Design Decisions in Suboptimal Heuristic Search-Based Systems

by

Richard Anthony Valenzano

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Heuristic search has been shown to be an effective way to solve state-space problems. While many heuristic search techniques are guaranteed to find the best solution, these are often not feasible given practical resource requirements. In such cases, it is necessary to sacrifice solution optimality in exchange for a faster search.

When a system designer is building a suboptimal heuristic search system to solve a set of given state-space problems, there are many decisions to be made that can greatly impact system performance. These include the need to select an appropriate algorithm from the many that have been proposed in the suboptimal heuristic search literature, selecting values for the parameters in these algorithms, and deciding on which search enhancements to employ.

The goal of this dissertation is to aid a system designer in this endeavour by increasing our understanding of what choices need to be considered and how these choices impact algorithm performance. In particular, we show that this large design space can be handled effectively through the use of an algorithm portfolio by building the state-of-the-art multi-core planner, `ArvandHerd`. Next we isolate the impact of inducing random exploration into greedy algorithms, and demonstrate that this option can often add useful variation into search. We then identify what choices are available to a system designer when there are given requirements on the quality of solutions found — even non-traditional ones — by showing that many existing algorithm frameworks can be modified accordingly to be guaranteed to satisfy a large range of possible requirements. Finally, we will examine the technique of "not re-expanding nodes" to better understand how the choice of whether or not to use this technique can impact the quality of solutions found.

Preface

Most of the research described in this thesis has been previously published, while some of it was performed in collaboration with others. Below, we mention these publications and identify my contributions to the corresponding projects.

Chapter 3 is based on a publication that appeared at the 2012 European Conference on Artificial Intelligence [96] which was preceded by a technical report [97]. The planner described in those papers was built in collaboration with Hootan Nakhost and Martin Müller. My contributions included parallelizing the system, and empirically evaluating the resulting planner.

Chapter 4 is based on a technical report [98] and a publication at the 2014 International Conference on Automated Planning and Scheduling. This latter publication was written in collaboration with Fan Xie, who initially proposed and performed the experiments regarding the use of randomly selected preferred operators. I re-ran these experiments and connected the experiment with that of isolating the impact of the knowledge used by GBFS enhancements. I am also responsible for the remaining work detailed in that chapter.

The content in Chapter 5 was first outlined in an extended abstract that appeared at the 2012 Symposium on Combinatorial Search [94], and then detailed in a full publication that appeared at the 2013 International Conference on Automated Planning and Scheduling [95]. These publications were written with Shahab Jabbari Arfaee, Jordan Thayer, and Roni Stern. I developed the theoretical foundations described in the chapter, ran the experiments on planning domains, helped design the remaining experiments, and was the lead in writing these papers.

The final Chapter 6 is based on a publication that appeared at the 2014 AAAI conference [99]. This paper is all original work.

In memory of my father,

Dr. Luch Valenzano

Acknowledgements

I am extremely grateful for the support and mentorship of my supervisors Jonathan Schaeffer and Nathan Sturtevant. They always provided me the freedom to explore what I found interesting, while still being there to offer guidance whenever I stumbled. I will always be very thankful for how how much I learned from them and for how much I have personally grown under their tutelage.

I would also like to thank my committee members Wheeler Ruml, Vadim Bulitko, and Michael Buro for their insights and kind words about my work. I am also grateful to Martin Müller and Rob Holte for their guidance and support over the year. They were always available for discussions about research, science, and life, and I will always be appreciative for that and for Martin's work as both a collaborator and on my committee. Ariel Felner and Carlos Linares López were also always very supportive of my work over the years, and I am also very thankful for that. I would also like to acknowledge the financial support of NSERC and GRAND.

Much of the work in this thesis has also been a collaborative effort with several others including Roni Stern and Jordan Thayer, and I would like to thank them for being such great co-authors. Of those I have worked with, I would particularly like to thank Levi Lelis, Shahab Jabbari Arfaee, Hootan Nakhost, and Fan Xie. You have all been both great friends to me and excellent colleagues and co-authors from whom I have learned a lot.

Many other friends have also been there for me over my years as a doctoral student, both academically and personally. These include Sheehan Khan, Mike Johanson, Nolan Bard, John Hawkin, Marc Gendron-Bellemare, Kit Chen, Rich Gibson, Joe Buscemi, Paul DiGiandomenico, Joe Calarco, Mark DiGiandomenico, Matt Fava, Fred Caprara, Dave Giralico, and many others. I will always be grateful

for your friendship and support over the years.

I also could not have gotten through this challenge and the others I have faced without my family, particularly my mother, Lorraine, and my sisters, Julie and Teresa. Thank you so much for your unending love and support.

Finally, I want to thank Jacqueline Smith. It has been a joy to experience the ups of the last few years with you, and I am equally as grateful for all the times you have been there to help me through the downs. Thank you for being there through both this and life's other endeavours.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the classical types of problems considered in the field of artificial intelligence are **planning tasks**. The goal in such tasks is to automatically find a sequence of actions to get from "point A" to "point B" in some given environment. For example, consider the task of finding a route between two locations on a map, or that of finding a sequence of cube twists that solves a given Rubik's cube configuration. In problems like these — which are said to be **discrete**, **perfect information**, and **deterministic** problems — one popular approach is called **heuristic search**. Heuristic search algorithms work by iteratively constructing sequences of actions that are candidates to be solutions. The key component of this procedure is the use of **heuristics**, which are functions that encode domain knowledge that are expected to aid the problem-solving process.

Heuristic search techniques have been shown to be successful in a wide range of domains including the aforementioned route pathfinding [27] and Rubik's cube problems [53], model-based diagnosis of faulty hardware and software systems [103], DNA sequence alignment [110], sewer placement for subdivisions [3], and robotics [58]. In such domains, it is often the case that there are multiple sequences of actions that achieve the desired goal conditions from the given initial **state** of the environment, and some of these solutions may be considered better than others. For example, when finding a route between two locations on a map, we would typically prefer shorter routes over longer ones. While many heuristic search algorithms can be guaranteed to only return the "best" of these solutions, these **optimal** approaches often require a very resource-intensive search. For example, finding the optimal so-

lution to a given Rubik's cube may take hours of computation [53]. When such time requirements are not acceptable — for example, if a human is waiting to manually implement the found solution — it is necessary to allow for the search to return **suboptimal** solutions in exchange for a less resource-intensive search.

There have been many such suboptimal heuristic search algorithms that have been developed in the literature, including WA* [71], A^*_ϵ [69], WIDA* [52], and EES [90]. These algorithms typically have a variety of parameters and other design decisions that must be set before the algorithm can be deployed. As such, the resulting space of possible design decisions can be very large. For example, suppose that we are designing a system to solve route pathfinding tasks in which it is possible to move in at most one of the four cardinal directions in any state, and assume we are only considering the four algorithms mentioned above as the basis of the system. All four have a parameter called a **weight** that can be set as any real number no smaller than 1, three of these algorithms (WA*, A^*_ϵ , and EES) have an option to **re-expand** nodes or not, and there will be a total of $4! = 24$ **static operator orderings**. Even if we only consider 10 possible values for the weight and 5 possible heuristics, this means that there are over $8,000$ possibilities for the system, and this does not even account for the fact that A^*_ϵ and EES can each be configured to employ multiple heuristics.

As each of these decisions can greatly impact algorithm performance, constructing a high performance system for a given domain requires the system designer to deal effectively with this large space of choices. This involves identifying what algorithm options can impact performance and which are even applicable given requirements that must be satisfied regarding the **quality** of the solutions found (*ie.* how close to optimal the solutions returned must be). Ideally, we would also like to know as much about the impact that each design decision will have on the system performance on the types of tasks being considered so that they each design decision can be set effectively. In the case that little is known about the problems to be solved ahead of time, building a strong system also requires minimizing the chance that the selected system is not well-suited for the given task.

1.1 Contributions

The goal of this dissertation is to aid a system designer in dealing with this space of design decisions when building a suboptimal heuristic search-based system. In particular, we make the following contributions.

1.1.1 Multi-Core Planning with a Portfolio

An automated planner is a system that must solve tasks about which little may be known prior to when problem-solving begins. Committing to a single algorithm and a corresponding set of design decisions can therefore be risky since different problems are best solved with different approaches. To deal with this issue, one can use multiple approaches simultaneously in an **algorithm portfolio** to combine of strengths of each. In Chapter 3, we will demonstrate that this approach can be used to build a state-of-the-art multi-core planner. The resulting multi-core planner, `ArvandHerd`, won the multi-core track of the last two International Planning Competitions in 2011 and 2014. It was built specifically to address some of the challenges inherent with using an algorithm portfolio in a shared memory system, and we will show that its strong performance is the result of the fact that it combines different techniques that are each best for different types of planning tasks.

Chapter 3 is based on two existing publications. The first was the description of `ArvandHerd` that was submitted along with the planner to the 2011 competition [97]. The second publication detailed the subsequent analysis of this planner that was performed after the competition results were released. This latter work was published at the European Conference on Artificial Intelligence in 2012 [96].

1.1.2 Random Exploration and Random Baselines

The heuristics used to guide search algorithms rarely provide perfect guidance and often identify states in the environment as being closer to goal states than they actually are. In algorithms like Greedy Best-First Search (GBFS), such heuristic error can lead the algorithm astray and thereby cause poor performance. In Chapter 4, we will demonstrate that by having GBFS occasionally ignore the advice of the heuris-

tic, this algorithm's performance often increases since its sensitivity to heuristic error is decreased. While existing algorithms have often employed such **random exploration**, they have typically done so in a combination with other techniques. In contrast, we introduce a simple technique called **ϵ-greedy node selection** that clearly isolates the impact of adding random exploration to a search, and demonstrates it to be positive in either a simple or a state-of-the-art planner. As such, this investigation will clearly identify random exploration as an option that designers of GBFS-based systems should consider.

A second technique for adding random exploration into a search, **heuristic perturbation**, is also introduced. Heuristic perturbation will be shown to lead to large improvements in performance in some domains and large decreases in others. While this "risky" behaviour may be undesirable, we will show that this riskiness can be exploited by pairing algorithms that use this technique with algorithms that use ϵ-greedy node selection in a portfolio.

We also re-examine existing techniques for enhancing the performance of GBFS. These techniques push the algorithm to occasionally go against the advice of the heuristic, though the decision to do so is based on some automatically derived problem structure instead of randomness. Given the success of random exploration, we argue that these techniques should be compared against appropriate random baselines to ensure that the structure they exploit is inducing variation that goes beyond what can be accomplished simply using randomness. The resulting study provides further confirmation of the value of the preferred operator and multi-heuristic best-first search enhancements [38, 74], and thereby provides further reason for system designers to use these techniques.

The work in Chapter 4 is based on two publications. The first was a technical report that introduced ϵ-greedy node selection and heuristic perturbation [98]. The second paper, which was published at the 2014 International Conference on Automated Planning and Scheduling [101], connected the idea of ϵ-greedy node selection to that of understanding random exploration and of suggesting the need to re-evaluate the existing planning enhancements.

1.1.3 Algorithms for Different Solution Quality Guarantees

If there are restrictions on how far from optimal the solutions returned can be, then a first step when building a heuristic search-based system is to identify the set of search algorithms that are guaranteed to satisfy that given solution quality requirement. While several existing algorithms were known to return solutions that cost no more than a given factor larger than the optimal solutions, it was previously not clear what algorithms were applicable for other types of quality guarantees such as the requirement that the solutions found are no more than an **additive** constant larger than optimal.

In Chapter 5, we will show that several existing algorithm frameworks will satisfy such alternative types of bounds, provided that they are modified appropriately. These frameworks include **anytime algorithms** [33, 58], **best-first search algorithms** [35, 71] , **iterative deepening algorithms** [50], and **focal list based algorithms** [69, 90]. In doing so, we generalize what these existing algorithm frameworks can be used to do and thereby identify what set of algorithm options are available to a system designer who must satisfy some given, perhaps non-traditional, solution quality requirements. We then test these results by using them to construct algorithms that satisfy additive requirements and which display the desired behaviour in practice: as the solution quality requirements are loosened, the algorithms typically increase their performance.

This chapter was published at the 2013 International Conference on Automated Planning and Scheduling [95]. That work was preceded by a two-page abstract that appeared at the 2012 Symposium on Combinatorial Search [94].

1.1.4 Not Re-Expanding Nodes and Solution Quality

Best-first search algorithms are a class of algorithms that are commonly used in the construction of a suboptimal heuristic search-based system. During the execution of these algorithms, it is common to find multiple action sequences to the same non-goal state. By default, if a lower cost action sequence is found that achieves a state through which other action sequences have already been explored, these al-

gorithms always reconsider (or **re-expand**) action sequences that pass through that state. Doing so is often necessary to satisfy given solution quality guarantees. However, doing so can often greatly increase the time necessary to find a solution. To avoid the overhead of re-expansions, some best-first search algorithms have often been configured so that they simply do not re-explore the action sequences that go through a state once that state has been explored for a first time. While doing so often speeds up the search, it often results in lower quality solutions being found. Understanding the impact that this technique has on solution quality is therefore necessary for a system designer who is deciding whether or not to configure their search to re-expand nodes.

In Chapter 6, we formally analyze the impact that not re-expanding nodes has on solution quality by showing that the worst-case quality can be bound in terms of a property of the heuristic called the heuristic's **inconsistency**. This bound will then be used to give worst-case guarantees for certain different types of heuristics. First, we will show that if the heuristic is **admissible** (*ie.* that it never over-estimates the cost to get to the nearest goal from any state) then the cost of solutions found can be at most quadratic in the optimal solution cost. Then we will consider the case in which the heuristic is the result of weighting an admissible heuristic, as it is in WA* , and offer bounds for when the weighted heuristic has inconsistency along it. Finally, we will identify the types of weighting in which the same bound is satisfied when weighting a consistent heuristic regardless of whether nodes are or are not re-expanded. This investigation will therefore not only increase our understanding of the impact of this design choice on performance, but will also identify when this technique can be used while still satisfying desired solution quality requirements.

The work in this chapter first appeared at the 2014 AAAI Conference on Artificial Intelligence [99].

1.2 Chapter Summary

In this chapter, we have described the problem facing a system designer who is building a heuristic search-based system: they must effectively deal with a large

space of possible design decisions. We then outlined four contributions made to help a system designer in this endeavour that aim at better identifying the options available, improve our understanding of how some of these options will impact performance, and handle the situation in which problem-solving must begin when little is known about the task to be solved. Each of these will be considered in a separate chapter below. Before doing so, we will first introduce many of the concepts and the notation to be used in the remainder of the thesis in the next chapter.

Chapter 2

Background

In this chapter, we provide a formal definition of a planning task and provide the notation that will be needed for reasoning about state-spaces and the algorithms that search for solutions in them. Once this notation is established, we will then define an algorithm framework called **open and closed list based algorithms**. This framework will generalize many existing algorithms like A*, WA*, and A$^*_\epsilon$ by allowing for the use of different policies for selecting nodes for expansion and for when to re-expand a node. Many of the properties that are known to hold for the aforementioned existing algorithms will then be shown to also hold for this generalization. We will then introduce automated planning and several other domains that will be used in our empirical evaluations, and describe related work on the use of automatic configuration tools for dealing with large spaces of design decisions.

Many of the formal proofs shown in this section represent generalizations of existing proofs. As many of these are quite technical when these generalizations are allowed — particularly when the path found to a node is not stored explicitly, but is instead stored implicitly using parent pointers as is typically done in practice — they have been moved to Appendix A.

2.1 Terminology and Notation

In this section, we define the planning task and introduce notation that will be used throughout this document. We begin by introducing the idea of a state-space with an example. This will then allow us to formalize the idea of a planning task.

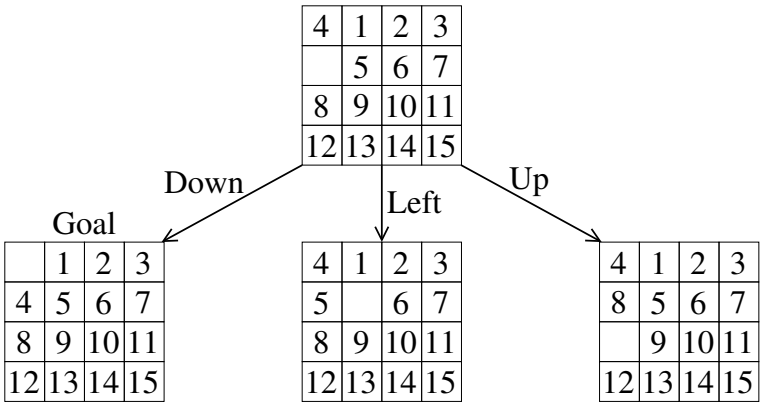


Figure 2.1: Example 15 puzzle states and transitions.

2.1.1 State-Space Examples

So as to more easily introduce the notion of a **planning task** and the notation to be used in this thesis, we begin with several examples of the types of problems we will be dealing with. The first is the **15 puzzle**, which is a common children's toy that is also a standard test domain for heuristic search algorithms. This puzzle consists of a grid with 4 rows and 4 columns. At any time, 15 of the grid locations contain a **tile**, each of which is labelled with a unique natural number from 1 to 15. The other location is empty, and this **blank space** is what allows for the puzzle to transition from one configuration into another. These transitions are the result of applying an **action** or **operator** to a puzzle configuration. In this domain, the actions correspond to sliding a single tile which is adjacent to a blank space into the blank space. Once a tile is moved into the empty location, that tile's previous location becomes empty. For example, Figure 2.1 shows a particular configuration of the 15 puzzle (the grid at the top), the actions applicable in this state (the labelled arrows), and the states that result from applying each of these actions (the three grids on the bottom of the figure). As the blank space is in the leftmost column, there is no tile to the left of the blank space. Therefore, the actions that are applicable to the puzzle in this configuration are to move the tile labelled 4 down into the empty location (the action labelled "Down"), move the tile labelled 5 left into the empty location (the action labelled "Left"), and move the tile labelled 8 into the empty location (the action labelled "Up").

In the language used when describing planning tasks, this description defines the 15-puzzle **state-space**. More generally, each configuration of the world (*ie.* each legal combination of locations for the tile and the blank space) corresponds to a **state**. A state-space then consists of the collection of all states in the environment, and a set of **transitions** between these states. In the 15 puzzle example, these transitions correspond to the four possible actions ("Up", "Down", "Left", and "Right") which define which states can be reached from a given state. As in all domains we consider in this thesis, actions will be assumed to be **deterministic**. This means that when applying an applicable action in a given state, there is only one possible outcome and that outcome is known prior to applying that action.

Given a state-space, the tasks we are considering are characterized by a given **initial state** and a given set of **goal states**. Intuitively, the start state corresponds to where you are starting at in the environment, and the goal states corresponds to where you want to get to. The objective is then to find a sequence of actions which, when applied to the start state, transform that initial state into one of the goal states. Such a sequence of actions is called a **solution**. For example, the standard goal state for the 15 puzzle problem is labelled "Goal" in Figure 2.1. Given this single goal state, if the top state in Figure 2.1 was the given initial state, then one possible solution would be the action sequence consisting only of the "Down" action.

In the 15 puzzle, it is easy to check if any of the four possible operators is applicable in a given state based on the location of the blank space. For example, the action "Up" is applicable unless the blank space is in the bottom-most row. Similar rules can be used to test the applicability of the other actions. This means the definition of the operators alone allows us to generate all the states which are exactly one action away from a given state.

In contrast, consider the problem of pathfinding from a given location in a map to a goal location. The typical way to approach this problem is to discretize the map and treat each location (*ie.* map address) as a separate state. Transitions in this state-space correspond to traversing along a road between adjacent locations. Determining which actions are applicable in a given state will therefore vary greatly from state-to-state. For example, it may not be possible to move northward in all

states and roads may not even correspond to the cardinal directions. This means that it is difficult (or at least inefficient) to encode these state transitions as operators whose applicability can be tested. In such cases, it will instead be more convenient to maintain a list of all possible transitions for each given state.

The type of representation just considered for pathfinding is called an **explicit state-space representation** due to the explicitly maintained a list of possible transitions. This representation also makes it clear that such planning tasks can be viewed as pathfinding in a graph. To see this, notice that we can construct a graph-search problem such that solving that problem would solve the initial pathfinding task. The graph being considered would have one unique **node** or **vertex** for each state in the state-space (*ie.* map location), and the set of **edges** would be given by the set of possible transitions. The problem is then to find a path in this graph from the initial node to the node corresponding to a goal state.

The same is also true of the 15 puzzle, even though the transitions are encoded using a set of rules. The representation used for the 15 puzzle is called an **implicit state-space representation**, but there is also a graph underlying such state-spaces. This graph would again have one unique node for each state in the state-space. The edge set could then be constructed by iterating over all the vertices, using the action rules to find the transitions that are applicable to each state, and then adding the corresponding edges to the graph.

While for many tasks actually building this graph may not be possible under realistic time or memory constraints, this construction does demonstrate the equivalence between looking for a solution to a state-space problem and pathfinding in a graph. This way of viewing the problem will be convenient to work with and so it will be the main one we use in the remainder of this thesis. We now formalize the terminology we will be using regarding pathfinding in a graph in the next section.

2.1.2 Planning as Pathfinding in a Graph

In this section, we will define some preliminary notation regarding graph edges, nodes, and heuristic functions. Included in this section will be the formal definition of a **planning task**.

Graph Definitions

As described above, we will consider planning tasks by their equivalent representation as pathfinding in a graph $G(V, E)$ where V is the set of nodes in G and E is the set of edges in G. Since each node corresponds to exactly one state in the state-space, we will often use the terms "node" and "state" interchangeably and often refer to G as being the state-space of a given problem.[1]

(p, c) will be be used to denote the edge from p to c in G, and edges will be assumed to be **directed** unless stated otherwise. Where $(p, c) \in E$, the node c will be said to be a **successor** or **child** of p, and p will be said to be a **predecessor** or **parent** of c. A domain will said to be **undirected** if $\forall n_1, n_2 \in V$, $(n_1, n_2) \in E$ if and only if $(n_2, n_1) \in E$. In undirected graphs, we will say that if $(n_1, n_2) \in E$, then n_1 and n_2 are **adjacent** or **neighbours** in G.

All edges will have an associated real-valued **transition cost**, enabling this representation to have actions with different costs. For example, in the pathfinding example given above, edge costs will allow for the definition of streets which are not all of the same length. The cost of an edge $(p, c) \in E$ will be denoted by $\kappa(p, c) \in \mathbb{R}$, where $\kappa(p, c) = \infty$ if $(p, c) \notin E$, and $\kappa(p, c)$ is finite otherwise. In this thesis, we will also assume that for any edge (p, c), $\kappa(p, c) \geq 0$.

For convenience, we will let $succ$ denote the set of successors of a given node. Formally, this means that $succ$ is defined as follows:

$$succ(n) = \{n' \mid (n, n') \in E\} \ .$$

In this thesis, we will always assume that $|succ(n)|$ is always finite.

This notation will now allow us to formally describe paths in the graph and to introduce notation for describing such paths. We do so in the next section.

Paths in a Graph

A **path** in a given state-space will be defined as a sequence of nodes $P = [n_0, n_1..., n_k]$ such that for any $0 \leq i < k$, $n_{i+1} \in succ(n_i)$. For convenience, we will use $n \in P$

[1] While the terms "state" and "node" are used interchangeably in this thesis, the term "node" has been used to mean various different things by different authors. We discuss some of the alternatives in Section B.1 of Appendix B.

to denote that n is one of the nodes along path P. Where $P = [n_0, n_1..., n_k]$, if $j > i$ then n_j will be said to be an **descendent** of n_i on P and n_i will be said to be a **ancestor** of n_j on P. Alternatively, if $j > i$ then we will say that n_j is **deeper** than n_i on P and n_i is **shallower** than n_j on P. If $P = [n_0, ..., n_k]$, then n_k will be said to be the **deepest** node on P.

Like edges, paths will also have an associated **path cost**, denoted by $C(P)$. This cost will be determined by a **path cost function**, which is based on the costs of the edges along the path. Where $P = [n_0, ..., n_k]$, the path cost function used in this thesis is as follows:

$$C(P) = \sum_{0 \leq j < k} \kappa(n_j, n_{j+1}) \ .$$

In this thesis, we will always assume that the "best" of two paths to the same node will be the path with the lowest cost.

Given a path $P = [n_0, ..., n_j, ..., n_k]$, we will say that the path $P' = [n_0, ..., n_j]$ is a **prefix** of P and that P is an **extension** of P'. We will also say that P' can be **extended into** P with path $[n_j, ..., n_k]$.

The Planning Task and Simplifying Assumptions

With this terminology established, we can now formally define a planning task.

Definition 2.1.1. *A **planning task** Γ is defined by the tuple $\Gamma = \langle G(V, E), n_{init}, V_{Goal} \rangle$ where G is a graph, $n_{init} \in V$, and $V_{Goal} \subseteq V$. The task is to find a path $P = [n_0, ..., n_k]$ such that $n_0 = n_{init}$ and $n_k \in V_{Goal}$.*

In this thesis, we refer to n_{init} as the **initial node**, V_{Goal} as the set of **goal nodes**, and the path P from n_{init} to some node in V_{Goal} as a **solution path**.

Notice that we defined planning tasks in terms of a graph instead of a **multi-graph**. As such, for any two nodes $n_i, n_j \in V$ there is at most one edge $(n_i, n_j) \in E$. If there is a state s in an implicit representation of a given state-space such that two different actions both result in a state s' when applied to s, then we will only include the action with the lowest cost in the corresponding graph representation. This simplification is possible since if there is a solution to the state-space problem

that includes a transition from s to s', we might as well use the lower cost edge since doing so will result in a better solution.

Solution Quality Guarantees

Even with the assumptions given, there are still several common planning task variants that differ in the requirements placed on the quality of any solutions returned. The first of these variants is the **Optimal Planning Task**, in which the task is only considered solved if the solution returned is an **optimal solution path**. A solution path is said to be optimal if there is no other solution path that has a lower cost, and we will use C^* to refer to the optimal solution cost to a given planning task.

This solution quality requirement is relaxed in the second variant, the **Bounded Planning Task**, in which the cost of any solution returned must satisfy some bound given *a priori* of any search. The most common of these types of bounds is the **linear suboptimality bound**. This bound requires that that any solution returned have a cost of no more than a factor w larger than the optimal solution cost for some predefined value for w of at least 1. As such, any solution returned must have a cost of no more than $w \cdot C^*$, and this must be guaranteed even if the value of C^* is unknown.[2]

While requiring that any solution found has a cost of no more than $w \cdot C^*$ is the most common type of bound, there are other ways to specify solution quality requirements. For example, one alternative is to require that any solution found has a cost that is no greater than $C^* + \gamma$ where γ is a predefined constant such that $\gamma \geq 0$. Such alternative bounding paradigms have seen very little previous consideration in the search literature, though we identify how existing algorithms can be modified to satisfy such requirements in Chapter 5.

Unlike the optimal planning task and the bounded planning task, the final planning task variant we consider does not place any additional constraints beyond those already given in Definition 2.1.1. This planning task variant, which is called a **Satisficing Planning Task**, allows for any solution found to be considered acceptable

[2]A solution that satisfies a linear suboptimality bound has traditionally been said to be ϵ-admissible where $\epsilon = w - 1$. However, we will not use this term in this thesis to avoid confusion since we will use ϵ later in a different context.

regardless of how suboptimal it is. While a lower cost solution is preferred over a higher cost solution, the main objective of this variant is to simply find any solution.

2.1.3 Path Sets and Node Costs

To reason about how planning algorithms explore a given state-space, it is necessary to introduce some notation for different kinds of paths and for various properties of a node. We do so in this section.

Π will be used to refer to the set of all paths that are possible in a given graph G. Any $P = [n_0, ..., n_k]$ is in Π as long as $n_{i+1} \in succ(n_i)$ for any $0 \leq i < k$, and there are no restrictions on what n_0 or n_k may be. If we instead want to refer to the set of paths that begin at the initial node, we will use Π_{init}. This means that for $P \in \Pi_{init}$, $n_0 = n_{init}$ where $P = [n_0, ..., n_k]$. A path $P = [n_0, ..., n_k] \in \Pi_{init}$ will often be referred to as **candidate path** since P could be a solution path (if $n_k \in V_{Goal}$), or it may be possible to extend P into a solution but this will not be known without further exploration of the state-space. Note that for the sake of simplicity, we will also assume that for any $P \in \Pi_{init}$ where $P = [n_0, ..., n_k]$, $n_i \notin V_{Goal}$ for any $0 \leq i < k$. Finally, we will use Π_{Goal} to denote the set of solution paths to a given task. This means that for any $P \in \Pi_{Goal}$, $n_0 = n_{init}$ and $n_k \in V_{Goal}$ where $P = [n_0, ..., n_k]$. For any $P \in \Pi_{Goal}$, we will assume that the only goal node on P is the deepest node on P.

This notation will now also allow us to formalize the notion of the "optimal solution cost" as follows:

$$C^* = \min_{P \in \Pi_{Goal}} C(P) \ .$$

We will also use Π^*_{Goal} to denote the set of all optimal solution paths. This set is defined as follows:

$$\Pi^*_{Goal} = \{P \mid P \in \Pi_{Goal} \wedge C(P) = C^*\} \ .$$

Notice that $\Pi^*_{Goal} \subseteq \Pi_{Goal} \subseteq \Pi_{init} \subseteq \Pi$.

Reachability and Node Costs

We now consider notation for various properties of a given node n such as how far n is from the initial node, how far n is from the nearest goal node, and the distance between two nodes n and n'. Such quantities will be denoted as follows:

- $g^*(n)$ will denote the cost of the lowest-cost path from n_{init} to n. A path $P \in \Pi_{init}$ which has a cost of $g^*(n)$ will be called an optimal path to n.
- $g^*(n, n')$ will denote the cost of the lowest-cost path from n to n'. A path $P \in \Pi$ from n to n' which has a cost of $g^*(n, n')$ will be called an optimal path from n_i to n_j.
- $h^*(n)$ will denote the cost of the lowest cost path to any of the nodes in V_{Goal}. A path $P \in \Pi$ from n to $n_g \in V_{Goal}$ which has a cost of $h^*(n)$ will be called an optimal solution path from n.

If there exists a path from a node n to any of the nodes in V_{Goal}, then $h^*(n)$ will be finite. In that case, the goal nodes will be said to be **reachable** from n. If no such path exists, then the goal nodes are **not reachable** from n and $h^*(n) = \infty$. Similarly, $g^*(n)$ will only be finite if n is reachable from n_{init}, and $g^*(n, n')$ will only be finite if there exists a path from n to n'.

Notice that according to these definitions, $g^*(n) = g^*(n_{init}, n)$ for any n and that $h^*(n_{init}) = g^*(n) + h^*(n) = g^*(n_g) = g^*(n_{init}, n_g) = C^*$ where $n_g \in V_{Goal}$ and $P = [n_{init}, ..., n, ..., n_g]$ is an optimal solution path.

2.1.4 Heuristic Functions

The planning community has developed several different approaches for solving planning tasks including SAT-solver based planners [78] and plan-space planners [29]. Our focus in this thesis on the use of **heuristic search algorithms** for planning. These algorithms explore the state-space in search of solution paths with the use of a **heuristic function**, defined as $H : V \rightarrow \mathbb{R}$.[3] The purpose of this function

[3]Many authors use h to refer to a heuristic instead of H. We will explain this difference below when describing **admissible** heuristics.

is to offer a relative ordering of nodes in terms of how close they are to the nearest goal nodes. This means that if $H(n) < H(n')$, the heuristic function is suggesting that n is closer to the goal than n'. Heuristic search algorithms can then use such information to give preference towards exploring the region of the state-space led to by n prior to the region led to by n'.

Ideally, the heuristic function would perfectly order nodes in terms of how close they are to the nearest goal node. This would be the case, for example, if $H = h^*$. In such a situation the heuristic would exactly identify which successor of a given node n was closer to a goal, and it would be trivial to find optimal solutions through a greedy exploration of the state-space. As h^* would therefore provide excellent guidance, and it also offers exact information regarding the cost to the nearest goal node from every node, it is often referred to as the **perfect heuristic**.

Such perfect guidance is rarely available, and instead, most heuristics are built in an effort to approximate h^* as accurately as possible. For example, when looking for routes from a given starting location to a given goal location in a road map, it is common to use the straight-line distance between the current location and the goal location. This heuristic implicitly encodes directional knowledge that favours nodes that involve moving in the direction of the goal as opposed to those that move away from it. Other heuristic functions do not attempt to approximate h^*, and are instead intended to offer information on the relative utility of how close a node is to a goal. For example, consider the heuristic $H = 1000 \cdot h^*$. Clearly H is not providing an accurate estimate of h^*, but $H(n) < H(n')$ is still an excellent indicator that n is closer to a goal than n'. This is also true of **distance-to-go** heuristics which estimate the number of nodes along the path to the nearest goal as opposed to the more typical **cost-to-go** heuristics which approximate h^*, the cost of the path to the nearest goal node [90].[4]

In the remainder of this thesis, a given heuristic will not be assumed to be estimating h^* unless otherwise stated. However, in all cases we will make the following assumptions:

- All nodes will have a non-negative heuristic value. Formally, this means that

[4] In unit-cost tasks, these two style of heuristics are the same.

$\forall n \in V, H(n) \geq 0$.

- If a goal node is reachable from a node n, the heuristic will not suggest otherwise. Formally, this means that for all n, if $h^*(n) \neq \infty$ then $H(n) \neq \infty$.

We will now define several heuristic properties which we will refer to throughout this dissertation.

Heuristic Admissibility

The following definition formalizes an important property of many heuristics in terms of their relationship with the perfect heuristic:

Definition 2.1.2. *A heuristic H is said to be **admissible** if for any $n \in V$*

$$H(n) \leq h^*(n) \ .$$

This means that a heuristic is admissible if it never over-estimates the optimal cost of the path from any node to the nearest goal node. Notice that if the heuristic in use is admissible then the heuristic value of any goal node will be 0.

If a heuristic H does not satisfy this property (*ie.* $\exists n, H(n) > h^*(n)$) then H is said to be **inadmissible**. Unless otherwise stated, we do not assume that H is either admissible or inadmissible in the remainder of this thesis, which is why we use the symbol H to refer to a heuristic instead of the symbol h. We will reserve the symbol of h to only refer to an admissible heuristic, which is how it is often used in the literature.

Heuristic Consistency

Let us now consider the property of heuristic **consistency**:

Definition 2.1.3. *A heuristic H is said to be **consistent** if for any $p, c \in V$ such that $c \in succ(p)$ the following is true:*

$$H(p) \leq H(c) + \kappa(p, c) \ .$$

This property of consistency ensures that the heuristic value does not decrease by "too much" (*ie.* by more than the edge cost) from any parent node to any of its children. Since the above definition is defined only over adjacent nodes, this property is sometimes referred to as **local consistency**. A heuristic is said to be **globally consistent** or **monotonic** if for any $n_i, n_j \in V$ it is true that

$$H(n_i) \leq H(n_j) + g^*(n_i, n_j) \ .$$

As shown by Pearl [68], these two forms of consistency turn out to be equivalent. Due to the equivalency of these properties, we will simply use the term "consistency" to refer to either the local or global form of this property, each of which will be used when it is most convenient.

Notice that the global form of consistency can be used to show that any consistent heuristic is also admissible. This is because global consistency implies that for any $n \in V$, if $n_g \in V_{Goal}$ is one of the nearest goal nodes for n (*ie.* $g^*(n, n_g) = h^*(n)$) then the following is true:

$$H(n) \leq H(n_g) + g^*(n, n_g) \ .$$

Since $H(n_g) = 0$ because $n_g \in V_{Goal}$ and $g^*(n, n_g) = h^*(n)$, this statement is equivalent to $H(n) \leq h^*(n_g)$. Therefore, any consistent heuristic is necessarily admissible. Note, however that the converse of this statement is not true.

2.2 Open and Closed List-Based (OCL) Algorithms

We now consider some of the algorithms which use heuristics for solving planning tasks. In the search literature, there have been many developed heuristic search algorithms including A* [35], IDA* [50], WA* [71], A$^*_\epsilon$ [69], and others. Some of these algorithms — including A*, WA*, and A$^*_\epsilon$ — have many similarities. These similarities include that they all run by iteratively building up candidate paths, and that they use data structures called the **open** and **closed** lists for doing so. We will refer to this class of algorithms as **Open and Closed List-Based (OCL) Algorithms** and define an abstract generalization of this class for which we can prove properties that hold true over all such algorithms. As it will be easier to understand

this generalization by starting with a concrete example and then showing what parts are generalized, we begin with a description of the A* algorithm.

2.2.1 A* as an Example of an OCL Algorithm

The A* algorithm — which is shown in Algorithm 1 — explores the state-space by iteratively building up candidate paths that start from n_{init} until a complete solution path is found. We begin this section with a high-level description of how this algorithm progresses through the state-space and the various structures used for doing so. We will then consider its execution in more detail below.

On each iteration, the algorithm selects the "most promising" candidate path P of those currently known (a notion we will formalize below), and **expands** the deepest node on P. This means that where $P = [n_0, ..., n_k]$, a list L_{n_k} is constructed consisting of all the successors of n_k. The nodes in L_{n_k} are said to have been **generated** by the expansion of n_k. For each successor $c \in L_{n_k}$, the result of the expansion of n_k is that the algorithm has found a new candidate path to c given by $P_c = [n_0, ..., n_k, c]$. P_c is then added to the list of candidate paths being considered if it is either the first or lowest-cost path found to c. This process then continues until a candidate path P_g is found for which the deepest node on P_g is a goal node.

At any time during the execution of the algorithm, the nodes in the candidate paths currently under consideration are stored using two sets, the **open** and **closed** lists, denoted as OPEN and CLOSED, respectively. The open list contains the deepest node of every candidate path currently under consideration and the closed list contains all nodes that have previously been expanded. Many of the nodes in the closed list will be the ancestor along some candidate path of some node (or many nodes) in the open list. At any time during the execution of A*, the union of these lists represents the complete set of all nodes that have been generated thus far.

To maintain the actual candidate paths currently under consideration, for each node n in the open list, A* could maintain the list of ancestors to n along the candidate path that was found to n. However, these paths may be very long and so maintaining such lists would require a lot of memory. As such, these candidate paths are instead maintained implicitly using **parent pointers**. For each node in

A* (Initial node n_{init}):

1: CLOSED $\leftarrow \{\}$
2: OPEN $\leftarrow \{n_{init}\}$
3: $parent(n_{init}) = NONE$
4: $g(n_{init}) = 0$
5: **loop**
6: **if** OPEN $= \{\}$ **then** ▷ OPEN is empty
7: **return** [] ▷ No solution path exists
8: $n \leftarrow \text{argmin}_{n' \in \text{OPEN}}\, g(n') + h(n')$
9: **if** $n \in V_{Goal}$ **then** ▷ Testing if n is a goal node
10: **return** `ReconstructPath`(n)
11: $L_n = \{c \mid c \in succ(n)\}$ ▷ Generate the children of n
12: **for all** $c \in L_n$ **do**
13: **if** $H(c) = \infty$ **then** ▷ Goal not reachable from c
14: **continue** ▷ Skip to next child
15: **if** $c \in$ OPEN **then**
16: **if** $g(c) > g(n) + \kappa(n, c)$ **then** ▷ Found better path to $c \in$ OPEN
17: $parent(c) \leftarrow n$
18: $g(c) = g(n) + \kappa(n, c)$
19: **else if** $c \in$ CLOSED **then**
20: **if** $g(c) > g(n) + \kappa(n, c)$ **then** ▷ Found better path to $c \in$ CLOSED
21: $parent(c) \leftarrow n$
22: $g(c) = g(n) + \kappa(n, c)$
23: CLOSED $\leftarrow$ CLOSED $- c$ ▷ Remove c from CLOSED
24: OPEN $\leftarrow$ OPEN $\cup \{c\}$ ▷ Add c to OPEN
25: **else**
26: $parent(c) \leftarrow n$
27: $g(c) = g(n) + \kappa(n, c)$
28: OPEN $\leftarrow$ OPEN $\cup \{c\}$ ▷ Add c to OPEN
29: OPEN $\leftarrow$ OPEN $- n$ ▷ Remove n from OPEN
30: CLOSED $\leftarrow$ CLOSED $\cup \{n\}$ ▷ Add n to CLOSED

Algorithm 1: The A* algorithm

```
ReconstructPath(node n):
 1: return ReconstructPathRecursive(n, [])
ReconstructPathRecursive(node n, path [n_0, ..., n_k]):
 1: if n = NONE then
 2:     return [n_0, ..., n_k]
 3: return ReconstructPathRecursive(parent(n), [n, n_0, ..., n_k])
```

Algorithm 2: Extracting a solution path from the parent pointers.

either the open or closed list, the parent pointer of a node n, denoted as $parent(n)$, is a single pointer to the parent node of n along some candidate path. For example, if $P = [n_0, .., n_k]$ is a candidate path currently under consideration, then the parent pointers for nodes $n_0, ..., n_k$ maintained by A* will be $parent(n_j) = n_{j-1}$ for each $1 \leq j \leq k$ and $parent(n_0) = NONE$ since $n_0 = n_{init}$ and therefore has no parent. For any node n which has been previously generated, the candidate path being stored to n can be reconstructed by following the parent pointers from n back to the initial node. This process is shown in Algorithm 2.

Since A* only maintains a single parent pointer for any node n, the nodes in the open and closed lists could be joined using only the parent pointers to form a tree with n_{init} as the root. This means that only a single candidate path is maintained to n at any time, even if there are many possible ways to get to n (this will be stated formally in Section 2.2.4). Only one such path is needed since even if a number of paths have been found to n at any time, only the lowest-cost path of these must be maintained. Therefore, when multiple paths are found to the same node, it is necessary for there to be a way to compare the costs of these paths to determine which one to keep. This is the reason that A* maintains what is called the **g-cost** of all nodes that have been generated. This value, denoted as $g(n)$, is an estimate of the cost of the candidate path to n that is currently under consideration. While $g(n)$ will often be equal to the cost of the candidate path currently being stored to n, this is not true in all cases. However, the g-cost of n can be guaranteed to be an upper bound on this candidate path to n as will be described in Section 2.2.4.

A Detailed Description of A*

Having described the various components of the A* algorithm, we now consider how it executes in more detail. The algorithm begins by initializing the g-cost of n_{init} to 0 and the parent pointer of n_{init} to $NONE$. n_{init} is then placed on the open list, and the main loop — each iteration of which corresponds to a single node expansion — then begins.

As described above, a single node is selected from the open list for expansion on each iteration of the algorithm. As shown in line 8 of Algorithm 1, the node selected is the node from the open list with the minimum value of $g(n)+h(n)$, with ties being broken using any tie-breaking scheme.[5] In the case of A*, we assume that h is an admissible heuristic function. Since $g(n)$ is an approximation of the cost of the path found to n and $h(n)$ is an estimate of the cost of the path from n to the nearest goal node, $g(n) + h(n)$ is an estimate of the cost of a solution path through n. The algorithm is therefore ordering candidate paths according to which are estimated to lead to the lowest cost solution.

Once a node n has been selected for expansion, the algorithm then checks if n is a goal node (line 9). If $n \in V_{Goal}$, the solution path from n_{init} to n is extracted using the `ReconstructPath` function given in Algorithm 2. If $n \notin V_{Goal}$, then it is expanded with L_n being the generated list of successors (line 11).

Each child c of n is then considered in turn. First, the heuristic is used to check if a goal node is reachable from c (line 13). If not, then the algorithm no longer needs to consider candidate paths through c since none of them can be extended into solution paths, and c can be safely discarded. If the heuristic does not identify c as a node from which all goal nodes are unreachable, then it will be handled as needed depending on whether c was already in the open or closed lists prior to the expansion of n, or if it is through the expansion of n that c has been generated for the first time.

In the case that c is already in the open list (line 15), the algorithm checks if the g-cost of c (*ie.* the upper-bound of cost of the candidate path previously found to c)

[5] In practice, there are several standard tie-breaking schemes that are used with A* and other OCL algorithms. We will detail some of these below.

is larger than $g(n) + \kappa(n, c)$ (which corresponds to an upper-bound on the cost of the newly found path to c through n). If it is larger, then the parent pointer of c is changed to n, and the g-cost of c is updated to be that for the path to c through n (lines 17 and 18), since the new path is expected to be of a lower cost. If the g-cost of c is no larger than $g(n) + \kappa(n, c)$, then no updates are needed and the new path is disregarded. This is because the new path does not improve the estimate of the best path from n_{init} to c.

In the case that c is in the closed list prior to the expansion of n (line 19), the g-cost and parent pointers of c are again only updated if the new path found to c has a lower expected cost than the maintained path (lines 21 and 22). However, as opposed to the case where c was already in the open list, if c is in the closed list it is then moved back into the open to make it available to be expanded (lines 23 and 24). By allowing c to be **re-expanded**, the g-cost of the improved path can be propagated to the ancestors of c that are on the open list.

The final case for c is that it has been generated for the first time by the expansion of n (line 25). In this case, the g-cost of c is set to $g(n) + \kappa(n, c)$, the parent pointer of c is set to n, and c is added to the open list thereby making it available for expansion.

The iteration is completed by removing the expanded node n from the open list (line 29) and adding it to the closed list (line 30). In doing so, n is removed from being considered for expansion on the next iteration, since it has just been explored. Notice that moving n from the open list to the closed list combined with the checks in lines 15 and 19 ensures that a node can never be on both the open and closed list at the same time.

The next iteration then begins with the selection of a new node for expansion. This process then continues until either a goal node is found, or the open list is found to be empty. If either of these conditions becomes true, the algorithm terminates. If the algorithm terminates because the open list is found to be empty, then it can be guaranteed that there is no solution. This will be shown in Section 2.2.4.

OCL(Initial node n_{init}):

1: CLOSED $\leftarrow \{\}$
2: OPEN $\leftarrow \{n_{init}\}$
3: $parent(n_{init}) = NONE$
4: $g(n_{init}) = 0$
5: **loop**
6: **if** OPEN $= \{\}$ **then** ▷ OPEN is empty
7: **return** [] ▷ No solution path exists
8: $n \leftarrow$ `SelectNode(OPEN)`
9: **if** $n \in V_{Goal}$ **then** ▷ Testing if n is a goal node
10: **return** `ReconstructPath`(n)
11: $L_n = \{c \mid c \in succ(n)\}$ ▷ Generate the children of n
12: **for all** $c \in L_n$ **do**
13: **if** $H(c) = \infty$ **then** ▷ Goal not reachable from c
14: **continue** ▷ Skip to next child
15: **if** $c \in$ OPEN **then**
16: **if** $g(c) > g(n) + \kappa(n, c)$ **then** ▷ Found better path to $c \in$ OPEN
17: $parent(c) \leftarrow n$
18: $g(c) = g(n) + \kappa(n, c)$
19: **else if** $c \in$ CLOSED **then**
20: **if** $g(c) > g(n) + \kappa(n, c)$ **and** `ShouldUpdateNode`(n, c) **then**
21: $parent(c) \leftarrow n$
22: $g(c) = g(n) + \kappa(n, c)$
23: **if** `ShouldReopenNode`(n, c) **then**
24: CLOSED $\leftarrow$ CLOSED $- c$
25: OPEN $\leftarrow$ OPEN $\cup \{c\}$
26: **else**
27: $parent(c) \leftarrow n$
28: $g(c) = g(n) + \kappa(n, c)$
29: OPEN $\leftarrow$ OPEN $\cup \{c\}$ ▷ Add c to OPEN
30: OPEN $\leftarrow$ OPEN $- n$ ▷ Remove n from OPEN
31: CLOSED $\leftarrow$ CLOSED $\cup \{n\}$ ▷ Add n to CLOSED

Algorithm 3: The open and closed list-based algorithm framework. The differences with the A* code given in Algorithm 1 are underlined and shown in red.

2.2.2 OCL Algorithms as a Generalization of A*

Having described A*, let us now define an abstract generalization of this algorithm that will allow us to reason about a large class of algorithms simultaneously. We refer to this class of algorithms as **Open and Closed List-Based (OCL) Algorithms**, pseudocode for which is shown in Algorithm 3.

This code involves two generalizations of the A* code given in Algorithm 1, with the corresponding changes being shown underlined and in red. The first is that the policy for selecting nodes from the open list is no longer specified. This can be seen by comparing line 8 of Algorithm 1, with line 8 of Algorithm 3. The OCL code replaces the use of the function $g(n)+h(n)$ for selecting a node from the open list as done by A* with a call to an undefined function called `SelectNode`. By leaving this function unspecified, the OCL definition allows each OCL algorithm to define this function differently. In the case of A*, this function is defined as follows:

`SelectNode`(OPEN):
1: **return** $\text{argmin}_{n \in \text{OPEN}}\, g(n) + h(n)$

Algorithm 4: The A* `SelectNode` function

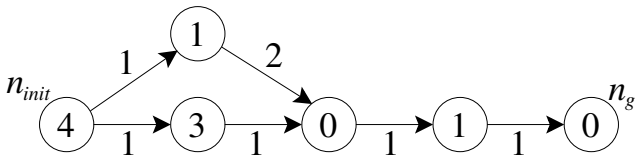
The abstract OCL definition also allows for some flexibility into the way algorithms handle the situation in which a path with a lower g-cost is found to a node c that is already on the closed list. Whereas A* always updates $g(c)$ and $parent(c)$, and always moves c back to the open list in this situation, these two actions are optional in the OCL definition. This flexibility is allowed through the use of two boolean functions. The first, `ShouldUpdateNode`, determines if the parent pointer and g-cost updates should be made, and it allows this decision to be made on a per-node basis. If these updates are performed then a second function, `ShouldReopenNode`, determines if the node should then be moved back onto the open list. This decision can also be made on a per-node basis.

In the case of A*, both of these functions will always return $True$. Any OCL algorithm which uses this same approach will be said to use a **full re-expansion policy**. Notice that this policy may result in the same node being expanded more than once. Another possible policy is to ensure that no node is ever re-expanded.

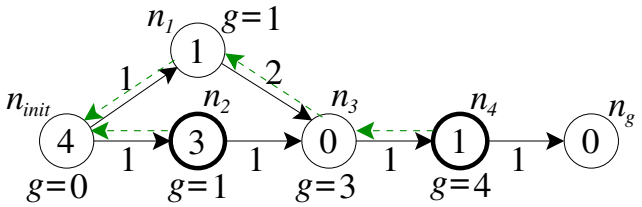
We will refer to such OCL algorithms as ones that do **not perform re-expansions**. This can be ensured by never moving nodes back from the closed list to the open list, either by having `ShouldUpdateNode` always return $False$ or by having `ShouldReopenNode` always return $False$.

This means there are two aproaches to ensuring that an OCL algorithm does not perform re-expansions. In the first, `ShouldUpdateNode` is set to always return $True$ and `ShouldReopenNode` is set to always return $False$. This technique will still update the parent pointer and g-cost of nodes that are in the closed list, but the lack of re-expansions prevents any g-cost improvements from being propagated to any ancestor nodes which are in the open list. In contrast, if `ShouldUpdateNode` is set to always return $False$, nodes can never be expanded more than once, but such updates will not be made. In terms of runtime, these two approaches will be identical (aside from the minor overhead in performing these updates). However, these two approaches may differ in terms of solution quality. If an OCL algorithm sets `ShouldUpdateNode` to always returns $True$ and `ShouldReopenNode` to always returns $False$, then it may allow for the candidate paths to nodes in the open list to be improved without propagating the g-cost improvement forward using re-expansions. The result is that if a goal node n_g is expanded when the g-cost and parent pointer updates are made, then the path reconstructed by `ReconstructPath` may have a lower cost than $g(n_g)$.

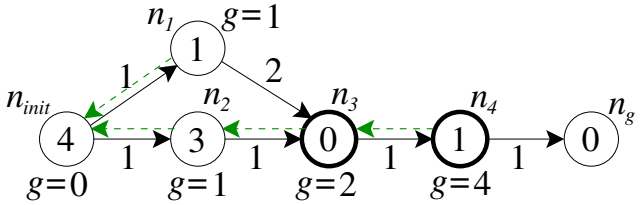
As a short-hand, we will use **rOCL** to refer to an OCL algorithm which uses the full re-expansion policy. For example, A* is an instance of an rOCL algorithm. Similarly, we will use **nrOCL** to refer to an OCL algorithm which never moves nodes back to the open list from the closed list, regardless of whether it updates the g-cost and parent pointers of a node or not. An OCL algorithm which does perform these updates is said to **always perform parent pointer updates**. Note, that this notation means we will use "OCL" as a short-hand for an OCL algorithm for which nothing is assumed about the re-expansion policy (*ie.* it may re-expand nodes all the time, never, or it may use some other policy).



(a) An example graph.



(b) An A* search after 3 node expansions.



(c) g-cost inaccuracy after 4 node expansions.

Figure 2.2: An example of g-cost inaccuracy.

2.2.3 The g-cost Function

Before considering some properties of these algorithms, it is first necessary to consider the g-cost function in more detail. The g-cost of a node n has typically been referred to in the search literature as being equal to the cost of the path held implicitly using the open and closed lists, and the parent pointers. While this would be true if the path to a node n were being maintained explicitly instead of when using parent pointers, and it is true in certain situations such as when using using a consistent heuristic with A*, this will not necessarily hold in general.

Figure 2.2 offers an example which demonstrates this behaviour. First, consider Figure 2.2a which shows the graph being used to demonstrate how g-cost inaccuracy may emerge during the execution of an OCL algorithm. As in all graphs in this thesis, nodes are represented as circles and edges are represented by arrows. The numbers besides the arrows correspond to the edge costs and the numbers inside the nodes correspond to the heuristic values. The node labelled "n_{init}" is the initial node, and the node labelled "n_g" is a goal node.

Figure 2.2b shows how an A* search would have explored the graph after 3 node

expansions, with the initial node being expanded first followed by n_1 and then n_3. The nodes that have been generated thus far have their g-cost at this time labelled, the bold nodes are currently those on the open list, and the green dashed arrows correspond to the parent pointers. Figure 2.2c then shows what happens after one more node expansion. In this case, the node marked n_2 is selected for expansion and a lower g-cost path is found to n_3. As a result, both the parent pointer and g-cost of n_3 are updated.

Now notice how the candidate path maintained to n_4 changes due to the 4-th node expansion from the path with a cost of 4 which passes through n_1, to the path of cost 3 that passes through n_2 and n_3. This change occurs without any updates being made to the g-cost and parent pointer of n_4. The result is that after 4 node expansions, $g(n_4)$ is inaccurately set as 4 even though the path implicitly stored to n_4 only has a cost of 3. This inaccuracy arises because the improvement made to the path found to n_4 is not immediately propagated to n_4. However, it is important to notice that while the g-cost of a node n does not necessarily exactly correspond to the cost of the path from n_{init} to n being maintained by the parent pointers, it still does correspond to the cost of some candidate path from n_{init} to n. In the case of Figure 2.2c, the value of $g(n_4)$ corresponds to the path to n_4 that passes through n_1 instead of the path through n_2 currently being maintained by the parent pointers. This property will be important when showing that any OCL algorithm terminates given a finite graph.

Figure 2.2 also demonstrates another important property of the g-cost function: the value of $g(n)$ can decrease over time. This can happen if a path with a lower g-cost is found to n while it is in the open list (line 18 of Algorithm 3) or if `ShouldUpdateNode` returns true and it is in the closed list (line 22). To track what the g-cost of a node is at a particular time, we introduce the following notation. We will use $g_t(n)$ to refer to the g-cost of n after t node expansions (or equivalently, after t iterations of an OCL algorithm). For example, consider node n_2 in Figures 2.2b and 2.2c. Recall that Figure 2.2b shows the state of the search after 3 node expansions and so $g_3(n_2) = 3$, while Figure 2.2c shows the state of the search after 4 expansions and so $g_4(n_2) = 2$. We will similarly use $parent_t(n)$, OPEN_t, and

CLOSED_t to refer to the value of the parent pointer of n after t expansions, the set of nodes in the open list after t expansions, and the set of nodes in the closed list after t expansions, respectively. Note that we will use $g(n)$, $parent(n)$, OPEN, and CLOSED instead of $g_t(n)$, $parent_t(n)$, OPEN_t, and CLOSED_t when the number of nodes that have been expanded is clear from the context.

g-cost Error

For the theoretical analysis presented later in the thesis, it will be convenient to refer to how much the g-cost of a node n deviates from the cost of the optimal path from n_{init} to n at different points during the algorithm's execution. We will refer to this deviation, denoted by $g^{\delta}_t(n)$, as the **g-cost error of n after t node expansions** and define it as follows:

Definition 2.2.1. *The g-cost error of a node n after t node expansions is given by*

$$g^{\delta}_t(n) = g_t(n) - g^*(n) \ .$$

Notice that the g-cost of a node n can only change during a particular iteration if n is newly generated, or if n is already in the open or closed list and the newly found path to n has a lower g-cost than $g(n)$. This means that if a node n is in the open or closed list after t node expansions, and no better path is found to it by the $t+1$-st expansion, then $g_{t+1}(n) = g_t(n)$ and $g^{\delta}_{t+1}(n) = g^{\delta}_t(n)$.

Simple Properties of the g-cost and g-cost Error Function

Having described $g(n)$ in detail, we now turn to showing simple properties of this function. We begin by considering how the g-cost of a node changes over time. Since the g-cost of a node n can only be updated to a lower value according to lines 18 and 22 of Algorithm 3, the value of $g(n)$ can only decrease during the execution of an OCL algorithm. This notion is formalized in the following observation:

Observation 2.2.2. *If a node n is first generated by an OCL algorithm by the t-th node expansion, then for any t' and t'' where $t \leq t' \leq t''$, it will be true that $g_{t'}(n) \geq g_{t''}(n)$ and $g^{\delta}_{t'}(n) \geq g^{\delta}_{t''}(n)$.*

Notice that the statement holds for the g-cost error of n since $g_t^\delta(n) = g_t(n) - g^*(n)$, $g^*(n)$ is a constant that does not change with t, and $g_t(n)$ can never increase.

The next property considers how the g-cost error is propagated from a parent n_{k-1} to a child node n_k where n_{k-1} is along an optimal path to n_k, and n_{k-1} is expanded before n_k. In this case, the g-cost error of n_k can be no larger than the g-cost error of n_{k-1} when it is first expanded for the remainder of the search. This is formalized by the following lemma:

Lemma 2.2.3. *Let $P = [n_0, ..., n_{k-1}, n_k]$ be an optimal path from $n_0 = n_{init}$ to n_k for $k > 0$. If n_{k-1} is expanded for the first time by the t-th node expansion of an OCL algorithm and n_k is not one of the first t nodes expanded, then the following is true for any $t' \geq t$*

$$g_{t'}^\delta(n_k) \leq g_t^\delta(n_{k-1}) \ .$$

The proof of this and the next lemma can be found in Section A.1.2 of Appendix A. If the OCL algorithm is always performing parent pointer updates, then the lemma can be strengthened so that the order in which n_k and n_{k-1} are expanded does not matter, and so that it does not only apply to the first time n_{k-1} is expanded.

Lemma 2.2.4. *Let $P = [n_0, ..., n_{k-1}, n_k]$ be an optimal path from $n_0 = n_{init}$ to n_k for $k > 0$. If n_{k-1} is expanded by the t-th node expansion of an OCL algorithm that always performs parent pointer updating, then the following is true for any $t' \geq t$*

$$g_{t'}^\delta(n_k) \leq g_t^\delta(n_{k-1}) \ .$$

These properties will now allow us to consider how the g-cost of a node n is related to the path from n_{init} to n that is stored implicitly using the parent pointers. The following theorem states that at any time for which n is in the open or closed list then this path is guaranteed to exist, it will be unique, it will have no cycles on it, and $g(n)$ will be an upper bound on its cost.

Theorem 2.2.5. *Suppose that there have been t node expansions of an OCL algorithm. If n is a node such that $n \in \text{OPEN}_t \cup \text{CLOSED}_t$, then there exists a unique path $P = [n_0, ..., n_k]$ from $n_{init} = n_0$ to $n = n_k$ such that the following are true:*

1. $\forall 0 \leq i \leq k$, $n_i \in \text{OPEN}_t \cup \text{CLOSED}_t$.

2. $\forall 1 \leq i \leq k$, $parent_t(n_i) = n_{i-1}$.

3. $\forall 0 \leq i < j \leq k$, $n_i \neq n_j$.

4. $g_t(n) \geq C(P)$.

The proof of this statement can be found in Section A.1.3 of Appendix A. Note that by stating that P is unique, we mean that there is no other path to n being maintained implicitly by the parent pointers.

2.2.4 Formal Properties of OCL Algorithms

In this section, we now turn to formally describing properties that are common to large sets of OCL algorithms. We begin by considering how OCL algorithms explore the state-space. This will be followed by an examination of how the open and closed lists can be used to find lower bounds for C^* during the search. Finally, we will identify cases in which OCL algorithms are guaranteed to find solutions.

The Exploration of Candidate Paths by OCL Algorithms

We begin by considering formal properties regarding how OCL algorithms progress along candidate paths when using different re-expansion policies. The first such theorem states that at any time during the algorithm's execution, an rOCL algorithm will either have already explored all the nodes along a given candidate path P, or it will continue to make progress along P. In addition, the statement guarantees that there will be some nodes along this path such that the g-cost of these nodes will be equal to the cost of the optimal path to these nodes. This is formalized in the following theorem:

Theorem 2.2.6. *Let $P \in \Pi_{init}$ be a candidate path for a given planning task Γ such that $P = [n_0, ..., n_k]$, P is an optimal path from n_{init} to n where $n = n_k$, and $\forall n_i \in P$, $H(n_i) \neq \infty$. Then after t iterations of a rOCL algorithm on Γ, one of the following will be true:*

1. $\forall n \in P$, $n \in \text{CLOSED}_t$ *and* $g_t(n) = g^*(n)$.

or

2. $\exists n_i \in P$ *such that* $n_i \in \text{OPEN}_t$, $g_t(n_i) = g^*(n_i)$, *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$ *and* $g_t(n_j) = g^*(n_j)$.

The proof of this theorem can be found in Section A.1.4 of Appendix A. It is similar to a lemma given by Hart, Nilsson, and Raphael [35] that was specific to the A* algorithm and how it progresses along optimal solution paths. Theorem 2.2.6 identifies that it can be generalized to apply to a path from n_{init} to any n even if $n \notin V_{Goal}$, and that this statement holds for any rOCL algorithm.

If the OCL algorithm in use does not employ the full re-expansion policy, then we can still guarantee that at any time during execution, the algorithm will either have already explored all the nodes along a given candidate path, or it will continue to make progress along that path. However, without the full re-expansion policy the optimality of the g-cost of some nodes along this path can no longer be guaranteed. This is formalized as follows:

Theorem 2.2.7. *Let* $P \in \Pi_{init}$ *be a candidate path for a given planning task* Γ *such that* $P = [n_0, ..., n_k]$ *and* $\forall n_i \in P$, $H(n_i) \neq \infty$. *Then after* t *iterations of an OCL algorithm on* Γ, *one of the following will be true:*

1. $\forall n \in P$, $n \in \text{CLOSED}_t$.

or

2. $\exists n_i \in P$ *such that* $n_i \in \text{OPEN}_t$, *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$.

As the proof of this statement is almost identical to that given for Theorem 2.2.6, it is also in the Section A.1.4 of Appendix A.

Theorems 2.2.6 and 2.2.7 can also be simplified considerably when they are applied to solution paths. In the case of an arbitrary OCL algorithm in which nothing is assumed about the re-expansion policy, this simplification is as follows:

Corollary 2.2.8. *Let* $P \in \Pi_{Goal}$ *be a solution path to a given planning task* Γ *such that* $P = [n_0, ..., n_k]$. *Then after* t *iterations of an OCL algorithm on* Γ *prior to the expansion of a goal node, there will exist a node* n *such that the following are true:*

1. $n \in \text{OPEN}_t$.

2. *Either* $n = n_{init}$ *or* $n = n_i$ *for* $1 \leq i \leq k$ *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$.

This simplification follows immediately from Theorem 2.2.7 since all nodes on a given solution path cannot be on the closed list prior to the first goal node having been expanded. Theorem 2.2.6 can similarly be simplified when using rOCL and only considering an optimal solution path. This is simplification is given as follows:

Corollary 2.2.9. *Let* $P \in \Pi^*_{Goal}$ *be an optimal solution path to a given planning task* Γ *such that* $P = [n_0, ..., n_k]$. *Then after* t *iterations of an rOCL algorithm on* Γ *prior to the expansion of a goal node, there will exist a node* n *such that the following are true:*

1. $n \in \text{OPEN}_t$ *and* $g(n) = g^*(n)$.

2. *Either* $n = n_{init}$ *or* $n = n_i$ *for* $1 \leq i \leq k$ *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$ *and* $g(n_j) = g^*(n_j)$.

Finding Lower Bounds for C^*

Another important property of OCL algorithms is that at any time during the search, the nodes that have been seen can be used to generate a lower bound on the optimal solution cost. In the following theorem, we consider how this can be done in the case of an OCL algorithm that uses the full re-expansion policy:

Theorem 2.2.10. *Let* h *be an admissible heuristic and suppose that there have been* t *iterations of an OCL algorithm that uses the full re-expansion policy, such that the search has yet to expand a goal node. Then the following is true:*

$$C^* \geq \min_{n \in OPEN_t} g_t(n) + h(n) \ .$$

Proof. Suppose there have been t node expansions of an rOCL algorithm such that the search has yet to expand a goal node. Let P be an optimal solution path to the

given task. By Theorem 2.2.6, there is a node n' from P on the open list such that $g_t(n') = g^*(n')$. This allows for the following derivation:

$$g_t(n') + h(n') \leq g^*(n') + h^*(n') \tag{2.1}$$

$$g_t(n') + h(n') \leq C^* \tag{2.2}$$

$$\min_{n \in OPEN_t} g_t(n) + h(n) \leq C^* \tag{2.3}$$

Line 2.1 holds since h is admissible and $g_t(n) = g^*(n)$. Line 2.2 holds since n' is on the optimal solution path P. The final line then holds since the minimum of $g_t(n) + h(n)$ over all nodes on the open list must be no larger than $g_t(n') + h(n')$ since n' is in the open list. □

This statement was shown to apply for A^*_ϵ by Pearl and Kim [69] and for WA* by Zhou and Hansen [33]. Both of these works presented essentially the same proof which has been replicated above so as to apply for any rOCL algorithm.

A lower bound can also be generated for an OCL algorithm that does not use a full re-expansion policy, but does always perform parent pointer updates. In this case, both the open list and a subset of the closed list needs to be consulted in order to find an appropriate lower bound. This subset, called the **unopened list** and denoted as UNOPENED, will consist of any node n in the closed list whose g-cost was lowered without that node being moved back into the open list. As defined by the following theorem, by taking the minimum of $g + h$ over all nodes on both the open and unopened list, we again have a guaranteed lower bound:

Theorem 2.2.11. *Let h be an admissible heuristic and suppose that there have been t iterations of an OCL algorithm that always performs parent pointer updates, such that the search has yet to expand a goal node. Then the following is true:*

$$C^* \geq \min_{n \in \text{OPEN}_t \cup \text{UNOPENED}_t} g_t(n) + h(n) \ .$$

Similar statements have been observed by Zhou and Hansen [110], as well as by Likhachev, Gordon, and Thrun [57, 58, 59]. These works have identified that this statement is true in the context of anytime algorithms that perform a sequence of WA* searches that delay the re-expansion of nodes to which lower g-cost paths are

found during a particular iteration until the next iteration.[6] Theorem 2.2.10 generalizes this statement so that it applies for any OCL algorithm that always performs parent pointer updates. The full proof of this theorem is given in Section A.1.5 of Appendix A. The basic idea behind this proof involves the construction of an rOCL algorithm that always selects the same node expansion as the given OCL algorithm but whose open list at any time is given by the union of the open and unopened list of the given OCL algorithm. This will then allow us to use Theorem 2.2.10 to get the desired bound.

Termination and Correctness of OCL Algorithms on Finite Graphs

In this section, we will show that given a finite state-space that has at least one solution path, any OCL algorithm will be guaranteed to terminate having found a solution. We begin by showing that given a state-space in which at least one solution exists, an OCL algorithm will only terminate having found a solution.

Theorem 2.2.12. *Let Γ be a planning task in which there is at least one solution path. Then any OCL algorithm that terminates when run on Γ will be guaranteed to return a solution path.*

Proof. The proof will be by contradiction. Assume that a given OCL algorithm terminates without having found a solution on a task Γ for which there exists at least one solution path P. OCL algorithms can only terminate having found a solution or if the open list ever becomes empty. By our assumption, this means that the OCL algorithm terminated when the open list was found to be empty. Suppose that this happens after t node expansions. Since no goal node was found during the first t expansions, Corollary 2.2.8 guarantees that there is some node $n \in P$ which is in the open list at that point in this search. This contradicts the fact that the open list is empty, and so the algorithm can only terminate once it expands a goal node. □

Let us now show that any OCL algorithm is guaranteed to terminate if the given state-space is finite.

[6] Likhachev, Gordon, and Thrun refer to the unopened list as *INCONS* instead of UNOPENED.

Theorem 2.2.13. *Any OCL problem will be guaranteed to terminate on a given planning task* $\Gamma = \langle G(V, E), n_{init}, V_{Goal} \rangle$ *if* V *is finite.*

Proof. If every node $n \in V$ is expanded at most a finite number of times, then the total number of node expansions must be finite if V is finite. Therefore, it suffices to show that for any n, it is expanded a finite number of times.

Let n be a node in V. Recall that the g-cost of a node refers to the cost of some path from n_{init} to n, even though that path may not be the one currently maintained by the parent pointers.[7] Now notice that each time n is re-expanded, it is given a lower g-cost, and that there are only a finite number of paths from n_{init} to n since V is finite and all nodes have a finite number of successors. This means that if there are K distinct paths to n, then $g(n)$ can be lowered at most K times. As such, n can only be re-expanded K or a finite number of times. Since n is an arbitrary node, any node in n can be expanded at most a finite number of times, there can be at most a finite number of expansions before the algorithm terminates. □

Together, these theorems show that given a problem with a finite state-space and for which a solution exists, any OCL algorithm will be guaranteed to terminate having found a solution. This was first shown for A* by Hart, Nilsson, and Raphael [35]. Pearl and Kim [69] recognized this proof also applied to A^*_ϵ. We have reproduced this proof above so that it also applies to any OCL algorithm, regardless of the re-expansion policy being used.

2.2.5 Best-First Search

Having described properties which hold for large sets of OCL algorithms, we now identify several OCL algorithms that have previously been described in the search literature. The set of such algorithms are called **Best-First Search (BFS)** algorithms, which includes A*, WA*, and greedy best-first search. Algorithms in this class iteratively select the "most promising" node for expansion, where an evaluation function Φ defines how promising a node is. This means that the `SelectNode` function used by best-first search algorithms is as shown in Figure 5.

[7] This is formalized by Lemma A.1.4 and proved in Section A.1.1 of Appendix A.

```
SelectNode(OPEN):
1: return argmin_{n∈OPEN} Φ(n)
```

Algorithm 5: The BFS$^{\Phi}$ `SelectNode` function

The evaluation function Φ may use any information available for estimating how promising a node n is, including information regarding the path found to that node. In general, the only requirement is that $\Phi(n)$ returns a real number. For example, the A* evaluation function is $\Phi(n) = g(n) + h(n)$ where h is an admissible heuristic [35]. This means that this algorithm is employing both the g-cost and heuristic functions for determining the value of a node.

Notice that for evaluation functions like the one used by A*, the value of $\Phi(n)$ can change over time. In the case of the A* function, this change will occur because g can change over time. When needed, we will use $\Phi_t(n)$ to refer to the value of $\Phi(n)$ after t expansions.

So as to be clear about the evaluation function in use, we will use rBFS$^{\Phi}$ to denote a best-first search that is guided by evaluation function Φ and uses the full re-expansion policy. nrBFS$^{\Phi}$ will be used to denote that the algorithm will never re-expand nodes. To the best of our knowledge, all existing best-first search algorithms use one of these two policies.[8]

We now define the evaluation function and re-expansion policy of several important best-first search algorithms below.

The A* Algorithm

As described above, A* is a best-first search algorithm that uses the full re-expansion policy and the following evaluation function:

$$f(n) = g(n) + h(n) \ ,$$

where h is admissible.[9] This means that A* is equivalent to rBFS^f.

[8] rBFS$^{\Phi}$ is similar to another generic best-first search definition given by Dechter and Pearl called BF* [12]. While BF* uses a slightly different re-expansion policy, most of Dechter and Pearl's results involve assumptions that make BF* equivalent to rBFS$^{\Phi}$. This is described in more detail in Section B.2 of Appendix B.

[9] Some authors use the term A* for rBFS^F where $F(n) = g(n) + H(n)$, regardless of whether H is admissible or not. We reserve this term for the case in which the heuristic is admissible to more

Hart, Nilsson, and Raphael [35] showed that any solution found by A* is guaranteed to be optimal. This will be shown in Section 5.4.

When there are multiple nodes in the open list that tie as having the best value for $g+h$, the typical policy used to break such ties is to select the node with the lowest heuristic value (or equivalently with the highest g-cost) for expansion. If there are still ties even when this second criteria is used, then ties are broken arbitrarily.

Djikstra's Search

Consider the special case of the A* algorithm in which the heuristic value of all nodes is 0. The evaluation of any node will therefore be given by $g(n) + h(n) = g(n)$. The resulting algorithm, which is equivalent to rBFS^g, will iteratively select the node with the lowest g-cost. The result is an algorithm that is equivalent to Djikstra's algorithm [13].

Weighted A*

Weighted A* (WA*) is a variant of A* proposed by Pohl [71]. This algorithm uses the following evaluation function:

$$f_w(n) = g(n) + w \cdot h(n) \ ,$$

where h is admissible and $w \geq 1$ is an algorithm parameter. The intuition behind this algorithm is that it emphasizes the importance of the heuristic in the evaluation function. This emphasis typically results in a faster search, though at the cost of decreased solution quality.

The standard version of this algorithm uses the full re-expansion policy and so WA* is equivalent to rBFS^{f_w}. Pohl [71] showed that when using this algorithm with an admissible heuristic and the full re-expansion policy, any solution found by WA* will be guaranteed to be no larger than $w \cdot C^*$. As we will show in Section 5.4.1, this linear loss in guaranteed solution quality is a consequence of the way the heuristic is weighted in the evaluation function. In that section, we will also show that other types of weighting can be used to induce other types of solution quality guarantees.

clearly distinguish this algorithm from the other BFS algorithms considered later.

WA* is commonly modified so that it does not re-expand nodes. When needed, we will therefore refer to nrWA* to indicate that this algorithm is being used such that it does not re-expand nodes, and rWA* to indicate that it uses the full re-expansion policy. It has also been shown that if the heuristic being weighted by an nrWA* algorithm is consistent, then this algorithm can also be guaranteed to return solutions that are no more costly than $w \cdot C^*$ [58]. This result will be shown in Section 6.5, in which we will also consider the solution quality of similar algorithms which weight a consistent heuristic in non-linear ways.

WA* is typically configured to use the same tie-breaking policy as A* if there are multiple nodes in the open list that tie as having the best value for $g + w \cdot h$. This means that ties are broken in favour of the node with the lowest h-cost, and further ties are broken arbitrarily.

Greedy Best-First Search

Greedy Best-First Search (GBFS) (or **pure heuristic search**) is an instance of best-first search that only uses heuristic information in its evaluation function. This means that GBFS uses the following evaluation function:

$$\Phi(n) = H(n) \ ,$$

where H is some heuristic function. GBFS is typically set to never re-expand nodes, which means that GBFS is equivalent to nrBFS^H.

This algorithm takes the idea of emphasizing the role of the heuristic in the evaluation function, as is done in WA*, as far as it can. The result is that it is typically the fastest of the best-first search algorithms described thus far, but it is not possible to bound the quality of solutions that will be found by GBFS.

When there are multiple nodes in the open list that all have the same lowest heuristic value of all those in the open list, the typically way to break such ties is to select the node with the lowest g-cost for expansion. This is in keeping with the understanding of GBFS as being like WA* as the weight approaches infinity. To see this, suppose that we are running WA* with a weight that is much larger than the length of any path in Π_{init}. In that case, $w \cdot h$ would dominate the evaluation

function and so the search would essentially be performing a GBFS on heuristic h. However, as g is still part of the evaluation function, the evaluation of a node with a lower g-cost will be less than a node with an equal h-cost but a lower g-cost. Therefore, the search would be equivalent to a GBFS that breaks ties in favour of the node with a lower g-cost.

2.2.6 Focal List Based Search

The other main category of existing OCL algorithms are **focal-list based algorithms**. This class of algorithms allows for the use of distance-to-go or other inadmissible heuristic information, while still guaranteeing bounded solution quality. These bounds are guaranteed by restricting the use of such inadmissible information to only be used to select nodes for expansion from a particular subset of the open list defined specifically so to ensure the required solution quality is met. We will introduce this class of algorithms through the example of the first such focal-list based algorithm: A^*_ϵ.

A^*_ϵ (or A^*_w)

A^*_ϵ was first described by Pearl and Kim [69]. As we will be using ϵ later in this thesis in a different context, we will refer to this algorithm as A^*_w. This algorithm also selects nodes from the open list according to an evaluation function, but it only uses this evaluation function on a subset of those nodes, called the **focal list**. This subset is defined as follows:

$$\text{FOCAL} = \left\{ n | g_t(n) + h(n) \leq w \cdot \min_{n' \in \text{OPEN}} g_t(n') + h(n') \right\} \ ,$$

where w is an algorithm parameter for A^*_w and h is an admissible heuristic.

Having defined the subset of the open list which it is restricted to selecting nodes from, A^*_w then greedily selects nodes from this subset using a secondary heuristic that may be inadmissible [69]. In the case of a graph with non-uniform edge costs, Pearl and Kim suggest the use of a distance-to-go heuristic H_d, which estimates the number of actions in the path to the nearest goal node and often provides a better measure of how much search effort will be needed to find a goal from a given node

[69, 90]. When taking this approach, the `SelectNode` function would look as shown in Figure 6.

`SelectNode`(OPEN):
1: FOCAL $= \{n | g_t(n) + h(n) \leq w \cdot \min_{n' \in \text{OPEN}} g_t(n') + h(n')\}$
2: **return** $\text{argmin}_{n \in \text{FOCAL}} H_d(n)$

Algorithm 6: The A^*_w `SelectNode` function

Notice that by this definition, A^*_w is using the same policy as a GBFS which uses the heuristic H_d. However, unlike GBFS which can consider all nodes on the open list, A^*_w is restricted to only use this policy to select amongst those nodes on the focal list. As a result, A^*_w will perform more and more like GBFS as w increases, since increasing w will increase what proportion of the open list is also in the focal list.

Properties of Focal List Based Algorithms

In their paper introducing the algorithm, Pearl and Kim showed that when using A^*_w with the full re-expansion policy and an admissible heuristic h for the construction of the focal list, any solution found will have a cost of at most $w \cdot C^*$ [69]. However, it is not the policy for selecting nodes from the focal list which ensures this bound, but the way the focal list is constructed. This was identified by Ebendt and Drechsler [15], and also by Farreny [17] who showed that this bound holds regardless of the policy used to select nodes from the focal list, assuming the use of the full re-expansion policy and that an admissible heuristic h is being used when constructing the focal list. For example, this bound also holds for the EES algorithm developed by Thayer and Ruml [90] which uses a different policy for selecting nodes from the focal list. We will describe this policy in more detail later in this thesis.

In Section 5.5, we will prove the A^*_w bound given above. This bound will also be shown to hold as a result of the way the focal list is constructed, regardless of what policy is then used to select from amongst the nodes on the focal list. We will also show that other ways to construct the focal list will admit other sorts of bounds. In particular, we will consider the use of functions of the form $\beta : \mathbb{R} \rightarrow \mathbb{R}$ where

$\forall x \geq 0, \beta(x) \geq x$ for building focal lists as follows:

$$\text{FOCAL} = \left\{ n | g_t(n) + h(n) \leq \beta \left(\min_{n' \in \text{OPEN}} g_t(n') + h(n') \right) \right\}$$

For example, in the case of A^*_w and related algorithms, the function β is defined as $\beta(x) = w \cdot x$. We will then use FOCAL$^\beta$ to denote the set of focal list based algorithms that use the function β to define its focal list. When we then wish to specify the re-expansion policy in use, we will use rFOCAL$^\beta$ and nrFOCAL$^\beta$.

Several authors have noted that rWA* is a special case of rFOCAL$^{\beta_w}$, where $\beta_w(x) = w \cdot x$, that uses the WA* evaluation function to select a node from the focal list [15, 17]. This is because the node with the minimum value of $g + w \cdot h$ is always guaranteed to be on the focal list when β_w is being used to define this list. However, Ebendt and Drechsler also showed that while any solution found by nrWA* is guaranteed to be no larger than $w \cdot C^*$ provided that the heuristic in use is consistent, this is not the case for nrFOCAL$^{\beta_w}$ algorithms in general [15]. Instead, a nrFOCAL$^{\beta_w}$ algorithm that builds its focal list like A^*_w using a consistent heuristic can only be guaranteed to return solutions that cost no more than $w^{\lfloor D/2 \rfloor} \cdot C^*$ where D is the number of edges along the optimal solution path.

2.2.7 Other Heuristic Search-Based Algorithms

In the remainder of this thesis, we will also refer to several other algorithms that use heuristics to explore a state-space in an effort to solve a given planning problem, aside from those in the class of OCL algorithms. These include random-walk based search [65, 66] and iterative deepening search [50]. As these algorithms will not be referred to as often as the set of OCL algorithms, they will be introduced as needed.

While there are also other paradigms for heuristic search-based algorithms — including bi-directional search [72] and real-time search [51] — a consideration of how the work in this dissertation applies to those frameworks is left as future work.

2.3 Automated Planning

In the remainder of this thesis, we will often test a heuristic search algorithm as part of an **automated planner**. An automated planner is a system that solves prob-

lems that are given to the system specified using a general problem task description language. For heuristic search-based automated planners, this means that such systems must be able to generate a heuristic on their own, based solely on the problem description. In this section, we will briefly describe how problems are represented, give an example of a heuristic used by automated planners, and describe several enhancements commonly used by such systems.

2.3.1 Automated Planning Representations

The most well-known planning problem description language is **STRIPS** [23]. This language offers an implicit representation for planning tasks. In a STRIPS representation of a problem, a state is defined by the set of **propositions** or **facts** that are true in that state. For example, a typical STRIPS representation of the 15 puzzle would define the set of facts as the set of all pairs of tiles and tile locations. Where F is the set of all possible facts for a given domain, a particular state s would be defined as the subset $F_s \subseteq F$ of those facts which are true in s. By the **closed world assumption**, this also means that for any fact $f \in F$ such that $f \notin F_s$, f does not hold in s. For example, if a particular 15 puzzle state s has the tile labelled 1 in the upper left corner, then F_s would include the proposition which corresponds to this fact, and s would not include the propositions corresponding to there being any other tile in that position.

A STRIPS problem is defined by a given set of facts F, a subset $F_{init} \subseteq F$ which defines the initial state, a set of propositions $F_{Goal} \subseteq F$ that define which states are goal states, and a set of actions A that define the legal state transitions, each of which will have an associated cost. A state s is said to be a goal state if $F_{Goal} \subseteq F_s$ where F_s is the set of facts that are true in s. An action $a \in A$ is defined by three components: the **preconditions**, the **add list**, and the **delete list**. The preconditions, $pre(a) \subseteq F$, are the propositions that must be true in a given state in order for a to be applicable. The add list, $add(a) \subseteq F$, are the set of propositions that will be true after a is applied to a state s in which a is applicable that were not true prior to applying a. The delete list, $del(a) \subseteq F$, are the set of propositions that will not be true after a is applied to a state in which a is applicable, but which were

true prior to the application of a.

Other important languages for describing planning problems include the **Planning Domain Definition Language (PDDL)** [62] and SAS+ [2]. PDDL, which is an extension to STRIPS that uses a restricted first-order language for defining problems, has become the standard language used by modern planners. In SAS+, a state-space is defined by a set of variables each of which has a finite, but not necessarily binary domain, and a state is defined by an instantiation of these variables. This language is notable since several modern planners such as `LAMA` [76] and `Fast Downward` [38] first translate the given PDDL representation into an SAS+ representation before building their heuristics. A full description of these languages is beyond the scope of this thesis, in which the above description of STRIPS will be enough to understand the techniques typically used by automated planners.

2.3.2 Automated Planning Test Suites

The planning problem definition languages just defined can all be used to encode a very large and diverse set of state-spaces. For example, PDDL has been used to model the operation of multi-engine printers [14] and greenhouse logistic management [42]. The automated planning benchmarks that we will use for empirical evaluations will be given by the problems from the **International Planning Competition (IPC)**. In this competition, teams submit automated planning systems that are tested on an unknown set of PDDL problems. For each competition, this test set is typically made up of multiple problems from different domains. The test set that we use consists of all of the problems from the last three planning competitions: IPC 2006 [28], IPC 2008 [39], and IPC 2011 [9].

Evaluating Planners

Depending on the context, we will be testing heuristic search algorithms in an automated planner in either a satisficing or a bounded context. In either case, our empirical evaluations will focus on how many of the problems in a given test set are solved by the planner given a specific time limit and under the given solution quality requirement. This value is referred to as the planner's **coverage** on the test

set. For example, if a planner solves 700 of the 790 problems in a given test set, the coverage of that planner is 700.

We will typically use the standard time limit of 1800 seconds per problem that is used in the International Planning Competition. As all the planners used first require a translation from PDDL to SAS+, the time needed for this translation was not included in the time limit. The empirical evaluations were also run on a variety of machines due to the high computational needs of running such experiments, and so the memory limit was varied depending on the machine in use. Despite the fact that different machines were used, direct comparisons will only be made between planners that were tested on the same hardware.

In some cases, we will be evaluating planners that use **Las Vegas algorithms**. Such algorithms have a variable runtime due to there being non-deterministic aspects of the algorithm, but they are guaranteed to return a valid solution path if they ever do terminate. When evaluating a Las Vegas planner (*ie.* a planner that uses a Las Vegas algorithm), our evaluation metric will be the planner's **expected coverage**. To define this value, notice that a Las Vegas planner p will have a probability $P(\Gamma, p, t)$ of solving planning task Γ in time limit t. Given a set of planning tasks $\{\Gamma_0, ..., \Gamma_n\}$, the expected coverage when p is limited to time t on each problem is therefore $P(\Gamma_0, a, 1800 \text{ s}) + ... + P(\Gamma_n, a, 1800 \text{ s})$.

Due to the stochastic nature of a Las Vegas algorithm, the expected coverage of a Las Vegas planner will vary when tested on a given problem set for multiple runs per problem. While this means that there is a probabilistic distribution of possible coverage values, we will not include confidence intervals or perform other statistical tests when comparing multiple planners. This approach is consistent with the existing planning literature, in which typically only average coverage results are reported due to the fact that it is still an open question regarding how meaningful such measures are when experimenting on the types of problem sets we consider below. One of the reasons for why this question is not settled is the fact that the problems in these test suites are often of varying difficulty, with many of the problems being exponentially more difficult than others. As such, even small coverage improvements can represent a substantial improvement in planner performance. Therefore,

the use of standard and known benchmark problems and average performance as a performance metric is usually considered to be sufficient when trying to determine the relative quality of planning systems.

2.3.3 The FF Heuristic

In heuristic search based automated planners, the heuristics used are automatically generated based on the problem description. Many of these heuristics are based on the idea of using the cost of the solution to a simplified version of the given problem to approximate the cost of reaching the goal in the unsimplified version of the problem. The most popular type of simplification is **delete relaxation** [44]. When using this technique to simplify a given planning problem, all the actions are modified so that their delete lists are emptied. This means that in the simplified version of the task, any proposition which is true in a given state s will necessarily be true in any descendent, since propositions can never be removed by applying a sequence of actions to s.

If the optimal solution from a given state to the nearest goal could be computed for the relaxed version of a problem, then the resulting solution cost could be used as an admissible heuristic to the original problem. Unfortunately, calculating the optimal solution for a problem that is simplified using delete relaxation has been shown to be NP-Complete [6]. As such, many existing automated planning heuristics find an approximation of the cost of the optimal solution for the relaxed problem that can be computed in polynomial time. This is the approach taken by the **FF heuristic** as developed by Hoffmann and Nebel [44]. In the case of this heuristic, the plan found to the relaxed problem is not guaranteed to even be admissible for the original problem, but it has been shown to be a very powerful heuristic for guiding satisficing search algorithms. For complete details on how this heuristic is computed, see the work of Hoffmann and Nebel [44].

Once the plan has been found to the relaxed version of the planner, it can be used in multiple ways to get a heuristic. The cost-to-go version of this heuristic is given by the sum of the actions of along this plan. This heuristic will be referred to as FFc. Alternatively, the cost of the actions can be ignored, in which case the

heuristic value corresponds to number of actions in the relaxed plan. The resulting distance-to-go heuristic will be referred to as FFd.

2.3.4 Deferred Heuristic Evaluation

The heuristics used by automated planning systems are typically quite expensive to compute. As such, it is often the case that an automated planning system is prevented from examining much of the state-space as it will spend the majority of its time performing heuristic calculations. Dealing with this issue is the motivation behind **deferred heuristic evaluation** which can often allow the search to examine a larger portion of the state-space in the same amount of time [38]. When using this technique, the heuristic value used for a node n is given by the heuristic value of the parent of n. For example, recall that the GBFS algorithm is equivalent to nrBFS^H where H is some heuristic function. If this same GBFS algorithm is modified to use deferred heuristic evaluation, then it will be equivalent to nrBFS$^{H'}$ where H' is a heuristic function where $H'(n)$ returns any finite number in the case that $n = n_{init}$ and $H'(n) = H(parent(n))$ otherwise.

When this technique has been tested experimentally, it has been shown to typically decrease the time needed to find a solution [74]. To see why, it is necessary to consider how OCL algorithms are typically implemented. In a standard OCL algorithm, the heuristic value of a node n is computed when n is first generated. The value of $H(n)$ is then stored for the remainder of search. The result is that it will often be necessary to perform $|succ(n)|$ heuristic computations each time a node n is expanded.

In the case of deferred heuristic evaluation, the heuristic value of a node n only needs to be computed once it is expanded. This is because until n is expanded, the algorithm only needs to lookup the value stored for the $H(parent(n))$ when evaluating n. As such, the expansion of n only requires a single heuristic evaluation. The result is that the heuristic value of many of the nodes on the open list will not have been computed, and the decrease in the number of node expansions will allow the planner to more quickly see a larger portion of the state-space.

2.3.5 Multi-Heuristic Best-First Search

Another common technique that modern automated planning systems employ is to use multiple heuristics in a process called **multi-heuristic best-first search**. This technique was introduced as part of the `Fast Downward` planning system [38]. Given heuristics H_0, ..., H_{k-1}, a system which uses multi-heuristic best-first search will simply alternate between using each of these k heuristics when selecting the next node for expansion. For example, when this technique is added to GBFS all k heuristics would be consulted when checking if the goal is reachable from a given node n, and the `SelectNode` function would be defined as shown in Figure 7. Note that in the code shown, i is a global variable initialized to 0.

```
SelectNode(OPEN):
1: n ← argmin_{n'∈OPEN} H_i(n')
2: i ← (i + 1) mod k
3: return n
```

Algorithm 7: The `SelectNode` function used by a GBFS enhanced with multi-heuristic best-first search.

The purpose of multi-heuristic best-first search is to allow a planner to simultaneously use the guidance offered by different heuristics. This is necessary due to the fact that different heuristics tend to offer guidance that will greatly vary in quality not only across different domains, but even within different parts of the same state-space. When this technique has been evaluated empirically, it has been shown that by using different heuristic functions that effectively complement one another, multi-heuristic best-first search can improve performance considerably [80].

In practice, a multi-heuristic best-first search using k heuristics is implemented using k copies of the open list, each as a priority queue sorted according to a different heuristic. The code then alternates between these lists when selecting a node for expansion.

2.3.6 Preferred Operators

The final planning enhancement that we will consider is that of **preferred operators**. A preferred operator of node n is an action a that is applicable in n, such

that we expect that a will be potentially useful when applied to n. A planner that uses preferred operators will then bias the search so that on every iteration it is more likely to select a node that was achieved using a preferred operator, called a **preferred successor**, for expansion.

Preferred operators are typically identified as a byproduct of heuristic computation. For example, the standard approach when using the FF heuristic function is to select the preferred operators as those that correspond to **helpful actions** [44]. As described above, the FF heuristic computes a possibly suboptimal plan to a relaxed version of the given problem. While the cost of this relaxed plan is used for the heuristic value of n, the actions in the relaxed plan that are applicable in n are identified as the helpful actions. The intuition behind this designation is that if the relaxed planning task is a reasonable approximation of the original planning task, then we would expect that the relaxed plan would be similar to one that would solve the original task. The actions in the plan for the relaxed planning task which are also applicable to n would therefore be identified as offering potentially useful guidance beyond what is given by the cost of this plan. As such, we will consider these nodes as preferred successors and seek to bias the search so as to expand these nodes for expansion more often than those that do not correspond to helpful actions.

The standard way to do so is to alternate between using the search's node selection policy to pick a node from the entire open list for expansion and using the search's node selection policy to pick only from amongst those nodes that correspond to preferred successors. For example, where $\mathrm{OPEN}_{\mathrm{pref}} \subseteq \mathrm{OPEN}$ is the subset of the open list that has been identified as preferred successors, the `SelectNode` function used by a GBFS enhanced with preferred operators is shown in Algorithm 8, where $count$ is a global variable initialized as 0.

Like multi-heuristic best-first search, preferred operators are typically implemented into a search using two priority queues, both of which are sorted according to the same heuristic. The first queue will contain a copy of all nodes in OPEN, while the second contains only those in $\mathrm{OPEN}_{\mathrm{pref}}$. As when using multiple heuristics, the code will then alternate between selecting nodes from each of these queues.

When using both multi-heuristic best-first search with k heuristics and preferred

```
SelectNode(OPEN):
1: n = NONE
2: if i = 0 or OPEN_pref = {} then
3:     n ← argmin_{n'∈OPEN} H(n')
4: else
5:     n ← argmin_{n'∈OPEN_pref} H(n')
6: i ← i + 1  mod 2
7: return n
```

Algorithm 8: The `SelectNode` function used by a GBFS enhanced with preferred operators.

operators simultaneously, $\mathrm{OPEN}_{\mathrm{pref}}$ will be set to contain all nodes that were identified as a preferred successor by at least one of the heuristics. The algorithm will then maintain $2 \cdot k$ priority queues which the algorithm will alternate between when selecting a node for expansion. k of these queues will each contain a copy of all nodes in OPEN and each will be sorted by a different heuristic. The remaining k queues will contain a copy of only those nodes in $\mathrm{OPEN}_{\mathrm{pref}}$, and each will be sorted according to a different heuristic. Notice that this means that for each of the k heuristics there will be two queues sorted by that heuristic, and that each preferred operator queue does not only contain the preferred successors identified by the heuristic being used to sort that queue. Instead, all preferred operator queues will contain the exact same set of nodes.

The `LAMA` planner further biases the search towards expanding preferred successors in a technique called **preferred operator boosting** [76]. For this technique, each of the queues are given a **priority** value, and whenever a node is to be selected for expansion, a node is retrieved from the list with the highest priority. The priority of each of these lists is initialized to zero. Whenever a node is selected from a particular list, the priority is decremented by 1. However, if a node n is found such that $H(n)$ is less than the heuristic value of any previously generated node, then the priority of each of the preferred operator queues is increased by some bonus value (typically set as 1000). Note that this bonus is given to $\mathrm{OPEN}_{\mathrm{pref}}$ even if this list was not the one which returned the node with the best heuristic value.

The result of this boosting is that preferred successors are selected for expansion

even more often than they are when using the standard alternating approach. In practice, boosting has been shown as an effective technique for improving planner coverage [74].

2.4 15 Puzzle Heuristics and Variants

Aside from automated planning, we will also use several additional domains for empirically evaluating algorithms. One of these is the aformentioned 15 puzzle. In this section, we describe some of the heuristics used when solving this problem, some of which will also be used when experimenting with other domains (which will be introduced as needed). We also describe several variants of the 15 puzzle that will be used in this thesis.

2.4.1 The Manhattan Distance Heuristic

A simple heuristic that is often used when solving 15 puzzle tasks is the **Manhattan distance heuristic**. This heuristic estimates the cost of getting to the goal node in terms of how far each of the tiles is from being in its goal location. For example, consider the tile labelled 2 whose goal location is in the first row and the third column. If this tile is in the third row and the first column of a node n, then that tile needs to move up two rows and right two rows to get into its goal location. The 2 tile is therefore a distance of $2 + 2 = 4$ steps away from its goal location, since this tile has to be moved at least 4 times to get into its goal location. The heuristic value of n according to the Manhattan distance heuristic is then given by the sum over all tiles of such distances. The result is an admissible heuristic since no action ever moves more than a single tile.

2.4.2 Pattern Database Heuristics

Pattern databases are another popular approach to building heuristics for 15 puzzle and other domains [11]. This heuristic requires that the state-space be **abstracted**, such that the abstract state-space is much smaller than the original space. Having built the abstract state-space, the exact cost from any abstract state to the nearest

abstract goal state is computed and stored in a lookup table. When the heuristic value of a node n is needed, the abstract version n_a of n is first constructed, and then the cost to get from n_a to the nearest goal in the abstract space is found in the lookup table. This cost is then used as the heuristic value of n in the original space.

The abstraction used by pattern databases involves treating certain elements of the domain as indistinguishable. For example, in the 15 puzzle, the abstraction may treat the tiles labelled 1, 2, 3, 4, and 5 as all the same tile. The abstract state for any given state n will correspond to the same state, just with all of the tiles from 1 to 5 having the same label. The resulting state-space will have $16!/5!$ states unlike the original state-space which has $16!$ states.

The resulting heuristic can be guaranteed to be admissible provided that for any state n, all actions applicable in n will also be applicable in the abstract version of n. In certain cases, the values returned by multiple different pattern databases for a given state can also be added together while still maintaining admissibility. While we will use such pattern databases, we omit a full description of additive pattern databases which can be found in the work of Felner, Korf, and Hanan [19].

2.4.3 The 24 Puzzle

The 24 puzzle is a larger variant of this state-space in which the grid has 5 rows and 5 columns instead of 4 rows and 4 columns as they are in the 15 puzzle. In the 24 puzzle, the tiles are labelled with the integers from 1 to 24. Like the 15 puzzle, all transitions will have a cost of 1 and the goal node of the 24 puzzle has the blank in the upper left corner and the tile labels proceed, starting from 1, in ascending order from left-to-right and top-to-bottom.

2.4.4 The Inverse Cost 15 Puzzle

Another variant on the 15 puzzle is the **inverse cost 15 puzzle**. The state-space of this puzzle is identical to that of the standard 15 puzzle with the exception that the edge costs are no longer all the same. In the inverse cost 15 puzzle, the edge cost is determined by the label on the tile being moved into the blank space. If the tile being moved during a transition has the label ℓ, then the cost of that transition is

$1/\ell$. For example, when moving the tile labelled 5, the cost will be $1/5 = 0.2$.

The admissible heuristic used in this domain will be the **weighted Manhattan distance heuristic**. This heuristic is computed in almost the same way as the standard Manhattan distance. The only difference is that when summing over the distances that each tile is from the goal, each distance is weighted by the cost of moving the corresponding tile. For example, recall that if the tile labelled 2 is in the third row and the first column, then the tile is 4 locations from being in its goal location. In the summation, this tile will be weighted by $1/2$ since the minimum cost to get this tile to its goal location is $4 \cdot 1/2 = 2$.

2.5 Dealing With Large Design Spaces Using Automatic Configuration Tools

To handle the large design space available to a system designer, a popular approach is to use an automatic configuration tool. For example, ParamILS is one such tool that, given a set of training tasks, runs a local search in the space of parameters to find a configuration that is expected to exhibit high performance on tasks drawn from a similar distribution as the training set [46]. This system has been successfully applied to a build high performance systems for a wide variety of computationally hard tasks including PDDL planning [18, 81, 102].

While systems like ParamILS can effectively find configurations even in a large design space, it is still up to the system designer to first enumerate the set of decisions that are to be tuned. This is one of the reasons that investigations such as that given in Chapter 5 are still necessary. In that chapter, we will consider the impact that random exploration can have on a search, and demonstrate that this technique can often positively impact the performance of GBFS. Therefore, if a system designer is employing a tool like ParamILS to develop a GBFS-base planner, our results suggest that it is important to include random exploration techniques in the set of decisions to be tuned, as doing so may be able to improve planner performance.

Automatic configuration tools also cannot guarantee that a given solution quality requirement will be satisfied unless it is restricted to only consider the portion of

the design space that will do so. Therefore, investigations such as those in Chapters 5 and 6 which help to identify the algorithm options that will satisfy a given solution quality requirement are needed before a tool like ParamILS can be deployed.

For these reasons, we consider the development of automatic configuration tools as orthogonal to the work described in this dissertation.

2.6 Chapter Summary

In this chapter, we have defined the planning task and introduced much of the notation that will be used in the remainder of this thesis. As described above, we will reason about state-spaces as graphs in which we are trying to find a path from a given initial node to one of a set of goal nodes. The notation used for referring to such graphs was then introduced in Section 2.1.

In Section 2.2, the OCL algorithm framework was introduced as a generalization that includes many existing algorithms. This framework not only allows additional flexibility in how nodes are selected for expansion but also when nodes are made available for re-expansion. Many properties of existing algorithms were then formally shown to still hold even when these generalizations are allowed.

In Section 2.3, we introduced the field of automated planning. In particular, we briefly describe the languages used for encoding planning problems, the test suites that will be used, and several enhancements that are commonly used in existing automated planning systems. This was then followed by a description of the 15 puzzle and a variant of this puzzle, the heuristics used in these puzzles, and some related work into the use of automatic configuration tools for dealing with large design spaces.

Chapter 3

Exploiting Variation to Build a State-of-the-Art Multi-Core Planner

Existing automated planning systems have their own strengths and weaknesses in terms of the types of state-spaces they are best suited for. This is not only true across different planners, but even within the space of design choices for a particular planner. In this chapter, we will show that the resulting variation in planner performance can be managed effectively through the use of an algorithm portfolio. We do so from the perspective of building a multi-core planner. The result will be `ArvandHerd` which won the multi-core sequential satisficing track of the 2011 and 2014 International Planning Competitions.

3.1 Introduction

In recent years, processor speeds have been increasing at a reduced rate, while the proliferation and availability of multi-core technology has substantially increased. This development suggests that to best utilize modern hardware when constructing satisficing planning systems, it is necessary to consider parallel approaches.

Past work on building multi-core planning systems, such as PBNF [5] and HDA* [49], has generally focused on parallelizing a single heuristic search algorithm. While these approaches have successfully improved run-time, satisficing planners that use these or similar techniques on shared memory machines should not be expected to solve many more problems than their single-core counterparts. This is because the parallelization of a planner will most likely have the same strengths

and weaknesses as the original single-core planner. For example, suppose we had a "perfect" parallelization of LAMA-2011 [77], the winner of the single-core sequential satisficing track of IPC 2011. Such a parallelization would run exactly k times faster than the single-core version when run on k cores. Given a time limit T, the performance of such a k-core system can be simulated by running LAMA-2011 for $k \cdot T$ time and counting any problem solved within this time limit as having been solved by the k-core parallelization in time T. This simulation indicates that even with such a speedup, coverage only increases slightly. For example, when given a 6 GB memory limit and a 30 minute time limit, even the 8-core version of this perfect parallelization of LAMA-2011 would solve only 6 more problems than the 721 solved by the standard single-core version when tested on all 790 problems from the 2006, 2008, and 2011 IPC competitions.

One of the issues with parallelizing a memory-heavy planner like LAMA-2011 in a shared-memory environment is that it is often the available memory that limits coverage. In these cases, any speedup will merely cause memory to be exhausted more quickly. This behaviour is seen in the simulated LAMA-2011 parallelization, as the 8-core simulation ran out of memory on 52 problems. This means that regardless of how many more cores are used, at most 738 of the 790 problems can be solved using LAMA-2011 without an increase in memory.

An alternative to parallelizing a single algorithm is to run members of an **algorithm portfolio** in parallel. This involves tackling each problem using a set of strategies that differ in either their **configuration** (*ie.* different parameter values or other settings) or in the underlying algorithm, and running these strategies simultaneously on different cores. This technique is inspired by two considerations. First, planners are expected to solve problems from a diverse set of domains, and no single existing algorithm dominates all others on all domains. Second, this approach offers a simple alternative to the difficult process of parallelizing a single-core algorithm and it mostly avoids overhead from communication and synchronization.

3.1.1 Contributions

In this chapter, we will show that two existing planners, `Arvand` and `LAMA`, exhibit variance across parameterizations and other design choices. We will then demonstrate that each of these planners can be enhanced through the use of multiple configurations and restarts. While these techniques have previously been successfully applied in the boolean satisfiability community as a way to deal with a large design space, we demonstrate that they are similarly successful in planning.

Even after using multiple configurations of each of these planners, these planners will still have substantial differences in the domains in which they are most effective. We will therefore describe how these planners have been combined effectively in a parallel portfolio called `ArvandHerd`. `ArvandHerd` represents the first system to successfully combine disparate planning approaches to create a state-of-the-art parallel planner for shared memory machines. It won the multi-core sequential satisficing track of IPC 2011 [9] and it was designed specifically to avoid the inherent limitations of parallelizing a single memory-heavy planning algorithm that were described above.[1] The effectiveness of this planner and the challenges in building a portfolio-based multi-core planner will be discussed below. In particular, we will show that `ArvandHerd` can solve more benchmark problems than several state-of-the-art planners, even if they could be effectively parallelized.

The work in this chapter is mainly based on a conference publication that appeared at the 2012 European Conference on Artificial Intelligence [96]. All of the described experiments were performed after the competition was completed in an effort to understand how the design choices made in the development of `ArvandHerd` contributed to its success. That paper was also preceded by a technical report that was part of the submission of `ArvandHerd` to the 2011 competition [97]. That paper simply presented the `ArvandHerd` architecture without results, including the design decisions aimed to improve solution quality. As most of the decisions made regarding solution quality were based on the work of others, they were not empirically evaluated and will not be explained further here.

[1] An updated version of this planner also won the multi-core track of the 2014 competition, though the impact of the updates made still needs to be evaluated.

The `ArvandHerd` planner was built in collaboration with Hootan Nakhost and Martin Müller, who developed the `Arvand` planner. I was the lead researcher developing `ArvandHerd` and my contributions included adding the ability to run multiple configurations of each planner, parallelizing the system, and empirically evaluating `ArvandHerd` after the competition. Selecting the members of the portfolio was done collaboratively with Hootan Nakhost.

3.2 Background and Related Work

In this section, we will consider some background regarding planning with a portfolio and multi-core planning, as well as related work in building planners for multi-core systems.

3.2.1 Algorithm Portfolios

There is substantial literature demonstrating that the performance of the solvers used for tackling many computationally hard tasks can vary greatly across different tasks [31, 32, 45]. This has also been shown to be true for automated planning systems [24, 30, 79]. To deal with this variability, Huberman, Lukose and Hogg proposed the use of an **algorithm portfolio** [45]. An algorithm portfolio A is given by a set of planners $\{p_0, ..., p_k\}$ that will be used together on the same problem so as to benefit from the strengths of each of the planners in A, which are called **portfolio members**.

Building a portfolio-based automated planner requires the selection of the planners to be included in the portfolio, and the selection of a policy to then use those planners collectively. We consider several approaches for each of these tasks below.

Using a Given Portfolio

There are several popular ways to deploy a given portfolio. The first is an approach called **dovetailing** which involves interleaving the independent execution of the planners by switching between them in an alternating fashion [32, 45, 100]. Given a set of planners $\{p_0, ..., p_k\}$ and some distribution over those algorithms given by

$\alpha_0, ..., \alpha_n$ where $0 \leq \alpha_i \leq 1$ and $\alpha_0 + ... + \alpha_k = 1$, dovetailing consists of a number of rounds. Each round works as follows: each portfolio member will, in order, advance its execution for some amount of time. This amount of time is set such that the ratio of the time spent on p_j to the time spent on the entire round is α_j. If some planner finds a goal during the time it is given to advance its search, the solution found is returned and dovetailing will stop. If a round completes without having found a solution, a new round then begins.

Another simple approach is to use **restarts** [31]. For this technique, the planners in A must be ordered and a restarting policy is needed. This policy is given by a sequence of runtimes $t_0, .., t_k$. If p'_0 is the first planner in the ordering of A, the portfolio is then run beginning with planner p'_0 for time t_0. If a solution is found in the given time limit, it is returned and the process terminates. If a solution is not found, then where p'_1 is the second planner in the order, p'_1 is run for time t_1. This process then continues until a solution is found or all planners have been run for their allotted time.

`ArvandHerd` will use both restarts and dovetailing-like approaches, but its main way of combining planners is in a multi-core setting. This approach involves assigning different portfolio members to different processors, each of which will run their portfolio member independently.

Selecting Portfolio Members

Prior to the 2011 International Planning Competition at which `ArvandHerd` competed, most portfolio-based planners selected some number of the top-ranking existing planners as the members of the portfolio [7, 24, 30, 79]. Some of these systems use the entire portfolio by restarting or dovetailing, while others would only use a subset of the portfolio which was selected on a per-task basis. For example, Roberts and Howe [79] learn a decision tree based on how the portfolio members perform on a set of training planning problems. When the planner is given a new task to solve, the decision tree then selects a subset of the portfolio and a restarting policy to use. A similar approach was taken by Gerevini, Saetti, and Vallati in their planner, PbP [30], and by Cenamor, de la Rosa, and Fernández [7].

Other approaches for selecting portfolio members start with a large set of planners and then use an offline phase to select the portfolio members. This is the approach used by `Fast Downward Stone Soup` (which we will refer to below as `Stone Soup`), a portfolio-based planner that competed in the single-core track of the 2011 International Competition [43]. This planner simultaneously develops a restarting strategy and selects the portfolio during an offline training phase. Given a large set of planners which have all been run on a test set, the portfolio is constructed by a hill-climbing search in the space of possible portfolios, starting with the empty portfolio. The process then iteratively adds a planner to the portfolio and a time limit for the added planner, where the added planner is selected such that it maximizes the coverage of the portfolio on the training set. The hill-climbing search then continues until the total time allocated to planners in the portfolio exceeds some given time limit.

In 2012, Seipp *et al.* [81] built a system for finding configurations of a highly configurable planner that could then be used as the members of a portfolio. To do so, they use the ParamILS automatic parameter tuner [46] to learn an effective configuration of the planner for each of a set of training domains. The planner they considered was `Fast Downward`, in which the space of design choices allows for the use of different heuristic functions, different planning enhancements, and different best-first search variants. The configurations found as part of this tuning process are then used as the members of the portfolio.

Like many of the portfolio-based planners built before the 2011 competition, the `ArvandHerd` portfolio was selected manually. The `ArvandHerd` portfolio includes both multiple planners and multiple configurations of these planners. It was designed specifically to avoid certain challenges unique to the use of portfolios for multi-core planning that we will detail below. As such, it includes a random-walk based planner and multiple operator orderings of a best-first search-based planner, unlike the other planners described above. Its performance could also be improved by the more sophisticated configuration selection techniques described above, but doing so is left as future work.

3.2.2 Expected Coverage of a Portfolio

Some of the planners we will use in the `ArvandHerd` portfolio will be Las Vegas planners. Recall that given a time limit t, a Las Vegas planner p will have a probability $P(\Gamma, p, t)$ of solving planning task Γ. Now let $A = \{p_0, .., p_m\}$ denote a portfolio of Las Vegas planners, and let A_T denote the set of tuples (p_j, t_j) where $p_j \in A$ and t_j is a maximum amount of time that planner p_j will be run for when using either a restarting strategy, assigning different planners to different cores, or dovetailing. As the algorithms will be run independently, the probability that A_T will solve a task Γ is therefore given by

$$P(\Gamma, A_T) = 1 - (1 - P(\Gamma, p_0, t_0)) \cdot ... \cdot (1 - P(\Gamma, p_m, t_m)) \quad .$$

The expected number of problems solved when restarting with a set of planner-time limit tuples on a set of tasks $\{p_0, ..., p_n\}$ is then given by $P(p_0, A_T)+...+P(p_n, A_T)$.

This provides an alternative technique for estimating the expected coverage of a portfolio aside from directly testing that portfolio. This alternative approach is desirable because it makes it less time consuming to evaluate a large set of portfolios. For example, if we are considering all portfolios of size three from a set of ten planners, there will be 120 possible portfolios that need to be tested. If there are 100 tasks in the test set, this means that we will need $12,000$ runs to even test each portfolio once on each task. If the included planners are Las Vegas planners, we will also need to run each planner on each problem multiple times. If we choose to run each portfolio 5 times per problem, this will require $60,000$ runs. In contrast, the data needed for the above estimation technique will only require $10 \cdot 100 \cdot 5 = 5,000$ runs, and then we can use the above estimation technique to calculate the expected coverage of each of the portfolios. This will mean that we will use empirical results as an estimate of $P(p, a_j, t_j)$.

3.2.3 Multi-Core Planning

Work in the area of parallel planning has typically focused on the parallelization of heuristic search algorithms. This includes PBNF, a recent parallelization of A* which exhibits substantial runtime improvement [5]. However, as described earlier

in the case of LAMA-2011, if a standard A* algorithm is unable to solve a given problem due to memory limitations, PBNF will merely run out of memory more quickly. As such, this algorithm cannot be expected to substantially improve planner coverage except in cases where the failure to solve the problem is a result of time limitations.

HDA* is another recent parallelization of A* that was designed explicitly for use on distributed memory systems [49]. By considering distributed memory architectures, this algorithm may have more memory at its disposal than when restricted to a single shared-memory machine. This additional memory may improve coverage in cases where memory is the bottleneck. Our focus in building ArvandHerd was in constructing a parallel planner for a shared memory architecture, in which case HDA* cannot be expected to substantially improve coverage for the same reason as PBNF. Using the portfolio technique in a distributed memory planner is left as future work.

There is nothing precluding the use of parallel algorithms in a portfolio. If a parallelized algorithm is included, it can be allotted several cores on which to run while the remaining cores will run the rest of the portfolio. Alternatively, the parallelized algorithm can be run until it hits some resource limit, at which point the remainder of the portfolio will be run. This approach would benefit from both the speedups seen with the parallelized algorithm and the coverage improvements seen with a portfolio. As such, research into parallelizing individual search algorithms remains as an important field of study.

3.3 Using Multiple Configurations in LAMA

In this section, we will examine the LAMA planner and demonstrate that the performance of this planner can change depending on how it tie-breaks between nodes with an equal heuristic value and how it is parameterized. We will then show that this variability can be exploited by using multiple configurations through restarts.

3.3.1 The LAMA Planner

LAMA is a best-first search-based planner that won the sequential satisficing track of the 2008 International Planning Competition [39]. An updated version of this planner then won the sequential satisficing track of the 2011 International Planning Competition [77]. In this section, we describe the 2008 version of this planner, which will be called LAMA-2008 for short. As our focus is on coverage, we will not describe the components of this planner that are aimed at improving solution quality once a first solution is found.

Both the 2008 and 2011 versions of LAMA use a number of different planning techniques including multiple heuristics, preferred operators, and deferred heuristic evaluation. The first heuristic used by this planner is the FF heuristic described in Section 2.3.3. In LAMA-2008, the version of this heuristic used considers both the cost-to-go version, FFc, and the distance-to-go version, FFd. This is done by using the heuristic **FF+** which is defined as the sum of FFc and FFd. The second heuristic used is the **landmark-count heuristic**, denoted as **LM**. An **action landmark** a is a set of actions $\{a_0, ..., a_{k-1}\}$, at least one of which must be used during any solution path. The landmark a is said to be **achieved** along a path of actions P if some a_i appears along P. For the LM heuristic, a set of action landmarks is generated and then the heuristic value of any node n is given by the number of unachieved landmarks along the path to n. For a full description of how this inadmissible heuristic is computed and how it can be used to identify additional preferred operators, see the work of Richter and Westphal [76].

Before the first solution is found, LAMA-2008 uses these two heuristics, LM and FF+, in a multi-heuristic best-first search version of GBFS that is also enhanced with boosted preferred operators. LAMA-2008 does allow for the search to use the WA* evaluation function for sorting the various copies of the open lists. However, LAMA-2008 only uses the other weights to try and improve solution quality once a first solution has been found using a process called **Restarting Weighted A*** [75]. As such, only the single GBFS configuration will have an impact on the coverage of this planner in its standard form.

3.3.2 Using Multiple Configurations in LAMA-2008

In this section, we will show that the performance of LAMA can vary depending on the order in which nodes are added to the open list, and that by mixing different configurations of this planner it is possible to improve its performance.

Consider how different **operator orderings** can impact the performance of a planner like LAMA. The operator ordering being used by an OCL algorithm is the sequence in which operators are tested for their applicability to a given state n. This sequence, which is typically static, determines the order in which the nodes in the list of successors of n will appear in when n is expanded, which in turn determines the order in which these successors are added to the open list. This sequence matters because it can determine which node is selected for expansion when there are multiple nodes in the open list that all have the lowest h-cost. This is because the open list of a best-first search algorithm (or the different queues in the case of multi-heuristic best-first search or when using preferred operators) are typically maintained using a heap, or using a set of buckets with each corresponding to the nodes in the open list with a particular heuristic value. In either implementation of a priority queue, the order in which nodes are added to the open list can determine how ties are broken and therefore in what order nodes are selected for expansion.

A given static operator ordering can therefore introduce a bias into the way that ties are broken by a best-first search planner. To see this, suppose that actions a_0, ..., a_k are always tested on a given state in that order. In any state in which a_0 and a_k are both applicable, a_0 will necessarily be added to the open list first. As such, if there are ties in the successors that arise as a result of applying a_0 and a_k, the node corresponding to action a_0 will more likely be ahead of the node corresponding to a_k (or vice versa depending on the implementation of the data structure) in the priority queue.

If there are not many ties, then the impact of this bias will be minimal. However, notice that deferred heuristic evaluation, which is used by LAMA, can increase the number of ties. This is because any two children c_1 and c_2 with the same parent p will have the same heuristic value and the same g-cost if they are achieved with equal cost actions. A unit cost domain represents an extreme case of this phe-

nomenon; all children of the same state will be assigned the same value for $g + H$ since the heuristic value of all children will be the same and equal to that of their parent. The ties that therefore result from this technique will then mean that a different operator ordering can substantially change the search.

This behaviour can be demonstrated by experimenting with *random operator ordering* in LAMA-2008. This technique involves randomly shuffling the list of successors before those new nodes are added to any open list. For this experiment, LAMA-2008 was configured to use random operator ordering in GBFS with the FFd and LM heuristics and preferred operators. We used FFd instead of FF+ as this heuristic was shown to lead to better coverage in experiments performed after the 2008 competition [76]. This system was tested on all 510 problems from the 2006 and 2008 competitions for 5 runs per problem with a 1800 second time limit and a 2 GB memory limit. These experiments were run on a cluster of machines each with two AMD Opteron 250 2.4 GHz processors, each with 1 MB of L2 cache. While the average number of problems solved is 431.8, if the best random seed had been selected on a problem-by-problem basis, 448 problems would have been solved. If the worst seed had been selected on a problem-by-problem basis, only 415 problems would have been solved.

Restarting in LAMA-2008

However, this variance can be exploited by using multiple runs of LAMA-2008 with random operator ordering on the same problem. In particular, we considered the use of multiple runs of this planner through the use of restarts. For the sake of simplicity, we will assume that each configuration is run for the same amount of time. This means that if there were k configurations used, then each is being used for $1800/k$ seconds and so there are $k - 1$ restarts.

The expected coverage when restarting in this way was then calculated using the technique described in Section 3.2.2. Table 3.1 shows this data for different types of search and different numbers of restarts. In these experiments, LAMA uses FFd and LM when running GBFS, and FFc and LM when using WA* (*ie.* $w = 7$ represents a weight 7 search).

Search Type	Number of Restarts 0	1	2	4	8	16
GBFS	431.8	437.0	438.5	440.3	437.3	427.8
$w = 10$	406.2	409.9	411.9	414.1	409.5	399.8
$w = 7$	403.6	408.2	409.1	409.0	405.4	397.7
$w = 5$	399.4	404.6	403.5	403.3	398.8	388.7
$w = 1$	207.2	209.1	209.8	207.3	205.4	194.9

Table 3.1: Expected coverage of `LAMA-2008` when using restarts.

Table 3.1 shows that a small number of restarts can help improve the expected number of problems solved, though too many restarts can degrade performance. The estimation technique also indicates that if `LAMA-2008` is set to restart not just with a new random seed but also with a different configuration, the additional diversity would help to further improve coverage. For example, when restarting 4 times such that each of GBFS, $w = 10$, $w = 7$, $w = 5$, and $w = 1$ are run for a maximum of six minutes, the expected coverage is 448.4. It is therefore not only important to mix multiple orderings, but also configurations.

These results extend some previous work in which we showed that assigning different WA* instance, each using a random operator ordering and a different random seed, to different processors of a multi-core system yielded an effective multi-core solver for certain combinatorial puzzles [100]. These ideas were central to the building of the `ArvandHerd` planner. However, that study only considered standard WA* without any planning enhancements, and only did so with low weights. In this section, we have shown that the `LAMA-2008` is also sensitive to the operator ordering and parameters used despite the fact that this planner is already using multiple heuristics as a way to induce diversity into the search. We have therefore shown that these ideas are still relevant to fully enhanced modern-day planners which can also be improved by mixing configurations and adding further diversity with random operator ordering.

3.4 Using Multiple Configurations in `Arvand`

In this section, we will show that mixing configurations and restarting can also be used to improve the performance of `Arvand`, a planner which explores the state-space in a much different way than the best-first search based `LAMA`.

3.4.1 The `Arvand` Planner

`Arvand` is a sequential satisficing planner that uses heuristically-evaluated **random walks** as the basis for its search [65, 66]. Starting from a node n, a random walk is a path starting from n that follows randomly selected state transitions through the state-space. Random walks are constructed iteratively starting with the path $[n]$. On every iteration, the random walk will extend the current path $[n_0, ..., n_k]$ to path $[n_0, ..., n_k, n_{k+1}]$ where n_{k+1} is a randomly selected node from $succ(n_k)$. This process then continues until either the deepest node on the path has no successors, or the number of nodes in the path has reached some given limit.

`Arvand` uses random walks to explore the state-space through a series of *search episodes*. In the simplest version of `Arvand`, each search episode begins with r random walks, each starting from n_{init} and containing at most $m + 1$ nodes. m is a parameter called the **walk length**. For each of these r random walks, the heuristic value of the final state reached is computed using some heuristic function. Once all r walks have been performed, the search **jumps** to the end of the walk whose final node, n, has the lowest heuristic value. In all experiments below, the heuristic function used is the FFd heuristic. `Arvand` then runs a new set of r random walks, only this time the walks originate from n. This is followed by another jump to the end of the most promising walk (according to the heuristic) from this set of new walks that started at n. This process repeats until either a goal state is found, or some number of consecutive jumps are made without any improvement being seen in the heuristic values of the nodes encountered. In the latter case, the current search episode is terminated, and the planner restarts with a new episode that begins with random walks originating from n_{init}.

During a search episode, the planner builds up a path consisting of the subpaths

found between each jump of the algorithm. This path will be called the **trajectory** of the search episode, and these trajectories can also be used to inform the search performed in future search episodes through the used of a **walk pool** [64]. A walk pool stores the most effective trajectories (*ie.* those which found nodes with the lowest heuristic value) of all those seen thus far. When using this structure, new episodes will start from a state randomly selected from a trajectory that itself was randomly selected from the walk pool, instead of from n_{init}. This technique has been shown to improve `Arvand`'s coverage [64]. Note, the walk pool technique has been disabled in the experiments in Sections 3.4.2 and 3.4.4 when experimenting with different ways to use multiple configurations. This was done to evaluate these techniques in isolation of the communication between configurations that a walk pool allows. However, the walk pool technique is employed when considering the more complete systems evaluated in Sections 3.5.1 and 3.5.3.

3.4.2 `Arvand` Configurations

There are a number of parameters in `Arvand` that can greatly affect its performance on a domain-by-domain basis. Perhaps the most important of these relates to the biasing of the random action selection. `Arvand` allows for random successor selection to be biased in several ways. The first is to bias this selection to avoid nodes that correspond to actions that have previously led to dead-ends. This biasing policy is referred to as **Monte-Carlo Deadlock Avoidance (MDA)**. A second technique biases the successor selection to increase the chance of selecting a successor that corresponds to the application of an action which has previously been identified as a helpful action by the heuristic. This technique is referred to as **Monte-Carlo with Helpful Actions (MHA)**. These different biasing strategies have been shown to each be useful for different domains [65].

There are also a set of parameters that are related to the walk length that can also greatly affect performance. In `Arvand`, this length is increased whenever a search episode terminates to allow the next search episode to get deeper into the state-space. The initial walk length, the frequency with which walks are lengthened, and the factor by which they are lengthened (called the *extending rate*) are all parameters

Config	Bias Type	Initial Walk Length	Extending Rate	Average Coverage
1	MDA	1	1.5	400.8
2	MDA	1	2.0	414.2
3	MDA	10	1.5	397.8
4	MHA	1	1.5	338.8
5	MHA	1	2.0	348.0
6	MHA	10	1.5	386.0

Table 3.2: Performance of different `Arvand` configurations.

affecting this process.

The average performance of six different configurations over 5 runs on each of 480 problems is shown in Table 3.2. The problem set consists of all problems in the 2006 and 2008 planning competitions, except for `sokoban` from IPC 2008. This domain was omitted in the interest of evaluation time since previous testing has indicated that `Arvand` performs poorly in this domain regardless of how it is configured.

These experiments were performed on a cluster of machines each with two 2.19 GHz AMD Opteron 248 processors with 1 MB of L2 cache. Configurations are given a maximum of 1800 seconds and 2 GB per run. Configurations 1 and 4 correspond to the default configurations that use one of MDA or MHA. The remaining four configurations are constructed by modifying these default configurations by either its initial walk length or its extending rate, but not both simultaneously.

While the MDA configurations outperform the MHA configurations, this is not true in all domains [65]. This is clear in Table 3.3, which shows the performance of configurations 1 (MDA) and 4 (MHA) from Table 3.2 on four different domains. This suggests that a combination of configurations is needed. Note that the naming convention used in this thesis for PDDL domains will be to refer to a domain by the concatenation of the year in which those tasks were used in the competition and the domain name. For example, "2008 woodworking" refers to the woodworking problems used in the 2008 competition. As needed, we will also write the number of tasks in the given domain in parentheses besides the domain name.

Domain	Config 1 Coverage	Config 2 Coverage
2006 pathways (30)	16.2	30.0
2006 tpp (30)	26.2	15.0
2008 transport (30)	15.4	23.2
2008 woodworking (30)	30.0	16.4

Table 3.3: Performance of different `Arvand` configurations on selected domains. The number of tasks in each domain are shown in parentheses.

3.4.3 Combining Different `Arvand` Configurations

A simple way to combine k configurations in a single-core version of `Arvand` is to use restarts to run each for $1800/k$ seconds. Given the 6 configurations tested in Table 3.2, there are $\binom{6}{k}$ possible portfolios for any portfolio size k such that $1 \leq k \leq 6$. For each such portfolio for all k, we calculated the expected performance using the technique described in Section 3.2.2 by using the experiments performed for Table 3.2 to estimate the probability that any particular configuration will solve a given problem. When $k = 2$, the best configuration set of all $\binom{6}{2} = 15$ possible portfolios is expected to solve 436.3 problems, an increase of 22.1 problems over the average number solved by the single best configuration alone.[2] All but 2 of these 15 portfolios improved over the single best configuration in its own set when used alone. For $k = 4$ and $k = 6$, the expected coverage of the best sets are 434.4 and 431.4, respectively. These diminishing returns are to be expected since an increase in k decreases the amount of time any individual configuration will run. However, notice that using all 6 of the configurations is still outperforming the use of any single configuration when used alone. The same is also true of even the worst performing of the portfolios containing 4 different configurations.

In practice, instead of starting with a new configuration every $1800/k$ seconds, we alternate amongst the configurations in a round-robin fashion. For each of $k = 2$, $k = 4$, and $k = 6$, we tested this approach with the configuration set of size k with the best expected performance when allocating time equally between

[2]Recall that the test set contains a total of 490 problems.

configurations. The coverage of alternation is slightly better, as it averages 435.4, 439.8, and 439.6 for k values of 2, 4, and 6, respectively, over 5 runs per problems. This occurs because when alternating between configurations, the different configurations are not necessarily used for an equal amount of time. In particular, those configurations which make more heuristic progress will necessarily be used for a higher proportion of the runtime. For example, if two configurations, c_1 and c_2, are used on a single task Γ, and c_1 is less effective on Γ than c_2, then search episodes using c_1 will stop making progress and restart more quickly than those using c_2. The available runtime will therefore skew more towards the longer, more effective c_2 configurations, than to the shorter, quickly-restarting c_1 configurations.

3.4.4 Configuration Selection as a Bandit Problem

We also tested the use of an online configuration selection system for use in `Arvand`. Given a set of configurations C, the system selects a configuration for the next search episode from C based on the performance of the configurations during previous episodes. This system views configuration selection as an instance of the *multi-armed bandit problem* in which C is the set of bandits (*ie.* slot machines) and the search episodes correspond to arm pulls. This paradigm requires the definition of a payoff function that assigns a reward to any search episode. The function used was defined such that the reward given to a search episode e performed with configuration c is given as follows: where n is the state on the trajectory of e with the lowest heuristic value of all nodes in the trajectory, the reward given to e is

$$\max(0, 1 - H(n)/H(n_{init})) \ .$$

Using this reformulation of configuration selection, configurations can be selected online using any multi-armed bandit algorithm. `Arvand` uses UCB [1], which begins by performing a single search episode with every configuration. After this stage, the configuration selected for the next episode is given by

$$\arg\max_{c \in C} Q(c) + \lambda \cdot \sqrt{\ln T(c)/T} \ .$$

where $Q(c)$ is the average reward seen thus far for configuration c, $T(c)$ is the number of search episodes performed with configuration c so far, T is the total

number of search episodes, and λ is a parameter called the *UCB constant value*. This algorithm was used as it has been shown to have strong theoretical guarantees on its ability to balance between focusing on effective selections and exploring the alternatives [1]. However, any multi-armed bandit algorithm could have been used.

To quickly seed the UCB configuration selection system with search episodes from which to learn, the frequency with which episodes restarted was initially set high and then gradually decreased. This was done by adjusting over time the maximum number of random walks performed before the search episode jumps, denoted r, and the number of jumps that can be performed without seeing heuristic progress before the search episode ends, denoted u. For these experiments, each configuration had its own value for r, all of which were initialized as 100 and doubled, up to a maximum of $2,000$, every time that configuration was used. Similarly, each configuration maintained its own value for u, all of which were initialized to 1 and incremented by 1, up to a maximum of 7, each time that configuration was used.

The resulting system was then tested on several of the possible sets of the configurations shown in Table 3.2. In general, the UCB system did not significantly change the coverage of `Arvand` when compared to the use of round-robin configuration selection, but it did improve run-time. For example, four different values of the UCB constant value were tested on the configuration set of size 2 with the best expected performance when allocating equal time to each configuration. Recall that round-robin selection solved 435.4 problems when applied to this same set. Of the UCB constant values tested (0.1, 0.5, 1.0, and 5.0), the most problems solved when using the UCB selector was 439.4 and the least was 437.2. However, the combination of the UCB configuration selection system and the higher rate of restarts did increase the speed of the system. For example, when we only consider the 399 problems solved on all five runs per problem by either the UCB system or round-robin, we see that even the UCB constant resulting in the longest run-time results in a 2.75 times speedup, while the value with the shortest run-time sees a 3.90 times speedup.

3.5 Multi-Core Planning with a Portfolio

As has just been shown, the performance of each of `LAMA-2008` and `Arvand` can be improved by using a portfolio containing multiple configurations of the respective planner. In this section, we will show that by combining the improved version of these planners in a portfolio, we can construct a state-of-the-art multi-core planner called `ArvandHerd`.

We begin by considering a parallelization of `Arvand` as an alternative to parallelizing a best-first search algorithm. This parallelization will later be used in constructing `ArvandHerd`.

3.5.1 Parallelizing `Arvand`

While using multiple configurations and restarts have been shown above to improve the performance of `Arvand` on a single-core machine, using a portfolio of configurations can also be used for constructing an effective parallel version of this planner. The parallelization considered uses each core to run an independent search episode. The only communication between cores is through the use of a shared walk pool and a shared UCB configuration selector. When a core has completed a search episode, it submits the corresponding trajectory to the shared walk pool, and it gets a trajectory in return. The core also submits the reward for its current configuration to the shared UCB configuration selection system and in return is given a configuration to use in its next search episode. This means that every core of the system has access to use any of the given configurations. The correctness of the walk pool and UCB configuration selection system are maintained by limiting access to each to only one thread at a time. As the search episodes dominate execution time, there is little synchronization or contention overhead caused by sharing these resources.

In the multi-core setting, the UCB configuration selection system was also modified to take into account that a new configuration may be requested while some threads are currently running search episodes. This may cause an issue if all configurations have all been used a similar number of times, have a similar average reward, and several processors request a configuration at around the same time. In

	Number of Cores				
	1	2	3	4	8
Coverage	660.4	668.0	671.4	677.8	679.6
Speedup Factor	1.0	1.9	2.5	3.0	3.4

Table 3.4: The performance of parallel `Arvand`.

such a situation, the first processor to get its request in will be given a configuration based on the UCB formula. If another processor then requests a configuration before the first processor completes its search episode, it will be given the same configuration since none of the parameters on which the configuration selection is made will have changed.

In the situation just described, we would prefer that the configurations are selected more evenly from amongst the set of available configurations due to the similarity in rewards and number of times each has been tested. To rectify this situation, once a configuration is returned, its average reward and count is immediately updated as if the yet-to-complete search episode returned a reward of 0. It is only when the search episode completes that this pessimistic reward is undone and the real award is used. This approach, which is inspired by the **virtual loss** idea often used by UCT parallelizations [8], will encourage more exploration amongst the configurations.

Parallel `Arvand` was tested with different numbers of cores on the 790 problems from IPCs 2006, 2008, and 2011. These experiments were performed on a cluster of machines, each with two 4-core 2.8 GHz Intel Xeon E546s processors with 6 MB of L2 cache. The configuration set used is identical to the set used in IPC 2011. It includes configurations 1, 4, and 6 from Table 3.2 and another MDA configuration with an extending rate of 1.5 and an initial walk length of 3. This configuration set was selected manually prior to IPC 2011 based on familiarity with `Arvand`, and also before the expected coverage analysis described above had been performed. We use this set in our experiments below so as to evaluate how parallel `Arvand` contributed to `ArvandHerd`'s success.

Table 3.4 shows the average number of problems solved over five runs per prob-

lem when using different numbers of cores, and how much faster the multi-core versions were in comparison to the single-core version on the 639 problems that were solved on all five runs regardless of the number of cores used. The table shows that the additional processors not only lead to increased coverage, but also improved runtime. However, while 8-core Arvand solved 19.2 more problems on average than the 1-core version, domain-by-domain analysis indicates that domains in which the single-core version exhibits poor performance are often also difficult for the multi-core versions. For example, 1-core Arvand only solves 4.4 of the 30 sokoban problems from IPC 2008 and none of the barman problems from IPC 2011. The 8-core version does improve in sokoban to an average of 6.8 problems solved, but is still unable to solve a single barman problem. As all Arvand configurations we have tested have performed poorly on these domains, this suggests that there is a limit in the coverage that can be achieved by adding more configurations to the system or through parallelizing Arvand. However, LAMA-2008 can solve 26 sokoban problems and 15 barman problems. This motivates the use of both LAMA-2008 and Arvand in a portfolio.

3.5.2 The ArvandHerd Portfolio

At the time when ArvandHerd was developed, LAMA-2008 was the winner of the sequential satisficing track of the most recent International Planning Competition in 2008. It was therefore a natural selection as the starting point when constructing a portfolio-based planner. While we have shown above that the performance of this planner can be improved by using multiple configurations and restarts, simply assigning different configurations to different processors of a shared memory machine requires that the available memory must be partitioned between the configurations. In the case of a planner like LAMA-2008 for which memory is essential, the coverage decrease seen by each individual configuration due to the limit on memory can harm the collective coverage of the portfolio. To see this, we simulated the performance of a portfolio containing subsets of size k of the GBFS, $w = 10$, $w = 7$, $w = 5$, and $w = 1$ configurations of LAMA-2008 tested in Section 3.3.2 on a k-core machine on which each used configuration is given $2/k$ GB of

memory. Regardless of the value of k considered, none of the portfolios matched the expected coverage of 448.4 seen when restarting over all 5 configurations on a single-core machine and giving each configuration a maximum of 6 minutes to run. For example, when simulating the performance of using all 5 configurations on a 5-core machine with each being given 400 MB of memory, the expected coverage is only 442.1 problems. Avoiding this behaviour — which will impact any portfolio to be used in parallel that contains multiple memory-heavy algorithms — is therefore integral for properly selecting a portfolio and was an important consideration when building `ArvandHerd`.

The `ArvandHerd` portfolio does include several configurations of `LAMA-2008`, but these configurations are employed using restarts. As such, these `LAMA-2008` configurations are never used simultaneously so as to avoid the memory issues just discussed. Alongside `LAMA-2008`, the portfolio includes several configurations of `Arvand`, for several reasons. First, this approach is very different from WA* and it is able to solve some problems that the systematic search of WA* is unable to handle. Secondly, domains in which it exhibits poor behaviour are often more successfully tackled by approaches based on WA*. Finally, `Arvand` has low memory requirements, and so when it is run alongside `LAMA` in a shared-memory system, the majority of the memory can be assigned to `LAMA`, thereby avoiding the memory-partitioning issue described above.

`ArvandHerd` Architecture

Let us now consider the architecture of the `ArvandHerd` planner and how this planner uses the configurations in its portfolio. As both `Arvand` and `LAMA-2008` are built on top of `Fast Downward` [38], `ArvandHerd` is run from a single binary. Like `Fast Downward`, this means that the execution of `ArvandHerd` consists of three phases. The first is the *translation* step in which the PDDL problem description is translated to SAS+. The second is the *knowledge compilation* step which builds the data structures that are needed for the landmark-count heuristic. The third is the search step during which the planner tries to solve the problem using search algorithms, multiple heuristics, and other planning enhancements. Note that

translation and knowledge compilation are pre-processing phases, and no attempt was made to parallelize them. While doing so would speed up the ArvandHerd system, we consider this an orthogonal problem to that of parallelizing the actual planning component and leave it as future work. As described previously, these steps are the same for all planners considered in this thesis, and so the time needed for these steps was not counted against the per problem time limit.

When the search phase of ArvandHerd begins, separate threads are spawned and each is assigned to run different members of the portfolio. In the competition setting in which four cores were made available, three threads were assigned to run the parallelization of Arvand described in Section 3.5.1 and the remaining one ran LAMA-2008. In the more general k-core machine setting, $k - 1$ threads will be running the parallelization of Arvand while the remaining thread will run LAMA-2008. The threads running Arvand use the same configuration as was detailed in Section 3.5.1. The LAMA-2008 configurations that were used will be described in the next section.

LAMA-2008 Configurations Used in ArvandHerd

LAMA-2008 as used in ArvandHerd actually uses three heuristics: FFd, FFc, and LM. This was because the experimental analysis performed by Richter and Westphal [76] suggested that FFc was offering important guidance in certain domains that the other two heuristics were not offering. Our own experimentation performed prior to IPC 2011 confirmed this. For example, when using all three heuristics LAMA-2008 solved 464 of the 550 problems taken from the 2008 and 2011 competitions, as opposed to only 449 when FFc was omitted. This is because even these very related heuristics also each have their own strengths and weaknesses. For example, in the 2011 barman domain, LAMA-2008 with FFd and LM solves 17 of the 20 problems, while LAMA-2008 with FFc and LM solves 7, and LAMA-2008 with all three heuristics solves 16. The 2011 woodworking domain provides another example of this type of behaviour as LAMA-2008 with FFd and LM solves 8 of the 20 problems, LAMA-2008 with FFc and LM solves 15, and LAMA with all three heuristics solves 17.

In the version of LAMA-2008 used in ArvandHerd at the time of IPC 2011, the FF implementation used at the time did not allow for both FFd and FFc to be computed using only a single solution to the relaxed version of the problem. Instead, the planner computed two plans to the relaxation of the problem, each based on a different implementation of FF. One of these plans was then used to compute FFd and the other was used to compute FFc. In subsequent analysis, we discovered that doing so increases coverage but at a significant cost to run-time. For this analysis, we modified our implementation of LAMA-2008 so that it could compute both FFd and FFc from the same plan. Over the test set given by all problems from the competitions in 2006, 2008, and 2011, the configuration that computed two plans solved 668 of 790 problems compared to the configuration which only used a single plan which solves 661. However, if we examine the total runtime by each of these approaches on the 644 problems that both configurations solved, we see that the version of LAMA-2008 that computed two relaxed plans is a factor of 1.36 times slower than the other configuration. Due to this time difference, the version of LAMA-2008 used in ArvandHerd in the evaluation in Section 3.5.3 will only require one plan to be computed for the relaxed version of the problem.

LAMA-2008 was also set to use random operator ordering at all times, and use multiple types of search using restarting. On the first iteration, the planner uses GBFS, which is followed by a set of WA* searches that use the following weights in the order given: 10, 5, 2, and 1. These configurations were not given an equal time however, as instead of restarting on a time limit, LAMA-2008 was set to restart on a memory limit of 4 GB. This limit was enforced through the use of an internal memory estimator. The closed list was also saved in between restarts so as to avoid recomputing the heuristic values of states seen in previous iterations. Subsequent experiments suggest that this was not necessarily the best approach. When restarting with a 2 GB memory limit and using the 5 configurations described in Section 3.3.2, an average of 440.2 problems were solved — which is lower than the 448.4 expected when restarting on a time limit. This gap is at least partially due to inaccuracies in our memory estimator which could only provide rough estimates. However, this approach was maintained in the experimentation detailed below.

Since the planner is set to restart on a memory limit instead of a time limit, it is necessary to have a policy in place for the case in which all configurations run out of memory and there is still time remaining. If this happens then the given cycle of searches — starting again with GBFS — is repeated indefinitely until the time limit is reached or a solution is found.

Note that the configuration set and the policy used after this set has been tried once both differ slightly from the version of `ArvandHerd` that competed in IPC 2011. In the competition version, the final weight 1 iteration was followed by 2 more weight 1 iterations and a weight 0 search. If the weight 0 search failed without having found a solution, the thread running `LAMA` would then join the others in running parallel `Arvand`. For the version of `ArvandHerd` tested below, the extra low-weight iterations were dropped as they were initially included for plan improvement and our focus in this chapter is on coverage. The switch to `Arvand` was also dropped since while this is a reasonable approach if the portfolio can be run from a single binary (as it can in `ArvandHerd`), this is may be much more difficult to do if other planners are added to the portfolio. In the interest of evaluating the general portfolio technique, `LAMA-2008` will instead cycle back to perform GBFS and the weighted searches in the order given above.

3.5.3 `ArvandHerd` on IPC Benchmarks

`ArvandHerd` was run 5 times on each of the 790 problems in the 2006, 2008, and 2011 planning competitions on the same cluster described in Section 3.5.1. In this section, the performance of this planner is compared with the parallelization of `Arvand` and the simulated parallelizations of `LAMA-2008`, `LAMA-2011`, and `Stone Soup` that assume a perfectly linear speedup.

Let us first briefly describe `LAMA-2011` and `Stone Soup`. `LAMA-2011` is an update to the original `LAMA-2008` that was submitted to the 2011 International Planning Competition [77]. It is built into the modern `Fast Downward` framework. `LAMA-2011` uses the distance-to-go FFd heuristic instead of FF+, since an experimental evaluation conducted by Richter and Westphal after IPC 2008 suggested doing so was beneficial for planner coverage [76]. The newer version of the

planner also includes improved memory management and faster heuristic computation, which is largely the reason for the improved performance.

`Stone Soup` is a portfolio planner that is also built into the modern `Fast Downward` framework. The way the portfolio and restarting strategy is constructed has previously been described in Section 3.2.3. The space of planners considered for the portfolio is given by best-first search variants that differ in the heuristics used, the use of GBFS or WA* with different weights, and the planning enhancements employed. Like with the other single-core planners, the k-core simulation was performed by running this planner for $k \cdot 1800$ seconds and counting any problem solved with that limit as being solved in 1800 seconds with k cores. We did not rerun the portfolio selection phase so that the planner could account for the additional time allowed, but instead extended the restarting times so that the proportion of time spent running any planner remained the same regardless of the value of k. This means that the simulated parallel performance of this planner is based on the assumption that all portfolio members could be parallelized with a perfectly linear speedup, and each of the resulting parallel planners would be run using the restarting policy given by the initial portfolio construction.

The performance of these systems and `ArvandHerd` can be seen in Table 3.5. The table clearly demonstrates that the coverage of `ArvandHerd` is significantly greater than the coverage of either a perfectly linear parallelization of `LAMA-2008` or the parallelization of `Arvand` described in Section 3.5.1. Note that the version of `LAMA-2008` tested in this experiment is the original version of this planner, and so it uses the FF+ heuristic instead of FFc or FFd. While the coverage of `LAMA-2008` and `Arvand` both lag significantly behind the coverage of `LAMA-2011`, the combination of these planners as used in `ArvandHerd` surpasses the simulated performance of even a perfectly linear parallelization of the two state-of-the-art single-core planners considered: `LAMA-2011` and `Stone Soup`.

Unlike `ArvandHerd`, `Stone Soup`'s portfolio does not consider a design space beyond best-first search variants. In contrast, `ArvandHerd` includes the random-walk based `Arvand` planner as a substantially different planning approach whose added diversity is very important to the planner's success as can be seen

Planner	Number of Cores 1	2	4	8
LAMA-2008 Simulation	639.0	641.0	643.0	NA
LAMA-2011 Simulation	721.0	724.0	726.0	727.0
Stone Soup Simulation	720.0	724.0	726.0	727.0
Parallel Arvand	660.4	668.0	677.8	679.6
ArvandHerd	NA	737.2	743.2	741.8
ArvandHerd +LAMA-2011	NA	750.4	754.2	755.2

Table 3.5: Performance of parallel planners.

when comparing the performance of ArvandHerd to that of LAMA-2008. This can also be seen when we consider the results on a domain-by-domain basis which is shown in Table 3.6. The table shows the performance of three planners on all domains tested. The first is a version of LAMA-2008 that uses the FFd and FFc (*ie.* the version of LAMA-2008 used inside ArvandHerd aside from the use of random operator ordering and restarts). The second is a single-core version of Arvand. The final planner is the 2-core version of ArvandHerd. Note that the table has been partitioned into three sections. The first shows the 2006 domains, the second shows the 2008 domains, and the third shows the 2011 domains.

As the table demonstrates that the portfolio is increasing coverage in the expected way: with Arvand and LAMA cancelling out each others' weaknesses. For example, recall that Arvand is unable to solve a single problem in barman (IPC 2011). With LAMA-2008 in the portfolio, the 2-core version of ArvandHerd solves an average of 15.4 out of 20 problems (similar to the 16 problems that LAMA handles when on its own). In contrast, LAMA-2008 solves 19 of 30 problems in storage (IPC 2006), while ArvandHerd solves an average of 29.4 (similar to the 30 that Arvand solves when on its own).

3.5.4 Using LAMA-2011 in ArvandHerd

LAMA-2008 was included in the portfolio instead of LAMA-2011 because Arvand had already been built into the LAMA-2008 code-base prior to the competition. The row labelled "ArvandHerd +LAMA-2011" in Table 3.5 shows that perfor-

Domain Name (# of Problems)	Planner LAMA-2008	Arvand	2-Core ArvandHerd
2006 openstacks (30)	30.0	25.8	30.0
2006 pathways (30)	28.0	30.0	30.0
2006 rovers (40)	40.0	40.0	40.0
2006 storage (30)	19.0	29.4	29.6
2006 tankage (50)	38.0	43.8	45.2
2006 tpp (30)	30.0	30.0	30.0
2006 trucks (30)	12.0	19.2	16.6
2006 Totals (240)	197.0	218.2	221.4
2008 cybersecurity (30)	30	30	30
2008 elevators (30)	30	30	30
2008 openstacks (30)	30	30	30
2008 parcprinter (30)	23	30	30
2008 pegsol (30)	30	29.8	29.8
2008 scanalyzer (30)	30	28.6	30
2008 sokoban (30)	24	4.4	25.8
2008 transport (30)	30	30	30
2008 woodworking (30)	30	30	30
2008 Totals (270)	257.0	242.8	265.6
2011 barman	16	0	15.4
2011 elevators	13	20	20
2011 floortile	2	1.6	5.2
2011 nomystery	11	19	18.2
2011 openstacks	20	20	20
2011 parcprinter	5	20	20
2011 parking	17	6.8	19
2011 pegsol	20	19.8	20
2011 scanalyzer	20	17.6	20
2011 sokoban	14	1.8	15.6
2011 tidybot	13	18.4	18.4
2011 transport	19	14.4	20
2011 visitall	20	20	19.8
2011 woodworking	17	20	18.6
2011 Totals (280)	207.0	199.4	250.2
All Totals (790)	661.0	660.4	737.2

Table 3.6: The coverage by LAMA-2008 using the FFc and FFd heuristics, Arvand, and 2-core ArvandHerd.

mance would further improve if LAMA-2011 had been used instead. For simulating the k-core performance of this new portfolio, parallel Arvand was run with $k-1$ cores on all problems that LAMA-2011 could not solve when given a 4 GB memory limit. The table shows the sum of the number of problems solved by LAMA-2011 and the average number of problems solved by $k-1$-cores running parallel Arvand. That the new portfolio successfully solves even more problems than it did before reflects the importance of Arvand in the portfolio. Arvand is not simply covering the weaknesses in LAMA-2008 that have been addressed with the release of LAMA-2011. As such, ArvandHerd is also capably handling problems that this state-of-the-art planner cannot.

3.6 Chapter Summary

In this chapter, we have demonstrated that two existing planners, LAMA-2008 and Arvand, each have their own strengths and weaknesses in terms of the domains they are best suited for, and that these strengths and weaknesses depend somewhat on how design choices are made. We have also shown that the use of multiple configurations and restarts can improve the coverage of each of the two planners used in the portfolio, namely LAMA-2008 and Arvand, even when used on only a single core. While these techniques have previously been used in the SAT-solving community, we have shown that their success extends into automated planning.

We then considered how these ideas would apply when building a parallel planner. This included demonstrating that parallelizing a single planning algorithm is not necessarily the best way to use a multi-core shared memory machine if the goal is to maximize coverage. While the parallelized algorithm may be faster, it will have similar limitations as the original single-core algorithm in terms of both resource usage and the domains it handles well. Instead of parallelizing a single algorithm, we used an algorithm portfolio approach to parallel planning in the development of ArvandHerd, which won the multi-core sequential satisficing track at IPC 2011. The combination of the planners used in ArvandHerd is then shown to outperform even the simulated performance of perfectly linear parallelizations

of several state-of-the-art single-core planners. It is also shown that the coverage can be further improved by replacing `LAMA-2008` with `LAMA-2011`in the `ArvandHerd` portfolio. More generally, we have demonstrated through the construction of `ArvandHerd` that the use of a portfolio is a powerful approach for building general parallel planners due to its ability to combine the strengths of different planners.

Chapter 4

Adding Random Exploration to GBFS

In this chapter, we introduce two simple ways of introducing random exploration into a GBFS-based planner to make GBFS less sensitive to errors in the heuristic. In doing so, we identify the use of random exploration as a useful design decision that can be employed when building a GBFS-based system. We then argue that these results suggest that GBFS enhancements that are based on additional problem structure need to be compared against random-based baselines to ensure that the variation these enhancements introduce into the way the state-space is examined is not merely equivalent to random exploration. This study further confirmed the value of the preferred operator and multi-heuristic best-first search techniques.

4.1 Introduction

Greedy Best-First Search (GBFS) is a popular algorithm that is used in many heuristic search-based satisficing planners including LAMA [76] and Fast Downward [38]. While it has no provable bounds on solution quality that can be guaranteed *a priori* of any search, GBFS is typically much faster than optimal algorithms such as A*, as well as other OCL algorithms like WA* that do have guarantees on the suboptimality of any solution found. Bounded algorithms often have to expand nodes that the heuristic identifies as less promising in order to ensure the bound is satisfied, while GBFS simply searches greedily according to the heuristic.

However, the greediness of GBFS can also cause the algorithm to exhibit poor

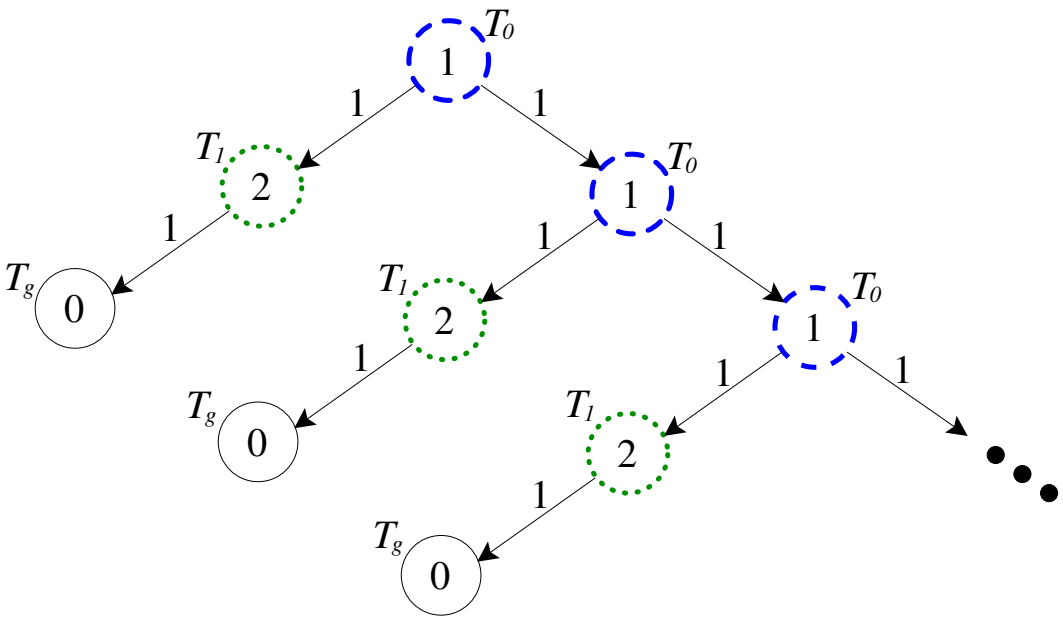


Figure 4.1: Infinite plateau with 3 types of nodes.

performance if the heuristic and the tie-breaking policy are not effectively guiding the search. As an example of such behaviour, let us recursively define the following simple, but infinite tree. In this graph, there are three types of nodes, denoted as T_0, T_1, and T_g. Every type T_0 node has b successors for some constant $b > 0$. $b - 1$ of these successors are T_0 nodes and the remaining successor is a type T_1 node. Type T_1 nodes have only a single successor: a goal, or type T_g, node. The heuristic value of type T_0 nodes are 1, while the heuristic value of the type T_1 nodes are 2. For example, Figure 4.1 shows an example of such a graph in the case that $b = 2$, where type T_0 nodes are shown as the dashed blue circles and type T_1 nodes are shown as dotted green circles.

In this type of graph, the type T_0 nodes form an infinitely large **heuristic plateau**. This means that these nodes form a contiguous section of the state-space on which all nodes have the same heuristic value. In the case of the graph shown in Figure 4.1, this plateau sits within a **heuristic local minimum** since any path from a node on this plateau must pass through a node with a higher heuristic value before a node with a lower heuristic value can be found. Heuristic local minima pose a problem for GBFS, since once the algorithm expands a single node in such a region of the

search space, it must expand all the nodes before it can escape the local minimum. If a given planning task has one of the infinite graphs defined above as a subgraph, this means that once a type T_0 node is expanded, GBFS will be stuck in that region forever. As such, GBFS will never find a solution path once it first expands a type T_1 node, despite the fact that in this case, all nodes in this region are at most a path of length two away from a goal node.

Any region of the search space which the heuristic incorrectly identifies as being more promising than the nodes along solution paths will similarly be problematic for GBFS which will have to exhaustively explore such areas before the algorithm can return to the nodes with higher heuristic values. In such cases, the heuristic will often not properly rank nodes according to the relative ease with which a solution can be found from them. This behaviour can be seen in Figure 4.2, which shows the performance of GBFS on each of 1000 randomly generated 15 puzzle states. For each state, the figure shows the heuristic value and the average number of nodes expanded over 100 runs per task when using random operator ordering, never allowing for re-expansions, and tie-breaking in favour of nodes with a lower g-cost. As the figure shows, nodes with the same heuristic value can differ greatly in how difficult it is to find a solution from them. Moreover, it is often substantially easier to find a solution from a node with a higher heuristic value than a node with a lower heuristic value.

To improve the performance of GBFS-based planners on such problems, the standard algorithm is often enhanced with techniques such as preferred operators or multi-heuristic best-first search. These enhancements use automatically identified problem structure to provide an alternative source of guidance. The goal of using such enhancements is to improve the way that the state-space is examined by making the search more informed. As the variation introduced into the search by these enhancements is based on knowledge, we refer to such enhancements as being **knowledge-based**.

Variation can also be introduced into GBFS by using **random exploration**. For example, we can change the way that nodes are iteratively selected for expansion by making the algorithm occasionally select a random node from the open list instead

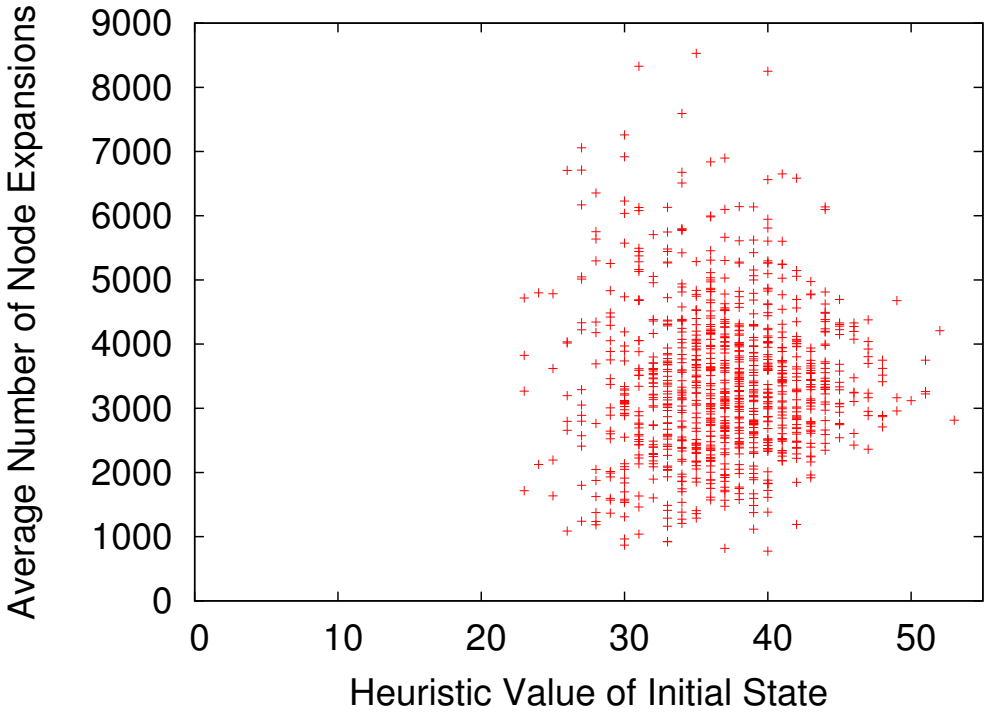


Figure 4.2: The average number of node expansions for 1000 randomly generated 15 puzzle states.

of the node suggested by the heuristic. This technique is easy to implement, has a low execution overhead, and does not require additional knowledge about the problem. The variation added by such a technique is not based on problem structure, and is therefore **knowledge-free**.

While several existing techniques have employed knowledge-free random exploration in some fashion, it has always been in combination with other approaches. As such, it has not previously been clear what the contribution of random exploration was to planner performance, and whether system designers should opt to use this technique when building a GBFS-based planner. Clearly identifying the impact of introducing random exploration into GBFS was therefore one of the goals of this line of research.

4.1.1 Contributions

In this chapter, we will introduce a simple technique for isolating the impact of adding random exploration to GBFS. The technique is called **ϵ-greedy node selection**. While several previous researchers have also introduced non-determinism to GBFS, this technique will more clearly isolate the impact of adding random exploration to GBFS, show that it is beneficial, and thereby demonstrate that it is a useful option to consider when building a GBFS-based planner. We will also introduce a second simple technique for encouraging exploration with randomness which will also improve the performance of this algorithm. This technique, called **heuristic perturbation**, will be shown to be much riskier than ϵ-greedy node selection, as it will result in substantial performance improvements in some domains and substantial decreases in others. Due to this variance, it will be shown to be quite useful when used alongside ϵ-greedy node selection in a portfolio.

Given the positive impact of adding random exploration to GBFS, we will argue that this result suggests a need to revisit the existing knowledge-based enhancements to determine if the knowledge they use is offering important guidance, or if the main impact of this knowledge is to add variation that can be replicated using simpler knowledge-free approaches. We therefore evaluate the impact of the knowledge used in the knowledge-based enhancements by comparing these enhancements to equivalent systems in which the knowledge has been replaced by randomness. This investigation confirms that preferred operators are offering much more to the search than simply adding random variation, and that the knowledge employed when using secondary heuristics is crucial in some domains while not offering enough variation in others.

This chapter is based on two publications. The first is a technical report that introduced ϵ-greedy node selection and heuristic perturbation [98]. This was then followed by a paper at the 2014 International Conference on Automated Planning and Scheduling which detailed ϵ-greedy node selection and a study of the impact of the knowledge used by existing GBFS enhancements [101]. Fan Xie was also involved in this latter publication as he first proposed and performed the experiments detailed below that test the value of the helpful actions when used as preferred oper-

ators in a GBFS guided by the FF heuristic. I later re-ran these experiments so that they were run on an equivalent system as was used in the remainder of the paper, and connected that study with the idea of using random baselines to evaluate the impact of the knowledge used by GBFS enhancements.

4.2 Related Work

We begin this section by describing several deterministic algorithms introduced to better deal with errors in the heuristic function in use. We will then describe several algorithms which also encourage exploration using non-determinism.

4.2.1 Dealing with Early Mistakes

An **early mistake** in the heuristic function is an error in the heuristic value of a node near the initial node that causes the algorithm to commit to first exploring a region of the state-space from which a goal node cannot be easily found. The approach taken by Felner, Kraus, and Korf for better dealing with early mistakes is called KBFS [20]. This algorithm framework is almost identical to rBFS, except on every iteration it expands k nodes instead of just one, where k is an algorithm parameter such that $k > 1$.[1] The k nodes selected for expansion are those in the open list with the lowest heuristic value.[2] Once this set of nodes is selected, the goal test is performed on the nodes it contains. If a goal is found among this set of nodes, the algorithm returns the solution extracted from the parent pointers just like in any OCL algorithm. If not, all the children of the k nodes are generated, and this list is treated just as the children of the single node selected for expansion in an OCL algorithm are. This means that they are considered individually for being added to the open list and any necessary g-cost and parent pointer updates are performed. By having the algorithm expand nodes that are not always at the top of the open list, this algorithm will often push the search to explore areas of the search against the advice of the heuristic function.

[1] Though it was not initially described as such, KBFS can defined as an OCL algorithm. We describe it as it was initially introduced for the sake of simplicity.

[2] KBFS was initially defined so as to use any evaluation function Φ instead of only considering the heuristic function. We ignore the more general case for the sake of brevity.

Harvey and Ginsberg introduced **limited discrepancy backtracking** as a technique for better handling early mistakes in graphs that are binary trees [36]. For defining this technique, we will need the following convention: we will assume that the **left child** of a node in such a tree is the child with the lower heuristic value than the **right child** (or which is considered more promising according to some tie-breaking policy). In such binary trees, the standard approach is to use a **depth-first search** that, when considering which child of a node to explore first, always goes to the left child. This means that the algorithm maintains a single candidate solution path, and on every iteration it adds the leftmost child of the deepest node on the candidate path for constructing the candidate path used in the next iteration. When backtracking, the deepest node on the candidate path is iteratively removed until a node is found for which the candidate paths along its right child have not been examined. The search then adds this right child to the remaining candidate path and performs a depth-first search from that node.

The result of such backtracking is that the search first explores alternatives near the leaves of the tree being searched before it can backtrack and consider alternatives near the initial state. In contrast, limited discrepancy backtracking causes the search to first explore alternatives near the initial node. It does so by incrementally increasing the number of **discrepancies** along the current candidate path. A discrepancy occurs whenever the candidate path follows the right child of a node along the path instead of the left child. On the first iteration, the algorithm does not allow for any discrepancies, and so it will follow the path consisting only of the left child nodes. If a goal node is found, the algorithm returns the path found to it. Otherwise, the algorithm restarts from the initial node and performs a depth-first search that allows for a single discrepancy along the candidate path. This discrepancy is also used as soon as possible, and so the first candidate path considered consists of first taking the right child of the initial node, and then proceeding to only follow the edges that correspond to the left children. The backtracking performed will then make the search examine every candidate path that contains at most one right edge. If all such paths are tried without a solution path being found, the search then restarts, and the number of allowed discrepancies is then increased to two. This

process then continues until a solution is found. Limited discrepancy backtracking was later generalized to apply to arbitrary graphs instead of just binary trees by Furcy and Koenig in an algorithm called BULB [25].

While both KBFS and limited discrepancy backtracking add variation to the search, this variation is still heavily influenced by the heuristic function. In KBFS, the node selected for expansion on every iteration will still correspond to those with a low heuristic value unless k is very large, in which case the algorithm will resemble breadth-first search. The depth-first nature of a limited discrepancy search, even in the case of BULB, also means that the search may still have to examine a large number of nodes before it ever looks at a node near the initial node that can only be found with even a small number of discrepancies. This is because these techniques still heavily favour those nodes that can be achieved greedily according to the heuristic. This differs from the techniques introduced below which will introduce exploration without heuristic bias.

4.2.2 Encouraging Exploration with Stochasticity

Several systems have been introduced which add non-determinism into search algorithms so as to introduce useful variation to the search. One such algorithm is **diverse best-first search (DBFS)** [47]. The execution of this algorithm consists of two phases that alternate. In the first, a node n is randomly selected from the open list according to a distribution which favours nodes with a low g-cost and a low heuristic value. In the second, a local GBFS search is initiated from n (*ie.* as if n is the initial node) with a node expansion limit. Once that limit is hit, the open and closed lists used by the local GBFS are merged with the open and closed lists of the main search, and algorithm returns back to the first phase. This process repeats until a solution is found.

Xie, Müller, and Holte have also shown that adding a local search to a GBFS as is done in DBFS can substantially improve planner performance, particularly in domains with large plateaus [105]. This is shown to be true regardless of if the local search is GBFS — as in DBFS — or if the local exploration is performed using `Arvand`-like random walks, as is the case in the Roamer planner [60].

Though ϵ-greedy node selection may not increase performance as much as these other approaches in all cases, we will see that it still increases coverage substantially. In any case, the main purpose of ϵ-greedy node selection is not to compete with these other approaches, but to isolate the impact of adding random exploration to GBFS and to clearly demonstrate its positive impact. In contrast, these other approaches often use a multitude of techniques like local search and random walks, and so it is not entirely clear as to the specific contribution of random exploration. This is also true of several other systems including Lamar and Randward [67], which use a stochastic version of the FF heuristic that is constructed by adding randomness into the way in which the heuristic is computed.

Recent work by Xie *et al.* [106] biases the random exploration through the introduction of a **type-system**. A type-system is a partitioning of the nodes in the state-space into groups that each contain nodes that are expected to be similar according to some system. The added random exploration is then biased to explore different types evenly. While ϵ-greedy node selection with a particular parameter setting is a special case of type-based exploration, ϵ-greedy node selection more clearly captures the impact of random exploration than does the introduction of a type-based biasing.

The ϵ-greedy approach described below has also been previously used in the context of generating a diverse set of plans [10], though not with GBFS. In that work, once a first solution was found using a standard search algorithm, this technique was used to introduce variation to an **enhanced hill-climbing search** to encourage the search to find solution paths that differed from those already found. In contrast, we will use this technique to demonstrate the positive impact of adding random exploration to GBFS to find a first solution, instead of in encouraging the search to find a differing set of plans.

4.3 ϵ-Greedy Node Selection

In this section, we describe **ϵ-greedy node selection**. This simple modification to the standard GBFS algorithm will clearly demonstrate the value in adding knowledge-

free random exploration to GBFS. This will be shown through experiments with this technique on PDDL planning problems.

Let us begin by recalling that GBFS is an nrOCL algorithm that uses the policy shown in Algorithm 9 for selecting nodes from the open list, where H is a heuristic function. By this definition, the algorithm always selects greedily according to the

`SelectNode`(OPEN):
1: **return** $\text{argmin}_{n \in \text{OPEN}} H(n)$

Algorithm 9: The GBFS `SelectNode` function.

heuristic function. This means that it always **exploits** the guidance it is given in the form of the heuristic and never goes against this guidance.

In contrast, ϵ-greedy node selection modifies this node selection policy to allow for random exploration to be explicitly added to the algorithm. This technique is inspired by the ϵ-greedy policies often used for multi-armed bandit problems [85]. When using ϵ-greedy node selection, the search uses the policy shown in Algorithm 10, where ϵ is an algorithm parameter that must be in the range $[0, 1]$.

`SelectNode`(OPEN):
1: With probability $(1 - \epsilon)$, **return** $\text{argmin}_{n \in \text{OPEN}} H(n)$
2: **return** a node randomly selected from OPEN

Algorithm 10: The `SelectNode` function used when employing ϵ-greedy node selection.

By this definition, ϵ-greedy node selection uses the same rule as GBFS to select a node for expansion with probability $(1 - \epsilon)$. However, with probability ϵ this technique selects a node uniformly at random from amongst all nodes in the open list. This means that the value of ϵ determines how often this node selection policy chooses greedily according to the heuristic (or how often it exploits the heuristic information), and how often the algorithm explores randomly. Notice that if $\epsilon = 0$ then search will be identical to a standard GBFS, while if $\epsilon = 1$ the search will be using a purely random node selection policy.

By introducing random node selection into its search, ϵ-greedy node selection will cause the algorithm to push the search into directions of the state-space it might

not otherwise consider. For example, this technique can be shown to have a finite expected runtime on the graph given in Figure 4.1, as is done in Section A.2 of Appendix A. Since this technique only changes GBFS by causing it to occasionally ignore the heuristic by selecting nodes from the open list uniformly at random, we can use this technique to isolate and evaluate the impact of adding knowledge-free random exploration to a GBFS-based planner. We do so in the next section.

4.3.1 Adding ϵ-Greedy Node Selection to Simple Planners

We begin by evaluating the impact of this technique when it is added to a simple planner that only uses a single heuristic. In this evaluation, we will only consider coverage. However, it should be noted that this technique could always be used as the first iteration in an iterative plan improvement system, such as the restarting WA* search used in `LAMA` [75] or in a more modern approach that also uses a planner post-processor like Diverse Any-time Search [107]. We expect that when doing so, the quality of solutions found would be similar.

The experimental setup for both this section and the rest of this chapter is as follows. All experiments are performed using the Fast Downward planning system. The test set is composed of the 790 problems from IPCs 2006, 2008, and 2011, and these tasks are treated as being unit-cost since our focus is on coverage. The experiments were performed on a cluster of 8-core machines, each with two 4-core 2.8 GHz Intel Xeon E546s processors and 6 MB of L2 cache.[3] Planners were run with a 4 GB per-problem memory limit, and a 1800 second per-problem time limit. All tested planners were set to break ties in their heuristic values in a first-in first-out manner and they never perform re-expansions. For stochastic planners, the coverage shown is the average coverage seen over 10 runs on each problem.

The first planner tested runs GBFS enhanced with deferred heuristic evaluation and the FF heuristic used as the single heuristic function providing guidance. Random operator ordering was also used in these experiments so as to avoid the bias

[3]The same machine was used in the experiments with `ArvandHerd` in Chapter 3. However, these numbers are not directly comparable. This is because in the experiments below, 7 processes were run at a time, while only 1 process was run at any time in the `ArvandHerd` evaluation (even when considering single-core planners) so as to fairly compare single-core and multi-core planners.

Domain Name (# of Problems)	ϵ 0	0.05	0.1	0.2	0.3	0.5	0.75	0.99
2006 openstacks (30)	30.0	30.0	29.9	29.8	29.9	30.0	29.4	29.3
2006 pathways (30)	9.2	**12.3**	**11.2**	**11.3**	**11.4**	**10.1**	10.0	*5.7*
2006 rovers (40)	22.3	**25.5**	**25.4**	**25.9**	**25.8**	**26.1**	**24.2**	*18.1*
2006 storage (30)	20.1	20.7	20.9	20.8	20.8	21.0	20.8	*18.6*
2006 tankage (50)	21.4	**26.0**	**25.7**	**26.5**	**26.6**	**26.6**	**26.9**	21.6
2006 tpp (30)	21.4	21.8	21.8	21.1	*20.1*	*17.6*	*16.1*	*13.1*
2006 trucks (30)	16.1	**18.2**	**18.2**	**18.1**	**17.6**	**17.8**	16.9	15.2
2006 Totals (240)	140.5	154.5	153.1	153.5	152.2	149.2	144.3	121.6
2008 cybersecurity (30)	25.3	**29.4**	**29.9**	**29.4**	**30.0**	**29.5**	**30.0**	**29.8**
2008 elevators (30)	30.0	30.0	30.0	30.0	30.0	30.0	30.0	*29.0*
2008 openstacks (30)	30.0	30.0	30.0	30.0	30.0	30.0	30.0	30.0
2008 parcprinter (30)	25.7	26.4	**26.8**	**27.2**	**26.8**	26.4	26.5	25.7
2008 pegsol (30)	30.0	30.0	30.0	30.0	30.0	30.0	30.0	29.9
2008 scanalyzer (30)	27.6	**29.6**	**29.1**	**28.7**	**28.8**	27.9	27.5	*22.1*
2008 sokoban (30)	29.0	29.0	29.0	29.0	29.0	29.0	29.0	*28.0*
2008 transport (30)	17.2	17.8	17.9	17.5	17.8	16.8	*15.8*	*13.7*
2008 woodworking (30)	15.4	**22.8**	**23.9**	**26.3**	**27.3**	**27.7**	**24.4**	16.1
2008 Totals (270)	230.2	245.0	246.6	248.1	249.7	247.3	243.2	224.3
2011 barman	11.4	**16.8**	**17.8**	**18.2**	**18.3**	**17.5**	**14.0**	*0.0*
2011 elevators	13.3	**14.4**	**14.7**	**14.7**	**14.9**	13.7	*11.9*	*8.6*
2011 floortile	4.2	**6.5**	**6.4**	**6.3**	**6.4**	**6.5**	**6.2**	5.1
2011 nomystery	8.4	8.2	9.1	8.6	8.5	9.3	9.1	7.7
2011 openstacks	18.5	18.7	18.5	17.9	*17.5*	*16.5*	*14.4*	*10.0*
2011 parcprinter	11.6	**12.6**	**13.7**	**14.1**	**13.6**	**13.8**	**13.0**	12.3
2011 parking	14.4	14.0	*12.5*	*12.6*	*11.5*	*10.0*	*6.4*	*0.3*
2011 pegsol	20.0	20.0	20.0	20.0	20.0	20.0	20.0	19.9
2011 scanalyzer	17.8	**19.2**	18.7	18.4	18.6	18.1	17.3	*12.1*
2011 sokoban	19.0	19.0	19.0	19.0	19.0	19.0	19.0	18.1
2011 tidybot	11.2	11.9	**13.0**	**13.8**	**14.4**	**14.6**	**13.7**	*6.8*
2011 transport	0.0	0.4	0.2	0.1	0.0	0.0	0.0	0.0
2011 visitall	3.8	**7.2**	**7.0**	**6.9**	**6.6**	**5.8**	4.6	*1.8*
2011 woodworking	2.6	**9.5**	**11.1**	**12.9**	**13.5**	**13.2**	**9.2**	2.0
2011 Totals (280)	156.2	178.4	181.7	183.5	182.8	178.0	158.8	104.7
All Totals (790)	526.9	577.9	581.4	585.1	584.7	574.5	546.3	450.6

Table 4.1: The average coverage when using ϵ-greedy node selection with the FF heuristic. Entries in bold and blue (italics and red) denote that ϵ-greedy node selection with the corresponding value for ϵ solved an average of at least one more (fewer) problem than standard GBFS on that domain.

Planner	ϵ 0	0.05	0.1	0.2	0.3	0.5	0.75	0.99
GBFS$^{\mathrm{FF}}$	533.2	591.4	599.1	602.3	596.8	584.8	555.6	456.4
GBFS$^{\mathrm{dFF}}$	526.9	577.9	581.4	585.1	584.7	574.5	546.3	450.6
GBFS$^{\mathrm{dCEA}}$	491.9	543.0	541.3	540.2	536.0	532.5	493.1	384.4
GBFS$^{\mathrm{dCG}}$	482.9	521.4	525.0	529.9	524.0	509.1	479.3	372.7
LAMA-2011	676.0	704.8	706.5	705.9	705.1	697.8	684.4	537.4

Table 4.2: The average coverage of various planners that use random operator ordering when ϵ-greedy is added to them. dH implies that the heuristic H was used with deferred heuristic evaluation.

introduced when using a single operator ordering. The average coverage of the standard GBFS algorithm (*ie.* $\epsilon = 0$) and GBFS enhanced with ϵ-greedy node selection on all domains tested is shown in Table 4.1. Entries in bold and blue denote that ϵ-greedy node selection with the corresponding value for ϵ solved an average of at least one more problem than standard GBFS on that domain. Entries in italics and red denote that ϵ-greedy node selection with the corresponding value for ϵ solved an average of at least one fewer problem than standard GBFS on that domain.

The table shows that by adding random exploration to a GBFS search we can substantially improve the coverage on the given test set. This is true for a wide range of values of ϵ and in many domains. In some of these domains the magnitude of this increase is also quite high. For example, for all values of ϵ tested where $\epsilon \leq 0.5$, ϵ-greedy was able to solve no less than an average of 29.4 of 30 2008 cybersecurity problems, 22.8 of 30 2008 woodworking problems, and 16.8 of the 20 2011 barman problems, while standard GBFS solved an average of 25.3, 15.4, and 11.4 of these problems, respectively. In the few domains in which ϵ-greedy node selection decreased the coverage, the effect was minimal unless ϵ was high. For example, for all values of $0.05 \leq \epsilon \leq 0.2$, the coverage decreased by more than one problem in a single domain — 2011 parking domain — and for any value of ϵ in that range, the average coverage never decreased in that domain by more than 1.8 problems.

To ensure this behaviour is not specific to the use of the FF heuristic with deferred heuristic evaluation, ϵ-greedy node selection was also tested with standard

heuristic evaluation and two other heuristics. The two heuristics used are the context enhanced additive heuristic (CEA) [41] and the casual graph (CG) heuristic [38]. The results of this study — along with the totals when using the FF heuristic and deferred heuristic evaluation – are shown in the first four rows of Table 4.2. The table indicates that when using these other configurations, we see similar behaviour to that seen with the FF heuristic and standard heuristic evaluation. Note that in the table, dH implies that the heuristic H was used with deferred heuristic evaluation.

These results indicate that while these popular automated planning heuristics offer effective guidance for standard GBFS in many cases, there is significant value in adding variation through knowledge-free random exploration.

4.3.2 Adding ϵ-Greedy Node Selection to `LAMA-2011`

Even in a fully enhanced planner like `LAMA-2011`, adding random exploration can still be beneficial. This was tested by adding ϵ-greedy node selection to this planner so that each of the planner's four queues used this technique on its own. This means that on every iteration, `LAMA-2011` uses its standard policy for selecting which queue to take a node from (based on boosting), and then the node in that queue with the lowest heuristic value is selected with probability $(1 - \epsilon)$, while a node is selected uniformly at random from amongst those in that queue with probability ϵ. For example, if the next node expanded is to be selected from one of the preferred operator queues, the search will expand the most promising preferred successor with probability $1 - \epsilon$, and a random preferred successor with probability ϵ.

When experimenting with a version of `LAMA-2011` that uses random operator ordering, the variation added through ϵ-greedy node selection again helps to improve coverage. This is shown in the last row of Table 4.2. Once again, the coverage improvements also hold over a wide range of values for ϵ, and it is only with a very high ϵ value that this technique hurts performance.

Experiments with `LAMA-2011` have also indicated that this planner can have a highly variable performance depending on the operator ordering used, much like the variability seen with different operator orderings in `LAMA-2008` as shown in Section 3.3.2. As such, we experimented with different operator orderings to en-

	Operator Ordering			
ϵ	Standard	ROO	Pref First	Pref First ROO
0.0	680.0	676.0	713.0	677.7
0.05	706.4	704.8	723.0	706.7
0.1	704.4	706.5	720.8	706.0
0.2	703.8	705.9	720.3	703.9
0.3	702.9	705.1	716.4	704.2

Table 4.3: The average coverage of `LAMA-2011` when ϵ-greedy node selection is added to it.

sure that ϵ-greedy node selection was not only overcoming harmful bias introduced through the use of a bad operator ordering. These experiments are shown in Table 4.3. The first row, which shows the performance without ϵ-greedy node selection, clearly demonstrates this variability in the standard version of `LAMA-2011`. The operator orderings tested are the standard ordering as given by the Fast Downward successor generator (Standard), random operator ordering (ROO), **preferred operators first** (Pref First), and random operator ordering with preferred operators first (Pref First ROO). Preferred operators first means that the preferred operators are put at the front of the generated successor list before these nodes are added to the open list. "Pref First ROO" refers to the random shuffling of the operators such that in the resulting list of children, the nodes corresponding to the preferred operators are at the front of the list, though not necessarily in the order they were generated by the successor generator. In the version submitted to IPC 2011, `LAMA-2011` uses the "Pref First" operator ordering.

The table shows that in all cases, this technique is able to improve upon the average performance of the standard version of `LAMA-2011` regardless of the operator ordering in use. In particular, it shows that even if a good operator ordering is used, this technique will not undo the positive impact of the ordering and can still yield further performance gains.

4.4 Heuristic Perturbation

In this section, we will introduce a second simple technique for introducing random exploration into GBFS. This technique, which is called **heuristic perturbation**, will show that different ways of introducing random exploration can induce very different behaviour. We will then exploit the fact that this technique leads to different behaviour by pairing it with ϵ-greedy node selection in a portfolio.

Unlike ϵ-greedy node selection which changes the node selection policy used by GBFS, heuristic perturbation changes the heuristic function. A GBFS which uses this technique will avoid overly trusting the heuristic by adding noise to it. In particular, if H is the given heuristic function and a is a non-negative integer parameter, the heuristic function being used to guide the search is given by the following:

$$H_{\mathrm{HP}}(n) = H(n) + r(n) \ ,$$

where $r(n)$ is a random integer from the range $[0, a]$ that is assigned to n when n is first generated. This means that $r(n)$ is set once, and never changes for the remainder of the search. The resulting search is then an instance of nrBFS$^{H_{\mathrm{HP}}}$. For the sake of convenience when considering heuristic perturbation, we will use the following terms: $H(n)$ as the **non-noisy** heuristic value of n, $r(n)$ will be called the **noise value** of n, and the value of the parameter a will be called the **noise level** of this technique. Notice that if the noise level is set to 0, the resulting heuristic is equivalent to the non-noisy heuristic, and so the search will be identical to a standard GBFS that uses H to guide the search.

Heuristic perturbation changes the order in which nodes are considered for expansion by changing how promising nodes appear to be. For example, suppose that a node n has a low non-noisy heuristic value. If n is assigned a high noise value, it will appear to be much less promising than it would have otherwise. In such cases, a node with a higher heuristic value which was assigned a low noise value may appear to be more promising and will be expanded first. However, the search induced when using heuristic perturbation is still biased towards nodes with low heuristic values. To see this, suppose that the noise level in use is 4, and that at some time during

the search the lowest non-noisy heuristic value of any of the nodes in the open list is 10. This means that the next node selected must have a non-noisy heuristic value of 14 or lower. Since the expected noise value for any node is 2, and the variance is the same for all nodes, the nodes with the highest probability of being selected are also still those with a non-noisy value of 10.

However, notice that since the noise value of a node is set when it is first generated, heuristic perturbation may greatly delay the expansion of nodes that the heuristic function correctly identifies as being near a goal node. For example, suppose that the heuristic function in use is accurate in a given domain. By adding noise to the heuristic, the search may be pushed in the wrong direction if the nodes along a solution path are randomly assigned a high noise value. The search may then be forced to exhaustively search large portions of the state-space until the algorithm gets back on track. As such, there is risk to employing this technique as the randomness may block the search from progressing in the right direction. However, if the heuristic function has many inaccuracies, the exploration induced by heuristic perturbation will cause it to explore areas of the state-space it would not have considered otherwise, which can be beneficial.

In contrast to heuristic perturbation, ϵ-greedy node selection will not significantly delay expanding nodes in the open list with the lowest heuristic values. While the exploratory node expansions made by this algorithm can push it into different areas of the state-space, the fact that ϵ-greedy node selection still selects greedily according to the heuristic with probability $(1 - \epsilon)$ means that those nodes with a low heuristic value will be expanded upon the next greedy node selection, unless the random exploration happens upon other nodes with a lower heuristic value. ϵ-greedy node selection can therefore be seen as being of lower risk than heuristic perturbation. This difference will be seen experimentally in the next section, as unlike ϵ-greedy node selection which generally leads to modest gains in many domains without much degradation of performance in others, the high-risk approach of heuristic perturbation will lead to large gains in some domains and large drops in others.

4.4.1 Adding Heuristic Perturbation to a Simple Planner

Experiments were run with heuristic perturbation on the same test set and under the same conditions as ϵ-greedy node selection. In all experiments below, this technique was employed using deferred heuristic evaluation. When these techniques are used together, the heuristic function being used is given by the following:

$$H_{\mathrm{HP}}(n) = H(parent(n)) + r(n) \ .$$

The average coverage on a per domain basis for different noise levels is shown in Table 4.4. The column labelled as using a noise level of 0 is equivalent to using GBFS without heuristic perturbation. The last row of the table shows that for noise levels of 16, 64, and 256, a GBFS enhanced using heuristic perturbation is able to solve more problems than the number solved without heuristic perturbation. In contrast, GBFS with heuristic perturbation at a noise level of 1 has worse coverage overall when compared to not using any noise, while noise levels of 2 and 4 have similar coverage to that seen without any noise.

Unlike ϵ-greedy node selection, which shows smaller improvements on many domains without large drops in performance unless ϵ is high, heuristic perturbation often leads to substantial changes in coverage in several domains. For example, when using this technique with a noise level of 256, the algorithm solves 18.0 of the 20 2011 barman problems, 18.2 of the 20 2011 visitall problems, and all 20 of the 20 2011 woodworking problems. In contrast, a standard GBFS that does not use heuristic perturbation only solves an average of 11.4, 3.8, and 2.6 of these problems, respectively. In other domains, such as 2011 parcprinter, 2011 parking, and 2006 openstacks, the opposite behaviour emerges. For example, heuristic perturbation at a noise level of 256 solves an average of 10.5 of the 30 2006 openstacks problems, 0.6 of the 20 2011 parcprinter problems, and 0.2 of the 20 parking problems, while a standard GBFS without heuristic perturbation solves 30, 11.6, and 14.4 in these domains, respectively.

Domain Name (# of Problems)	Noise Level 0	1	2	4	16	64	256
2006 openstacks (30)	30.0	*7.4*	*7.6*	*7.3*	*7.9*	*9.0*	*10.5*
2006 pathways (30)	9.2	9.1	9.7	*7.4*	*4.7*	*4.8*	*5.1*
2006 rovers (40)	22.3	22.9	23.2	**23.8**	**24.3**	**29.0**	**32.2**
2006 storage (30)	20.1	19.3	20.4	**21.8**	**22.2**	**23.8**	**21.6**
2006 tankage (50)	21.4	22.3	**22.4**	**22.4**	**26.0**	**33.9**	**38.1**
2006 tpp (30)	21.4	22.0	**23.3**	**24.6**	**30.0**	**30.0**	**30.0**
2006 trucks (30)	16.1	*15.1*	15.4	15.5	16.5	**19.0**	**18.8**
2006 Totals (240)	140.5	118.1	122.0	122.8	131.6	149.5	156.3
2008 cybersecurity (30)	25.3	**26.8**	**26.8**	**28.7**	**29.1**	**29.4**	*17.9*
2008 elevators (30)	30.0	30.0	30.0	30.0	30.0	30.0	30.0
2008 openstacks (30)	30.0	30.0	30.0	30.0	30.0	30.0	30.0
2008 parcprinter (30)	25.7	24.9	*24.4*	*21.8*	*20.9*	*18.4*	*14.5*
2008 pegsol (30)	30.0	30.0	30.0	30.0	30.0	29.7	29.1
2008 scanalyzer (30)	27.6	27.7	27.4	28.1	**29.6**	27.3	*24.1*
2008 sokoban (30)	29.0	29.0	29.0	29.0	28.4	28.4	*27.1*
2008 transport (30)	17.2	17.6	17.9	16.9	**19.4**	**21.1**	**24.3**
2008 woodworking (30)	15.4	15.7	16.0	**16.5**	**21.9**	**30.0**	**30.0**
2008 Totals (270)	230.2	231.7	231.5	231.0	239.3	244.3	227
2011 barman (20)	11.4	**15.7**	**18.9**	**19.7**	**19.9**	**20.0**	**18.0**
2011 elevators (20)	13.3	13.9	**15.2**	**15.1**	**17.2**	**20.0**	**20.0**
2011 floortile (20)	4.2	4.4	4.4	4.7	4.6	5.0	**6.6**
2011 nomystery (20)	8.4	*6.2*	*5.8*	*5.7*	*5.5*	*5.6*	*4.9*
2011 openstacks (20)	18.5	**20.0**	**20.0**	**20.0**	**20.0**	**19.9**	**19.7**
2011 parcprinter (20)	11.6	10.9	11.1	*9.7*	*7.4*	*4.1*	*0.6*
2011 parking (20)	14.4	16.9	16.9	16.5	*7.2*	*2.4*	*0.2*
2011 pegsol (20)	20.0	20.0	20.0	20.0	20.0	19.9	19.4
2011 scanalyzer (20)	17.8	17.9	17.7	17.8	**19.4**	17.6	*14.4*
2011 sokoban (20)	19.0	19.0	19.0	19.0	18.7	18.4	*16.9*
2011 tidybot (20)	11.2	**13.1**	**14.2**	**13.8**	**16.6**	**16.4**	11.8
2011 transport (20)	0.0	0.0	0.0	0.1	0.6	**2.6**	**4.7**
2011 visitall (20)	3.8	4.6	**5.4**	**6.3**	**8.7**	**12.5**	**18.2**
2011 woodworking (20)	2.6	2.2	2.5	2.9	**6.8**	**19.0**	**20.0**
2011 Totals	156.2	164.8	171.1	171.3	172.6	183.4	175.4
All Domains Totals (790)	526.9	514.6	524.6	525.1	543.5	577.2	558.7

Table 4.4: The average coverage when using heuristic perturbation with the FF heuristic. Entries in bold and blue (italics and red) denote that heuristic perturbation at the corresponding noise level solved an average of at least one more (fewer) problem than standard GBFS on that domain.

	Operator Ordering			
Noise Level	Standard	ROO	Pref First	Pref First ROO
0	680.0	676.0	713.0	677.7
1	681.9	681.2	701.6	680.0
2	678.4	678.4	701.8	676.6
4	673.6	673.7	692.4	675.4
16	676.2	676.0	684.1	676.0

Table 4.5: Adding heuristic perturbation to the first iteration of LAMA-2011.

4.4.2 Adding Heuristic Perturbation to LAMA-2011

We now consider the performance of the heuristic perturbation when added to a state-of-the-art planner like LAMA-2011. In this experiment, each of this planner's four queues is set to use this technique independently. This means that the noise value of a node may be different in the different queues.

The performance of this technique when it is added to LAMA-2011 is shown in Table 4.5 for different operator orderings. This technique was also tested when using random operator ordering with a noise level of 64 and 256. The average number of problems solved with these noise levels were 665.7 and 632.6 respectively.

As the table shows, heuristic perturbation does not improve the coverage of LAMA-2011 on the entire test set. This is because the gains made in some domains are cancelled out by the losses in others. For example, when using random operator ordering and a noise level of 256, the planner solves an average of 20 out of the 20 2011 woodworking problems but only 1 of the 20 2011 parking domain, while a standard GBFS without heuristic perturbation solves an average of 12.8 and 14.4 of the problems in these domains, respectively.

4.4.3 Combining ϵ-Greedy Node Selection and Heuristic Perturbation in a Portfolio

When considering Tables 4.1 and 4.4, it is clear that there are both similarities and differences in the domains in which ϵ-greedy node selection and heuristic perturbation improve planner coverage. For example, both lead to improvements in the 2006 tankage, 2011 barman, 2008 woodworking, and 2011 woodworking domains,

though heuristic perturbation typically yields larger coverage gains. In contrast, ϵ-greedy node selection leads to modest improvements in the 2011 floortile, 2011 nomystery, and 2011 parcprinter domains, while heuristic perturbation does not help in these tasks. Furthermore, heuristic perturbation improves planner coverage in the 2006 tpp and 2008 transport domains while ϵ-greedy node selection does not.

			Heuristic Perturbation				ϵ-Greedy Node Selection			
	Alone	GBFS	NL= 4	NL= 16	NL= 64	NL= 256	$\epsilon = 0.1$	$\epsilon = 0.2$	$\epsilon = 0.3$	$\epsilon = 0.5$
GBFS	526.9	523.1(3)	561.2	588.9	**628.2**	625.0	573.7	580.2	581.6	577.8
NL= 4	525.1	561.2	543.7(6)	563.1	603.9	**618.0**	593.6	598.4	599.9	598.1
NL= 16	543.5	588.9	563.1	566.4(7)	592.4	608.7	607.2	611.1	**611.5**	609.6
NL= 64	577.2	628.2	603.9	592.4	589.5(3)	597.7	**636.2**	636.1	635.5	633.3
NL= 256	558.7	625.0	618.0	608.7	597.7	562.5(2)	643.6	**645.2**	644.2	641.2
$\epsilon = 0.1$	581.4	573.7	593.6	607.2	636.2	**643.6**	582.4(2)	588.1	589.1	586.7
$\epsilon = 0.2$	585.1	580.2	598.4	611.1	636.1	**645.2**	588.1	587.6(2)	589.5	587.5
$\epsilon = 0.3$	584.7	581.6	599.9	611.5	635.5	**644.2**	589.1	589.5	587.3(2)	587.1
$\epsilon = 0.5$	574.5	577.8	598.1	609.6	633.3	**641.2**	586.7	587.5	587.1	580.4(2)

Table 4.6: The expected performance of portfolios constructed by pairing the planning technique shown in the row and the column. When the row and column planner is the same, the table shows the best performance of any portfolio of size 2 to 10 (the number in the best portfolio is shown in parentheses) when restarting multiple times with the same planner.

The variance in the strengths of these techniques suggests that they can be combined effectively in a portfolio, and Table 4.6 shows that this is indeed the case. Every entry in this table shows the expected coverage of a different portfolio, calculated using the data summarized in Tables 4.1 and 4.4.[4] The planners used in each portfolio are given by the planners listed in the corresponding row and column. For example, the entry in the row labelled "GBFS" and the column labelled "Noise Level 16" shows that a portfolio containing one instance of standard GBFS and one instance of GBFS with heuristic perturbation at a noise level of 16 is expected to solve an average of 588.9 of the 790 problems in our test set. Where the configurations differ between the row and column planner, the portfolio contains exactly two planners, each of which is run for 900 seconds. The rows marked NL$= a$ refer to GBFS using heuristic perturbation at a noise level of a, while $\epsilon = j$ refers to ϵ-greedy node selection with $\epsilon = j$. The "Alone" column shows the performance of the corresponding technique when it is used alone for 1800 seconds and not in a portfolio. The "Alone" column data is taken directly from the "All Domains Totals" rows in Tables 4.1 and 4.4.

The entry in bold highlights the best of the portfolios containing the planner in the corresponding row. For example, in the row labelled "GBFS", the bold entry lies in the noise level of 64 column because the portfolio containing both standard GBFS and GBFS at a noise level of 64 had the highest expected coverage of all portfolios containing a standard GBFS instance.

When the column and row refer to the same planner, we show the performance of a portfolio which contains multiple instances of the same technique which only differ in their random seed. For these entries — which lie along the diagonal of the table – we considered every portfolio containing anywhere from 2 to 10 instances such that all instances are given an equal amount of time, with the best coverage seen by any of these portfolio sizes being shown. The number of planner instances used in the best such portfolio is shown in parentheses.

Notice that none of the portfolios containing only multiple instances of the same

[4]Recall that the technique used for calculating the expected coverage of a portfolio was described in Section 3.2.2.

planning technique appear in bold, as in all tested cases it is better to mix-and-match techniques. In particular, the best portfolios appear to be constructed by combining a low-risk technique (such as standard GBFS, GBFS at a low noise level, or ϵ-greedy node selection), with a high-risk technique (such as GBFS with a high noise level). For example, the best portfolios containing standard GBFS or GBFS at the low noise level of 4, are achieved when these are each combined with GBFS instances with noise levels of 64 and 256, respectively. Similarly, the high noise level GBFS instances pair best with ϵ-greedy instances, while the ϵ-greedy instances are best paired with GBFS instances at a high noise level. In particular, the best expected coverage is achieved by the portfolio that contains one instance of GBFS at a noise level of 256 and an instance of GBFS using ϵ-greedy node selection with $\epsilon = 0.2$. This portfolio solves 22.5% more problems than standard GBFS.

In contrast, combining only multiple low-risk approaches or only high-risk approaches is much less effective. This can be seen when two different ϵ-greedy approaches are used in a portfolio, or when combining ϵ-greedy with standard GBFS. Similarly, using a portfolio that only contains GBFS instances with high noise levels also leads to only minor coverage improvements.

The table also shows that in almost all cases, using multiple instances of the same technique in a portfolio outperforms using a single instance alone. This is consistent with the results seen with `LAMA-2008` in the previous chapter. The only outlier is GBFS, which does not improve when using a portfolio. The reason for this is mostly due to several 2011 domains that were introduced into the competition to be problematic for delete relaxation-based heuristics like FF. For example, in the 2011 barman domain, the FF heuristic has very large plateaus that the search is forced into before GBFS begins to make progress towards a goal state. These plateaus are searched regardless of the operator ordering. Due to their size, the runtime of GBFS even in the cases that it solves these problems is often larger than 900 seconds. The result is that by only allotting each instance 900 seconds in a portfolio, the coverage will decrease.

We also considered using portfolios that contain two instances of `LAMA-2011`, one using ϵ-greedy node selection and one using heuristic perturbation. In this

planner, there are fewer domains in which random exploration leads to performance improvements, and in those domains in which random exploration increases the coverage, these two approaches often lead to similar performance. As such, the coverage does not improve much over using ϵ-greedy node selection alone. For example, when using random operator ordering, `LAMA-2011` enhanced with ϵ-greedy node selection has an expected coverage of 706.5 problems when $\epsilon = 0.1$, while the portfolio containing one instance of `LAMA-2011` with $\epsilon = 0.1$ and one instance with a noise level of 16 has an expected coverage of 711.2.

4.5 Comparing Knowledge-Based to Knowledge-Free Enhancements

We have shown above that adding random exploration to GBFS introduces valuable variation into the search that can improve performance. Given that even random variation is valuable, this raises the question of whether the structure exploited by existing planning enhancements is actually adding important guidance or if it is merely causing the search to ignore the heuristic from time-to-time. To answer this question, we will isolate the impact of the structure being used in a given enhancement by replacing the resulting knowledge with randomness. This study will provide further confirmation of the importance of preferred operators and multi-heuristic best-first search.

4.5.1 Evaluating the Variation Added by Preferred Operators

Recall that the knowledge being exploited by the preferred operator enhancement is given by the helpful actions suggested by the heuristic, and that the variation introduced by using these operators is the result of biasing the search so as to expand preferred successors more often. This is done by alternatively selecting the node with the lowest heuristic from two queues: one containing all of the nodes in the open list and one containing only those nodes in the open list that were achieved with a helpful action. Yet the search would most likely vary if the second queue was populated by any proper subset of the open list. Therefore, we can evaluate the

Heuristic	No Prefs	Prefs	Random Prefs	Avoid Prefs
FF	526.9	606.8	554.1	531.6
CEA	491.9	583.6	534.6	486.8

Table 4.7: Comparing the use of preferred operators to the prioritization of randomly selected operators.

effectiveness of the helpful action knowledge by populating the preferred operator queue with randomly selected nodes instead of those suggested by the heuristic.

In this experiment, we ensured that the number of random successors of a given node that are identified as preferred operators is equal to the actual number of helpful actions suggested by the heuristic. We use random operator ordering to avoid the inherent bias introduced through the use of a static operator ordering with first-in first-out tie-breaking, and we did not use boosting for the sake of simplicity. The results are shown in Table 4.7, in which the results are shown for two different heuristics: the FF heuristic, and the CEA heuristic [41]. The columns in the table are labelled as follows. "No Prefs" refers to the use of a GBFS that does not use preferred operators. "Prefs" refers to the use of the actual helpful actions suggested by the heuristic as the preferred operators. "Random Prefs" refers to the use of randomly selected nodes as the preferred successors. The final column, "Avoid Prefs," populates the preferred operator queue with randomly selected nodes as the preferred successors, but those selected are restricted from including those identified as a preferred successor by the heuristic. Intuitively, a search using the "Avoid Prefs" approach is prioritizing nodes against the advice of the preferred operators.

As shown in the table, with either heuristic tested, the use of helpful actions as the preferred operators outperforms both the use of no preferred operators and the baselines which give preference to randomly selected operators. This suggests that the helpful action information is offering important guidance that goes beyond what random variation adds. However, the results in the "Random Prefs" column indicate that useful variance is introduced into the search even if the preferred operators are set randomly when using these heuristics. It is only when the preferred operators are set so as to bias the search against the advice of the helpful actions provided by

the heuristic functions — advice which is clearly informative — that the use of the second queue is not helpful.

4.5.2 Evaluating the Variation Added by Multi-Heuristic Best-First Search

Let us now consider the variation added by multi-heuristic best-first search. Recall that when using this heuristic, the search alternates between using each of the given heuristics when selecting a node from the open list. The extra knowledge used in the case of this enhancement is that given by the second heuristic. As with preferred operators, we will evaluate the importance of the knowledge being used by replacing that knowledge with randomness. In the case of multi-heuristic best-first search, this means that we will still use a second heuristic, but it will be a purely random heuristic. For the experiments below, this was done by defining the second heuristic so that the heuristic value of a node was given by a random integer in the range from 0 to 100.

The coverage of the planner using this random heuristic is shown in Table 4.8 in the row labelled "FF & Random" both when using boosted preferred operators and when not using preferred operators. The table shows that the variation added by the random heuristic leads to substantially better coverage than the single-heuristic baseline planner. We include the results over different operator orderings since this attribute did affect the relative ordering of the planners tested. In particular, when using random operator ordering, the use of a random heuristic led to slightly better coverage than the use of the knowledge-based LM heuristic, though the opposite is true with the other orderings. The use of the random heuristic as a secondary heuristic also outperforms a multi-heuristic best-first search that uses the CEA heuristic as a second heuristic, regardless of the operator ordering and whether or not preferred operators are used.

Despite the similarity in the total coverage results, domain-by-domain analysis showed that using the knowledge-based heuristic is resulting in variation that is quite different than random exploration. For example, consider the results when using standard operator ordering and no preferred operators. Just as with ϵ-greedy

	No Prefs		Boosted Prefs	
Heuristics	ROO	Standard	ROO	Pref First
FF	526.9	528.0	657.5	675.0
CEA	491.9	491.0	617.3	626.0
FF & Random	586.0	584.6	686.8	698.0
CEA & Random	542.6	537.5	650.9	661.3
FF & CEA	540.8	545.0	646.5	666.0
FF & LM	587.4	604.0	676.0	713.0

Table 4.8: Knowledge-based and knowledge-free multi-heuristic BFS.

node selection, the random exploration added when using the random heuristic increased coverage in the 2008 cybersecurity and 2008 woodworking domains, from 20 out of 30 and 15 out of 30 respectively when using a single heuristic, to averages of 30.0 and 25.4 when using the random heuristic. In contrast, the use of the knowledge-based LM heuristic as a secondary heuristic actually hurts coverage in these domains, as the resulting planner solves only 12 and 19 problems, respectively. However, the LM heuristic does add important guidance in the 2008 transport, 2011 parking, and the 2011 visitall domains. In these domains, the single heuristic planner solved 34 of the 70 total problems, while adding the random heuristic improved coverage to 45.7 which was still not as much as the 67 solved when using the LM heuristic. Similar results are seen with the CEA heuristic.

The table also shows that the use of a random heuristic was also an effective way to increase coverage when used alongside the CEA heuristic. For example, when using random operator ordering and boosted preferred operators, CEA solved an average of 617.3 problems, while the addition of a random heuristic increased coverage to an average of 650.9. The use of the random heuristic even compares well to a multi-heuristic best-first search that uses both the CEA and FF heuristics when using random operator ordering and boosted preferred operators, as such a system solves an average of 646.5 problems.

4.6 Chapter Summary

In this chapter, we have studied the impact of adding random exploration to GBFS. While several existing techniques have included a random component, they have typically used it in a combination with other approaches. As such, it has never been clear what the impact of random exploration alone was having on performance. To better measure the impact of random exploration, we have introduced ϵ-greedy node selection. This technique was shown to improve the performance of both simple planners and a state-of-the art planner in `LAMA-2011`. As such, our study confirms that adding random exploration is an option that system designers should consider when building a GBFS-based planner.

We also introduced a second technique, called heuristic perturbation, for introducing random exploration. This technique was shown to be of higher risk than ϵ-greedy node selection, as it would greatly improve the performance in some problems while it would hurt in others. We then showed that this higher variance can be exploited by combining this technique with ϵ-greedy node selection in a portfolio.

Given that GBFS can be improved by random exploration and existing planning enhancements, we argued that this suggests that the impact of the knowledge being used by such enhancements needs to be isolated to ensure it is not merely adding exploration which can be achieved in simpler ways. To do so, we propose that such enhancements should be compared to random baselines that are equivalent to the original enhancement, just with the knowledge replaced by randomness. We performed such a comparison between appropriate randomized baselines and two existing enhancements. Our results indicate that the knowledge used by preferred operators is essential to the success of this technique, while the use of a secondary heuristic in a multi-heuristic best-first search is offering important guidance in certain domains while not varying the search effectively in others.

Chapter 5

Heuristic Search Algorithms with Alternative Solution Quality Requirements

One of the choices a system designer is faced with when developing a heuristic search based system is that of deciding what quality of solutions is needed. Once a solution quality requirement is decided upon, it is then necessary to find an algorithm that is guaranteed to satisfy it. In this chapter, we identify what sets of algorithm options are available for a given solution quality requirement, even if these requirements are not of the commonly used form of linear suboptimality bounds. In particular, we will show how four existing frameworks of algorithms can be modified to apply to other bounds than those they were initially intended for. We then demonstrate that the generalizations of each of these frameworks effectively tradeoff suboptimality for runtime in the types of domains they are best suited for, even when they are used to satisfy additive bounds.

5.1 Introduction

As described previously, it is often not feasible due to practical runtime and memory constraints to require that the solutions found be optimal. In such situations, it is necessary to allow for suboptimal solutions in exchange for a less resource-intensive search. However, it may not be desirable to accept any solution, as there may be requirements on just how suboptimal a solution may get before it is con-

sidered unacceptably suboptimal. As an example of such a requirement, consider the common linear suboptimality bound. When given this bound, a solution is only considered acceptable if the cost C satisfies the inequality $C \leq w \cdot C^*$ for some given parameter $w \geq 1$, where w is set before any problem solving begins.[1]

Notice that requiring an algorithm to satisfy this bound for some particular w involves setting a maximum of w on the solution suboptimality, where suboptimality is measured by C/C^*. Since the development of rWA* [71] as the first algorithm guaranteed to satisfy a linear bound, this suboptimality measure has been by far the most commonly used measure. For example, ever since 2008, the International Planning Competition satisficing track scoring function has been based on the idea that the relative suboptimality of two plans is given by their ratio [9, 39]. The popularity of this suboptimality measure can also be seen in the fact that the majority of algorithms with suboptimality guarantees that have been developed since rWA* have been designed to satisfy linear suboptimality bounds. Such algorithms include Optimistic Search [88], EES [90], and rA$^*_w$ [69]. However, there also exists other ways to measure suboptimality and other types of suboptimality bounds. For example, one alternative to C/C^* is to measure suboptimality by $C - C^*$. A bound corresponding to this measure would require that the cost C of any solution returned must satisfy the inequality $C \leq C^* + \gamma$ for some $\gamma \geq 0$.

In this chapter, we consider such alternative suboptimality measures and bounds to evaluate the generality of existing methods and to give more choice into how suboptimality guarantees can be specified. If a system designer wishes to have guarantees (or to use suboptimality measures) that do not correspond to a linear suboptimality bound, it was not previously clear as to what algorithms or design choices were available to them. This was the case despite the fact that researchers in other fields, such as those who investigate and develop approximation algorithms [104], routinely consider alternative types of solution quality requirements.

Moreover, even though almost all previous on bounded heuristic search algorithms has concentrated on linear suboptimality bounds, such bounds are not nec-

[1] Recall that we are using the term "linear suboptimality bound" to refer to what is more commonly referred to as ϵ-admissibility and rA$^*_w$ to refer to rA$^*_\epsilon$ to avoid confusion with the ϵ in ϵ-greedy node selection.

essarily suitable in all cases. To see this, we present two such examples. In the first, consider a problem in which plan cost refers to the amount of money needed to execute the plan. The value (or **utility**) of money is known to be a non-linear function (*ie.* $10 is more valuable to someone with no money than it is to a billionaire). In particular, assume that for some individual the utility of a plan with cost C is given by $K - \log_2(2 + C)$ for some constant $K > 0$ and the desired guarantee on suboptimality is that the utility of any plan found will be no more than 10% worse than is optimally possible. While any solution found by a rWA* instance parameterized with $w = 1.1$ is guaranteed to be at most 10% more costly than the optimal solution, clearly this does not actually correspond with the desired requirement on utility. Moreover, it is unclear how to parameterize rWA* correctly to satisfy the given requirement without prior knowledge of C^*.

In the second example we consider, let us look again at the scoring function used in the International Planning Competition. The score given to a planner p on a task Γ is 0 if p is unable to solve Γ in the time allotted, and C^*/C where C is the cost of the solution found by p. If the cost of the optimal solution for Γ is unknown, then the score is given by C^{best}/C where C^{best} is the cost of the best solution known for Γ. The score given to a planner p over a task set is then given by the sum of the scores that planner p was awarded on each individual problem in the task set.

Another way to interpret this scoring function is as $1/M_w(C, C^*)$ where $M_w(x, y)$ is a measure of the relative suboptimality of solutions with cost x and y given by $M_w(x, y) = x/y$. This interpretation also indicates how other scoring functions can be constructed that correspond to any other suboptimality measure M. To do so, simply use the scoring function $1/M(C, C^*)$ and ensure that M is defined such that the optimal solution has a measure of 1. For example, this is possible for suboptimality measures such as $M(x, y) = 1 + x - y$ and $M(x, y) = \log_y x$, which correspond to using an absolute measure of suboptimality and a logarithmic measure of suboptimality, respectively.

Each of these measures represent a different perspective on what it means for a solution to be suboptimal, and how a planner should be penalized for not finding a solution relative to finding a suboptimal one. To see this, consider running three

	Task			
Planner	Γ_0	Γ_1	Γ_2	Γ_3
p_0	100	100	None	None
p_1	200	200	100	None
p_2	300	300	300	100

Table 5.1: Example of planner performance on a test set. Each entry corresponds to the cost of the solution found, with "None" indicating that the respective planner was unable to solve the task.

planners p_0, p_1, and p_2 on a task set containing Γ_0, Γ_1, Γ_2, and Γ_3, such that the cost of the solutions are found by the planners is given in Table 5.1. If all the tasks have an optimal solution cost of 100, then by the standard IPC scoring function, p_2 will get a score of $100/300 = 1/3$ on tasks Γ_0, Γ_1, and Γ_2, and $100/100 = 1$ on task Γ_3. The total score of planner p_2 will therefore be 2. By similar calculations, it can also be shown that p_0 and p_1 would get a score of 2 on this test set.

The fact that these three planners have achieved an equal score means that the current scoring function views these planners as having performed equally well on this test set. Whether or not the reader believes this to be true — or alternatively, as to which planner should be considered "the best" — is a matter of personal preference and reflective of a different notion of how suboptimality should be measured or how the inability to find a solution should be penalized. Suboptimality measures other than $M_w(x, y) = x/y$ can often allow for such preferences to be better represented. For example, when using the suboptimality measure of $M(x, y) = 1+x-y$, planner p_1 would be awarded a score of $1/(1 + 200 - 100) \approx 0.01$ on tasks Γ_0 and Γ_1, and 1 on task Γ_2 for a total score of 1.02. A similar calculation would show that planners p_0 and p_2 would be awarded a total score of 2 and 1.01, respectively. When using the measure $M(x, y) = \log_y x$, planner p_1 would be awarded a score of $1/\log_{100} 200 \approx 0.87$ on tasks Γ_0 and Γ_1, and 1 on task Γ_2 for a total score of 2.74. A similar calculation would show that planners p_0 and p_2 would be awarded a total score of 2 and 3.4, respectively. The suboptimality measure $M(x, y) = 1 + x - y$ would therefore correspond to a preference for planners that usually find high quality solutions at the expense of coverage, while $M(x, y) = \log_y x$ would correspond

to the opposite.

This is not to say that there is anything wrong with the current IPC scoring function. The purpose of the above example is merely to point out that the scoring function is based in a particular way of measuring suboptimality that may not capture everyone's preferences. In this way, it motivates the need to allow for other types of suboptimality bounds based on other ways of measuring suboptimality.

Allowing more choice in the type of bounds that can be defined then raises the following question: what algorithms or algorithm enhancements can be used such that they are guaranteed to satisfy a given suboptimality bound? In this chapter we show that four different classes of existing algorithms can be modified in this end. These classes are anytime algorithms, best-first search algorithms, iterative deepening algorithms, and focal list based algorithms. As each of these algorithm paradigms is best-suited for different types of problems, by extending them all to be able to handle arbitrary bounding constraints, we are allowing a system designer to not only specify a desired form of bounding, but to also select the best search framework for their particular domain.

5.1.1 Contributions

The main contributions of this chapter are as follows. First, we introduce a functional notion of a suboptimality bound to allow for the definition of alternative bounding paradigms. We then develop a theoretical framework which identifies how existing algorithms can be modified to satisfy alternative bounds. The existing algorithm classes that we consider include anytime algorithms and iterative deepening. We will also consider best-first search and focal list based algorithms that use the full re-expansion policy (*ie.* rBFS and rFOCAL). In doing so, we increase our understanding of what the existing algorithm frameworks can be used to do, and we more clearly identify what choices are available to a system designer when selecting an algorithm that must satisfy a given bound. Finally, we demonstrate that the theoretical framework leads to practical algorithms that can effectively trade-off guaranteed solution quality for improved runtime when considered for additive bounds.

An early version of this work in this chapter was first published as a two-page abstract [94], with a complete version appearing later [95]. Shahab Jabbari Arfaee, Jordan Thayer, and Roni Stern were also involved in this work. I was the lead on this project as I initially developed the theoretical foundations of this work, though the other authors helped to shape the notation used. I also implemented and ran the experiments on planning domains. Designing the remaining experiments was a collaborative process, though the other authors were responsible for implementing and running them.

5.2 Background and Related Work

In this section, we will define a functional way to describe a solution quality bound, and then describe related work on the use of non-linear suboptimality bounds.

5.2.1 Bounding Functions

Recall that the linear suboptimality bound requires that the cost C of any solution returned must satisfy the inequality $C \leq w \cdot C^*$ for a given $w \geq 1$. We generalize this idea by allowing for an acceptable level of suboptimality to be defined using a function, $B : \mathbb{R} \rightarrow \mathbb{R}$. This **bounding function** is used to define the set of acceptable solutions as those with cost C for which $C \leq B(C^*)$. This yields the following definition:

Definition 5.2.1. *For a given bounding function B, an algorithm will be said to **satisfy** B on a given task Γ, if any solution for Γ returned by that algorithm will have a cost C for which $C \leq B(C^*)$.*

As an example of how this definition applies, notice that the linear suboptimality bound corresponds to the bounding function $B_w(x) = w \cdot x$. This is true since an algorithm will satisfy B_w if and only if any solution it finds satisfies the inequality $C \leq B_w(C^*)$ which is equivalent to $C \leq w \cdot C^*$. Similarly, an algorithm is optimal if and only if it satisfies the bounding function $B_{opt}(x) = x$.

Let us now consider some other possible bounding functions. We begin with **sub-linear** bounding functions. The first is the **additive bounding function** given

by $B_\gamma(x) = x + \gamma$ for some $\gamma \geq 0$. Unlike the linear suboptimality bound, this function does not allow for the difference between C and C^* to increase as C^* increases, as solutions found must satisfy the inequality $C \leq C^* + \gamma$ for a constant γ. There are also other sub-linear bounds that allow the difference between C and C^* to increase, but at a slower rate than the linear suboptimality bound. Examples of such bounding functions include $B_{\log_a}(x) = x + \log_a(x)$ where $a > 1$, and $B_\surd(x) = x + \sqrt{x}$. These bounding functions correspond to the requirements that $C \leq C^* + \log_a(C^*)$ and $C \leq C^* + \sqrt{C^*}$, respectively.

Alternatively, we can allow the difference between C and C^* to grow faster than the linear suboptimality bound. Examples of such bounding functions include $B_p(x) = x^p$ for $p \geq 1$ and $B_a(x) = a^x$ for $a > 1$. These bounding functions correspond to the requirements that $C \leq (C^*)^p$ and $C \leq a^{C^*}$, respectively.

Notice that for all $x \geq 0$, $B_\gamma(x) \geq x$, $B_\surd(x) \geq x$, and $B_w(x) \geq x$. The remaining functions can also be adjusted appropriately — for example, by defining $B_{\log_a}$ as $B_{\log_a}(x) = x + \log_a(\max(x, a))$ — such that the other bounding functions introduced above are defined such that $\forall x \geq 0, B(x) \geq x$. When this criteria does not hold, there may be cases in which the bounding function requires better than optimal solution quality. For example, this may be the case when using the **bound-cost search** paradigm introduced by Stern, Puzis, and Felner [83]. This paradigm corresponds to the use of bounding function $B_K(x) \leq K$ for some constant $K \geq 0$. The intuition behind this form of bound is that K is a budget that cannot be exceeded. To allow for such bounds requires that the definition of a bounding function be modified so that an algorithm satisfying B is not only guaranteed to only return solutions that have a cost of C where $C \leq B(C)$ when such a solution exists, but that the algorithm can provably show this is not possible when such a solution does not exist. However, we will not consider this case in this chapter for the sake of simplicity. As such, we will focus on bounding functions for which $\forall x \geq 0, B(x) \geq x$.

It should also be noted that for any bounding function B for which $\forall x \geq 0, B(x) \geq x$, B will be trivially satisfied by any optimal algorithm. However, selecting an optimal algorithm for satisfying B where $B \neq B_{opt}$ defeats the purpose of defining an acceptable level of suboptimality, which was to avoid the resource-

intensive search typically required for finding optimal solutions. The goal is therefore not only to find algorithms that satisfy B, but to find algorithms which satisfy B and can be expected to be faster than algorithms satisfying tighter bounds. The approach we take is similar to that of rWA*: by allowing the algorithm to become greedier. We do so in several well-known heuristic search frameworks later in this chapter.

5.2.2 Alternative Bounding in the Literature

Dechter and Pearl [12] previously provided bounds on the cost of a solution found when using a best-first search guided by an arbitrary evaluation function. Further generalizations were considered by Farreny [17] who increased the types of path cost and evaluation functions that could be used. These bounds are given in terms of the highest evaluation of any node on an optimal path. This means that the objective of the work of Dechter and Pearl, as well as by Farreny, was the inverse of ours. Whereas they seek to improve our understanding of how a given evaluation function will affect the quality of solutions found, our goal is to better understand what algorithms are applicable for a given bounding paradigm.

Some of the algorithms proposed below for satisfying a given bounding function will be based on the idea of building the focal list in different ways than is done by rA$^*_w$. Farreny also considers different ways of constructing focal lists than rA$^*_w$ [17] and offers bounds on the solutions found by such algorithms. However, just as with best-first search, Farreny's bounds are given in terms of the value of $g+h$ for nodes on the focal list at the time a solution node is expanded. As such, that component of Farreny's work can also be seen as having a different objective as ours, just as it was in the case of best-first search.

An alternative form of focal list construction has also previously been used to construct algorithms for bounded-cost search problems by Thayer *et al.* [92]. As mentioned above, this paradigm corresponds to the use of the bounding function $B_K(x) = K$ for some constant K [83]. Below we will generalize the focal list construction technique used by Thayer *et al.* and show that it can be used to satisfy many other bounding functions. However, note that unlike in the case of focal list

based search, the techniques we propose for best-first search and iterative deepening algorithms will not be immediately applicable for B_K. This is because the bounding function B_K associated with bounded-cost search does not satisfy the condition that $\forall x \geq 0, B(x) \geq x$.

The best-first search algorithms proposed for satisfying a given bounding function will often use non-traditional evaluation functions. BUGSY is another best-first search algorithm which also uses an evaluation function different from that used by A*, WA*, and GBFS [4]. When using this algorithm, the objective is to find the solution with the best **utility**, where the utility of a solution is a linear combination of the solution cost and the time needed to find that solution. This differs from the objective of the algorithms considered in this chapter of finding solutions that satisfy a given solution quality requirement. Using the theory in this chapter to identify requirements being satisfied by BUGSY, or introducing runtime requirements into the algorithms constructed in this chapter both remain as future work.

When evaluating our framework below, we will use the additive bounding function as an example of an alternative bounding paradigm. Additive bounding has previously been considered by Harris [34] who showed that any solutions found by rBFS^{g+H} will cost no more than $C^* + \gamma$ if the heuristic H is such that for any node n, $H(n) \leq h^*(n) + \gamma$. This condition will be generalized below. More recently, it was shown that a particular termination criterion induced an additive bound when using bi-directional search [73]. We consider generalizing bi-directional search to satisfy other bounding functions as future work.

5.3 Bounding with Anytime Algorithms

In this section, we will demonstrate that we can use **anytime algorithms**, regardless of the suboptimality paradigm they were initially developed for, to satisfy any monotonically non-decreasing bounding function. We then argue that this approach is problematic and that we instead need algorithms tailored specifically for the given bounding function.

Anytime algorithms are designed for the situation in which a good solution

is needed by some unknown deadline. The approach these algorithms take is to quickly find an initial solution, and then to continue to search for better solutions until the deadline is reached. As a result, these algorithms typically find a sequence of improving solutions, the last of which is returned when the deadline is reached.

As an example of such an algorithm, consider Anytime Weighted A* (AWA*) [33], which runs rWA* to find a first solution, and then continues to search the space as rWA* would. This means that when the goal node is selected for expansion, the solution path is stored, and the search moves on to the selection of the next node for expansion according to the WA* evaluation function. At any time after the first solution is found, the search also discards any node n for which $g(n)+h(n) \geq C^{inc}$ where h is admissible and C^{inc} is the cost of the **incumbent solution** (*ie.* the best solution found thus far). Discarding such nodes is possible since if $g(n) + h(n) \geq C^{inc}$, then no path through n can improve on the incumbent path unless the path to n is first improved, and so expanding n with its current path will not further the efforts to find a better solution. This is also why the children of the goal node n_g which has been expanded will not be added to the open list, since for any child c of n_g, the cost of the new path to c will have a g-cost of $g(n_g) + \kappa(n_g, c)$, which must be at least C. This process will continue, with the algorithm storing each new best solution path it finds, and then moving on to select nodes from the open list as rWA* would. At any time, the algorithm can be interrupted at which point it will simply return the best solution it has found thus far.

To use an anytime algorithm to satisfy a monotonically non-decreasing bounding function B, we will require that during the execution of the algorithm, there is a lower bound L on the optimal cost that is available. Instead of using a time limit as a deadline, this lower bound will be used. To see how, let C^{inc} be the cost of the incumbent solution. Since B is monotonically non-decreasing, this means that if $C^{inc} \leq B(L)$ then $C^{inc} \leq B(C^*)$. Therefore, if we run the anytime algorithm and terminate once $C^{inc} \leq B(L)$, we are guaranteed that the last incumbent solution will satisfy the bounding function.

As an example of this approach, let us again consider AWA*. In this case, one

lower bound that can be used after t expansions is as follows:

$$L = \min(C^{inc}, \min_{n \in \text{OPEN}_t} g_t(n) + h(n)) \ ,$$

where h is an admissible heuristic function and $C^{inc} = \infty$ if no solution has been found. This is because at any time during the search, if the incumbent solution is not optimal, then for some optimal solution path $P = [n_0, ..., n_k]$, there will be a node $n_i \in P$ in the open list such that $g(n_i) = g^*(n_i)$ [33].[2] This lower bound holds because $g(n_i) + h(n_i) \leq g^*(n_i) + h^*(n_i) \leq C^*$ since h is admissible, and so the minimum of $g + h$ over all nodes in the open list must be less than $g(n_i) + h(n_i)$. After each node expansion, we can therefore use the value of this lower bound to check if $C^{inc} \leq B(L)$ is true, in which case we can terminate. Note that the special case of this statement of $C^{inc} \leq L$ (or equivalently $B = B_{opt}$) is also a termination condition in the original definition of AWA* since there is no need to continue searching once the optimal solution has provably been found.

This technique was first considered by Thayer and Ruml [88] who ran AWA* to satisfy the linear suboptimality bound $C \leq B_w(C^*)$ where $B_w(x) = w \cdot x$. To do so, they parameterize AWA* so that it can be expected to find an initial solution quickly, even if the cost of that solution does not satisfy $C^{inc} \leq B_w(C^*)$. The algorithm, which they refer to as Bounded AWA*, then continues its search until the incumbent solution does satisfy this condition. Above, we have extended this technique to satisfy any monotonically non-decreasing B, even when the initial algorithm was designed for some other bounding function B'. This means that we can use AWA*— or any anytime algorithm initially designed to satisfy linear suboptimality bounds — to handle other bounding functions provided that the algorithm maintains a lower-bound for C^*.

Notice that if we are satisfying the additive bounding function $B_\gamma(x) = x + \gamma$ and the value of C^* is known, then we can ensure that the first solution found by AWA* will satisfy the bound by setting the weight as $1 + \gamma/C^*$. However, this approach cannot be used without prior knowledge about C^*. If the value of C^* is

[2]This holds by the same arguments as given for Corollary 2.2.9 and Theorem 2.2.10 since AWA* is essentially an rOCL with a slightly different termination condition. The same is also true of related algorithms like Optimistic Search [88].

not known, then it is not clear as to how the weight parameter should be set. This is problematic since this parameter can have a large impact on performance. If AWA* is set to be too greedy, it may find a first solution quickly, but then take too long to improve its incumbent solution to the point of satisfying the bounding function. If AWA* is not set to be greedy enough, it may take too long to find an initial solution. Such behaviour will be shown experimentally below. This suggests the need for algorithms specifically targeted towards the given bounding function. Constructing such algorithms is therefore the goal of the coming sections.

5.4 Bounding with rBFS Algorithms

In this section, we will show that if a best-first search algorithm that uses the full re-expansion policy employs an appropriate evaluation function, then that algorithm can be guaranteed to satisfy a given bounding function B.

Recall that rBFS$^\Phi$ iteratively selects the most promising node from the open list according to the evaluation function Φ. In the case of rWA*, $\Phi = f_w$ where $f_w(n) = g(n) + w \cdot h(n)$, $w \geq 1$, and h is admissible. In using this evaluation function, rWA* is guaranteed to find solutions that cost at most $w \cdot C^*$. In practice, rWA* often solves problems faster than A* as a result of f_w which emphasizes the relative importance of h relative to g, and therefore allows rWA* to search more greedily on h than does A*.

Also notice that $f_w(n) = g(n) + B_w(h(n))$ where B_w is the linear suboptimality bounding function $B_w(x) = w \cdot x$. This suggests the use of the evaluation function $\Phi_B(n) = g(n) + B(h(n))$ for satisfying a given bounding function B since it similarly puts additional emphasis on h. Below, we will show that this approach will suffice for a large family of bounding functions. To do so, consider the following theorem which provides sufficient conditions on Φ for satisfying a given bounding function B.

Theorem 5.4.1. *Given a bounding function B, rBFS$^\Phi$ will satisfy B if the following conditions hold:*

1. *$\exists$ an optimal solution path $P_{opt} \in \Pi^*_{Goal}$ such that $\forall n \in P_{opt}$ and $\forall t \geq 0$,*

$\Phi_t(n) \leq B(g_t(n) + h^*(n))$.

2. $\forall n_g \in V_{Goal}$ *and* $\forall t \geq 0$, $g_t(n_g) \leq \Phi_t(n_g)$.

Proof. Assume that a goal node n_g is the t-th node expanded by rBFS$^\Phi$. Let P_{opt} be an optimal solution path for which condition 1 holds. In addition, assume condition 2 also holds. Now consider the state of the search immediately after the $t-1$-st expansion. Recall that Corollary 2.2.9 in Section 2.2.4 stated that at any time prior to an rOCL algorithm expanding a goal node there will be a node n on the open list from any optimal path such that $g_t(n) = g^*(n)$. Since rBFS$^\Phi$ is an rOCL algorithm, this means that after $t-1$ expansions, there will be some node $n_{opt} \in P_{opt}$ such that $g_t(n_{opt}) = g^*(n_{opt})$. Since n_g is the t-th node selected for expansion, this implies that $\Phi_t(n_{opt}) \geq \Phi_t(n_g)$ by the definition of rBFS. By our assumptions, this means that $B(g_t(n_{opt}) + h^*(n_{opt})) \geq \Phi(n_g)$. As $g_t(n_{opt}) = g^*(n_{opt})$ and $g^*(n_{opt}) + h^*(n_{opt}) = C^*$ since n_{opt} is on P_{opt}, this also means that $B(C^*) \geq \Phi(n_g)$. When this is combined with the fact that $g_t(n_g) \leq \Phi_t(n_g)$ by assumption 2, it yields $B(C^*) \geq g_t(n_g)$.

Now recall that when n_g is selected for expansion, the path to n_g implicitly maintained with the parent will be extracted and returned. As shown by Theorem 2.2.5 in Section 2.2.4, the cost of this path will be at most $g_t(n_g)$. Therefore, the solution returned will have a cost of at most $B(C^*)$ and so rBFS$^\Phi$ satisfies B. □

We can now use this theorem to develop evaluation functions for rBFS such that the resulting algorithm will satisfy B. This is done in the following sections.

5.4.1 WA*-Style Heuristic Weighting

To concretely demonstrate the implications of Theorem 5.4.1, consider how it applies to B_w and the WA* evaluation function $f_w = g + w \cdot h$ when h is admissible. For B_w, the first condition on Φ simplifies to $\Phi_t(n) \leq B_w(g_t(n) + h^*(n))$. For any

node n on an optimal path, f_w then satisfies this condition since

$$\begin{aligned} f_w(n) =& g(n) + B_w(h(n)) && (5.1)\\ \leq& B_w(g(n) + h(n)) && (5.2)\\ \leq& B_w(g(n) + h^*(n)) && (5.3) \end{aligned}$$

where line 5.2 holds because $w \cdot (g(n)+h(n)) \geq g(n)+w \cdot h(n)$. Since we also have that $g(n_g) = f_w(n_g)$ for any goal n_g by the fact that $h(n) = 0$ for any $n \in V_{Goal}$, rWA* (*ie.* wBFS^{f_w}) satisfies B_w by Theorem 5.4.1. The following corollary then extends this result to a large class of bounding functions:

Corollary 5.4.2. *Given bounding function B, if $\Phi(n) = g(n) + B(h(n))$ where h is admissible, then rBFS$^\Phi$ will satisfy B if the following holds for B:*

$$\forall x \geq 0, y \geq 0, B(x + y) \geq B(x) + y \ .$$

This corollary holds because $\Phi_t(n_g) = g_t(n_g)$ for any goal node n_g, and because the exact same derivation as was performed for f_w applies for Φ. This means that for any bounding function B such that for all x, y, $B(x + y) \geq B(x) + y$, we immediately have an algorithm, specifically rBFS$^\Phi$, for satisfying it. This condition can be viewed as requiring that the maximum difference between C and C^* (*ie.* $C - C^*$) allowed by B cannot be smaller for problems with a large optimal cost than for those with a small optimal cost. Alternatively, notice that if B is differentiable, this means that $B'(x) \geq 1$ for all $x \geq 0$.

Notice that the A* evaluation function $g+h$ is an instance of this form of evaluation function for the bounding function $B_{opt}(x) = x$. As the conditions of corollary 5.4.2 hold for B_{opt}, this corollary can therefore be used to prove that any solution returned by A* will be optimal.

5.4.2 Other Types of Evaluation Functions for rBFS

Since $B(x) \geq x$ for all x, the evaluation function $\Phi(n) = g(n) + B(h(n))$ will intuitively increase the emphasis on $h(n)$ in a similar fashion as the WA* evaluation function. One exception is the case of the additive bounding function $B_\gamma(x) =$

$x + \gamma$ (and the resulting evaluation function denoted as Φ^{B_γ}), for which nodes will be ordered in the open list in the same way as in A*. To see this, consider any two nodes n_1 and n_2 that are in the open list after t node expansions such that $\Phi_t^{B_\gamma}(n_1) \geq \Phi_t^{B_\gamma}(n_2)$. This inequality allows for the following derivation:

$$\Phi_t^{B_\gamma}(n_1) \geq \Phi_t^{B_\gamma}(n_2) \tag{5.4}$$

$$g_t(n_1) + h(n_1) + \gamma \geq g_t(n_2) + h(n_2) + \gamma \tag{5.5}$$

$$g_t(n_1) + h(n_1) \geq g_t(n_2) + h(n_2) \tag{5.6}$$

This last line shows that the A* evaluation function $g + h$ would also order these two nodes in the exact same way. Since this will be true for any pair of nodes in the open list, this means that the search performed using the evaluation function Φ^{B_γ} will be identical to A*.

This is not ideal behaviour since the whole motivation behind allowing for sub-optimality is so that the result is a less resource-intensive search than A*. As such, for this bounding function, we need to consider other types of evaluation functions. One such type is defined by the following corollary:

Corollary 5.4.3. *Given bounding function B, if $\Phi(n) = g(n) + D(n)$ where h is admissible and $0 \leq D(n) \leq B(h(n))$ for all n, then rBFS$^\Phi$ will satisfy B if the following holds for B:*

$$\forall x \geq 0, y \geq 0, B(x + y) \geq B(x) + y \ .$$

This corollary, which follows from an almost identical derivation as Corollary 5.4.2, introduces a new function D which must be bound by B. For example, consider the following possible D function:

$$D_\gamma(n) = \begin{cases} 0 & \text{if } n \text{ is a goal} \\ h(n) + \gamma & \text{otherwise} \end{cases}$$

When D_γ is used in a rBFS search in the evaluation function $\Phi(n) = g(n) + D_\gamma(n)$, the search is guaranteed to satisfy B_γ by Corollary 5.4.3. This search will be identical to A*, at least until the first time a goal node is expanded, for the same reason that the search is the same as A* when using the evaluation function

$\Phi(n) = g(n) + h(n) + \gamma$. As such, this function will also have minimal impact on the search. However, in Section 5.7.1 we will consider other D functions which will allows us to successfully trade-off speed for solution quality.

5.4.3 Inadmissibility Limiting

While Corollary 5.4.2 offers $\Phi(n) = g(n) + B(h(n))$ as a general way to use an admissible heuristic to construct an evaluation function to satisfy B, an alternative is to use an inadmissible heuristic with a **bounded amount of inadmissibility**. This is possible due to the following corollary:

Corollary 5.4.4. *Given bounding function B, if $\Phi(n) = g(n) + H(n)$ where $0 \leq H(n) \leq B(h^*(n))$ for all n, then rBFS$^\Phi$ will satisfy B if the following holds for B:*

$$\forall x \geq 0, y \geq 0, B(x + y) \geq B(x) + y \ .$$

This corollary, which follows from an almost identical proof as was given for Corollary 5.4.2, generalizes a theorem by Harris [34] which was specific to the additive bounding function.

Unfortunately, for many modern-day inadmissible heuristics such as the FF heuristic, there are no known bounds on heuristic inadmissibility. However, we can still employ an inadmissible heuristic H_{in} in conjunction with an admissible heuristic h, by forcing a limit on the difference between the two. This means that we will use the following evaluation function $\Phi(n) = g(n) + H(n)$ where $H(n) = \min(B(h(n)), H_{in}(n))$. The heuristic $H(n)$ will satisfy the inequality $H(n) \leq B(h^*(n))$ since the fact that h is admissible and B is monotonically increasing implies that $B(h(n)) \leq B(h^*(n))$. As such, the evaluation function given by $g + H$ satisfies the conditions of Corollary 5.4.4.

We refer to this approach as **inadmissible limiting** since it employs a perhaps arbitrarily inadmissible heuristic H_{in} by limiting how much of the potential inadmissibility of H_{in} can be used in the evaluation function. When the inadmissible heuristic H_{in} and the admissible heuristic h disagree by too much (*ie.* $H_{in}(n) > B(h(n))$) we cannot simply use $H_{in}(n)$ and still be guaranteed to satisfy

the bound. As such, we use the value of $B(h(n))$ since this penalizes the node appropriately (by increasing its evaluation) while still satisfying B_γ.

Notice, that Theorem 5.4.1 also says that the following evaluation could be used when both an inadmissible H_{in} and an admissible h is available:

$$\Phi(n) = \min(B(g(n) + h(n)), g(n) + H_{in}(n))$$

This approach is similar to an existing technique called **clamped adaptive** [91], but generalized so that it applies to arbitrary evaluation functions. Since we will evaluate the theory developed in the construction of algorithms with additive bounds and because $B_\gamma(g(n)+h(n)) = g(n)+B_\gamma(h(n)) = g(n)+h(n)+\gamma$ in the case of the additive bounding function B_γ, we will not further consider techniques of this form in this thesis.

5.5 Bounding with rFOCAL Algorithms

Let us now consider how focal list based algorithms can be used to satisfy an arbitrary bounding function B. These algorithms were described in Section 2.2.6, in which we introduced the term rFOCAL$^\beta$ where $\beta : \mathbb{R} \to \mathbb{R}$ and $\forall x \geq 0, \beta(x) \geq x$ to refer to a focal list based algorithm that uses the full re-expansion policy and defines its focal list was defined as follows:

$$\text{FOCAL} = \left\{ n | g_t(n) + h(n) \leq \beta \left(\min_{n' \in \text{OPEN}} g_t(n') + h(n') \right) \right\} \ ,$$

where h is an admissible heuristic. An rFOCAL$^\beta$ algorithm then uses some secondary policy for iteratively selecting nodes from this subset of the open list.

To understand how β influences the search, let us first look closer at the way the original focal list based algorithm, A$^*_w$, defines its focal list. This is done as follows:

$$\text{FOCAL} = \left\{ n | g_t(n) + h(n) \leq w \cdot \min_{n' \in \text{OPEN}} g_t(n') + h(n') \right\} \ ,$$

where h is an admissible heuristic. Recall that by this definition, the focal list contains all nodes with a value for $g + h$ (*ie.* the A* evaluation function f) that is no more than a factor of w greater than the node in the open list with the lowest

value of of $g + h$. From this list, any policy for iteratively selecting nodes from the focal list for expansion can be used while still satisfying the requirement that any solution found is no larger than $w \cdot C^*$ [15].

Now consider how the size of the focal list changes with different values of w. When $w = 1$, the focal list will only contain those nodes that have the minimum value of $g + h$ of all nodes in the open list. As such, the policy being used to select nodes from the focal list can be seen as simply determining how ties are broken between nodes with an equal value of $g + h$. The resulting search will therefore be identical to an A* search that uses that policy to break ties.

For larger values of w the focal list will contain a larger set of nodes with a variety of values of $g + h$, and thus will be allowed to explore them more greedily according to the focal list selection policy. For example, when $w \to \infty$, the focal list will consist of all nodes in the open list. In such a situation, if the policy being used selects from the focal list greedily according to some secondary heuristic H_2, as A$^*_w$ does, then the search will be equivalent to a GBFS using H_2.

A similar behaviour occurs in rFOCAL$^\beta$ for a particular monotonically non-decreasing bounding function β. Where $f_{\min}$ is the minimum value of $g_t + h$ of any node on the open list after t expansions, the larger the difference between $B(f_{\min})$ and $f_{\min}$, the larger the proportion of the open list that can be expected to be contained in the focal list.

Moreover, notice that A$^*_w$ is an instance of rFOCAL^{B_w} where B_w is the bounding function $B_w(x) = w \cdot x$. The following theorem shows that for any monotonically non-decreasing bounding function B, any rFOCAL^B algorithm will similarly satisfy B. This is shown by the following theorem:

Theorem 5.5.1. *Given any monotonically non-decreasing bounding function B, rFOCAL^B will satisfy B.*

Proof. Assume that rFOCAL^B has expanded a goal node n_g with the t-th node expansion, and consider the search immediately prior to the expansion of n_g. Let P_{opt} be an optimal solution path to the given task Γ. Now consider the state of the open list immediately after the $t - 1$-st expansion. Since rFOCAL is a rOCL

algorithm, we can appeal to Corollary 2.2.9 in Section 2.2.4 — as we did in the proof of the sufficient conditions on the evaluation function used by rBFS (Theorem 5.4.1) above — to guarantee that after $t-1$ node expansions of rFOCAL^B there will be some node $n_{opt} \in P_{opt}$ such that n_{opt} is in the open list and $g_{t-1}(n_{opt}) = g^*(n_{opt})$.

Now let us consider the t-th iteration, for which goal node n_g is selected for expansion. By the definition of rFOCAL algorithms, this requires that $n_g \in \text{FOCAL}_{t-1}$. Let n be the node in the open list after $t-1$ expansions such that n has the lowest value of $g+h$ of all nodes in the open list. Since n_g is in the focal list, this means that the first line of the following is true, which allows for the following derivation:

$$g_{t-1}(n_g) + h(n_g) \leq B(g_{t-1}(n) + h(n)) \tag{5.7}$$

$$g_{t-1}(n_g) \leq B(g_{t-1}(n) + h(n)) \tag{5.8}$$

$$g_{t-1}(n_g) \leq B(g_{t-1}(n_{opt}) + h(n_{opt})) \tag{5.9}$$

$$g_{t-1}(n_g) \leq B(g^*(n_{opt}) + h^*(n_{opt})) \tag{5.10}$$

$$g_{t-1}(n_g) \leq B(C^*) \tag{5.11}$$

Line 5.8 holds since n_g is a goal node and so $h(n_g) = 0$. Line 5.9 holds since n has the minimum value of $g+h$ of all nodes on the focal list, and B is monotonically non-decreasing. Line 5.10 then holds since $g_{t-1}(n_{opt}) = g^*(n_{opt})$ as shown above, h is admissible and B is monotonically non-decreasing. The last line holds since $n_{opt} \in P_{opt}$, P_{opt} is an optimal solution path, and by the definition of C^*.

As in the proof of Theorem 5.4.1, the path that is returned will have a cost of at most $g_t(n_g)$ by Theorem 2.2.5 in Section 2.2.4. Therefore, the solution returned will have a cost of at most $B(C^*)$. □

Again, notice that this proof holds regardless of the policy used to select nodes from the focal list for expansion. This will allow us to construct versions of both A$^*_w$ and more modern focal list based algorithms like EES [90] to satisfy any monotonically non-decreasing bounding function. We will do so in the additive setting below.

5.5.1 WA*-like Weighting as a Focal List Based Algorithm

Ebendt and Drechsler [15] demonstrated that rWA* is an instance of rFOCAL^{B_w} where $B_w(x) = w \cdot x$ in which the policy used to select nodes from the focal list chooses the node from the focal list with the lowest value of $g+w\cdot h$. In this section, this result is generalized so that it applies a larger class of bounding functions. This is done by the following theorem:

Theorem 5.5.2. *Any rBFS$^{\Phi}$ algorithm is a rFOCAL^{B} algorithm for bounding function B, provided that for any node in the open list after t node expansions, the following condition holds for Φ:*

$$g_t(n) + h(n) \leq \Phi_t(n) \leq B(g_t(n) + h(n)) \ ,$$

where h is the admissible heuristic being used to construct the focal list.

Proof. Let a_{bfs} be a given rBFS$^{\Phi}$ algorithm such that the above condition holds on Φ. Since a rFOCAL^{B} algorithm always selects a node from its focal list for expansion, to show this theorem holds we need to the show that the node selected for expansion on any iteration will always be in the focal list if one were constructed. In this end, suppose there have been t node expansions and let n be the $t+1$-st node selected for expansion. Let $n_{\min}$ be the node in the open list with the lowest value of $g + h$. We can now perform the following derivation:

$$\begin{aligned} g_t(n) + h(n) &\leq \Phi(n) && (5.12)\\ &\leq \Phi(n_{\min}) && (5.13)\\ &\leq B(g_t(n_{\min}) + h(n_{\min})) && (5.14) \end{aligned}$$

The first line holds by our assumption on Φ, the middle line holds since n was selected for expansion instead $n_{\min}$, and the final line holds by our assumption on Φ. Since $n_{\min}$ is the node in the open list with the lowest value of $g + h$, the last line of this derivation also shows that n is necessarily in the focal list. Therefore, the statement is true. □

This proof is based on the proof given by Ebendt and Drechsler for the specific case of WA*. Notice that if for all x, y, $B(x + y) \geq B(x) + y$, then the evaluation

function $\Phi(n) = g(n) + B(h(n))$ satisfies the conditions of the theorem. Therefore, an rBFS$^{g+B(h)}$ algorithm is an instance of a rFOCAL^{B} where for all x, y, $B(x+y) \geq B(x) + y$.

5.6 Bounding with Iterative Deepening Algorithms

While the best-first and focal list based search algorithms detailed above will generally successfully exchange solution quality for speed improvements, they often suffer in practice due to the high memory overhead required for maintaining the open and closed lists. To combat this issue, Korf developed **depth-first iterative deepening search** [50]. In this section, we will define this algorithm framework and then show how it too can be used to satisfy a large family of bounding functions.

5.6.1 The Depth-First Iterative Deepening Framework

Like best-first search algorithms, depth-first iterative deepening algorithms require the use of an evaluation function Φ. As such, we will use DFID$^{\Phi}$ to denote an instance of depth-first iterative deepening search that uses Φ as its evaluation function. This framework is shown in Algorithm 11.

The execution of DFID$^{\Phi}$ consists of a series of threshold-limited depth-first searches, where the threshold is set at the beginning of each depth-first search iteration. For convenience, we will refer to the threshold used on iteration i as T_i. The definition of this algorithm means that during iteration i, the algorithm will visit every path $[n_0, ..., n_k]$ where $n_0 = n_{init}$ and $\Phi(n_j) \leq T_i$ for all $0 \leq j \leq k$, and every path $[n_0, ..., n_{k-1}, n_k]$ where $n_0 = n_{init}$, $\Phi(n_k) > T_i$, and $\Phi(n_j) \leq T_i$ for $0 \leq j < k$. This latter set of paths corresponds to those that induce the algorithm to backtrack. The minimum of $\Phi(n_k)$ over these paths in this set is then used as T_{i+1} if no solution is found during iteration i. This means that T_{i+1} is given by the minimum Φ-cost of all nodes that were generated but not expanded during iteration i. Notice that the initial threshold, denoted by T_0 since this iteration will be referred to as iteration 0, is set as $\Phi(n_{init})$.

As alluded to earlier, the main advantage that DFID$^{\Phi}$ has over rBFS$^{\Phi}$ is its low

DFID$^{\Phi}$(Initial node n_{init}):

```
1: T ← Φ(n_i)
2: loop
3:     T, P ← RecursiveDFID^Φ([n_init], T)
4:     if P ≠ [] then                          ▷ Test if solution path was returned
5:         return P
```

RecursiveDFID$^{\Phi}$(Path $[n_0, ..., n_k]$, threshold T):

```
1: if Φ(n_k) > T then
2:     return Φ(n_k), []
3: if n_k ∈ V_Goal then
4:     return T, [n_0, ..., n_k]
5: T_next ← ∞
6: L_k = {c | c ∈ succ(n_k)}                   ▷ Generate the children of n
7: for all c ∈ L_k do
8:     if H(c) = ∞ then                        ▷ Goal not reachable from c
9:         continue                                          ▷ Skip to next child
10:    T_recurse, P ← RecursiveDFID^Φ([n_0, ..., n_k, c], T)
11:    if P ≠ [] then                          ▷ Test if solution path was returned
12:        return T, P
13:    T_next ← min(T_next, T_recurse)
14: return T_next, []
```

Algorithm 11: Pseudocode for the Depth-First Iterative Deepening Algorithm.

memory overhead. This is because BFS$^{\Phi}$ does not maintain open and closed lists and only needs to maintain the current path being considered, along with some bookkeeping to keep track of which children of any node along this path have previously been explored. However, one of the purposes of the open and closed lists of rBFS$^{\Phi}$ is to detect when a node is visiting with a longer path than it reached with previously. Since the standard version of DFID$^{\Phi}$ does not maintain anything like these structures, it will often expand the same node many times even within the same iteration, with each expansion corresponding to a different path to that node. This increase in re-expansions represents the sacrifice that this algorithm makes in exchange for a low-memory overhead.

5.6.2 Existing DFID Algorithms: IDA* and WIDA*

Just as with BFS$^\Phi$, there are several known variants of DFID$^\Phi$ that only differ in the evaluation function they use. We detail two of these algorithms here.

IDA*

The first iterative deepening algorithm we consider is the IDA* algorithm defined by Korf [50]. Where $P = [n_0, ..., n_k]$ is the path found from $n_{init} = n_0$ to n_k, the evaluation function used by this algorithm is

$$\Phi(n_k) = g(n_k) + h(n_k) \ ,$$

where h is an admissible function and $g(n_k)$ the cost of path P. This means that IDA* is using the same evaluation function as A*. As a result, IDA* can be shown to have the same solution quality guarantee as A*: any solution found by IDA* is guaranteed to be optimal [50].

WIDA*

While IDA* uses the same evaluation function as A*, WIDA* uses the same evaluation function as WA* [52]. Specifically, where $P = [n_0, ..., n_k]$ is the path found from $n_{init} = n_0$ to n_k, the evaluation function used by WIDA* is

$$\Phi(n_k) = g(n_k) + w \cdot h(n_k)$$

where h is admissible and $w \geq 1$ is an algorithm parameter.

Just as IDA* had the same solution quality guarantee as A*, so too does WIDA* have the same solution quality guarantee as WA*. In particular, any solution found is guaranteed to be no more costly than $w \cdot C^*$. While this has been known for quite some time, the only proofs to appear in the literature that we are aware of are those given in my Master's thesis [93], a proof given by Hatem, Stern, and Ruml [37], and the proof given below that first appeared in the paper that is the basis of this chapter [95].

5.6.3 Using DFID to Satisfy a Given Bounding Function

As with best-first search above, we now identify a set of sufficient conditions on Φ that will guarantee that DFID$^{\Phi}$ will satisfy a given bounding function B. In particular, we will prove a theorem that is analogous to Theorem 5.4.1 for best-first search. The proof of that theorem started with a demonstration that if a goal node n_g is expanded, there will be a node n_{opt} on some optimal solution path P_{opt} that is in the open list such that $\Phi(n_{opt}) \geq \Phi(n_g)$. The following lemma offers a similar statement that guarantees that if n_g is expanded during a DFID$^{\Phi}$ search, there must be a node n_{opt} on P_{opt} such that $\Phi(n_{opt}) \geq \Phi(n_g)$.

Lemma 5.6.1. *Let $P \in \Pi_{Goal}$ be a solution path $P = [n_0, ..., n_k]$ where $n_0 = n_{init}$ and $n_k \in V_{Goal}$. If a DFID$^{\Phi}$ expands n_k when it reaches n_k along P, then for any optimal solution path $P_{opt} \in \Pi^*_{Goal}$, there is some node $n_{opt} \in P_{opt}$ such that $\Phi(n_i) \leq \Phi(n_{opt})$ for all i where $0 \leq i \leq k$.*

Proof. Let P_{opt} be any optimal solution path to the given task. Assume that a goal node n_k is expanded on iteration j when it reaches n_k along P. Since every node on P must have been expanded to find the path to n_k along P, this means that $\Phi(n_i) \leq T_j$ for any i where $0 \leq i \leq k$.

We must now consider two cases regarding j. In the first, $j = 0$, in which case $T_j = T_0 = \Phi(n_{init})$. Since $n_{init} \in P_{opt}$ and $\Phi(n_i) \leq T_j$ for any i where $0 \leq i \leq k$, the statement is clearly true.

Now suppose that $j > 0$. In this case, we will show by contradiction that there exists a node n_{opt} on P_{opt} such that $\Phi(n_{opt}) \geq T_j$. Since we have shown above that $\Phi(n_i) \leq T_j$ for any i where $0 \leq i \leq k$, the statement will then immediately follow.

We begin by assuming that for any $n \in P_{opt}$, $\Phi(n) < T_j$. Now notice that there must be at least one node $n \in P_{opt}$ such that $\Phi(n) > T_{j-1}$, as otherwise all nodes on P_{opt}, including the goal node, would have been found during iteration $j - 1$. If that was the case, then the algorithm would have terminated during that iteration. Assume that n_{opt} is the shallowest such node. Since $\Phi(n_{opt}) < T_j$ by our assumption, this means that $T_{j-1} < \Phi(n_{opt}) < T_j$. Since n_{opt} is the shallowest on P_{opt} for which $\Phi(n_{opt}) > T_{j-1}$, this means that all nodes shallower than n_{opt} on P

must have a Φ-cost of no more than T_{j-1}. This means that the parent of n_{opt} on P_{opt} must have been expanded during iteration $j-1$ and so n_{opt} must have been generated during iteration $j-1$. Notice that n_{opt} must have a parent on P_{opt} as otherwise $n_{opt} = n_0 = n_{init}$, and all thresholds must be as least as large as $\Phi(n_{init})$ since $T_i < T_{i+1}$ and $T_0 = \Phi(n_{init})$.

Now recall that T_j is selected as the minimum of $\Phi(n)$ of any node n that was generated but not expanded during iteration $j-1$. Since n_{opt} is one of these nodes, $T_j \leq \Phi(n_{opt})$ which contradicts the fact that $\Phi(n_{opt}) < T_j$. Therefore, there must be a node n on P_{opt} such that $\Phi(n) \geq T_j$ and so the statement is true when $j > 0$.

Having handled all cases, the lemma has been shown to be true. □

This lemma will now allow us to show that the same sufficient conditions on the evaluation function Φ that guaranteed that rBFS$^{\Phi}$ would satisfy a given bounding function B will also be sufficient for DFID$^{\Phi}$.

Theorem 5.6.2. *Given a bounding function B, DFID$^{\Phi}$ will satisfy B if the following conditions hold:*

1. $\exists$ *an optimal solution path* $P_{opt} \in \Pi^*_{Goal}$ *such that* $\forall n \in P_{opt}$, $\Phi(n) \leq B(g(n) + h^*(n))$.
2. $\forall n_g \in V_{Goal}$, $g(n_g) \leq \Phi(n_g)$.

Proof. Suppose that DFID$^{\Phi}$ expands node $n_k \in V_{Goal}$ on iteration j along the path $P = [n_0, ..., n_k]$ where $n_0 = n_{init}$. Let P_{opt} be an optimal solution path that satisfies condition 1 of the theorem. By Lemma 5.6.1, there will be a node $n_{opt} \in P_{opt}$ such that $\Phi(n_k) \leq \Phi(n)$.

The remainder of the proof then follows exactly as did the proof of sufficient conditions for Φ in the case of rBFS (Theorem 5.4.1) did once $\Phi(n_k) \leq \Phi(n_{opt})$ was guaranteed for some node $n_{opt} \in P_{opt}$ By assumption 1, this means that $\Phi(n_k) \leq B(g(n_{opt}) + h^*(n_{opt}))$ and so $\Phi(n_k) \leq B(g^*(n_{opt}) + h^*(n_{opt}))$ since the path from n_{init} to n_{opt} along P_{opt} is necessarily optimal. This means that $\Phi(n_k) \leq B(C^*)$ since P_{opt} is an optimal solution path, and so $g(n_k) \leq B(C^*)$ by condition 2. Since $g(n_k)$ is the cost of P which is returned, the statement then holds. □

Since Theorems 5.4.1 and 5.6.2 show that the same sufficient conditions hold on Φ for either rBFS$^{\Phi}$ or DFID$^{\Phi}$, corollaries 5.4.2 and 5.4.3 can easily be extended to also apply to DFID$^{\Phi}$. Similarly, the techniques like inadmissibility limiting that were shown to be used with rBFS can also be used with DFID. This means that in practice, when we want to satisfy a bounding function B, we simply need to find an evaluation function Φ that satisfies the properties in these theorems and then decide based on domain properties (such as state-space size, or the number of transpositions) on whether to use rBFS$^{\Phi}$ or DFID$^{\Phi}$.

5.7 Experimenting with Additive Bounds

In this section, we consider the additive bounding function $B_\gamma(x) = x + \gamma$ as a test case for demonstrating that the above theory can be used to construct effective algorithms for satisfying bounding paradigms aside from linear suboptimality bounds. Note that if the optimal solution cost to a given problem is known before search begins, it is possible to use WA* — or some other algorithm that was initially developed to satisfy linear suboptimality bounds — to satisfy a given additive bound γ. If the optimal solution cost is not known, then it will not be obvious as to how to parameterize these algorithms to guarantee that they satisfy B_γ. As such, we will only experiment with the use of such algorithms for satisfying a given additive bound γ when deployed as part of an anytime system like AWA*, as described above in Section 5.3. However, this issue of finding a proper parameter value will also be shown to impact anytime algorithms like AWA* when used for additive bounding as the effectiveness of these algorithms, relative to the use of an additive rBFS algorithm, will depend on how they are parameterized.

5.7.1 Evaluation Functions for Additive Bounds

As described in Section 5.4.2, the evaluation function $g + B(h)$ will be ineffective when used for satisfying $B_\gamma(x) = x + \gamma$ since the resulting search will be identical to A*. As an alternative, we can use inadmissible limiting if both an inadmissible heuristic H_{inad} and an admissible heuristic h is available. In the case of additive

bounding, this corresponds to the use of the following evaluation function:

$$IL(n) = g(n) + \min(h(n) + \gamma, H_{inad}(n)) \ .$$

We can also construct evaluation functions that only use an admissible heuristic function h. To do so, we consider evaluation functions of the type $\Phi(n) = g(n) + D(n)$ where $0 \leq D(n) \leq h(n) + \gamma$, which were shown to satisfy B_γ by Corollary 5.4.3. The guiding principle which we will use for constructing evaluation functions of this type will be to follow the example of the WA* evaluation function and further emphasize the role of the heuristic. This is the idea behind our next evaluation function, which penalizes nodes with a high heuristic value by adding a term that is linear in the heuristic. The key difference between the new function and the WA* function is that in order to satisfy B_γ, this penalty must be guaranteed to be no greater than γ. To this end, consider the following function:

$$\Phi'(n) = g(n) + h(n) + \frac{h(n)}{h_{max}} \cdot \gamma \ ,$$

where h_{max} is a constant such that for all n, $h(n) \leq h_{max}$. This condition on h_{max} guarantees that the corresponding D-function, given by $D(n) = h(n) + h(n) \cdot \gamma/h_{max}$ satisfies the required relation that $0 \leq D(n) \leq h(n) + \gamma$ for all n.

Also notice that this evaluation function is equivalent to WA* evaluation function $g + w \cdot h$ where $w = 1 + \gamma/h_{max}$. This means that if h_{max} is a loose upper bound and we were using best-first search, then our algorithm would be equivalent to a WA* search that is not using as high of a weight as it could. Using a tight upper bound on h for h_{max} is therefore crucial for achieving good performance.

Unfortunately, a tight upper bound on the heuristic value of any state may not be immediately available without extensive domain knowledge. Moreover, such an upper bound may not even be relevant for a given starting state, as it may over-estimate the heuristic values that will actually be seen during search. For example, suppose that only a small number of nodes in the state-space actually have a heuristic value corresponding to h_{max}. If these nodes are not encountered during the search, then the search can safely be made greedier by using a lower value for h_{max}.

This motivates the use of the following evaluation function:

$$\Phi_\gamma(n) = g(n) + h(n) + \frac{\min(h(n), h(n_{init}))}{h(n_{init})} \cdot \gamma$$

Intuitively, this function uses $h(n_{init})$ as a more instance relevant upper bound and penalizes nodes according to how much **heuristic progress** has been made, where progress is measured by $h(n)/h(n_{init})$.

Notice that if $h(n_{init})$ is the largest heuristic value seen during the search, then the '$\min$' can be removed from the Φ_γ evaluation function, which simplifies to

$$\Phi_\gamma(n) = g(n) + h(n) + \frac{h(n)}{h(n_{init})} \cdot \gamma$$

In this case, Φ_γ is now equivalent to the WA* evaluation function in which $w = 1 + \gamma/h(n_{init})$. In general, the '$\min$' cannot be omitted as it is necessary for the additive bound, making it the crucial difference between Φ_γ and the WA* evaluation function. However, omitting the '$\min$' gives us insight into what to expect from Φ_γ and the Φ' function defined above. Specifically, since Φ_γ corresponds to the WA* evaluation function when $w = (1 + \gamma/h(n_{init}))$ and Φ' corresponds to the WA* evaluation function when $w = (1 + \gamma/h_{max})$, we expect that when comparing the two, rBFS$^{\Phi_\gamma}$ will be greedier (and therefore usually faster), since $h(n_{init}) \leq h_{max}$.

We performed a simple experiment to verify this behaviour using the 15 puzzle domain. The test set considered was given by the 100 puzzles given by Korf in the original IDA* paper [50]. The heuristic used was the 7-8 additive pattern database heuristic, which consists of two additive pattern database heuristics [19]. For this test, we ran rBFS using both of the Φ_γ and Φ' evaluation function for different additive bounds. The bounds tested correspond to the different possible values of γ in the set 2, 4, 8, ..., and 256. The value of h_{max} used in Φ' was given by scanning each database to find the largest value therein, and taking the sum of these two values. The performance of rBFS$^{\Phi_\gamma}$ and rBFS$^{\Phi'}$ can be found in the first section of Table 5.2, which shows the average number of node expansions needed (in tens of nodes) to find a solution. As the table shows, both evaluation functions exhibit the desired trade-off of suboptimality for time as it expands fewer nodes as the suboptimality bound is loosened. However, rBFS$^{\Phi_\gamma}$ was found to result in a lower

Algorithm	Additive Bound / γ 0	2	4	8	16	32	64	128	256
rBFS$^{\Phi_\gamma}$	3,732	**2,460**	**1,175**	**403**	117	66	44	**33**	**30**
rBFS$^{\Phi'}$	3,732	2,565	1,527	815	245	104	51	45	34
AWA*, $w = 1.5$	**3,013**	2,850	2,180	606	117	99	99	99	99
AWA*, $w = 2.0$	4,614	4,553	4,222	2,330	343	53	53	53	53
AWA*, $w = 2.5$	5,248	5,215	4,979	3,185	678	**49**	**40**	40	40
OS, $w = 1.5$	3,179	2,784	2,072	717	**107**	99	99	99	99
OS, $w = 2.0$	3,483	3,279	2,739	2,350	386	53	53	53	53
OS, $w = 2.5$	3,557	3,358	3,068	2,214	919	52	**40**	40	40

Table 5.2: The average number of expanded nodes (in tens of nodes) by different additively bound algorithms in the 15 puzzle.

number of average nodes expanded than rBFS$^{\Phi'}$ in all values for γ considered larger than 0. Note that $\gamma = 0$ corresponds to an A* search when using either of these algorithms. As such, in all further experiments we will use Φ_γ instead of Φ'.

5.7.2 Using Anytime Algorithms for Additive Bounding

In this section, we will experiment with using anytime algorithms to satisfy a given additive bound, and show that there are issues with doing so. We considered two different anytime algorithms. The first is the AWA* algorithm which has been described previously. The second is **optimistic search** [88]. Like AWA*, this algorithm also uses rWA* to find an initial solution. Optimistic search then continues to iteratively expand nodes in the open list according to a policy that more proactively tries to prove that the incumbent solution already satisfies the given bound.

Where w is the weight used by this algorithm, the policy used to select nodes from the open list after the first solution is found is defined as follows. First, the algorithm identifies the node n_w on the open list with the lowest value f_w-cost where f_w is the WA* evaluation function $f_w = g + w \cdot h$. If $f_w(n_w)$ is less than the cost of the incumbent solution, then n_w is the next node selected for expansion. When nodes are expanded according to this criteria, the algorithm can be viewed as looking for better solutions. If $f_w(n_w)$ is not less than the cost of the incumbent

solution, then the algorithm expands the node n on the open list with the lowest value of $g + h$. Notice that n is the node corresponding to the lower bound. If every child c of n has a heuristic value larger $h(n) - \kappa(n, c)$, then this can cause the lower bound to increase. Therefore, when nodes are expanded according to this criteria, the algorithm can be viewed as looking to increase the lower bound to more quickly show that the incumbent solution already satisfies the bound.

Both AWA* and optimistic search (denoted by "OS") were tested on the set of 100 15 puzzle problems given by Korf using the aforementioned 7-8 additive pattern database heuristic. The performance of these algorithms is shown in Table 5.2. Both AWA* and optimistic search were tested with several different weights since it is not obvious how to parameterize it for a specific γ. For every bound, the algorithm that expanded the fewest nodes is marked in bold.

Like rBFS$^{F_\gamma}$, both of these anytime algorithms also improve their runtime as the bound is loosened. However, while rBFS$^{F_\gamma}$ is the best or nearly the best algorithm for all values of γ, the general trend when using the anytime approaches with a particular weight is that they work well over some range of γ values, but no single weight gets strong performance everywhere. As described previously, this is because the weight determines the initial greediness of these algorithms. If the weight is too low for a given task and a given bound, then the search will take too long to find the initial solution when the search could have been greedier. If the weight is too high, the search may be too greedy and the result can be that the first solution may be too far from optimal and it may take too long to improve upon that solution to find one satisfying the required bound.

The best-first search approach also has a second advantage over the anytime approaches: there is less per-node overhead when using best-first search. This is because the anytime approaches require two copies of the open list to be maintained. The first copy is sorted by the WA* evaluation function and is used to extract the node from the open list according to that measure as needed. The second copy is sorted by $g + h$ and is used to quickly calculate the value of the lower bound, or to extract the node with the lowest value of $g + h$ as needed by optimistic search. The extra overhead needed to maintain these lists properly often results in a slower

runtime than best-first search. For example, while bounded AWA* with a weight of 2.5 required 35% fewer node expansions than rBFS$^{F_\gamma}$, it actually took 20% more runtime. Similar behaviour was also seen in the few other instances of an anytime search outperforming rBFS$^{F_\gamma}$ in terms of average number of nodes expanded.

These experiments demonstrate that using an arbitrary anytime algorithm instead of an algorithm specific for a given bound can therefore see reduced performance due to two issues: it is unclear as to how to initially select a weight for a particular bound, and the computational expense of maintaining two open lists. As weight selection will be even more difficult in an automated planner (since less is known about the domains), we restrict our experiments with such tasks to the additive rBFS algorithms.

Best-First Search for Additive Bounding on PDDL Planning Tasks

We will now show that additive BFS algorithms can also be effective solving PDDL planning tasks. For these experiments, we used the `Fast Downward` framework [38] in which we implemented Φ_γ and inadmissibility limiting. For inadmissible limiting, the inadmissible heuristic used was the FF heuristic. The admissible heuristic used is the LM-Cut heuristic which is a strong admissible heuristic based on landmarks [40]. On each problem, planning approaches were given a 1800 seconds time limit and 4 GB memory limit when running on a machine with two 2.19 GHz AMD Opteron 248 processors. For the test suite, we used the 280 problems from the optimal track of IPC 2011 with action costs ignored. This means that all tasks are treated as unit cost tasks. This is because non-unit cost tasks are best handled by focal list based approaches [90], and so we wished to test these algorithms on the types of problems they are generally best suited for.

We begin by considering the coverage of rBFS$^{\Phi_\gamma}$ on these planning domains, which is shown in the first row of Table 5.3. As in the 15 puzzle, the table shows that this algorithm exhibits the desired behaviour: as suboptimality is allowed to increase, so too does the coverage. This is consistent with the behaviour of WA*, which also benefits from the additional greediness allowed with a looser bound.

The second row of Table 5.3 then shows the performance when the inadmissi-

	Additive Suboptimality Bound / γ									
	0	1	2	5	10	25	50	100	500	1000
rBFS$^{\Phi_\gamma}$	132	**145**	**148**	**163**	**175**	191	204	**216**	**218**	**218**
rBFS^{IL}	NA	142	146	157	172	199	199	199	199	199
P-rBFS^{IL}	NA	142	146	159	**175**	**213**	**214**	214	214	214

Table 5.3: The coverage of additive BFS algorithms in planning domains.

bility of the FF heuristic is limited using LM-Cut (IL is being used to refer to the inadmissibility limiting evaluation funcion). Again we see that as the amount of suboptimality is increased, so too does the coverage. However, the coverage generally lags behind that of BFS$^{F_\gamma}$. Our hypothesis was that this was caused by the overhead of calculating two heuristic values for each node since heuristic computation is expensive in automated planners. To combat this effect, we used a technique inspired by **pathmax** [63] to decrease the number of LM-cut heuristic computations that need to be performed. For this technique, notice that for any node n, $h^*(c) \geq h^*(n) - \kappa(n,c)$ where c is a child of n.Therefore, for any admissible h and inadmissible H_{in}, if $H_{in}(c) \leq h_a(n) - \kappa(n,n) + \gamma$, we can infer that $H_{in}(c) \leq h^*(n') + \gamma$ without calculating $h(n')$ by Corollary 5.4.3. We can similarly avoid calculating $h(c')$ for any successor c' of n for which $H_{in}(c') \leq h(c') - C(P_{n,c'})$ where $P_{n,c'}$ is the path found from n to c'.

For the implementation of this technique, the inadmissible FF heuristic value of a node c generated by the expansion of a node n is always calculated first. If $h(n) - \kappa(n,c) + \gamma$ is no smaller than the FF heuristic value of c, then the admissible heuristic value of c is not calculated, but is assumed to be $h(n) - \kappa(n,c)$. This value can then be used to determine if the LM-Cut heuristic value of any child of c must be calculated to determine if it limits the FF heuristic value of that node. Notice that this technique can also easily be extended to apply to any bounding function.

The coverage seen when this technique is added to rBFS^{IL} is shown in the third row of Table 5.3 (denoted as P-rBFS^{IL}). The pathmax-based technique clearly offers significant gains, and now inadmissibility limiting is equal or better than rBFS$^{F_\gamma}$ for $10 \leq \gamma < 100$. For low values of γ, P-BFS^{IL} suffers because the inad-

missible heuristic is limited too often. To illustrate this effect, consider what happens when the inadmissible heuristic is always limited. This means that $\min(h_n(c)+\gamma, H_{in}(n)) = h_a(n)+\gamma$ for all n. When this happens, the inadmissible limiting evaluation function degenerates to $g(n)+h_a(n)+\gamma$, which, as discussed in Section 5.4.2 will result in a search that is identical to A^*. As γ gets smaller, the frequency with which H_{in} is limited increases and so we approach the case in which it is always limited.

When γ gets large enough, the inadmissible heuristic will never be limited and the inadmissible limiting evaluation function degenerates to $g(n) + H_{in}(n)$. We can see this happening in Table 5.3, as P-rBFS^{IL} stops increasing in coverage at $\gamma = 50$. We confirmed this as the cause by using the FF heuristic without limiting, in which case BFS solves 218 problems. This small discrepancy in coverage is caused by the overhead of using LM-Cut.

Though P-BFS^{IL} has a lower coverage than BFS$^{\Phi_\gamma}$ for low and high values of γ, it still sits on the Pareto-optimal front when comparing these two algorithms, and it does show that we can use inadmissible heuristics in BFS and still have guaranteed bounds. We will see similar results below when evaluating additive depth-first iterative deepening algorithms.

5.7.3 Additive Bounding with Depth-First Iterative Deepening

Depth-first iterative deepening search is typically used on large combinatorial state-spaces in which the memory requirements of best-first search can limit its effectiveness. In this section, we will test the evaluation functions used previously in rBFS algorithms with depth-first iterative deepening and show that these functions also lead to effective algorithms for additive bounds when using this framework.

The first domain we tested DFID on was the 15 blocks world problem. In this domain, there are 15 wooden blocks on top of a table, each labelled with a different integer value from 1 to 15. These blocks can be stacked up, with only one block immediately above another, and actions correspond to picking up a block and putting it down either on top of another block or onto the table. Each task consists of some given starting configuration of the blocks with the goal in the tasks we consider be-

ing a single stack in which the blocks appear in ascending order of their label from bottom to top.

For the admissible heuristic value of a state, we will use a count of the number of blocks out of place. For the inadmissible heuristic, we will use the **bootstrap learning heuristic** [48]. This heuristic is built iteratively in an offline process. Given an initial heuristic H_0 and a set of planning tasks, the learning process first runs DFID$^{g+H_0}$ on each problem in the task set with a given time limit. In the case of the blocks world domain, this initial heuristic is given by the number of out of place blocks. The result will be a set of solved problems, and a solution cost for each of these problems. Using a set of simple but manually selected features, an artificial neural network is then trained for estimating the cost-to-go from any state to the nearest goal. In the case of the blocks world domain, the features are given by the values returned by several small pattern databases, the number of out of place blocks, and the number of stacks of blocks.

The neural network can then be used as a new heuristic, H_1. The next step is to run DFID$^{g+H_1}$ on the tasks that were not solved using H_0. This in turn will produce a new set of solved instances and corresponding solution costs that can be used to train another neural network to produce heuristic H_2. This process then continues until some time limit is hit or the number of new problems solved during an iteration is very low. The final heuristic produced is then the heuristic that is used to solve new problems. Note, the system also provides a way to generate additional training instances if the given task sets turn out to be too difficult for the initial heuristic. For a full description on how this heuristic is constructed, see the work of Jabbari Arfaee, Zilles, and Holte [48].

The results for the blocks world experiment are shown in Figure 5.1. The general trend for DFID$^{\Phi_\gamma}$ and DFID^{IL} is that they both improve their runtime as the bound increases, though DFID^{IL} stops improving when the inadmissible heuristic is no longer ever limited. This happens at $\gamma = 9$ in this domain due to the relatively high accuracy of the admissible heuristic. Because DFID$^{\Phi_\gamma}$ can continually become greedier on h, it is able to surpass the performance of DFID^{IL} for $\gamma \geq 20$.

We also tested these two DFID approaches in the 24 puzzle domain. In this

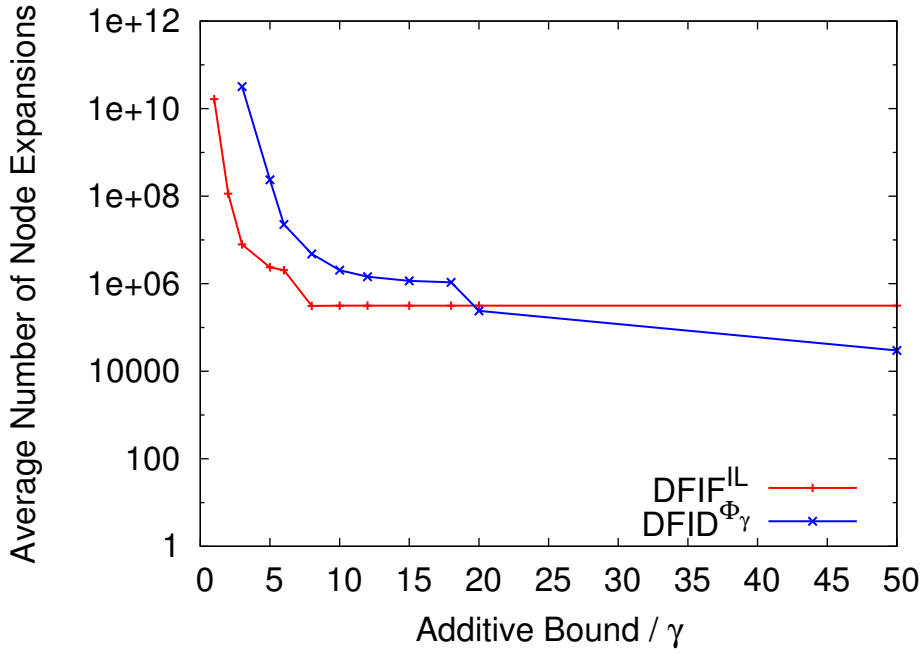


Figure 5.1: Additive DFID in the 15 blocks world domain.

domain, the admissible heuristic is given by the 6-6-6-6 additive pattern database heuristic, which consists of 4 additive patter databases [19]. The bootstrap learning heuristic was also used as the inadmissible heuristic when experimenting with inadmissible limiting. In this case, the initial heuristic is given by the Manhattan distance heuristic and the feature set is given by the Manhattan distance heuristic, 5 small pattern database heuristics, the number of out of place tiles, and the position of the blank.

The results in this domain are similar to those seen in the 15 blocks world domain, as both DFID^{IL} and DFID$^{\Phi_\gamma}$ improve their performance as the bound is loosened. As in the 15 blocks world domain, DFID^{IL} again shows better performance for an intermediate range of values for γ before it reaches a γ at which it stops improving, while DFID$^{\Phi_\gamma}$ continues to improve its performance as the bound is loosened. For example, at $\gamma = 18$ and $\gamma = 20$, DFID^{IL} requires 2.5 and 3.2 times fewer node generations on average, respectively. However, the performance of DFID^{IL} stops improving at $\gamma = 30$, and by $\gamma = 50$ DFID^{IL} is 1.96 times slower than DFID$^{\Phi_\gamma}$. This means that just as with BFS, the theory developed above led to the successful construction of additive iterative deepening algorithms.

5.7.4 Additive Bounding with Focal List Based Algorithms

Focal list based algorithms have been shown to be an effective way to approach state-spaces with a wide range in the edge costs. This is because such algorithms make it easy to employ both cost-to-go and distance-to-go heuristics while still satisfying desired bounds. In this section, we will demonstrate that the above theory can also be used to construct focal list based algorithms that are also effective for satisfying additive bounds in domains with non-uniform edge costs.

We considered two focal list based algorithms. The first is A^*_w. In the additive case, this algorithm will build its focal list as follows:

$$\text{FOCAL} = \left\{ n | g_t(n) + h(n) \leq \left[\min_{n' \in \text{OPEN}} g_t(n') + h(n') \right] + \gamma \right\} \tag{5.15}$$

where h is the admissible cost-to-go heuristic. This algorithm will then select from the focal list greedily according to a distance-to-go heuristic.

We also considered a newer focal list based algorithm, **Explicit Estimation Search (EES)** [90]. This algorithm was initially developed to satisfy linear suboptimality bounds. It uses not only the admissible cost-to-go heuristic h and the distance-to-go heuristic H_d, but also a potentially inadmissible cost-to-go heuristic H_{in} that estimates h^*. In the case of a linear suboptimality bound determined by weight w, the focal list used by EES is defined in the same way as A^*_w, but the way it selects nodes from the focal list will differ. For this node selection policy, the algorithm first identifies the node n_{in} which is defined as follows:

$$n_{in} = \operatorname*{argmin}_{n \in \text{OPEN}} g_t(n) + H_{in}(n) \ .$$

Intuitively, this is the node in the open list which the inadmissible cost-to-go heuristic identifies as leading to the lowest cost solution. Having defined n_{in}, the algorithm then defines an alternative focal list according to H_{in} instead of h. This secondary focal list is defined as follows:

$$\text{FOCAL}_{in} = \{ n | g_t(n) + H_{in}(n) \leq w \cdot [g_t(n_{in}) + H_{in}(n_{in})] \} \ .$$

The intuition behind FOCAL_{in} is that this list will contain the set of nodes whose expected cost to get to the goal is no more than a factor of w larger than that of the

node with the best expected cost. This focal list differs from the standard one in that it is built using the potentially inadmissible heuristic which may be more accurate than the admissible one. However, since H_{in} might be inadmissible, this focal list does not only include those nodes through which any solution cost is guaranteed to be no more than a factor of w larger than optimal. From this secondary list, the algorithm then identifies the node with the lowest distance-to-go heuristic, defined formally as follows:

$$n_d = \underset{n \in \mathrm{FOCAL}_{in}}{\operatorname{argmin}} \; H_d(n)$$

The policy EES uses for iteratively selecting nodes for expansion is then given by the following:

1. If n_d is the in the focal list (the actual focal list FOCAL, not FOCAL_{in}) then it is the next node selected for expansion.

2. If n_d is not in the focal list, then n_{in} is selected for expansion provided that it is in the actual focal list.

3. If neither n_d nor n_{in} are in the actual focal list, then the algorithm selects the node from the open list with the lowest value of $g + h$.

The intuition behind the first case is that n_d is expected to be the node from which it will be easiest to find a solution that will satisfy the given bound. However, the algorithm can only pursue the solution through n_d if n_d is in the actual focal list, as otherwise the bound will no longer be guaranteed. Otherwise, it uses the second case: n_{in} is selected for expansion provided that it is in the actual focal list. This node is selected since n_{in} is expected to lead to the lowest cost solution according to the inadmissible — but often more accurate — heuristic and expanding it may also increase the size of the secondary focal list. However, once again the algorithm can only pursue the solution through n_{in} if n_d is in the actual focal list, as otherwise the bound will no longer be guaranteed. In the final case which occurs, when the algorithm is not allow to pursue the solutions through either n_d nor n_{in} while still guaranteeing the bound will be satisfied, the algorithm selects the node from the open list with the lowest value of $g + h$. This node will necessarily be

in the actual focal list by the definition of FOCAL. When following this final node selection criteria, the hope is that the focal list size will be increased thereby allowing for the algorithm to pursue the solution paths through n_d or n_{in} on later iterations. Notice that regardless of which of these criteria is used to select a node for expansion during any iteration, the algorithm is always restricted to only select nodes from the focal list for expansion. As such, EES is guaranteed to satisfy the linear suboptimality bound by Theorem 5.5.2.

To modify this algorithm so that it applies for an additive bound, two changes have been made. Both of these were inspired by the BEES algorithm, a previous modification of EES for satisfying the $B_K(x) = K$ bounding function for some constant $K \geq 0$ [92]. The first change that was made is that the actual focal list has been modified so that it is defined for additive bounds (see equation 5.15 above). In addition, the alternative focal list based on H_{in} is defined as follows:

$$\text{FOCAL}_{in} = \{n | g_t(n) + H_{in}(n) \leq [g_t(n_{in}) + H_{in}(n_{in})] + \gamma\}$$

While only the first change is needed to guarantee that a given additive bound is satisfied, the second change was made to remain in the spirit of initial definition of the alternative focal list in the case of linear suboptimality bounds: to identify the nodes that are expected to lead to solutions that satisfy the bound according to the inadmissible heuristic.

A^*_w and EES were both tested on the inverse cost 15 puzzle. In both algorithms, the admissible heuristic was given by the weighted Manhattan distance. In A^*_w, the distance-to-go heuristic is given by the standard Manhattan distance which ignores action costs. For EES, the distance-to-go heuristic and the inadmissible heuristic are given by a technique called **one-step correction** [87]. This technique is based on the observation that $h^*(n) = h^*(n_c) + \kappa(n, n_c)$ where n is the parent of n_c along an optimal path. If the heuristic value of n_c does not decrease by $\kappa(n, n_c)$ according to some heuristic function H, then this indicates there is a one-step error in H. One-step correction inflates the heuristic value of a node n by assuming that the frequency of such errors as seen in the path found to n will be the same as that from n to the goal. The distance-to-go and inadmissible heuristic used by EES

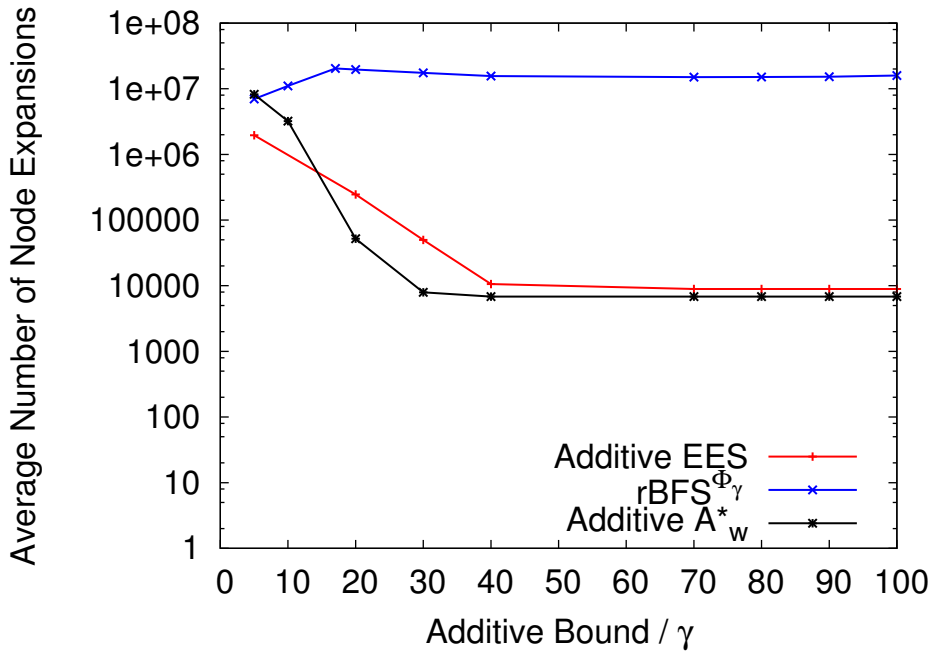


Figure 5.2: Additive focal list based algorithms in the inverse cost 15 puzzle domain.

are then given by applying one-step correction to the standard Manhattan distance heuristic and the weighted Manhattan distance heuristic, respectively.

Figure 5.2 shows the performance of these two algorithms, along with rBFS$^{\Phi_\gamma}$ on this domain. FOCAL$^{B_\gamma}$-EES is used to denote EES for additive bounds since it is a FOCAL$^{B_\gamma}$ that uses the EES policy to select nodes from the focal list. Similarly, FOCAL$^{B_\gamma}$-A$^*_w$ is used to denote the additively bound focal list algorithm that selects greedily according to the distance-to-go heuristic, since such an algorithm uses the A$^*_w$ node selection policy.

Notice that with the introduction of action costs, rBFS$^{\Phi_\gamma}$ is not exchanging guaranteed suboptimality for speed and it is much weaker than the focal list based algorithms. In addition, we see that FOCAL$^{B_\gamma}$-EES outperforms FOCAL$^{B_\gamma}$-A$^*_w$ for low values of γ, while FOCAL$^{B_\gamma}$-A$^*_w$ shows a slight advantage for large values of γ. These results, including the poor performance of rBFS in this domain, are consistent with those seen when using the versions of these algorithms initially constructed for linear suboptimality bounds in this domain. This demonstrates that these algorithms, which were initially developed for a different bounding paradigm,

have retained their relative strengths and weaknesses when modified to satisfy a different bounding function.

5.8 Chapter Summary

In this chapter, we have considered the problem of identifying what algorithm choices are available for a system designer who requires some kind of solution quality guarantees. To do so, we have introduced a functional notion of a solution quality requirement that allows a system designer more flexibility in how they define bounds. We then showed that existing anytime algorithms, best-first search algorithms, focal list based algorithms, and depth-first iterative deepening algorithms can all be modified to apply to many possible kinds of bounds.

The additive bounding function was then used to test whether the modifications suggested by the theory could be used to construct effective algorithms for a different form of bounding than these algorithms were initially developed for. In particular, the different search techniques were each tested on the types of problems they are known to be well-suited for in the case of linear suboptimality bounds, in which they were shown to still effectively sacrifice solution quality for improved runtime when used with additive bounds.

Chapter 6

Worst-Case Solution Quality Analysis When Not Re-Expanding Nodes in Best-First Search

When deploying a best-first search based algorithm, a system designer must decide on whether to use rBFS, nrBFS, or some other re-expansion policy. This decision can greatly impact algorithm performance in terms of both runtime and the quality of solutions found. The goal of this chapter is to better equip the system designer in making this decision. We do so by formally proving a bound on just how bad solution quality can get when using policies aside from the full re-expansion policy. This bound will be given in terms of a measure of the inconsistency of the heuristic. We then consider consequences of these bounds when using evaluation functions of the form $g + B(h)$, where B is a bounding function and h is an admissible heuristic.

6.1 Introduction

In the previous chapter we showed that the A* algorithm, which is an rBFS algorithm that uses the evaluation function $g + h$ where h is admissible, will only find solutions that are optimal. This was first shown by Hart, Nilsson, and Raphael [35]. In that work, it was also shown that if h is consistent, then once a node n is expanded, it can be guaranteed that $g(n) = g^*(n)$. As such, once a node is expanded, it will never be moved back into the open list or re-expanded. This means that for a consistent heuristic h, rBFS^{g+h} is equivalent to nrBFS^{g+h}.

Weight	Nodes Expanded by rWA*Relative to A*	Percentage of Re-expansions	Nodes Expanded by nrWA*Relative to A*
1.0	1.0	0%	1.0
1.1	0.94	5%	0.95
1.2	0.89	10%	0.89
1.5	0.86	16%	0.74
2	1.52	65%	0.56
3	2.25	83%	0.40
5	3.17	90%	0.33
10	3.28	91%	0.30

Table 6.1: The impact of re-expanding nodes on WA*.

If the heuristic in use is admissible but inconsistent, then it can no longer be guaranteed that when a node n is expanded, the g-cost of n will be optimal. An example of this behaviour is given by Martelli who identified a family of graphs, which will be described in more detail below, for which A* will perform $\Theta(2^{|V|})$ expansions [61]. As such, the number of re-expansions performed on these graphs can dominate runtime and thereby greatly decrease algorithm performance.

This behaviour is not isolated to the use of an admissible but inconsistent heuristic with A*, as re-expansions can also greatly harm the performance of a rBFS that uses an inadmissible heuristic. For example, this can happen when using rWA*, which can be thought of as being a rBFS^{g+H} instance that uses an inadmissible heuristic given by $H = w \cdot h$ where h is admissible. To demonstrate this behaviour, we tested this algorithm on a set of pathfinding problems given by Sturtevant [84]. The set consists of 10 512×512 grids in which 40% of the grid locations have been marked as obstacles. The objective is to find a path from a given initial location to an given goal location, where motion can be made in the 8 possible directions of north, northeast, east, southeast, south, southwest, west, or northwest, provided that there is no obstacle in the way. The heuristic used is called the **octile heuristic**. Like the Manhattan distance described for the sliding tile puzzle, this consistent heuristic counts up how many locations away from the goal the current location is, with an adjustment for the ability to move in 8 directions instead of 4.

Given these 10 maps, we used the total of $35,360$ pairs of start and goal po-

sitions from the benchmark set [84]. The average optimal solution cost of these problems is 733.87, and A* performs an average of $36,003$ expansions per problem. Table 6.1 shows the average performance of rWA* on these problems for a variety of weights. The first column shows the weight, the second column shows the total number of expansions relative to A*, and the third column shows what percentage of the total expansions were re-expansions. The table shows that the higher weights actually expand more nodes than A*, largely because of re-expansions. For example, 91% of the expansions made by the weight 10 rWA* search are re-expansions, which is why it is slower than A* despite expanding only 30% as many unique nodes.

When re-expansions dominate runtime as they do in the examples just given, many system designers opt to never allow for re-expansions as opposed to using the full re-expansion policy. This approach has been particularly popular in the case of WA* as it has been applied successfully in domains such as robot path planning [58] and binary decision diagram minimization [15]. This approach would similarly improve the runtime in the example domains considered above. For example, the final column of Table 6.1 shows the total number of node expansions made by nrWA* relative to A*. As shown, all weights greater than one lead to a faster search than A*, which was not the case when the full re-expansion policy is used. Similar improvements would be seen if the search used on Martelli's graphs was set to not re-expand nodes. In that case, only $\Theta(|V|)$ expansions would be performed instead of the $\Theta(2^{|V|})$ performed when using the full re-expansion policy.

Yet there is still much that is not fully understood about the impact that not re-expanding nodes has on a search. For example, when this technique has been tested empirically it was shown to improve algorithm runtime in some problems while harming it in others, and it also typically decreases the quality of the solutions found [15, 88]. This is true not even just across domains, but across different tasks within the same domain [82]. However, there are still no theoretical results that identify the properties of a state-space that determine whether or not re-expanding nodes is beneficial. Most of the empirical studies on this topic have focused on the use of this technique when using a consistent heuristic in nrWA* or in focal list

based algorithms. The use of a consistent heuristic in nrWA* and focal list based algorithms are also the only cases in which there are known bounds on the quality of the solutions found when not re-expanding nodes. As such, if a system designer is building a search system, it is not always clear when they should use a re-expansion policy aside from the full re-expansion policy.

6.1.1 Contributions

The goal of this chapter is to begin to address this gap in our understanding of re-expansion policies other than the full re-expansion policy. Our specific focus will be on formally analyzing how not re-expanding nodes can impact the solution quality. In particular, we will show that the worst-case loss in quality that can result from any re-expansion policy can be bound based on the amount of inconsistency along optimal solution paths. This will be proven for a large class of best-first search algorithms and will apply regardless of whether the heuristic is admissible or inadmissible. The bound will then be used to show that for admissible heuristics, the worst-case when using an A* which does not re-expand nodes is to find solutions that are quadratic in the optimal solution cost. We will then identify a family of worst-case graphs and corresponding heuristics for which the given bound is exact when using nrBFS. Finally, we will consider the known bound on solution quality that is specific to the use of a consistent heuristic with WA*, extend this bound so that it applies to other types of heuristic weighting, and provide bounds when weighting inconsistent heuristics.

Note that this chapter is based on work that was published at the 2014 AAAI conference [99].

6.2 Background and Related Work

In this section, we will define a metric for measuring the inconsistency of a path and then describe several related studies on the impact of inconsistent heuristics and different re-expansion policies.

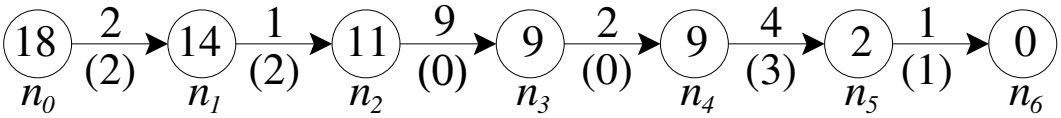


Figure 6.1: Example path with INC_H values shown below the edge costs in parentheses.

6.2.1 Heuristic Inconsistency

Consider Figure 6.1 which shows a path of nodes n_0, ..., n_6 with an associated heuristic value for each (ignore the values in parentheses for now). Recall that in order for a heuristic H to be consistent, it must be true that for any edge $(p, c) \in E$, $H(p) \leq H(c) + \kappa(p, c)$. This property is clearly not true along this path. For example, consider the edge from n_1 to n_2 and that $H(n_1) = 14 > H(n_2) + \kappa(n_1, n_2) = 11 + 1 = 12$. This heuristic is therefore inconsistent along this path.

Notice that the consistency property does hold on some edges, such as (n_3, n_4), it just does not hold on all of them. Intuitively, this means that the heuristic can be thought of as being consistent on some edges, and inconsistent on others. To compare the relative amount of inconsistency of the heuristic on different edges, we define a metric called the **inconsistency of a heuristic** H **on an edge** from p to c. This metric, which will be denoted by $INC_H(p, c)$, is formally defined as follows:

$$INC_H(p, c) = \max(H(p) - H(c) - \kappa(p, c), 0) \ .$$

Intuitively, this metric measures how far away the heuristic H is from being consistent on the edge from p to c. This is why the $\max$ is included to prevent the value of INC_H from becoming negative. If $H(p) - H(c) - \kappa(p, c)$ is negative, then the distance is consistent on this edge. The "distance" of H from being consistent on such an edge is therefore 0, regardless of how negative $H(p) - H(c) - \kappa(p, c)$ is. Furthermore, notice that $INC_H(p, c) > 0$ if and only if H is inconsistent on edge (p, c), and 0 otherwise. As such, the standard definition of consistency is equivalent to requiring that $INC_H(p, c) = 0$ for any $(p, c) \in E$.

We will also be able to use this metric to measure the **inconsistency of a heuristic** H **along a path** P as the sum of the inconsistency of H on each of the edges

along P. Where this value is denoted as $INC_H(P)$ and $P = [n_0, ..., n_k]$, this value is formally given by the following:

$$INC_H(P) = \sum_{i=0}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

If P is the empty path [] consisting of 0 nodes, or a path $[n]$ consisting of a single node, then we will define $INC_H(P)$ such that $INC_H(P) = 0$.

As an example of how these metrics are calculated, let us reconsider the path $[n_0, ..., n_6]$ given in Figure 6.1. The numbers given in parentheses below each edge correspond to the value of the inconsistency of the heuristic on the corresponding edge. For example, where H is the heuristic in use, H is inconsistent on edge (n_4, n_5), which is why $INC_H(n_4, n_5) = 9 - 2 - 4 = 3$, while the value of $INC_H(n_2, n_3)$ and $INC_H(n_3, n_4)$ are both 0 since H is consistent on these edges.

As described above, the inconsistency of H along the path in Figure 6.1 is then given by the sum of the numbers in the parentheses. In this case, the result is $2+2+0+0+3+1 = 8$. The path in Figure 6.1 also demonstrates the importance of using the $\max$ function in the definition of INC_H to prevent this metric from returning negative values. If $INC_H(p, c)$ were calculated as $H(p) - H(c) - \kappa(p, c)$ without taking the maximum with 0, the sum of the inconsistency of H on each edge along this path would be -1, and this value does not capture that there is inconsistency along this path.

It is for this reason that INC_H is used instead of an existing metric for measuring heuristic inconsistency called the *inconsistency rate of an edge (IRE)* [22, 109]. The IRE of an edge, which is calculated as $|H(p)-H(c)|$, was initially intended for graphs with only unit-cost, undirected edges. The directed version of IRE would be $H(p) - H(c)$, though this is still only suitable for unit-cost edges. To see this, notice that the IRE of edges (n_2, n_3) and (n_5, n_6) in Figure 6.1 are both 2, even though H is consistent on (n_2, n_3) and inconsistent on (n_5, n_6). Since the value of IRE can be negative in the directed case, we also cannot use the sum of the IRE values along a path P as a measure of the inconsistency along P for the same reason that INC_H is forced to be non-negative. As such, INC_H was found to be a more convenient metric than IRE for the analysis performed in this chapter.

6.2.2 The Impact of Heuristic Inconsistency

There have been several studies looking at the impact of heuristic inconsistency on search performance, but they have focused on admissible but inconsistent heuristics when used in A*. This began with the work of Martelli [61] mentioned above. Martelli also offered a technique which decreased the worst-case runtime from being $O(2^{|V|})$ when using standard A* to being $O(|V|^2)$ while still maintaining optimality. A different technique with the same runtime guarantees was then proposed by Mero [63].

Felner et al. [21] then offered a way to improve Mero's technique in undirected graphs. They also looked at the worst-case exponential behaviour identified by Martelli and showed that this could only happen in graphs in which the edge costs are exponential in $|V|$.

6.2.3 Studies on nrOCL Algorithms

There are two main previous results that offer upper bounds on the impact that not re-expanding nodes has on solution quality. The first is that any solution found by nrWA* is guaranteed to have a cost of no more than $w \cdot C^*$ provided that the heuristic being weighted is consistent [58]. Ebendt and Drechsler then offered an upper bound on the cost of any solution returned by any nrFOCAL algorithm which builds its focal list using the linear suboptimality bounding function $B_w(x) = w \cdot x$, provided that the heuristic used for the construction of the focal list is consistent. This upper bound is given by $w^{\lfloor D/2 \rfloor} \cdot C^*$, where the optimal solution has D edges in it [15]. However, there are currently no results for either of these algorithms in the case of inconsistent heuristics.

Several authors have noted that not re-expanding nodes can help on some problems while it can hurt in others. For example, Hansen and Zhou [33] noticed that nrWA* often expanded more unique nodes than rWA*, particularly in domains in which the solution paths are relatively sparse. Thayer [86] similar identified that the impact of this technique varied from domain to domain and suggested that it will most likely be useful in domains with many transpositions.

More recently, Sepetnitsky, Felner, and Stern [82] have shown that even in the same search problem, the choice of whether re-expansions will improve or harm performance in a WA* search on a particular task can depend on the value of the weight. Their experiments show that in the 15 puzzle and in pathfinding, a significant proportion of the test set is solved faster when re-expanding nodes than when not for low weights, though not re-expanding is better for higher weights. They also demonstrate that in certain problems, nrWA* actually finds better quality solutions than rWA*, though this is not typical.

Sepetnitsky, Felner, and Stern also demonstrated that there is potential for re-expansion policies aside from the full and never re-expansion policies to further improve the runtime of WA* [82]. In this chapter, we will prove that such policies will also be guaranteed to satisfy the linear suboptimality bound provided that the heuristic being weighted is consistent.

Not re-expanding nodes is also a common technique used by anytime heuristic search algorithms. The first such anytime algorithm to use this technique was ARA* [58]. This algorithm iteratively runs a set of nrWA* searches that perform parent pointer updates to find a set of iteratively improving solutions. During each nrWA* search, the algorithm stores the nodes in the closed list to which a a lower g-cost path is found — which we defined as the unopened list earlier — in a separate list. Once a solution is found, a new nrWA* search is performed, though in this case the open list that the search starts with is not given by the set $\{n_{init}\}$, but with the open list that remained at the end of the previous nrWA* run merged with the unopened list. The weight used for the next iteration is also lowered. ARA* therefore consists of a sequence of nrWA* searches with decreasing weights, each of which delays the re-opening of nodes and merges the open list and unopened list of a previous iteration to be used as the starting open list of a new iteration. This process then continues either until the time limit is reached or a weight 1 search is completed. Thayer and Ruml [89] also showed that this "repairing" paradigm could be used to turn other bounded suboptimal search algorithms into effective anytime algorithms.

6.3 Worst-Case Solution Quality Analysis

In this section, we will show that the quality of the solutions found by not only nrBFS^{g+H}, but any BFS^{g+H} algorithm, can be bound based on the inconsistency of H along optimal solution paths. This will hold regardless of the re-expansion policy used by the algorithm. Notice that this means we are only considering evaluation functions of the form $g+H$. This restriction still allows for the framework to include A* (which corresponds to the use of an admissible H), WA* (which corresponds to $H = w \cdot h$ where h is admissible), and the use of many other evaluation functions. However, this framework does not include GBFS since that algorithm does not include g in its evaluation function.

In addition to this restriction, we will also assume that the heuristic value of any goal node is 0. Formally, this means the following:

$$\forall n \in V_{Goal}, H(n) = 0 \ .$$

6.3.1 Bounding with Nodes on an Optimal Solution Path

In order to prove the bound given below, let us first consider the situation in which the $t+1$-st node expanded by a BFS^{g+H} instance is a goal node $n_g \in V_{Goal}$. This means that no goal node had been expanded in the first t expansions. Now recall that if there is an optimal solution path P_{opt} to a given problem, then at any time prior to the expansion of a goal node there will be a node from P_{opt} in the open list. This was guaranteed by Corollary 2.2.8 in Section 2.2.4. In particular, let n be the node from P_{opt} that is guaranteed to be in the open list after t expansions by this corollary, and consider the following derivation of the evaluation of n:

$$\begin{aligned} g_t(n) + H(n) &= g^*(n) + g_t^\delta(n) + H(n) \\ &= g^*(n) + g_t^\delta(n) + h^*(n) - h^*(n) + H(n) \\ &= C^* + g_t^\delta(n) + (H(n) - h^*(n)) \ . \end{aligned} \tag{6.1}$$

This shows that the evaluation of n differs from C^* by exactly the sum of the g-cost error and $(H(n) - h^*(n))$ which, if positive, is how inadmissible H is on n. Since

n_g was selected for the $t+1$-st node expansion instead of n, this means that

$$g_t(n_g) + H(n_g) \leq g_t(n) + H(n) \ ,$$

by the definition of BFS. Where C is the cost of the solution found by BFS, this inequality can be simplified to $C \leq g_t(n) + H(n)$ since $n_g \in V_{Goal}$ implies that $H(n_g) = 0$ and the fact that $C \leq g_t(n_g)$. This latter fact holds by Theorem 2.2.5 from Section 2.2.4 which guarantees that the path implicitly maintained using the parent pointers has a cost of no more than $g_t(n_g)$. Substituting $C \leq g_t(n) + H(n)$ into equation 6.1 allows for the following observation:

Observation 6.3.1. *If the $t+1$-st node expanded by BFS^{g+H} is a goal node n_g, then there is a node n from some optimal path P_{opt}, where $n \in \text{OPEN}_t$ and the cost of the solution found to n_g will be no more than*

$$C^* + g_t^{\delta}(n) + H(n) - h^*(n) \ .$$

Below we will show that $g_t^{\delta}(n) + H(n) - h^*(n)$ is bound by the inconsistency of H along the optimal path that n is on. This proof will have two components. In the first, we will show in Section 6.3.2 that for any node n_i which is on an optimal solution path $P_{opt} = [n_0, ..., n_k]$, the inadmissibility of H on n_i (*ie.* $H(n_i) - h^*(n_i)$) is bound by the inconsistency of H along the portion of P_{opt} from n_i to n_k. We will then show in Section 6.3.3 that at any time prior to the expansion of a goal node there is at least one node n_i from P_{opt} which is in the open list such that the g-cost error of n_i is no larger than the inconsistency of H along the portion of P_{opt} from n_1 to n_i. These will then be used to derive the main theorem given in Section 6.3.4.

6.3.2 Bounding Inadmissibility with Inconsistency

In this section, we will show that the inadmissibility of the heuristic value of a node n can be bound by the inconsistency along the optimal path from n to the nearest goal node. We begin by showing the following more general statement:

Lemma 6.3.2. *If $[n_0, n_1, ..., n_k]$ is an optimal path from n_0 to n_k such that $H(n_i) \neq \infty$ for all $0 \leq i \leq k$, then the following is true:*

$$H(n_0) - H(n_k) - g^*(n_0, n_k) \leq \sum_{i=0}^{k-1} INC_H(n_i, n_{i+1}) \ ,$$

which holds with equality if $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \geq 0$ *for all* $0 \leq i < k$.

Proof. Let $[n_0, n_1, ..., n_k]$ be an optimal path from n_0 to n_k such that $H(n_i) \neq \infty$ for all $0 \leq i \leq k$. Now notice that if $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) > 0$ then the following is true:

$$INC_H(n_i, n_{i+1}) = H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \ .$$

This follows directly from the definition of INC_H, which is given by the maximum of the right side of this equation and 0. If $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \leq 0$, then $INC_H(n_i, n_{i+1}) = 0$. Together, these two facts mean that we can guarantee that the following is true:

$$INC_H(n_i, n_{i+1}) \geq H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \ .$$

This in turn implies the following:

$$\sum_{i=0}^{k-1} H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \leq \sum_{i=0}^{k-1} INC_H(n_i, n_{i+1}) \ . \tag{6.2}$$

Notice that this will hold with equality if $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \geq 0$ for all $0 \leq i < k$.

Now we will show that the left side of Equation 6.2 is equal to $H(n_0) - H(n_k) - g^*(n_0, n_k)$. In particular, we will show show the following is true for any $j \leq k$ by induction:

$$H(n_0) - H(n_j) - g^*(n_0, n_j) = \sum_{i=0}^{j-1} H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \ .$$

This clearly holds true in the base case when $j = 1$ by the following:

$$\begin{aligned} H(n_0) - H(n_k) - g^*(n_0, n_1) &= H(n_0) - H(n_1) - \kappa(n_0, n_1) \\ &= \sum_{i=0}^{0} H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \ . \end{aligned}$$

This derivation holds because $\kappa(n_0, n_1) = g^*(n_0, n_1)$ since P is an optimal path from n_0 to n_1. Now suppose the statement is true for $1 \leq j < k - 1$, and consider

the following derivation:

$$\begin{aligned}
&\sum_{i=0}^{j} H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \\
&\qquad = \left[\sum_{i=0}^{j-1} H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1})\right] \\
&\qquad\qquad + H(n_j) - H(n_{j+1}) - \kappa(n_j, n_{j+1}) && (6.3) \\
&\qquad = [H(n_0) - H(n_j) - g^*(n_0, n_j)] \\
&\qquad\qquad + H(n_j) - H(n_{j+1}) - \kappa(n_j, n_{j+1}) && (6.4) \\
&\qquad = H(n_0) - H(n_{j+1}) - [g^*(n_0, n_j) + \kappa(n_j, n_{j+1})] && (6.5) \\
&\qquad = H(n_0) - H(n_{j+1}) - g^*(n_0, n_{j+1}) \ . && (6.6)
\end{aligned}$$

Line 6.3 follows by an expansion of the summation term. Line 6.4 is true by the induction hypothesis. Line 6.5 then follows by cancelling out the $H(n_j)$ and $-H(n_j)$ terms. The final line then follows since $g^*(n_0, n_j) + \kappa(n_j, n_{j+1}) = g^*(n_0, n_{j+1})$ since $[n_0, ..., n_{j+1}]$ is an optimal path from n_0 to n_{j+1}. Therefore, the statement holds by induction.

The theorem then holds by substituting this result into Equation 6.2. □

Since we have assumed that $H(n) \neq \infty$ for any n from which a goal node is reachable, this lemma immediately leads to the following bound on the inadmissibility of the heuristic value of such nodes:

Theorem 6.3.3. *If $P = n_0, n_1, ..., n_k$ is a path from n_0 to some $n_k \in V_{Goal}$ such that $h^*(n_0) = g^*(n_0, n_k)$, then the following holds:*

$$H(n_0) - h^*(n_0) \leq \sum_{i=0}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

This follows from Lemma 6.3.2 since $g^*(n_0, n_k) = h^*(n_0)$ and by the fact that $n_k \in V_{Goal}$ implies that $H(n_k) = 0$.

6.3.3 Bounding g-cost Error with Inconsistency

In this section, we will show that whenever a node n is selected for expansion during the execution of a BFS^{g+H} algorithm, the g-cost error of n can be bound

by the inconsistency along the optimal path from n_{init} to n, regardless of the re-expansion policy in use. This will then be used to show that at any time prior to the expansion of a goal node, there will be a node n from an optimal solution path P_{opt} that is in OPEN_t whose g-cost error is bound by the inconsistency along P_{opt}.

We begin with the following lemma which shows how bounds of the type we will consider can be propagated from parent to child along an optimal path, if the child is first expanded after the parent. Notice that the bound on the g-cost error of a node n_k is given in terms of the sum of the inconsistency along an optimal path to n_k, not including the inconsistency of the heuristic on the first edge on this path. This will be true of all bounds on g-cost error considered in this section for reasons we will describe later.

Lemma 6.3.4. *Let $P = [n_0, ..., n_k]$ be an optimal path from $n_{init} = n_0$ to n_k such that $H(n_i) \neq \infty$ for all $0 \leq i \leq k$. If n_{k-1} is expanded for the first time by the t-th node expansion by a BFS^{g+H} algorithm and n_k is not one of the first t nodes selected for expansion, then the following holds:*

$$g_t^\delta(n_{k-1}) \leq \sum_{i=1}^{k-2} INC_H(n_i, n_{i+1}) \Longrightarrow g_t^\delta(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

Proof. Assume the conditions described above on P, that n_{k-1} is expanded for the first time by the t-th node expansion of a BFS^{g+H} algorithm, that n_k is not one of the first t nodes selected for expansion, and that $g_t^\delta(n_{k-1})$ is bound as specified by the theorem statement. Recall that because n_{k-1} is expanded before n_k, we can guarantee that $g_t^\delta(n_k) \leq g_t^\delta(n_{k-1})$. This was formalized by Lemma 2.2.3 in Section 2.2.3. Along with the assumptions, this allows for the following derivation:

$$\begin{aligned}
g_t^\delta(n_k) &\leq g_t^\delta(n_{k-1}) && (6.7)\\
&\leq \sum_{i=1}^{k-2} INC_H(n_i, n_{i+1}) && (6.8)\\
&\leq \left[\sum_{i=1}^{k-2} INC_H(n_i, n_{i+1})\right] + INC_H(n_{k-1}, n_k) && (6.9)\\
&\leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ . && (6.10)
\end{aligned}$$

Line 6.8 follows by our assumption on $g_t^{\delta}(n_{k-1})$. Line 6.9 then follows since $INC_H(n_{k-1}, n_k) \geq 0$. The final line then follows by folding $INC_H(n_{k-1}, n_k)$ into the summation. Therefore, the statement is true. □

We now turn to the main theorem in this section, which states that whenever a node n_k is selected for expansion during the execution of BFS^{g+H}, the g-cost error of n_k can be bound by the inconsistency of H along the optimal path P to n_k, excluding the inconsistency on the first edge of P.

Theorem 6.3.5. *Let n_k be the t-th node expanded by a BFS^{g+H} algorithm. If there exists an optimal path $P = [n_0, n_1, ..., n_k]$ from $n_{init} = n_0$ to n_k where $H(n_i) \neq \infty$ for all $0 \leq i \leq k$, then the following holds:*

$$g_t^{\delta}(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

Proof. The proof is by induction on the number of node expansions, denoted by t. In the base case of $t = 1$, the node selected for expansion is n_{init}. Since $g_0(n_{init}) = g^*(n_{init})$, this means that when it is selected for expansion, the g-cost error of n_{init} is also 0. Therefore, the statement is true in this base case.

Assume the statement is true for all of the first $t \geq 1$ nodes selected for expansion. Let n_k be the $t+1$-st node to be expanded. If there does not exist an optimal path $P = [n_0, ..., n_k]$ from n_{init} to n_l such that $H(n_i) \neq \infty$ for all $0 \leq i \leq k$, then the statement is vacuously true. Now suppose that there does exist such a path. If n_k was also the t'-th node expanded for some $t' \leq t$, then the following is true by the induction hypothesis:

$$g_{t'}^{\delta}(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

Since the g-cost error of n_k cannot increase as BFS^{g+H} makes further expansions (where this notion was formalized by Observation 2.2.2 in Section 2.2.3), this means that the following holds:

$$g_t^{\delta}(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

Therefore, the statement holds if n_k was previously expanded.

Now suppose that n_k is expanded for the first time by the $t+1$-st node expansion. There are now two cases to consider: either n_{k-1} has been previously expanded or it was not. Let us first assume that n_{k-1} was previously selected as the t'-th node expanded for some $t' \leq t$. Since P can only be an optimal path to n_k if $[n_0, ..., n_{k-1}]$ is an optimal path to n_{k-1}, the induction hypothesis guarantees the following:

$$g^{\delta}_{t'}(n_{k-1}) \leq \sum_{i=1}^{k-2} INC_H(n_i, n_{i+1}) \ .$$

We can therefore use Lemma 6.3.4 to get the following bound:

$$g^{\delta}_{t'}(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ .$$

The desired bound on $g^{\delta}_{t}(n_k)$ then follows immediately since the g-cost error of n_k cannot increase with further node expansions.

Notice that if $k = 1$ and $n_k = n_1$ is expanded for the first time, $n_{k-1} = n_0$ must have been previously expanded. This is because $n_0 = n_{init}$ is always necessarily the first node expanded. Therefore, the situation in which $n_k = n_1$ is expanded for the first time is an instance of the case just handled. The g-cost error of n_1 will therefore necessarily be 0 since the g-cost error of n_{init} is 0. This "omission" of $INC_H(n_0, n_1)$ from the g-cost error of n_1 will then be propagated to any n which is deeper along P (either by the case above or that below), which is why the inconsistency along the first edge of P is not incorporated into the theorem statement.

In addition, notice that the same argument used to prove this last case can also be used to guarantee that for any node n_j from P that is in OPEN_t where n_{j-1} has been previously expanded, this bound will also apply to n_j. This will be used in proving the next case.

In the second case to be considered, n_{k-1} has not been previously expanded. However, recall that we are guaranteed that there will be a node n_j from P such that $n_j \in \mathrm{OPEN}_t$ and $n_{j'} \in \mathrm{CLOSED}_t$ for any $0 \leq j' < j$. This is true since BFS is an OCL algorithm, not all nodes from P are in the closed list (since n_k is in the open list), and by Theorem 2.2.7 in Section 2.2.4. By using the argument just

applied for when n_{k-1} was previously expanded, we can therefore guarantee the following:

$$g_t^{\delta}(n_j) \leq \sum_{i=1}^{j-1} INC_H(n_i, n_{i+1}) \ . \tag{6.11}$$

Since n_{j-1} has previously been expanded and n_{k-1} has not, this means that $n_j \neq n_k$. Moreover, n_j must be shallower on P than n_k is since all nodes on P that are shallower than n_j are in the closed list and n_k is in the open list. As the algorithm selected n_k instead of n_j as the $t+1$-st node to be expanded, the first line of the following is true which then allows for the following derivation:

$$g_t(n_k) + H(n_k) \leq g_t(n_j) + H(n_j) \tag{6.12}$$

$$g_t^{\delta}(n_k) \leq g_t^{\delta}(n_j) + H(n_j) - H(n_k) + g^*(n_j) - g^*(n_k) \tag{6.13}$$

$$g_t^{\delta}(n_k) \leq g_t^{\delta}(n_j) + [H(n_j) - H(n_k) - g^*(n_j, n_k)] \tag{6.14}$$

$$g_t^{\delta}(n_k) \leq g_t^{\delta}(n_j) + \left[\sum_{i=j}^{k-1} INC_H(n_i, n_{i+1})\right] \tag{6.15}$$

$$g_t^{\delta}(n_k) \leq \left[\sum_{i=1}^{j-1} INC_H(n_i, n_{i+1})\right] + \left[\sum_{i=j}^{k-1} INC_H(n_i, n_{i+1})\right] \tag{6.16}$$

$$g_t^{\delta}(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ . \tag{6.17}$$

Line 6.13 is achieved by expanding $g_t(n)$ to $g^*(n) + g_t^{\delta}(n)$, and rearranging the terms. Line 6.14 then holds since n_j is along an optimal path to n_k and so $g^*(n_k) = g^*(n_j) + g^*(n_j, n_k)$. By using the upper bound on $H(n_j) - H(n_k) - g^*(n_j, n_k)$ given by Lemma 6.3.2 results in line 6.15. Line 6.16 then holds by substituting in the bound on $g_t^{\delta}(n_j)$ given above in equation 6.11. The final line then follows by combining the summations. Therefore, the statement holds in the case that n_{k-1} has not been previously expanded.

Having handled all cases for n_k, the theorem holds by induction. □

While this theorem can be used to immediately offer a bound on the solution quality found by BFS^{g+H} in domains in which there is only a single goal node, we will need the following corollary in the case that there are multiple goal nodes.

Corollary 6.3.6. *Let $P_{opt} \in \Pi^*_{Goal}$ be an optimal solution path to a given planning task Γ where $P_{opt} = [n_0, ..., n_k]$. If no goal node has been expanded during the first t iterations of a BFS^{g+H} on Γ, then there will exist a node $n_i \in P_{opt}$ such that all of the following conditions hold:*

1. $n_i \in \text{OPEN}_t$.
2. $\forall j$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$.
3. $g^\delta_t(n_k) \leq \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1})$.

Proof. After t expansions, Corollary 2.2.8 in Section 2.2.4 guaranteed the existence of some n_i from P_{opt} such that $n_i \in \text{OPEN}_t$, and for any j where $0 \leq j < i$, $n_j \in \text{CLOSED}_t$. Therefore, the first two conditions hold.

If $i = 0$ (*ie.* $n_i = n_{init}$), the third condition clearly also holds for n_i. If $n_i \neq 0$ but n_i was previously expanded, then the third condition immediately follows for n_i by Theorem 6.3.5. If $n_i \neq 0$ and n_i was not previously expanded, notice that n_{i-1} must have been previously expanded since it is in the closed list. As such, the g-cost error of n_{i-1} can be bound by the inconsistency of H along $[n_1, ..., n_{i-1}]$ by Theorem 6.3.5. This is possible since P is an optimal solution path, and so $H(n_j) \neq \infty$ for any $n_j \in P$. The third condition then follows for n_i using this result and Lemma 6.3.4. □

6.3.4 Bounding Solution Quality with Inconsistency

In this section, we will use Theorems 6.3.3 and 6.3.5 to show that the inconsistency of H along any optimal solution path (not including the inconsistency of H on the first edge) can be used to bound the quality of any solution found. This is formalized by the following theorem:

Theorem 6.3.7. *If $P_{opt} \in \Pi^*_{Goal}$ is an optimal solution path to a given planning task Γ such that $P_{opt} = [n_0, ..., n_k]$, then the following will be an upper bound on the cost of any solution returned by BFS^{g+H} on Γ:*

$$C^* + \sum_{j=1}^{k-1} INC_H(n_j, n_{j+1}) \ .$$

Proof. If $k = 0$, the search will obviously find the optimal solution with the first expansion. Clearly the statement is satisfied in that case.

If $k > 0$, assume that a goal node n_g is first expanded by the the $t+1$-st expansion for some $t \geq 0$. Let C be the cost of the solution found. By Corollary 6.3.6, there will be a node n_j from P_{opt} which is in the open list after t expansions and which satisfies the following condition:

$$g_t^\delta(n_j) \leq \sum_{i=1}^{j-1} INC_H(n_i, n_{i+1}) \ . \tag{6.18}$$

Recall that Observation 6.3.1 identified that any solution found by BFS^{g+H} will have a cost no greater than $C^* + g_t^\delta(n_j) + H(n_j) - h^*(n_j)$. This means the following is true:

$$C \leq C^* + g_t^\delta(n_j) + H(n_j) - h^*(n_j) \tag{6.19}$$

$$C \leq C^* + g_t^\delta(n_j) + [H(n_j) - H(n_k) - g^*(n_j, n_k)] \tag{6.20}$$

$$\leq C^* + \left[\sum_{i=1}^{j-1} INC_H(n_i, n_{i+1})\right] + \left[\sum_{i=j}^{k-1} INC_H(n_i, n_{i+1})\right] \tag{6.21}$$

$$\leq C^* + \sum_{i=1}^{k-1} INC_H(n_i, n_{i+1}) \ . \tag{6.22}$$

Line 6.20 is achieved since $H(n_k) = 0$ as $n_k \in V_{Goal}$ and $h^*(n_j) = g^*(n_j, n_k)$ by the fact that P_{opt} is an optimal path to n_k. Line 6.21 then follows by bounding $g_t^\delta(n_j)$ using Equation 6.18, and by bounding $H(n_j) - H(n_k) - g^*(n_j, n_k)$ by Theorem 6.3.3. The final line is then achieved by combining the two summations. The statement therefore holds. □

This theorem shows a formal relationship between the inconsistency of the heuristic and solution quality. We now consider this bound in the special case that H is admissible.

6.3.5 Worst-Case Bound when Using Admissible Heuristics

In this section, we will show that when the heuristic H being used is admissible, then the cost of the solutions returned by BFS^{g+H} will be at most quadratic in

the optimal solution path, regardless of the re-expansion policy being used. To show this result, we first consider the following which holds when there is a known maximum on the heuristic value of any node along an optimal solution path:

Lemma 6.3.8. *Let $P_{opt} \in \Pi^*_{Goal}$ be an optimal solution path to a given planning task Γ that consists of D edges. If $H_{\max} \geq 0$ is a constant such that for any $n \in P_{opt}$, $H(n) \leq H_{\max}$, then the following will be an upper bound on the cost of any solution returned by BFS^{g+H} on Γ:*

$$D \cdot \max(H_{\max}, C^*) \ .$$

Proof. Let P_{opt} be an optimal solution path such that $P_{opt} = [n_0, ..., n_k]$ where $n_{init} = n_0$ and $n_k \in V_{Goal}$. Assume that $H_{\max}$ is defined as given above, let C be the cost of the solution found, and consider any two consecutive nodes n_i and n_{i+1} on P_{opt}. Since $H(n_i) \leq H_{\max}$, $H(n_{i+1}) \geq 0$, and $\kappa(n_i, n_{i+1}) \geq 0$, this means that the following is true:

$$H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1}) \leq H_{\max} \ .$$

Now recall that $INC_H(n_i, n_{i+1})$ is given by the maximum of the left side of this inequality and 0. Since $H_{\max} \geq 0$, the following is true:

$$INC_H(n_j, n_{j+1}) \leq H_{\max}$$

As this holds for any consecutive pair of nodes on P_{opt}, this allows for the following derivation which begins with the bound on C guaranteed by Theorem 6.3.7:

$$\begin{aligned}
C &\leq C^* + \sum_{j=1}^{k-1} INC_H(n_j, n_{j+1}) && (6.23)\\
&\leq C^* + \sum_{j=1}^{k-1} H_{\max} && (6.24)\\
&\leq C^* + (k-1) \cdot H_{\max} && (6.25)\\
&\leq C^* + (k-1) \cdot \max(H_{\max}, C^*) && (6.26)\\
&\leq k \cdot \max(H_{\max}, C^*) \ . && (6.27)
\end{aligned}$$

Therefore, the lemma holds since k is the number of edges on P_{opt} (*ie.* $D = k$). □

If all edges in the state-space have a cost of at least θ where θ is a constant that is larger than 0, then there can be at most C^*/θ edges along the optimal path. Therefore, $D \leq C^*/\theta$ in which case we can apply Lemma 6.3.8 to immediately yield the following corollary:

Corollary 6.3.9. *Let Γ be a task such that for any $(p, c) \in E$, $\kappa(p, c) \geq \theta$ for some constant $\theta > 0$. If $H_{\max} \geq 0$ is a constant such that for any $n \in P_{opt}$, $H(n) \leq H_{\max}$, then the following will be an upper bound on the cost of any solution returned by BFS^{g+H} on Γ:*

$$(C^*/\theta) \cdot \max(H_{\max}, C^*) \ .$$

If H is admissible, then $C^* \geq H(n)$ for any $n \in P_{opt}$. In that case, we can set $H_{\max}$ as C^* and Corollary 6.3.9 implies the following:

Corollary 6.3.10. *Let Γ be a task such that for any $(p, c) \in E$, $\kappa(p, c) \geq \theta$ for some constant $\theta > 0$. If H is an admissible heuristic function, then the following will be an upper bound on the cost of any solution returned by BFS^{g+H} on Γ:*

$$[C^*]^2/\theta \ .$$

As such, the worst-case solution quality for BFS^{g+H} when H is admissible is quadratic in C^* if there is some positive minimum edge cost.

When using an admissible heuristic, this result identifies that there is a trade-off of worst-case solution quality for worst-case runtime when deciding what re-expansion policy to use with BFS^{g+H}. When using the full re-expansion policy and an admissible heuristic H, Martelli [61] and Mero [63] offered best-first variants that can guarantee that any solutions found by rBFS^{g+H} will require $O(|V|^2)$ node expansions if V, while still ensuring that the solutions found are optimal. In contrast, if the search is set to never perform node re-expansions and all edges have a cost that is larger than some positive constant θ, then a solution will be found in no more than $|V|$ iterations, but the cost of the solution found may be quadratic in C^*.

In the next section, we will show that there exists graphs of any size in which the solutions found by nrBFS^{g+H} exactly match the inconsistency of the heuristic along

an optimal solution path. This means that there are graphs in which the solutions found by nrBFS have a cost that is equal to the upper bound given in Theorem 6.3.7. In contrast, while it is possible to construct graphs in which the solutions found are $\Theta([C^*]^2)$ (as we do in the next section), nrBFS^{g+H} will always outperform the bounds given in Lemma 6.3.8, and Corollaries 6.3.9 and 6.3.10 for any graph where the optimal solution has more than two edges. To see why, notice that a key part of the proof of Lemma 6.3.8 — from which the remaining bounds are built — was the bounding of $INC_h(n_i, n_{i+1})$ by $H_{\max}$. However, it is not possible for all edges along an optimal solution path P_{opt} to have an inconsistency of $H_{\max}$. To see why, notice that if $INC_H(n_i, n_{i+1}) = H_{\max}$ then it must be the case that $H(n_i) = H_{\max}$, $H(n_{i+1}) = 0$, and $\kappa(n_i, n_{i+1}) = 0$. Yet the fact that $H(n_{i+1}) = 0$ implies that $INC_H(n_{i+1}, n_{i+2})$ cannot equal anything except 0. This means that there is a gap between the bound on the inconsistency of an edge and the actual inconsistency. As a result, BFS^{g+H} will necessarily outperform the bound given in Lemma 6.3.8, and Corollaries 6.3.9 and 6.3.10.

6.4 Worst-Case Bound Accuracy

In this section, we describe a process for constructing graphs that demonstrate the tightness of the bound in Theorem 6.3.7 in the case of nrBFS^{g+H}. We will then show that this worst-case behaviour is not always happening in practice and use the worst-case graphs to better understand why.

6.4.1 Worst-Case Graphs

We begin by identifying a set of worst-case graphs on which nrBFS^{g+H} will find solutions that are arbitrarily close to the inconsistency of H along the optimal path, excluding the inconsistency on the first edge. In particular, a graph G_P can be constructed for any given path P with any possible heuristic values for the nodes on P(admissible or inadmissible), such that P is the optimal solution path in G_P and this worst-case behaviour is observed when using nrBFS^{g+H}. For example, Figure 6.2 shows such a graph that has been built around the path $P = [n_0, ..., n_6]$

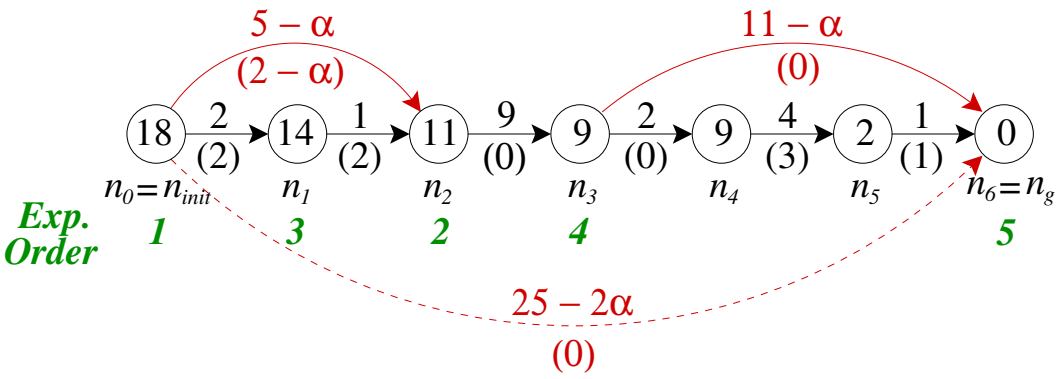


Figure 6.2: Example worst-case graph for nrBFS^{g+H}. The order in which nodes are selected for expansion is shown italicized and in green.

given previously in Figure 6.1. The order in which these nodes are expanded is shown italicized and in green.

This graph is given by the original path with the addition of two types of edges, both shown in red. The first type of edge, shown in solid red, offers suboptimal routes for the search to take around the portions of the optimal path on which there is heuristic inconsistency. However, the cost of these edges is not so expensive that the resulting path found to a node n does not prevent it from being expanded before ancestors of n on the optimal solution path that have a lower g-cost error. For example, notice the red edge from n_3 to n_6 which bypasses the edges (n_4, n_5) and (n_5, n_6) on which the heuristic is inconsistent. The cost of this red edge is given by the sum of $g^*(n_3, n_6)$ and the inconsistency of the heuristic on path $[n_4, n_5, n_6]$. The costs of these edges include a constant $\alpha \geq 0$ whose purpose is to ensure that the worst-case solution is found regardless of how ties are broken between nodes with equal values for $g + H$. If ties are broken in favour of the node with higher g-cost as is typical for such algorithms, then α can be set to 0.[1] For other tie breaking policies, α can be set as a positive value arbitrarily close to 0.

The second type of edge that has been added is shown as a dashed red arrow from n_0 to n_6. This edge offers an alternative path to the goal node that has no nodes in common with the optimal solution path except for n_0 and n_6. It is only needed when using parent pointer updating, the solution returned by nrBFS that makes such

[1]This tie-breaking policy is equivalent to favouring the node with the lower H-cost.

updates will outperform the bound if this alternative path is not included.

The resulting graph is one in which nrBFS^{g+H} will find a solution whose cost is equal to the sum of the optimal solution cost and the inconsistency of H along P from n_1 to n_6, or arbitrarily close to that value if the tie-breaking policy does not favour nodes with a higher g-cost. In particular, the solution found when using nrBFS^{g+H} on the graph in Figure 6.2 will cost $25 - 2\epsilon$, while C^* is 19 and the inconsistency of P from n_1 to n_k is 6.

The worst-case graph also demonstrates how solution suboptimality occurs as a result of heuristic inconsistency. Heuristic inconsistency causes nodes that are shallower on the optimal solution path to have a relatively higher heuristic value than those deeper on the path. The result is that the shallower node with a higher value "blocks" the algorithm from progressing along the optimal solution path. For example, this is what happens with node n_1 in Figure 6.2 as the heuristic value of n_1 relative to that of n_2 pushes the search to first pursue the suboptimal path around it. The result is that n_2 is expanded prior to n_1.

While the better path to n_2 will later be found, the g-cost improvement will not be propagated to the ancestors of n_2 when re-expansions are not performed. The result is that the value of $g + H$ for any ancestor n_a of n_2 will be inflated by the g-cost error of n_2. As such, suboptimal paths to the goal node — such as the one given by the red dashed edge — will look more promising relative to the paths through nodes on the optimal solution path.

The example in Figure 6.2 also demonstrates why the value of $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1})$ for edges on which H is consistent do not factor into the bound. To see this, consider the edge from n_2 to n_3. If the cost of this edge was 10 instead of 9 and the cost of dotted red edge was increased by 1, then nrBFS^{g+H} would find a solution of cost $26 - 2\epsilon$. This solution is still a cost of $6 - 2\epsilon$ larger than C^*, just like the solution found by nrBFS^{g+H} when $\kappa(n_2, n_3) = 9$. This would also be true for any $\kappa(n_2, n_3) \geq 2$ provided that the dotted red edge was adjusted accordingly. Intuitively, this shows that nrBFS^{g+H} cannot "make up" on g-cost error that has been accrued along optimal paths using an edge from n_i to n_{i+1} for which $H(n_i) - H(n_{i+1}) - \kappa(n_i, n_{i+1})$ is a negative number.

Below we will describe in more detail how such graphs can be constructed around an arbitrary given path.

Constructing Worst-Case Graphs

To construct such graphs around any given path $P = [n_0, ..., n_k]$ it is necessary to identify the inconsistent portions of P that do not include (n_0, n_1). This means that we will be looking for every maximally long subpath $[n_i, ..., n_j]$ of P where $1 \leq i < j \leq k$ on which all edges are inconsistent. Formally, this means that $[n_i, ..., n_j]$ satisfies the all of the following conditions:

1. $\forall i'$ where $i \leq i' < j$, $INC_H(n_{i'}, n_{i'+1}) > 0$.
2. Either $i = 1$ or $INC_H(n_{i-1}, n_i) = 0$.
3. Either $j = k$ or $INC_H(n_j, n_{j+1}) = 0$.

For example, consider the subpath $[n_4, n_5, n_6]$ in Figure 6.2. This is a maximally long inconsistent subpath since $INC_H(n_4, n_5) = 3$ and $INC_H(n_5, n_6) = 1$ (satisfying condition 1), $INC_H(n_3, n_4) = 0$ (satisfying condition 2), and n_6 is the last node on the path (satisfying condition 3).

For each such maximally long inconsistent subpath $[n_i, ..., n_j]$, an edge is added from n_{i-1} to n_j with a cost given as follows:

$$g^*(n_{i-1}, n_j) - \epsilon + \sum_{i'=i}^{j-1} INC_H(n'_i, n_{i'+1}) \ .$$

As an example, let us again consider $[n_4, n_5, n_6]$ in Figure 6.2. The edge added to go around this subpath will go from n_3 (which is the parent of n_4) to n_6, and the cost of this edge will be $g^*(n_3, n_6) - \epsilon + 4$. Since $g^*(n_3, n_6) = 7$, the cost of this edge is $11 - \epsilon$. This edge starts at n_3 and not n_4 so that the high heuristic value of n_4 relative to its descendents will block the search from progressing along the optimal solution path and push it to take the route around $[n_4, n_5, n_6]$. The other maximally long inconsistent subpath of $[n_1, n_2]$ is handled similarly, though notice that this subpath does not include n_0 since the inconsistency of (n_0, n_1) cannot impact the quality of solutions found.

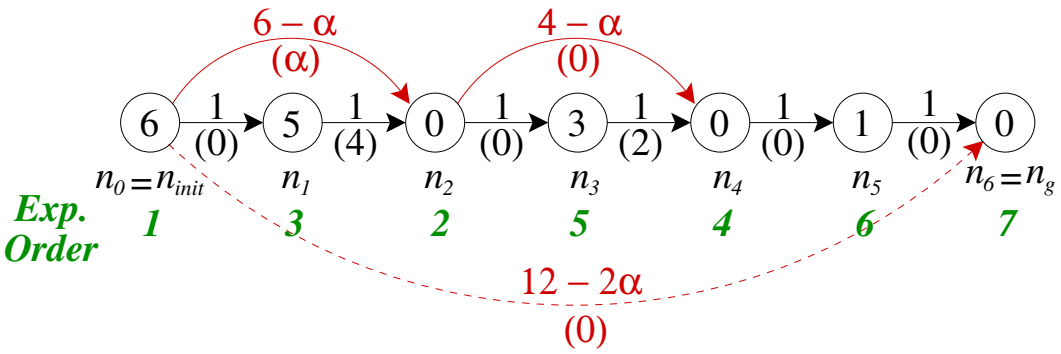


Figure 6.3: Example worst-case graph when H is admissible. The order in which nodes are selected for expansion is shown italicized and in green.

When using parent pointer updating, an additional edge is needed from n_0 to n_k whose cost is equal to the sum of C^* and the inconsistency of H along P_{opt} from n_1 to n_k, less $d \cdot \epsilon$ where d is the number of maximally long inconsistent subpaths of P_{opt}. For example, $d = 2$ in Figure 6.2.

6.4.2 Worst-Case Graphs for Admissible Heuristics

The graph construction procedure described above can also be used to generate graphs that have an admissible heuristic on which nrBFS^{g+H} will find solutions that are quadratic in C^* as suggested by Corollary 6.3.10. To do so, start with any path $[n_0, ..., n_k]$ that is to be the optimal solution path in the completed graph, where all edges on this path have a cost of at least θ for some constant $\theta > 0$. For each node n_i, we will assign the heuristic values as follows. If $i = 0$ or i is odd, set $H(n_i)$ as the cost from n_i to n_k along P. Otherwise, set $H(n_i)$ to 0. To this path add the edges as defined by the construction above. For example, given path $P = [n_0, ..., n_6]$ such that all the edges on P have a cost of 1, the result of this process is the graph with an optimal solution cost of 6 shown in Figure 6.3. In this graph, nrBFS will find the path of cost $12 - 2\epsilon$.

Now let us demonstrate that when using this assignment of heuristic values, nrBFS^{g+H} will find solutions that are quadratic in C^* when the edges are added according to the construction given above. To do so, consider the path $P = [n_0, ..., n_k]$ such that the edge costs between these nodes are all 1. When using the heuristic

value assignment described above, this means that $H(n_i) = k-i$ if i is 0 or odd, and $H(n_i) = 0$ if i is a positive even number. As a result, $INC_H(n_i, n_{i+1}) = k - i - 1$ if $i > 1$ if i is odd and $INC_H(n_i, n_{i+1}) = 0$ if i is even and $i > 0$. If k is odd, then the inconsistency along P is given by the following sum:

$$\sum_{j=1}^{k-1} INC_H(n_j, n_{j+1}) = (k-2) + 0 + (k-4) + 0 + ... + 1 + 0 \tag{6.28}$$

$$= \sum_{j=1}^{(k-1)/2} 2 \cdot j - 1 \tag{6.29}$$

$$= -(k-1)/2 + 2 \cdot \sum_{j=1}^{(k-1)/2} j \tag{6.30}$$

$$= -(k-1)/2 + 2[(k-1)/2][(k-1)/2 + 1]/2 \tag{6.31}$$

$$= (k-1)^2/4 \tag{6.32}$$

By Theorem 6.3.7, any solution found by nrBFS^{g+H} on a graph in which P is an optimal solution path will therefore cost no more than $C^* + (C^* - 1)^2/4 = (C^* + 1)^2/4$ since $C^* = k$.[2] By the graph construction described above, we can actually construct graphs of any size in which the solutions found by nrBFS^{g+H} will be arbitrarily close to this value. Therefore, the quadratic behaviour suggested by Theorem 6.3.10 is possible.

If $H(n_0)$ is changed from being equal to $h^*(n_0)$ to being equal to $h^*(n_0) - \alpha$ (or $h^*(n_0) - \mu$ for any $\mu \geq \alpha$), then even if nrBFS^{g+H} is enhanced by pathmax [63], this worst-case solution will still be found on such graphs. When using this technique, the heuristic value given to a node c generated by the expansion of p is given by the following:

$$\max(H(c), H(p) - \kappa(p, c)) \ .$$

This means that the heuristic value for c will be $H(c)$ unless $H(c) < H(p) - \kappa(p, c)$ or $H(p) - H(c) - \kappa(p, c) > 0$. This can only be true if $INC_H(p, c) > 0$.

When the modification to $H(n_0)$ described above is made, then all of the added red edges will have an inconsistency of 0. As such, pathmax will not increase the

[2] If k is even, a similar procedure can be used to derive a bound of $C^* \cdot (C^* + 2)/4$.

heuristic value of nodes generated along a red edge and so these nodes will still be expanded before their ancestors with less g-cost error. The result is that the same solutions are found whether pathmax is used or not.

6.4.3 The Worst-Case Bound in Example Domains

While the graphs just constructed demonstrate that the worst-case suggested by the bounds is possible, even nrBFS^{g+H} will not always find solutions of such quality. For example, consider any graph in which there is only a single solution path. In such a graph, any solution found will necessarily be optimal regardless of how much inconsistency there is along the optimal path.

Even in graphs with multiple solution paths, the bound will still often be outperformed. For example, let us reconsider the worst-case graphs given above. In these graphs, g-cost error is accrued by using the added (*ie.* red) edges to bypass inconsistent subpaths on the optimal solution. If the cost of these edges is not as high as defined in the construction, the endpoints of the edges are changed, or if these edges do not appear at all, nrBFS^{g+H} will find higher quality solutions.

As a result, nrBFS^{g+H} can often outperform the bound in problems that are not crafted specifically for inducing worst-case behaviour. For example, consider the case in which a weight 10 WA* is used on the pathfinding problems described above. In this domain, nrWA* finds solutions that are an average of 18% more costly than optimal, which is worse than the 12% solution suboptimality seen when nodes are re-expanded. However, this is substantially better than the upper bound of 475% suboptimality that is guaranteed by Theorem 6.3.7, where this value has been found by calculating the inconsistency of the heuristic over the optimal solution paths found by A*.

The bound is more accurate when the domain in question is given by Martelli's family of graphs. For any $M \geq 3$, this family includes a graph with $M + 1$ vertices on which A* will require $\Theta(2^M)$ node expansions to find a solution. If re-expansions are not performed, then the solutions found can be shown to be no larger than 1.5 times larger than the optimal solution.[3] In these graphs, the upper

[3] All results regarding the solution quality found by nrBFS when applied to Martelli's graphs are

bound given by Theorem 6.3.7 is $C + |V| - 2$ where C is the cost of the solution actually found. Since both C^* and C are $O(\Theta^{|V|})$, this deviation from the bound is insignificant.

6.5 Weighting with Bounding Functions

Recall that rWA* uses the evaluation function $g + w \cdot h$ for some admissible h. In Chapter 5, it was shown that the WA* evaluation function is of the form $g + B(h)$ where B is a bounding function and that for a large class of bounding functions, rBFS$^{g+B(h)}$ is guaranteed to satisfy B. In this section, we will use the theory developed above to find bounds for a BFS that uses an evaluation function of the form $g + B(h)$, regardless of the re-expansion policy being used. We will begin with a bound for BFS$^{g+B(h)}$ in the case that the heuristic h is admissible but inconsistent. This will be followed by an extension of a well-known result that any solution found by nrWA* will cost no more than $w \cdot C^*$ if the heuristic h being weighted is consistent. In particular, we will demonstrate that BFS$^{g+B(h)}$ will satisfy B provided that the rate of growth of B never slow downs.

Notice the heuristic H given by $H = B(h)$ will not necessarily satisfy the assumption made earlier that $H(n) = 0$ for any goal node n. However, $H = B(h)$ will satisfy a weaker assumption that $H(n) = b_0$ for any goal node n where b_0 is some non-negative constant (*ie.* $b_0 = B(0)$). The stronger assumption that all goal nodes have a heuristic value of 0 was used instead of this weaker assumption so as to maintain the clarity of the theorem statements. However, the bounds derived above can also be shown to be true even if only this weaker condition is assumed, and so these bounds will be used in the analysis below. For an overview of why these bounds apply even if only this weaker condition is assumed, see Section B.3 of Appendix B.

---

shown in Section A.3 of Appendix A.

6.5.1 Weighting An Inconsistent but Admissible Heuristic

Let us first consider a BFS$^{g+B(h)}$ search in which h is admissible, but perhaps inconsistent. This algorithm can be viewed as an instance of BFS^{g+H} where $H = B(h)$, which will allow us to use the theory developed above to find bounds for BFS$^{g+B(h)}$. Now let $P_{opt} = [n_0, ..., n_k]$ be an optimal solution path to a given task. Since h is admissible, this means that $h(n_i) \leq h^*(n_i) \leq C^*$ for any node $n_i \in P_{opt}$. If B is monotonically non-decreasing, this also means that $B(h(n_i)) \leq B(C^*)$ for any $n_i \in P_{opt}$. This allows us to immediately use Lemma 6.3.8 and Corollary 6.3.9 to show the following:

Theorem 6.5.1. *Let Γ be a task such that for any $(p, c) \in E$, $\kappa(p, c) \geq \theta$ for some constant θ where $\theta > 0$. If h is an admissible heuristic and B is a monotonically non-decreasing function such that for any $x \geq 0$, $B(x \geq x)$, then the following will be an upper bound on the cost of any solution returned by BFS$^{g+B(h)}$ on Γ:*

$$C^* \cdot B(C^*)/\theta \ .$$

For example, this result guarantees that any result found by nrWA* will have a cost of no more than $w \cdot (C^*)^2/\theta$ even if the heuristic being weighted is inconsistent.

6.5.2 Bounding B-Consistent Heuristics

While the previous bound applies regardless of how inconsistent the admissible heuristic h is, we will now show that for many bounding functions we can also guarantee that BFS$^{g+B(h)}$ will satisfy B if h is consistent, regardless of the re-expansion policy in use. To demonstrate this fact, we will first require the following definition:

Definition 6.5.2. *Let B be a bounding functions such that for all $x \geq 0, B(x) \geq x$ and B is monotonically non-decreasing. Given nodes p and c where $c \in succ(p)$, a heuristic H is said to be B-**consistent** on edge (p, c) if the following is true:*

$$H(p) - H(c) \leq B(h^*(p)) - B(h^*(c)) \ .$$

Like the standard definition of heuristic consistency, this property describes limitations on how much the heuristic can decrease from parent to child. Notice that if B is the optimal bounding function $B(x) = x$ and c is along the lowest-cost path from p to a goal node, then this condition simplifies as follows:

$$\begin{aligned} H(p) - H(c) &\leq B(h^*(p)) - B(h^*(c)) \\ &\leq h^*(p) - h^*(c) \\ &\leq \kappa(p, c) \ . \end{aligned}$$

which is the standard definition of consistency.

In the following theorem, we show that if a heuristic is B-consistent for every edge along some optimal path, then BFS^{g+H} will satisfy B, regardless of the re-expansion policy being used.

Theorem 6.5.3. *Let B be a bounding function such that $\forall x \geq 0, y \geq 0, B(x+y) \geq B(x) + y$, and let $P_{opt} = [n_0, ..., n_k]$ be an optimal solution path to a task Γ. If the heuristic function H is B-consistent on every edge (n_i, n_{i+1}) where $1 \leq i < k$, then BFS^{g+H} will satisfy B.*

Proof. Let B and P_{opt} be defined as described above, and let H be a heuristic that is B-consistent on every edge (n_i, n_{i+1}) where $1 \leq i < k$. This proof will work as follows: first we will show that under these conditions, the inconsistency of H along P_{opt} will be at most $B(C^*) - C^*$, and then we will use Theorem 6.3.7 to show that any solution found will cost no more than $B(C^*)$.

The bound on the inconsistency of H along P_{opt} will be shown by induction. In particular, we will show that the inconsistency of H along $[n_0, ..., n_j]$ is bounded by $B(h^*(n_0)) - B(h^*(n_j)) - g^*(n_0, n_j)$ for any j where $0 \leq j \leq k$.

Let us begin with the base case of $j = 1$. When $j = 1$, the path consists solely of the edge (n_0, n_1) so the sum of the inconsistency along this path is given by $INC_H(n_0, n_1)$. There are now two cases to consider. In the first, $INC_H(n_0, n_1) = 0$. To show that $B(h^*(n_0)) - B(h^*(n_1)) - g^*(n_0, n_1) \geq INC_H(n_0, n_1)$ notice that $B(h^*(n_0)) > B(h^*(n_1)) + g^*(n_0, n_1)$ by the assumption made about B and the fact that $h^*(n_0) = h^*(n_1) + g^*(n_0, n_1)$ since n_1 is along the optimal solution path P_{opt}

from n_0 to the nearest goal node. As $INC_H(n_0, n_1) = 0$ in this case, this shows that $B(h^*(n_0)) - B(h^*(n_1)) - g^*(n_0, n_1)$ is at least as large as $INC_H(n_0, n_1)$. Thus, this case is handled.

In the second case, $INC_H(n_0, n_1) > 0$ and so $INC_H(n_0, n_1) = H(n_0) - H(n_1) - \kappa(n_0, n_1)$. We can now perform the following derivation:

$$\begin{aligned} INC_H(n_0, n_1) &= H(n_0) - H(n_1) - \kappa(n_0, n_1) \\ &\leq B(h^*(n_0)) - B(h^*(n_1)) - g^*(n_0, n_1) \ . \end{aligned}$$

This last line holds since H is B-consistent on this edge and by the fact that $g^*(n_0, n_1) = \kappa(n_0, n_1)$ since n_0 and n_1 are on P_{opt}. Therefore, the base case holds.

Now assume that the statement is true for the path $[n_0, ..., n_j]$ and consider $[n_0, ..., n_j, n_{j+1}]$. Once again, there are two cases. In the first, $INC_H(n_j, n_{j+1}) = 0$. The inconsistency along $[n_0, ..., n_j, n_{j+1}]$ will therefore be the same as the inconsistency along $[n_0, ..., n_j]$, and so $B(h^*(n_0)) - B(h^*(n_j)) - g^*(n_0, n_j)$ will bound the inconsistency along $[n_0, ..., n_j, n_{j+1}]$ by the induction hypothesis. This case will then be handled by the fact that $B(h^*(n_0)) - B(h^*(n_{j+1})) - g^*(n_0, n_{j+1})$ is no smaller than $B(h^*(n_0)) - B(h^*(n_j)) - g^*(n_0, n_j)$ due to the following:

$$\begin{aligned} B(h^*(n_j)) + g^*(n_0, n_j) &= B(\kappa(n_j, n_{j+1}) + h^*(n_{j+1})) + g^*(n_0, n_j) && (6.33) \\ &\geq B(h^*(n_{j+1})) + \kappa(n_j, n_{j+1}) + g^*(n_0, n_j) && (6.34) \\ &\geq B(h^*(n_{j+1})) + g^*(n_0, n_{j+1}) \ . && (6.35) \end{aligned}$$

Line 6.33 holds since $h^*(n_j) = \kappa(n_j, n_{j+1}) + h^*(n_{j+1})$ by the fact that n_{j+1} is the child of n_j along an optimal path to the nearest goal node from n_j. Line 6.34 follows by the assumption made on B. The final line holds due to the fact that $g^*(n_0, n_{j+1}) = \kappa(n_j, n_{j+1}) + g^*(n_0, n_j)$ since an optimal path from n_0 to n_{j+1} passes through n_j. Therefore, the statement holds if $INC_H(n_j, n_{j+1}) = 0$.

Now suppose that $INC_H(n_j, n_{j+1}) > 0$. In this case, we can perform the fol-

lowing derivation:

$$\sum_{i=0}^{j} INC_H(n_i, n_{i+1})$$

$$= INC_H(n_j, n_{i+j}) + \sum_{i=0}^{j-1} INC_H(n_i, n_{i+1}) \tag{6.36}$$

$$= H(n_j) - H(n_{j+1}) - \kappa(n_j, n_{j+1}) + \sum_{i=0}^{j-1} INC_H(n_i, n_{i+1}) \tag{6.37}$$

$$\leq B(h^*(n_j)) - B(h^*(n_{j+1})) - \kappa(n_j, n_{j+1}) + \sum_{i=0}^{j-1} INC_H(n_i, n_{i+1}) \tag{6.38}$$

$$\leq B(h^*(n_j)) - B(h^*(n_{j+1})) - \kappa(n_j, n_{j+1})$$
$$+ B(h^*(n_0)) - B(h^*(n_j)) - g^*(n_0, n_j) \tag{6.39}$$

$$\leq B(h^*(n_0)) - B(h^*(n_{j+1})) - g^*(n_0, n_{j+1}) \ . \tag{6.40}$$

The first line follows by expanding the summation. Line 6.37 then holds since $INC_H(n_j, n_{i+j}) > 0$, while line 6.38 follows by the assumption that H is B-consistent on every edge along P_{opt}. Line 6.39 holds by the induction hypothesis. The final line then follows since $\kappa(n_j, n_{j+1}) + g^*(n_0, n_j)$ is equal to $g^*(n_0, n_{j+1})$. Having handled all cases, we have shown that the inconsistency of H along $[n_0, ..., n_j]$ will be at most $B(h^*(n_0)) - B(h^*(n_j)) - g^*(n_0, n_j)$.

Notice how this statement applies when $j = k$. Since $h^*(n_0) = g^*(n_0, n_k) = C^*$, this means that the inconsistency along P_{opt} is no more than $B(C^*) - B(h^*(n_k)) - C^*$. As $B(h^*(n_k)) = B(0) \geq 0$, this means that the value of this inconsistency is at most $B(C^*) - C^*$.

Now this is the inconsistency along the entire length of P_{opt}, which is an upper bound on the portion from n_1 to n_k since $INC_H(n_0, n_1) \geq 0$. Therefore, $B(C^*) - C^*$ is an upper bound on the inconsistency along the relevant portion of the path needed to apply Theorem 6.3.7. That theorem states that the solution found by BFS^{g+H} will therefore be no larger than $B(C^*) - C^* + C^* = B(C^*)$, and so the theorem is proven. □

Having shown that BFS^{g+H} will satisfy B if H is B-consistent, we now identify a set of bounding functions for which $B(h)$ is B-consistent if h is consistent.

6.5.3 Bounding Functions with a Non-Decreasing Rate of Growth

Recall that the evaluation function used by nrWA* is of the form $g + H$ where $H = w \cdot h$ for some admissible heuristic h. If h is also consistent, then notice the following is true where $c \in succ(p)$:

$$\begin{aligned} H(p) - H(c) - \kappa(n_i, n_{i+1}) &= w \cdot h(p) - w \cdot h(c) - \kappa(p, c) && (6.41) \\ &= w \cdot (h(p) - h(c)) - \kappa(p, c) && (6.42) \\ &\leq w \cdot \kappa(p, c) - \kappa(p, c) && (6.43) \\ &\leq (w - 1) \cdot \kappa(n_i, n_{i+1}) \ , && (6.44) \end{aligned}$$

where the third line of this derivation holds since $h(n_i) \geq h(n_{i+1}) - \kappa(n_i, n_{i+1})$.

Now suppose that c is along a path from p to the nearest goal node. In that case, $h^*(p) - h^*(c) = \kappa(p, c)$. This also means that $w \cdot h^*(p) - w \cdot h^*(c) = w \cdot \kappa(p, c)$. This can alternatively be written as $B_w(h^*(p)) - B_w(h^*(c)) = w \cdot \kappa(p, c)$ for the linear suboptimality bounding function $B_w(x) = w \cdot x$. When this is substituted in to the inequality in line 6.44, we see that if c is along a path from p to the nearest goal node then the following is true:

$$H(p) - H(c) \leq B_w(h^*(p)) - B_w(h^*(c)) \ .$$

This of course means that the heuristic $H = w \cdot h$ is B_w-consistent along any edges on an optimal solution path provided that h is consistent on this edges. We can now use Theorem 6.5.3 to show that any solution found by BFS$^{g+w \cdot h}$ will cost no more than $B_w(C^*) = w \cdot C^*$ if h is consistent As such, this proves the known bound on the solutions found by nrWA* when the heuristic being weighted is consistent.

The linear suboptimality bounding function is not the only bounding function for which BFS$^{g+B(h)}$ will find solutions of at most $B(C^*)$ regardless of the re-expansion policy in use. In particular, we can use Theorem 6.5.3 to show that this will be true of any bounding function whose rate of growth is non-decreasing. This is formalized by the following theorem:

Theorem 6.5.4. *BFS$^{g+B(h)}$ will satisfy B if all of the following conditions hold:*

1. *h is consistent.*

2. $\forall x \geq 0, y \geq 0, B(x+y) \geq B(x) + y$.

3. $\forall x \geq 0, y \geq 0, z \geq 0, x \geq y \Rightarrow B(x+z) - B(y+z) \geq B(x) - B(y)$.

Proof. Let $P_{opt} = [n_0, ..., n_k]$ be an optimal solution path, h be a consistent heuristic, and assume B satisfies the conditions given above. We will now show that $B(h)$ is B-consistent on every edge on P_{opt}. To do so, consider any pair of consecutive nodes n_i and n_{i+1} on P_{opt} where $0 \leq i < k$.

There are two cases to be handled. In the first, $h(n_i) < h(n_{i+1})$. In that case, $B(h(n_i)) - B(h(n_{i+1})) < 0$ since B is monotonically non-decreasing. However, as $h^*(n_i) \geq h^*(n_{i+1})$, this similarly means that $B(h^*(n_i)) - B(h^*(n_{i+1})) > 0$. Therefore, $B(h(n_i)) - B(h(n_{i+1})) \leq B(h^*(n_i)) - B(h^*(n_{i+1}))$ and so $B(h)$ is B-consistent on edge (n_i, n_{i+1}).

In the second case, $h(n_i) \geq h(n_{i+1})$ and so $B(h(n_i)) - B(h(n_{i+1})) \geq 0$. This allows for the following derivation:

$$\begin{aligned} B(h(n_i)) - B(h(n_{i+1})) &\leq B(h(n_i) + h^*(n_{i+1}) - h(n_{i+1})) \\ &\quad - B(h(n_{i+1}) + h^*(n_{i+1}) - h(n_{i+1})) \qquad (6.45) \\ &\leq B(h(n_{i+1}) + \kappa(n_i, n_{i+1}) + h^*(n_{i+1}) - h(n_{i+1})) \\ &\quad - B(h^*(n_{i+1})) \qquad (6.46) \\ &\leq B(h^*(n_i)) - B(h^*(n_{i+1})) \ . \qquad (6.47) \end{aligned}$$

Since h is consistent, $h^*(c) - h(c) \geq 0$ and so line 6.45 holds by the assumption that property 3 holds for B. Line 6.46 then holds since h is consistent which implies that $h(n_i) \leq h(n_{i+1}) + \kappa(n_i, n_{i+1})$. The final line then holds by the fact that $\kappa(n_i, c) + h^*(n_{i+1}) = h^*(n_i)$ since n_{i+1} is the child of n_i along P_{opt}.

Having handled both cases, the heuristic $B(h)$ is B-consistent on all edges along P_{opt}. Therefore, BFS$^{g+B(h)}$ satisfies B by Theorem 6.5.3. □

Let us now look at the third condition on B specified by this theorem. If B is twice differentiable and for all $x \geq 0, B''(x) \geq 0$, then condition 3 will necessarily hold. This means that BFS$^{g+B(h)}$ will satisfy B if the rate of growth of this function is never decreasing, regardless of the re-expansion policy in use. As such,

the theorem identifies that for bounding functions such as the linear suboptimality bounding function $B_w(x) = w \cdot x$ and other fast growing bounding functions like $B_p(x) = \max(x, x^p)$ for $p \geq 1$ and $B_a(x) = \max(x, a^x)$ for $a > 1$, a BFS$^{g+B(h)}$ can be set to never re-expand nodes and it will still satisfy the bound provided that h is consistent. Thus, the theorem extends the well-known bound which was known for nrWA* to apply to both other bounding functions and any re-expansion policy.

6.5.4 Bounding Functions with a Decreasing Rate of Growth

While Theorem 6.5.4 can be used to show that nrBFS$^{g+B(h)}$ will satisfy B if B never slows the rate at which it grows, it cannot be applied in the case of bounding functions that do slow the rate at which they grow. This is not simply a matter of the argument not applying to such graphs, as we can construct graphs for a given slow-growing bounding function for which B is not satisfied. For example, consider the following bounding function:

$$B_{\log_2}(x) = \begin{cases} x + \log_2 x, & \text{if } x \geq 1 \\ x, & \text{if } x < 1 \end{cases} .$$

Figure 6.4 shows an example in which nrBFS$^{g+B_{\log_2}(h)}$ will find a solution that is worse than $B_{\log_2}(C^*)$ regardless of the tie-breaking policy in use and whether or not parent pointer updates are performed. In the figure, the nodes on the optimal path $[n_0, n_1, n_2, n_3, n_4]$ are labelled where $n + 0 = n_{init}$ and the goal node is $n_4 = n_g$. The cost of this path is 14. The value of the consistent heuristic h is shown inside the circle representing a node and the expansion order is shown italicized and in green. The evaluation of each node according to $g + B_{\log_2}(h)$ at both the time the node is first generated when it is expanded (since it will be the same in all cases) is shown in the figure, and is denoted by Φ.

The path found by nrBFS$^{g+B_{\log_2}(h)}$ is the rightmost path from top to bottom which has a cost of 18. This is more suboptimal than is allowed by the bounding function which requires that any solution returned have a cost of no more than $B_{\log_2}(C^*) \approx 17.8$.

The issue that nrBFS^{g+H} has with such bounding functions with a slowing rate of growth is that the amount of inconsistency introduced on edges between nodes

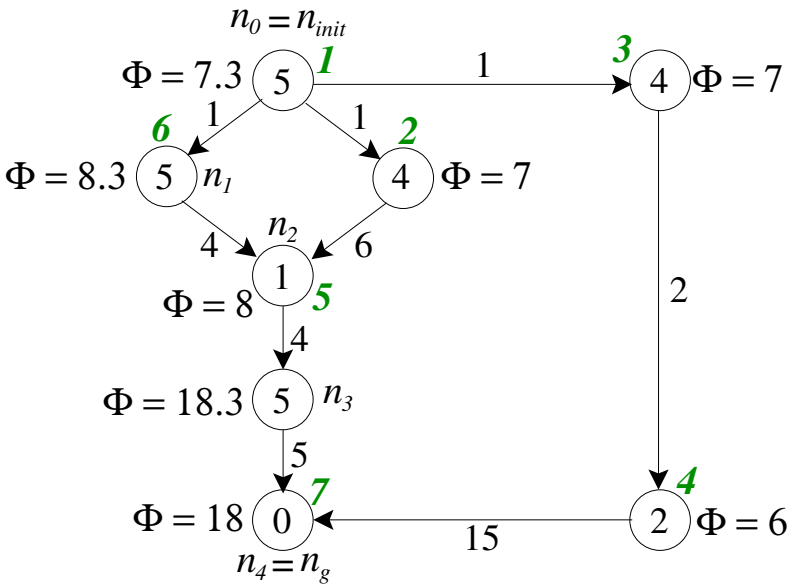


Figure 6.4: nrBFS$^{g+B_{\log_2}(h)}$ does not satisfy the bound in the given graph. The expansion order is shown italicized and in green and the value of $\Phi(n) = g + B_{\log_2}(h(n))$ is shown for each node at the time it is first generated/expanded.

that have a low heuristic value is larger than the inconsistency introduced on edges between nodes with a higher heuristic value. To see this, consider the edge (n_1, n_2) in Figure 6.4 and notice that the following is true on this edge:

$$h(n_1) - h(n_2) = h^*(n_1) - h^*(n_2) = \kappa(n_1, n_2) = 4 \ .$$

Now consider the difference in the H values of these nodes where $H = B_{\log_2}(h)$:

$$H(n_1) - H(n_2) \approx 7.3 - 1 = 6.3 \ .$$

In contrast, notice that $h^*(n_1) = 13$, $h^*(n_2) = 9$, and so the following is also true:

$$B_{\log_2}(h^*(n_1)) - B_{\log_2}(h^*(n_2)) \approx 16.7 - 12.2 = 4.5 \ .$$

The inconsistency along this edge introduced by the bounding function in the case that $h = h^*$ would therefore be less than the amount introduced as is. Moreover, $H(n_1) - H(n_2) > B(h^*(n_1)) - B(h^*(n_2))$ and so the heuristic $H = B_{\log_2}(h)$ is not $B_{\log_2}$-consistent on this edge.

While nrBFS$^{g+B(h)}$ will not satisfy B in general if B is a bounding function with a decreasing rate of growth, there are cases in which we can show that it will.

One of these is if there is an optimal solution path along which the heuristic values are monotonically non-decreasing. This is formalized by the following theorem:

Theorem 6.5.5. *BFS$^{g+B(h)}$ will satisfy B if all of the following conditions hold:*

1. *h is consistent.*
2. *$\forall x \geq 0, y \geq 0, B(x+y) \geq B(x) + y$.*
3. *There exists an optimal solution path $[n_0, ..., n_k]$ such that $h(n_i) \geq h(n_{i+1})$ for all $0 \leq i < k$.*

The proof of this theorem can be found in Section A.4 of Appendix A. Notice that in the case of $B_{\log_2}$, the theorem guarantees that if h is consistent and monotonically non-decreasing along an optimal solution path, then nrBFS$^{g+B_{\log_2}(h)}$ is guaranteed to find a solution that satisfies $B_{\log_2}$. As such, the reason why nrBFS$^{g+B_{\log_2}(h)}$ does not satisfy the bound in Figure 6.4 is not solely due to the edge (n_1, n_2) on which the $B_{\log_2}(h)$ is not $B_{\log_2}$-consistent. The problem nrBFS$^{g+B_{\log_2}(h)}$ faces in this graph is that there are edges like (n_1, n_2) which are highly inconsistent and there are edges like (n_2, n_3) on which the heuristic increases, thus allowing there to be further inconsistency further on down the optimal path when the heuristic later decreases. For example, in the case of Figure 6.4, the increase in the heuristic along edge (n_2, n_3) allows for another highly inconsistent edge in (n_3, n_4).

Note that Theorem 6.5.5 identifies one set of heuristic conditions in which nrBFS$^{g+B(h)}$ can be used safely to satisfy B even for bounding functions with a decreasing rate of growth. Identifying others remains a topic for future work.

6.6 Chapter Summary

This chapter presents a formal analysis of the impact that not re-expanding nodes can have on the quality of solutions returned by a best-first search. In doing so, we have increased our understanding of how this technique can affect the performance of a search algorithm. In particular, the inconsistency of a heuristic along an optimal solution path is shown to bound the suboptimality of the solutions found when not

performing re-expansions, regardless of the re-expansion policy being used. This bound was shown to be tight in the case of nrBFS through the introduction of a family of worst-case graphs on which this algorithm finds solutions with a cost that is equal to the upper bound. The bound was then used to show that the impact of not re-expanding nodes in A* would be at worst a quadratic decrease in the quality of solutions found. Finally, we considered evaluation functions of the form $g + B(h)$ where B is a bounding function and h is an admissible heuristic. In doing so, we identified a set of bounding functions for which nrBFS could be guaranteed to satisfy B provided that the heuristic being weighted was consistent. The bound was also used to derive worst-case bounds in the case that the heuristic being weighted was not consistent.

Chapter 7

Conclusion

The problem we have considered in this dissertation is that facing a system designer constructing a suboptimal heuristic search-based system: there is a large space of design choices to be made and it is often not obvious how to make these choices effectively. The goal in this thesis has been to better identify the space of options available to such a system designer, improve our understanding of how certain design decision impact search, and consider how to handle a large design space when problem-solving must begin immediately without knowing much about the problem at hand. In this chapter, we summarize the contributions made in this dissertation and then describe several directions for future research.

7.1 Contributions

In this section, we briefly describe the contributions made in this dissertation.

7.1.1 Multi-Core Planning with a Portfolio

In Chapter 3, it was demonstrated that we can deal effectively with a large space of design options in the case that little is known ahead of time about the problems to be solved through the use of an algorithm portfolio. We do so in building the shared memory, multi-core `ArvandHerd` planner. This planner was built specifically to avoid the memory issues that can arise when trying to use a portfolio on a shared memory machine. `ArvandHerd` was the winner of the multi-core track of both the 2011 and 2014 International Planning Competitions. The study in Chap-

ter 3 identifies several of the design decisions that can introduce variance into the performance of the different components of the ArvandHerd portfolio, Arvand and LAMA-2008, and demonstrates that the strength of ArvandHerd is due to it being able to mix and match the strengths of multiple techniques. As part of this investigation, we also developed an effective parallel version of the random walk-based Arvand planner.

7.1.2 Random Exploration and Random Baselines

In Chapter 4 we considered the use of random exploration when added to a GBFS search. In particular, we introduced ϵ-greedy node selection as a simple technique for isolating the impact that adding random exploration has on the performance of a planner. This technique demonstrates that adding variation into GBFS using random exploration can greatly improve the performance of the algorithm, as it improves the coverage of both basic planners and a state-of-the-art planner in LAMA-2011. This investigation thereby identifies random exploration as an option for system designers to consider using when building a GBFS-based planner.

We also introduced heuristic perturbation as a second simple technique for adding random exploration into a search. This approach is riskier than ϵ-greedy node selection, as it may force the search to disregard nodes with low heuristic values for a long time. However, this riskiness means that the technique pairs well with ϵ-greedy node selection when they are used together in a portfolio.

Finally, we argued that given the positive impact seen when adding random exploration, it is necessary to evaluate GBFS enhancements that are based on problem structure against random baselines to ensure that the improvements seen with these techniques is not merely the result of the enhancements varying the search in a way that is equivalent to random variation. In particular, we suggest testing such enhancements against equivalent techniques that replace the influence of the problem structure with randomness. By doing so, we are more clearly isolating and identifying the impact of that structure. The resulting study provided further confirmation of the value of two existing techniques: preferred operators and multi-heuristic best-first search.

7.1.3 Satisfying Alternative Solution Quality Requirements

In Chapter 5, we introduced the concept of a bounding function to allow for a system designer to have greater choice in the definition of solution quality requirements. We then analyzed existing algorithm frameworks and showed that if they are modified appropriately, then they can be used to satisfy a large class of possible bounding paradigms aside from the commonly used linear suboptimality bound. The algorithm frameworks considered are anytime algorithms, best-first search algorithms, iterative deepening algorithms, and focal list based algorithms. In doing so, we have not only given system designers the opportunity to define solution quality requirements as they wish, but more clearly identify the set of algorithm options that are available to them for a given requirement. Based on other properties of the domain, the best suited algorithm can then be used.

To demonstrate that the modifications do lead to effective algorithms for bounding functions aside from the linear suboptimal bound, these algorithms were each tested in the types of problems they are best suited for when using a additive bound. In all cases, the new versions of these algorithms were shown to effectively trade-off solution quality for improved run-time.

7.1.4 Analyzing the Impact of Not Re-Expanding Nodes on Solution Quality

In Chapter 6, we formally analyze the impact that not using the full re-expansion policy in a best-first search has on the quality of solutions. In doing so, we increase our understanding of the never re-expand technique often used with best-first search and thereby better equip a system designer in their decision of whether or not to configure their system to re-expand nodes or not.

Our particular contributions are as follows. First, we show that the worst-case solution quality when using any re-expansion policy in a BFS search is bound by a measure of the inconsistency of the heuristic along the optimal solution path. This bound is then shown to be tight through the description of a process for constructing worst-case graphs in which the solutions found are arbitrarily close to those given by the bound. We then consider the case in which the heuristic used by the algorithm

involves weighting an admissible heuristic using some bounding function B. In this analysis, we show that any solution found when not using the full re-expansion policy will have a cost of at most $C^* \cdot B(C^*)$. This means that if the heuristic is admissible, any solution found will be at most quadratic in the optimal solution cost, while nrWA* will find solutions that cost at most $w \cdot (C^*)^2$. Finally, we identify a set of bounding functions for which weighting a consistent heuristic is still guaranteed to satisfy the given solution quality requirements.

7.2 Future Directions For Research

Dealing with large spaces of design choices will continue to be an important challenge when building high performance heuristic search-based systems. As such, it continues to be important to improve our understanding of the space of design choices available, how these decisions impact performance, and how to best combine the strengths of different design decisions. In this section, we describe directions for future research that build upon the contributions in this thesis.

7.2.1 Improving Portfolio Construction

As mentioned in Chapter 3, the `ArvandHerd` portfolio was initially selected manually prior to the 2011 International Planning Competition. Since that competition, other researchers have introduced effective techniques for automatically generating effective sets of parameters for a configurable system like `Arvand` and `LAMA-2008` that can then be used in a portfolio [81]. Using such an approach would be expected to further improve the performance of a system like `ArvandHerd`.

Further analysis is still needed into what domains are still not handled well by `ArvandHerd`. Such an investigation would be expected to help identify what other planning approaches should be included alongside best-first search and random walk-based search going forward. The inclusion of ϵ-greedy node selection and heuristic perturbation should also be considered.

7.2.2 Sharing Candidate Paths in `ArvandHerd`

While we have shown that `ArvandHerd` can effectively combine the strengths of different planning approaches across different problems, it is often the case that different techniques are each better suited for different parts of the same state-space. To better handle such problems, ideally the different approaches could share information about progress made. In `ArvandHerd`, the information shared could take the form of the candidate paths found. `Arvand` generates candidate paths through its search episodes and stores the best of these paths in a walk pool. Every node in `LAMA`'s open list represents a candidate path which can be easily extracted. By adding candidate paths from `LAMA`'s open list into `Arvand`'s walk pool, and by adding nodes from the `Arvand` search episodes into `LAMA`'s open list, these two techniques may be able to share the progress each has made. The result will be that random walk-based search may be used in regions of the state-space that are only reached by `LAMA`, and vice versa. Doing so may improve the performance of the planner if each of these approaches is best used in different regions of the space.

7.2.3 Characterizing the Impact of Random Exploration

While ϵ-greedy node selection has been shown to improve the performance of GBFS in some domains, a better understanding is needed into what form of heuristic error this approach is best able to help with. A similar investigation is needed into better understanding the heuristic perturbation. A related question is whether it is possible to automatically analyze a given domain and to use the collected information to inform us on whether exploration is needed, and if so, how much should be included (*ie.* what should be the value for ϵ or the noise level).

7.2.4 Adding Random Exploration to Other Algorithms

Further study is needed to determine if random exploration can also help improve the performance of other algorithms, both when solution quality guarantees must be satisfied and when any solution will suffice. For example, recall that A^*_w greedily exploits the heuristic from amongst those nodes in the focal list. Preliminary

results suggest that modifying this approach to use ϵ-greedy node selection from amongst those nodes in the open list can also improve search. Similarly, **beam search** algorithms may also benefit from not always selecting the nodes with the lowest heuristic values for expansion.

7.2.5 Satisfying Alternative Bounding Functions in Other Algorithm Frameworks

In Chapter 5, we showed how existing algorithm frameworks could be modified to satisfy a given bounding function. However, there are still several algorithm frameworks which currently can only be used to satisfy linear suboptimality bounds. For example, the theory provided does not immediately apply to bi-directional search, though Rice and Tsotras [73] have demonstrated that bi-directional search could be made to satisfy the additive bounding function by changing its termination criteria. Generalizing this criteria so as to allow for other types of bounds represents an important step in better understanding what algorithms are available to be used when satisfying a given bounding function. The same is also true of the existing parallelizations of rOCL algorithms that are known to either be optimal or to satisfy linear suboptimality bounds [5, 49].

7.2.6 Estimating Execution Time of Suboptimal Algorithms

In recent years, there has been extensive work on estimating the runtime of optimal algorithms, particularly IDA* [54, 56, 108]. These techniques have not been used much for predicting the performance of suboptimal algorithms. There are some limitations to doing so, since it is unclear how well these techniques will work when the heuristics in use are inconsistent. However, if this line of research could be effectively applied towards predicting the performance of suboptimal algorithms at a fixed solution quality requirement, such prediction techniques may allow for automatic methods for effectively making design decisions between different evaluation functions, heuristics, or algorithm frameworks.

7.2.7 Preventing Re-Expansions in Other Algorithms

To satisfy a given bounding function B, focal list based algorithms are required to use the full re-expansion policy. The only known bound when not doing so is that the solution found by nrA$^*_w$ is exponential in w if the heuristic being weighted is consistent [15]. It is currently not known how to bound the search if the focal list is built using other bounding functions, or if the heuristic is not consistent.

Another possible direction for research is into developing other re-expansion policies to be used with focal list based algorithms that will allow them to again satisfy a desired bounding function. Our current hypothesis is that the g-cost error of nodes in the open list can also be bound by the inconsistency along the optimal paths to them if the search re-expands a node whenever the paths found to it would not have been found in that order by nrBFS. This may allow the results in Chapter 6 to be extended to this other framework. It is expected that a similar approach may also be used to develop policies for when to re-expand a node during an iterative deepening search that uses a transposition table, while still guaranteeing some given solution quality requirements.

Determining whether the analysis in Chapter 6 also extends to this and other parallel OCL algorithms also remains as future work. It will also be necessary to better understand the relationship between our analysis and recent work by Phillips, Likhachev, and Koenig [70] who demonstrate how to use a never re-expand policy in a parellel version of WA* search while still guaranteeing that the linear suboptimality bound is satisfied.

7.2.8 The Impact of Not Re-Expanding on Runtime

While we have introduced worst-case analysis of the impact of not re-expanding nodes on the quality of solutions found on the algorithm, there is currently no theoretical analysis of the impact that it will have on the runtime. In our experience with WA*, not re-expanding nodes tends to harm performance for low weight values that are larger than 1, but improve performance for higher weight values. However, this behaviour has not been formally characterized. The analysis in Chapter 6 may aid

in such research since it identifies what is the maximum value that the evaluation function reach for all nodes that will be expanded.

7.2.9 Domain Specific Bounding

Most of the bounds given in Chapters 5 and 6 make no assumptions about the state-space or the heuristic. As such, the worst-case bounds derived often greatly underestimates the quality of solutions actually found in practice. Identifying properties of the state-space for which these bounds can be tightened could therefore have substantial practical consequences by allowing for the use of more greedy algorithms.

Another open question is in regards to finding lower-bounds of the optimal solution cost after a suboptimal solution has been found. Currently, the only way to do this is to consider all of the nodes in the open list and unopened lists at the end of the search. Whether state-space or heuristic properties can be used to improve these estimates is unknown.

7.3 Summary

In this dissertation, we have considered several aspects of the problem facing someone developing a heuristic search-based system: there is a large set of algorithm options and design choices that need to be set before the system can be deployed and each of these can greatly impact performance. Our contributions have been in more clearly identifying the options available, improving our understanding of the impact of some of these options, and in demonstrating the effectiveness of using an algorithm portfolio to deal with a large space of design choices when problem-solving must begin without knowing much about the task to solve.

In terms of identifying the set of options available, we demonstrated that the use of random exploration in a GBFS-based system can effectively improve the performance of such systems. We have therefore clearly identified this option as one that designers developing such systems should consider.

A second contribution made towards better identifying the set of options available concerns the case in which all solutions found must satisfy some given quality

requirements. In particular, we have shown that four existing algorithm frameworks can be made to satisfy even non-traditional forms of requirements. In doing so, we allow the system designer the choice to select the most suitable of these frameworks for the tasks at hand while still satisfying desired bounds.

We then focused on the best-first search framework and considered the impact that not re-expanding nodes can have on solution quality. In particular, we showed that the suboptimality of any solution found when not re-expanding nodes can be bound in terms of a measure of the inconsistency of the heuristic along the optimal solution paths. This investigation furthers our understanding of how this decision decision can impact performance. As part of this work, we also identified when a bounding function B could be used to weight a consistent heuristic such that B will still be satisfied even when not re-expanding nodes.

Finally, we demonstrated that an algorithm portfolio could be used effectively in the situation which occurs in automated planning in which little is known about a given task before problem-solving begins. In particular, we described how an effective portfolio-based planner could be constructed for use on a shared-memory architecture. The resulting planner, `ArvandHerd`, not only won two consecutive planning competitions, but was also shown to outperform several state-of-the-art planners even in the case that they could be effectively parallelized.

Bibliography

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[2] Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–656, 1995.

[3] Neil Burch, Robert C. Holte, Martin Müller, David O'Connell, and Jonathan Schaeffer. Automating Layouts of Sewers in Subdivisions. In *19th European Conference on Artificial Intelligence*, pages 655–660, 2010.

[4] Ethan Burns, Wheeler Ruml, and Minh Binh Do. Heuristic Search When Time Matters. *Journal of Artificial Intelligence Research*, 47:697–740, 2013.

[5] Ethan Burns, Wheeler Ruml, Sofia Lemons, and Rong Zhou. Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research*, 39:689–743, 2010.

[6] Tom Bylander. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[7] Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. Learning Predictive Models to Configure Planning Portfolios. *Proceedings of the Fourth ICAPS Workshop on Planning and Learning*, pages 14–22, 2013.

[8] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, pages 60–71, 2008.

[9] Amanda Jane Coles, Andrew Coles, Angel García Olaya, Sergio Jiménez Celorrio, Carlos Linares López, Scott Sanner, and Sungwook Yoon. A Survey of the Seventh International Planning Competition. *AI Magazine*, 33(1), 2012.

[10] Alexandra Coman and Hector Muñoz-Avila. Generating Diverse Plans Using Quantitative and Qualitative Plan Distance Metrics. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[11] Joseph C. Culberson and Jonathan Schaeffer. Searching with Pattern Databases. In *Canadian Conference on AI*, pages 402–416, 1996.

[12] Rina Dechter and Judea Pearl. Generalized Best-First Search Strategies and the Optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] Minh Binh Do and Wheeler Ruml. Lessons Learned in Applying Domain-Independent Planning to High-Speed Manufacturing. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 370–373, 2006.

[15] Rüdiger Ebendt and Rolf Drechsler. Weighted A* Search - Unifying View and Application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[16] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications*. Academic Press, 2012.

[17] Henri Farreny. Completeness and Admissibility for General Heuristic Search Algorithms — A Theoretical Study: Basic Concepts and Proofs. *Journal of Heuristics*, 5(3):353–376, 1999.

[18] Chris Fawcett, Malte Helmert, Holger Hoos, Erez Karpas, Gabriele Röger, and Jendrik Seipp. FD-Autotune: Domain-Specific Configuration using Fast Downward. In *ICAPS-2011 Workshop on Planning and Learning*, pages 13–20, 2011.

[19] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[20] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):19–39, 2003.

[21] Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan R. Sturtevant, and Zhifu Zhang. Inconsistent Heuristics in Theory and Practice. *Artificial Intelligence*, 175(9-10):1570–1603, 2011.

[22] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual Lookups in Pattern Databases. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 103–108, 2005.

[23] Richard Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[24] Eugene Fink. How to Solve It Automatically: Selection Among Problem Solving Methods. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 128–136, 1998.

[25] David Furcy and Sven Koenig. Limited Discrepancy Beam Search. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 125–131, 2005.

[26] Ángel García-Olaya, Sergio Jiménez, and Carlos Linares López. The 2011 International Planning Competition. Technical report, Universidad Carlos III de Madrid, Madrid, Spain, July 2011. http://hdl.handle.net/10016/11710.

[27] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[28] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic Planning in the Fifth International Planning Competition: PDDL3 and Experimental Evaluation of the Planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[29] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

[30] Alfonso Gerevini, Alessandro Saetti, and Mauro Vallati. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 350–353, 2009.

[31] Carla Gomes, Bart Selman, and Nuno Crato. Heavy-Tailed Distributions in Combinatorial Search. In *Principles and Practice of Constraint Programming*, pages 121–135, 1997.

[32] Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[33] Eric A. Hansen and Rong Zhou. Anytime Heuristic Search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[34] Larry R. Harris. The Heuristic Search under Conditions of Error. *Artificial Intelligence*, 5(3):217–234, 1974.

[35] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[36] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, 1995.

[37] Matthew Hatem, Roni Stern, and Wheeler Ruml. Bounded Suboptimal Heuristic Search in Linear Space. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search*, pages 98–104, 2013.

[38] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[39] Malte Helmert, Minh Do, and Ioannis Refanidis. IPC 2008 Deterministic Track, 2008. http://ipc.informatik.uni-freiburg.de.

[40] Malte Helmert and Carmel Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, 2009.

[41] Malte Helmert and Hector Geffner. Unifying the Causal Graph and Additive Heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 140–147, 2008.

[42] Malte Helmert and Hauke Lasinger. The Scanalyzer Domain: Greenhouse Logistics as a Planning Problem. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 234–237, 2010.

[43] Malte Helmert and Gabriele Röger. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *ICAPS-2011 Workshop on Planning and Learning*, pages 28–35, 2011.

[44] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[45] Bernardo Huberman, Rajan Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275:51–54, January 3 1997.

[46] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[47] Tatsuya Imai and Akihiro Kishimoto. A Novel Technique for Avoiding Plateaus of Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 985–991, 2011.

[48] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[49] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 201–208, 2009.

[50] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[51] Richard E. Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[52] Richard E. Korf. Linear-Space Best-First Search. *Artificial Intelligence*, 62(1):41–78, 1993.

[53] Richard E. Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 700–705, 1997.

[54] Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[55] Levi H. S. Lelis. *Cluster-and-Conquer: A Paradigm for Solving State-Space Problems*. PhD thesis, University of Alberta, 2013.

[56] Levi H. S. Lelis, Sandra Zilles, and Robert C. Holte. Predicting the size of IDA*'s search tree. *Artificial Intelligence*, 196:53–76, 2013.

[57] Maxim Likhachev. *Search-based Planning for Large Dynamic Environments*. PhD thesis, Carnegie Mellon University, 2005.

[58] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *Advances in Neural Information Processing Systems 16*, pages 767–774, 2003.

[59] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. ARA*: Formal Analysis. Technical Report CMU-CS-03-148, Carnegie Mellon University, 2003.

[60] Qiang Lu, You Xu, Ruoyun Huang, and Yixin Chen. The Roamer Planner: Random-Walk Assisted Best-First Search. In *The 2011 International Planning Competition*, pages 73–76, 2011.

[61] Alberto Martelli. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[62] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL The Planning Domain Definition Language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

[63] Lazlo Mero. A Heuristic Search Algorithm with Modifiable Estimate. *Artificial Intelligence*, 23:13–27, 1984.

[64] Hootan Nakhost, Jörg Hoffmann, and Martin Müller. Resource-Constrained Planning: A Monte Carlo Random Walk Approach. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 181–189, 2012.

[65] Hootan Nakhost and Martin Müller. Monte-Carlo Exploration for Deterministic Planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1766–1771, 2009.

[66] Hootan Nakhost and Martin Müller. Towards a Second Generation Random Walk Planner: An Experimental Exploration. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 2336–2342, 2013.

[67] Alan Olsen and Daniel Bryce. Randward and Lamar: Randomizing the FF Heuristic. In *The 2011 International Planning Competition*, pages 55–57, 2011.

[68] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[69] Judea Pearl and Jin H. Kim. Studies in Semi-Admissible Heuristics. *IEEE Tranactions on Pattern Recognition and Machine Intelligence*, 4(4):392–399, July 1982.

[70] Mike Phillips, Maxim Likhachev, and Sven Koenig. PA*SE: Parallel A* for Slow Expansions. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 208–216, 2014.

[71] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1(3-4):193–204, 1970.

[72] Ira S. Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[73] Michael N. Rice and Vassilis J. Tsotras. Bidirectional A* Search with Additive Approximation Bounds. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, pages 80–87, 2012.

[74] Silvia Richter and Malte Helmert. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

[75] Silvia Richter, Jordan Tyler Thayer, and Wheeler Ruml. The Joy of Forgetting: Faster Anytime Search via Restarting. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 137–144, 2010.

[76] Silvia Richter and Matthias Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[77] Sylvia Richter, Matthias Westphal, and Malte Helmert. LAMA 2008 and 2011. [26], pages 117–124. http://hdl.handle.net/10016/11710.

[78] Jussi Rintanen. Heuristics for Planning with SAT and Expressive Action Definitions. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, pages 210–217, 2011.

[79] Mark Roberts and Adele Howe. Directing a Portfolio with Learning. In *AAAI 2006 Workshop on Learning for Search*, pages 129–135, 2006.

[80] Gabriele Röger and Malte Helmert. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 246–249, 2010.

[81] Jendrik Seipp, Manuel Braun, Johannes Garimort, and Malte Helmert. Learning Portfolios of Automatically Tuned Planners. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, pages 368–372, 2012.

[82] Vitali Sepetnitsky, Ariel Felner, and Roni Stern. To Reopen or Not to Reopen in the Context of Weighted A*? Classifications of Different Trends. In *Proceedings of the 6th ICAPS Workshop on Heuristics and Search for Domain Independent Planning*, pages 92–97, 2014.

[83] Roni Tzvi Stern, Rami Puzis, and Ariel Felner. Potential Search: A Bounded-Cost Search Algorithm. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, pages 234–241, 2011.

[84] N. Sturtevant. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[85] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[86] Jordan Tyler Thayer. *Heuristic Search Under Time and Quality Bounds*. PhD thesis, University of New Hampshire, 2012.

[87] Jordan Tyler Thayer, Austin J. Dionne, and Wheeler Ruml. Learning Inadmissible Heuristics During Search. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011.

[88] Jordan Tyler Thayer and Wheeler Ruml. Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, pages 355–362, 2008.

[89] Jordan Tyler Thayer and Wheeler Ruml. Anytime Heuristic Search: Frameworks and Algorithms. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, pages 121–128, 2010.

[90] Jordan Tyler Thayer and Wheeler Ruml. Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 674–679, 2011.

[91] Jordan Tyler Thayer, Wheeler Ruml, and Ephrat Bitton. Fast and Loose in Bounded Suboptimal Heuristic Search. In *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics*, pages 120–126, 2008.

[92] Jordan Tyler Thayer, Roni Stern, Ariel Felner, and Wheeler Ruml. Faster Bounded-Cost Search Using Inadmissible Estimates. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[93] Richard Anthony Valenzano. Simultaneously Searching with Multiple Algorithm Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search. Master's thesis, University of Alberta, 2009.

[94] Richard Anthony Valenzano, Shahab Jabbari Arfaee, Jordan Tyler Thayer, and Roni Stern. Alternative Forms of Bounded Suboptimal Search. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, pages 175–176, 2012.

[95] Richard Anthony Valenzano, Shahab Jabbari Arfaee, Jordan Tyler Thayer, Roni Stern, and Nathan R. Sturtevant. Using Alternative Suboptimality Bounds in Heuristic Search. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pages 233–241, 2013.

[96] Richard Anthony Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan R. Sturtevant. ArvandHerd: Parallel Planning with a Portfolio. In *20th European Conference on Artificial Intelligence*, pages 786–791, 2012.

[97] Richard Anthony Valenzano, Hootan Nakhost, Martin Müller, Nathan R. Sturtevant, and Jonathan Schaeffer. ArvandHerd: Parallel Planning with a Portfolio. In *The 2011 International Planning Competition* [26], pages 113–116. http://hdl.handle.net/10016/11710.

[98] Richard Anthony Valenzano, Nathan R. Sturtevant, and Jonathan Schaeffer. Adding Exploration to Greedy Best-First Search. Technical Report TR13-06, University of Alberta, 2013.

[99] Richard Anthony Valenzano, Nathan R. Sturtevant, and Jonathan Schaeffer. Worst-Case Solution Quality Analysis When Not Re-Expanding Nodes in Best-First Search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[100] Richard Anthony Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto. Simultaneously Searching with Multiple Settings: An Alternative to Parameter Tuning for Suboptimal Single-Agent Search Algorithms. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 177–184, 2010.

[101] Richard Anthony Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, and Fan Xie. A Comparison of Knowledge-Based GBFS Enhancements and Knowledge-Free Exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, pages 375–379, 2014.

[102] Mauro Vallati, Chris Fawcett, Alfonso Gerevini, Holger H. Hoos, and Alessandro Saetti. Automatic Generation of Efficient Domain-Optimized Planners from Generic Parametrized Planners. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search*, pages 184–192, 2013.

[103] Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

[104] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.

[105] Fan Xie, Martin Müller, and Robert C. Holte. Adding Local Exploration to Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[106] Fan Xie, Martin Müller, Robert C. Holte, and Tatsuya Imai. Adding Local Exploration to Greedy Best-First Search in Satisficing Planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[107] Fan Xie, Richard Anthony Valenzano, and Martin Müller. Better Time Constrained Search via Randomization and Postprocessing. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, pages 269–277, 2013.

[108] Uzi Zahavi, Ariel Felner, Neil Burch, and Robert C. Holte. Predicting the Performance of IDA* using Conditional Distributions. *Journal of Artificial Intelligence Research*, 37:41–83, 2010.

[109] Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, and Nathan R. Sturtevant. Inconsistent Heuristics. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1211–1216, 2007.

[110] Rong Zhou and Eric A. Hansen. Multiple Sequence Alignment Using Anytime A*. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 975–977, 2002.

Appendix A

Additional Formal Results

In this appendix, we include proofs of the theorems which are either largely based on the work of others, or for simple statements that require a technical proof.

A.1 Algorithm Properties From Chapter 2

In this section, we include the proofs for the basic algorithm properties that were described in Chapter 2.

A.1.1 Simple Observations about OCL Algorithms

In this section we define some simple properties of OCL algorithms that will be useful in the following analysis. These properties include statements that nodes will remain in the open or closed list for the remainder of the search after they have been generated, the node selected for expansion can never decrease its own g-cost, only the initial node can have a parent pointer set to $NONE$, and that the g-cost of any node n corresponds to the cost of some path from n_{init} to n.

The first observation simply states that once a node is generated for the first time, then at any time during the remainder of the search it will be in exactly one of the open or closed list.

Observation A.1.1. *If $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ after $t \geq 0$ iterations of an OCL algorithm, then for any $t' \geq t$, one of the following will be true:*

1. $n \in \text{OPEN}_{t'}$ *and* $n \notin \text{CLOSED}_{t'}$.

or

2. $n \notin \text{OPEN}_{t'}$ *and* $n \in \text{CLOSED}_{t'}$.

This observation holds since once a node is put on the open list, it can only ever be moved back and forth between the open and closed lists. Moreover, it can only be added to the closed list if it is removed from the open list, and a node on the closed list can only be added to the open list if it is removed from the closed list.

The next observation states that identifies exactly the set of nodes whose g-cost can change due to a given expansion.

Observation A.1.2. *Let n be the t-the node selected by an OCL algorithm, and let n' be a node such that $n' \in \text{OPEN}_{t-1} \cup \text{CLOSED}_{t-1}$. Then $g_{t-1}(n') \neq g_t(n')$ if and only if $n \neq n'$, $n' \in succ(n)$, and $g_t(n') = g_t(n) + \kappa(n, n')$.*

The next observation states that at any time during the execution of an OCL algorithm, n_{init} will have a parent pointer set to $NONE$, and it will be the only node on either the open or closed list such that its initial node is set to $NONE$.

Observation A.1.3. *Suppose there have been t node expansions of an OCL algorithm. Then $parent_t(n_{init}) = NONE$ and for any $n \in \text{OPEN}_t \cup \text{CLOSED}_t$, $parent(n) = NONE$ if and only if $n = n_{init}$.*

This observation holds since the parent pointer of a node n can only be set to a node that generated n and since it is not possible to find a path to n_{init} that has a lower cost than the initial g-cost of 0 assigned to that node.

The next lemma states that the g-cost of a node n corresponds to some path from n_{init} to n consisting of nodes that are on either the open or closed list.

Lemma A.1.4. *Suppose there have been t node expansions of an OCL algorithm and let n be a node such that $n \in \text{OPEN}_t \cup \text{CLOSED}_t$. Then there exists a path $P \in \Pi_{init}$ where $P = [n_0, ..., n_k]$ such that the following are true:*

1. $n_0 = n_{init}$ *and* $n_k = n$.

2. $g_t(n) = C(P)$.

3. $\forall 0 \leq i \leq k$, $n_i \in \text{OPEN}_t \cup \text{CLOSED}_t$.

4. $\forall 0 \leq i < j \leq k$, $n_i \neq n_j$.

5. $\forall n_i \in P$, $\exists t' \leq t$ *such that* $g_{t'}(n_i) = C([n_0, ..., n_i])$.

Proof. This proof is by induction on the number of node expansions. In the base case, there have been 0 node expansions, only n_{init} is in either the open or closed list, and let $P = [n_{init}]$. Since $g_0(n_{init}) = 0 = C(P)$, $n_0 = n_{init}$, and P clearly satisfies conditions 3, 4, and 5, the statement holds in the base case.

Now suppose the statement holds after $t \geq 0$ expansions. Let n' be the $t+1$-st node selected for expansion and let n be an arbitrary node in $\text{OPEN}_{t+1} \cup \text{CLOSED}_{t+1}$. We now have to handle a number of cases.

In the first case, $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ and $g_{t+1}(n) = g_t(n)$. This means that the $t+1$-st expansion did not change the g-cost of n. Let P be the path from n_{init} to n guaranteed to exist by the induction hypothesis such that P satisfies conditions 1 through 5 after t expansions. Since all $g_{t+1}(n) = g_t(n)$, all nodes along P must remain in the open or closed list through the $t+1$-st expansion by Observation A.1.1, this path clearly still satisfies conditions 1 through 5 after the $t+1$-st expansion. Therefore, the statement holds in this case.

Now suppose that $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ and $g_{t+1}(n) < g_t(n)$. This requires that the g-cost of n was updated because the expansion of n' resulted in the generation of a lower g-cost path to n. As such, $g_{t+1}(n) = g_{t+1}(n') + \kappa(n', n)$. Let $P' = [n_0, ..., n_k]$ be the path to n' that is guaranteed to exist after t expansions such that $n_0 = n_{init}$, $n_k = n'$, $g_t(n') = C(P')$, and the remaining conditions are also satisfied. By the previous case, these conditions will also apply for P' after the $t+1$-st expansion.

We now let $P = [n_0, ..., n_k, n]$ and will show that P satisfies conditions 1 through 5. Clearly P satisfies condition 1. P also satisfies condition 2 since $C(P) = C(P') + \kappa(n', n) = g_{t+1}(n)$. P also satisfies condition 3 since the nodes on P' will be in the open or closed list before and after the expansion of n' by Observation A.1.1 and so will n. Condition 5 holds for $n_i \in P$ since it held prior to the $t+1$-st expansion, and it holds for n by condition 2 which was just shown.

Let us now consider condition 4. Since condition 4 held for P before the $t+1$-st expansion, $n_i \neq n_j$ for any $0 \leq i < j \leq k$. Therefore, in order for condition 4 to not hold for P after the $t+1$-st expansion requires that $n = n_i$ for some $0 \leq i \leq k$. If this is true, then $g_t(n_i) = g_t(n) = C([n_0, ..., n_i])$ by the induction hypothesis, which implies that $g_t(n) \leq C(P)$ since $C(P) = C([n_0, ..., n_i]) + C([n_i, ..., n_k])$. However, $g_{t+1}(n) = C(P) + \kappa(n', n)$ since the g-cost of n was updated by the expansion of t. Since $\kappa(n', n) \geq 0$ this means that $g_{t+1}(n) \geq C(P)$ which, when combined with the previously shown fact that $g_t(n) \leq C(P)$, contradicts the fact that $g_{t+1}(n) < g_t(n)$. Therefore $n \neq n_i$ and condition 4 holds in this case. Having handled all conditions, the statement is true if $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ and $g_{t+1}(n) < g_t(n)$.

The final case is if n is generated for the first time by the $t+1$-st expansion. As with the proof in the previous case, we will select P as $[n_0, ..., n_k, n]$ where $P' = [n_0, ..., n_k]$ is the path to n' guaranteed to exist and satisfy conditions 1 through 4 after t expansions by the induction hypothesis. The proof of conditions 1 through 4 are exactly the same as for $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ and $g_{t+1}(n) \leq g_t(n)$. Condition 5 then holds since $n_i \neq n_j$ for any $0 \leq i < j \leq k$ by the induction hypothesis, and $n \neq n_i$ since this is the first time n was generated. Therefore, the statement is true in all cases. □

A.1.2 The g-cost of Parent and Children Nodes

Lemma 2.2.3 states if a node n_k is not expanded prior to its parent n_{k-1} along some optimal path from n_{init} to n_k, then the g-cost error of n_k will be bound by the g-cost error of n_{k-1} at the time it is first expanded. The proof is as follows:

Lemma 2.2.3. *Let $P = [n_0, ..., n_{k-1}, n_k]$ be an optimal path from $n_0 = n_{init}$ to n_k for $k > 0$. If n_{k-1} is expanded for the first time by the t-th node expansion of an OCL algorithm and n_k is not one of the first t nodes expanded, then the following is true for any $t' \geq t$*

$$g^{\delta}_{t'}(n_k) \leq g^{\delta}_{t}(n_{k-1}) \ .$$

Proof. Suppose n_{k-1} is the t-th node expanded. If n_k is not one of the first t nodes expanded, then it is either generated for the first time by this expansion, or it was

already on the open list. If n_k has been generated for the first time by the expansion of n_{k-1}, then its g-cost is set as $g_t(n_{k-1}) + \kappa(n_{k-1}, n_k)$. If n_k was already on the open list, then its g-cost will be updated to $g_t(n_{k-1}) + \kappa(n_{k-1}, n_k)$ if its g-cost is any larger than that. As such, the g-cost of n_k will be at most $g_t(n_{k-1}) + \kappa(n_{k-1}, n_k)$ regardless of whether n_k is generated for the first time by the t-th expansion or if it was previously on the open list. This allows us to perform the following derivation:

$$g_t(n_k) \leq g_t(n_{k-1}) + \kappa(n_{k-1}, n_k) \tag{A.1}$$

$$g_t^\delta(n_k) + g^*(n_k) \leq g_t^\delta(n_{k-1}) + g^*(n_{k-1}) + \kappa(n_{k-1}, n_k) \tag{A.2}$$

$$g_t^\delta(n_k) + g^*(n_k) \leq g_t^\delta(n_{k-1}) + g^*(n_k) \tag{A.3}$$

$$g_t^\delta(n_k) \leq g_t^\delta(n_{k-1}) \tag{A.4}$$

Line A.2 holds by the definition of g-cost error, line A.3 holds by the fact that $g^*(n_k) = g^*(n_{k-1}) + \kappa(n_{k-1}, n_k)$ since n_{k-1} is along an optimal path to n_k, and the final line holds by subtracting $g^*(n_k)$ from both sides. The full statement then follows by the fact that the g-cost of a node is non-decreasing over time, where this statement was formally stated by Observation 2.2.2. □

We now consider the case in which the OCL algorithm is always performing parent pointer updates:

Lemma 2.2.4. *Let $P = [n_0, ..., n_{k-1}, n_k]$ be an optimal path from $n_0 = n_{init}$ to n_k for $k > 0$. If n_{k-1} is expanded by the t-th node expansion of an OCL algorithm that always performs parent pointer updating, then the following is true for any $t' \geq t$*

$$g_{t'}^\delta(n_k) \leq g_t^\delta(n_{k-1}) \ .$$

Proof. If n_{k-1} is the t-th node expanded by an OCL algorithm that always performs parent pointer updating, then either $g_t(n_k)$ will be updated to $g_t(n_{k-1}) + \kappa(n_{k-1}, n_k)$ (because it is generated for the first time, updated in the open list, or moved back from the closed list), or it will already have a g-cost that is no larger than this value. In either case, $g_t(n_k) \leq g_t(n_{k-1}) + \kappa(n_{k-1}, n_k)$ and the same derivation can be performed as in Lemma 2.2.3. The result then follows for the same reasons. □

A.1.3 The g-cost as an Upper Bound on Path Cost

In this section, we will show that for any node n on the open or closed list of an OCL algorithm, there exists a unique path from n_{init} to n that follows the parent pointers and that $g(n)$ is an upper bound on the cost of this path. So as to show this theorem, we begin with the following lemma:

Lemma A.1.5. *Let $P = [n_0, ..., n_k]$ be a path in Π such that after t iterations of an OCL algorithm the following are true:*

- $\forall 0 \leq i \leq k$, $n_i \in \text{OPEN} \cup \text{CLOSED}$.
- $\forall 1 \leq i \leq k$, $parent_t(n_i) = n_{i-1}$.

Then $\forall 0 < i < j \leq k$, $n_i \neq n_j$ and the following holds:

$$g_t(n_k) \geq g_t(n_0) + C(P) \ .$$

Proof. The proof is by induction on the number of node expansions. In the base case, there have been no expansions, the closed list is empty, and the open list contains only n_{init}. In this case, the only path consisting of nodes on either the open or closed list is $[n_{init}]$. Since $C([n_{init}]) = 0$ and $g_0(n_{init}) = 0$, this means that $g_0(n_{init}) \geq g_0(n_{init}) + C(P)$ and the statement is true in this case.

Now suppose the statement is true after $t \geq 0$ node expansions. Let n be the $t+1$-st node expanded, and let $P = [n_0, ..., n_k]$ be some path in Π such that after $t+1$ node expansions it is true that for all $0 \leq i \leq k$, n_i is on either the open or closed list and $parent_t(n_i) = n_{i-1}$. We now consider two cases for P.

In the first case, assume that for any node $n_i \in P$, the g-cost and parent pointer of n_i were not updated by the $t+1$-st node expansion, nor was n_i generated for the first time by the $t+1$-st node expansion. If this is true, then all the nodes on P were on either in the open and closed list and that P existed as is prior to $t+1$-st node expansion. As such, the statement holds true for P after the expansion of n since it held true prior to the expansion of n by the induction hypothesis.

Notice that the previous case applies in the situation of any path $P = [n_0, ..., n_k]$ for which $n = n_k$ and P satisfies the conditions of the lemma. This is because the

expansion of n cannot update the g-cost or parent pointers of any such path to n. To see this, suppose that n made such updates for some n_i on P where $i < k$. Let n_i be the deepest such node on P, which means that the g-cost and parent pointer of any node along $[n_i, ..., n_k]$ was not changed by the expansion of $n = n_k$,. As such, $[n_i, ..., n_k]$ must have satisfied the conditions of the lemma after t expansions, and so $g_t(n_i) \leq g_t(n_k)$ since $g_t(n_k) \geq C([n_i, ..., n_k] + g_t(n_i)$ by the induction hypothesis. However, $g_t(n_i) \leq g_t(n_k)$ contradicts the fact that the g-cost and parent pointers of n_i were updated by the expansion of $n = n_k$, since doing so requires that $g_t(n_i) > g_t(n_k) + \kappa(n_k, n_i)$. Therefore, the g-cost and parent pointers of n_i cannot have been updated by the expansion of n, and so any path on which n is the deepest node will be covered by the first, already handled, case.

In the second case that we consider, there is some node $n_i \in P$ which was either generated for the first time by the $t+1$-st expansion, or whose parent pointer and g-cost was updated as a result of this expansion. Since the parent pointer of n_i will in both cases be set to n, this means that $n = n_{i-1}$. Notice that this means that at most one node from P can have been generated or updated by the $t + 1$-st expansion, as otherwise, the expansion of $n = n_i$ will update a node $n_j \in P$ where $j < i$ and this was just shown to not be possible. This also means that n_i cannot be one of the nodes in $[n_0, ..., n_{i-1}]$ and so all the nodes on $[n_0, ..., n_{i-1}, n_i]$ are necessarily unique by the induction hypothesis.

Now let us show first that $g_t(n_k) \geq g_t(n_0) + C(P)$. Since the g-cost of n_i is being updated by the expansion of $n = n_{i-1}$, $g_{t+1}(n_{i-1}) = g_t(n_{i-1})$ and so $g_{t+1}(n_i) = g_{t+1}(n_{i-1}) + \kappa(n_{i-1}, n_i)$ by the definition of an OCL algorithm . This allows for the following derivation:

$$g_{t+1}(n_i) = g_{t+1}(n_{i-1}) + \kappa(n_{i-1}, n_i) \tag{A.5}$$

$$\geq g_{t+1}(n_0) + C([n_0, ..., n_{i-1}]) + \kappa(n_{i-1}, n_i) \tag{A.6}$$

$$\geq g_{t+1}(n_0) + C([n_0, ..., n_i]) \tag{A.7}$$

Line A.6 is true since $g_{t+1}(n_{i-1}) \geq g_{t+1}(n_0) + C([n_0, ..., n_{i-1}])$ follows from the case just proven because all nodes on $[n_0, ..., n_{i-1}]$ will have the same g-cost and parent pointers before and after the expansion of n_{i-1}. The final line then holds by

the definition of the cost function.

Notice that this last argument proves that $g_t(n_k) \geq g_t(n_0) + C(P)$ is true in the case that $i = k$. Let us now consider the case in which $i < k$, and notice that this means that n_i cannot have been generated for the first time by the $t+1$-st expansion. This is because if $i < k$ then there is some n_{i+1} whose parent pointer is n_i, and this is not possible if n_i has just been generated and therefore has not previously been expanded. As a result, $[n_i, ..., n_k]$ existed as is prior to the $t+1$-st expansion, with n_i being the only node on this path that had its g-cost and parent pointer updated. This means that $g_t(n_k) \geq g_t(n_i) + C([n_i, ..., n_k])$ by the induction hypothesis. Since $g_{t+1}(n_i) < g_t(n_i)$ (because the g-cost of n_i is updated by the expansion of n_{i-1}) and $g_{t+1}(n_k) = g_t(n_k)$ (as n_i is the only node from P that had its g-cost and parent pointers updated), this means that $g_{t+1}(n_k) \geq g_{t+1}(n_i)+C([n_i, ..., n_k])$. This allows for the following derivation:

$$g_{t+1}(n_k) \geq g_{t+1}(n_i) + C([n_i, ..., n_k]) \tag{A.8}$$

$$\geq g_{t+1}(n_0) + C([n_0, ..., n_i]) + C([n_i, ..., n_k]) \tag{A.9}$$

$$\geq g_{t+1}(n_0) + C([n_0, ..., n_k]) \tag{A.10}$$

Line A.9 follows from line A.7 above, and the last line holds by the definition of C. Therefore $g_{t+1}(n_k) \geq g_{t+1}(n_0) + C(P)$ holds in the case that $i < k$.

Now let us show that in this case, $\forall 0 < i < j \leq k$, $n_i \neq n_j$. By the inductive hypothesis, all nodes on $[n_0, ..., n_{i-1}, n_i]$ are unique, as are all nodes on $[n_i, ..., n_k]$, since the nodes on these paths did not change by the expansion of n_{i-1}. Therefore, it can only be false if there are nodes n_j and $n_{j'}$ such that $0 \leq j < i - 1 < j' \leq k$ and $n_j = n_{j'}$. The path between these nodes is $P' = [n_j, ..., n_{i-1}, n_i, ..., n'_j]$. As just shown, $g_{t+1}(n_j) \geq g_{t+1}(n_{j'}) + C(P')$, which is only possible if $C(P') = 0$ since $n_j = n_{j'}$. This requires that all the nodes along P' have an equal g-cost of 0 after $t + 1$ expansions. In particular, it means that $g_{t+1}(n_i) = g_{t+1}(n_{j'})$.

Since the parent pointers along $[n_i, ..., n_{j'}]$ were unchanged by the $t+1$-st expansion, this means that $g_t(n_{j'}) \geq g_t(n_i) + C([n_i, ..., n_{j'}])$ by the inductive hypothesis. This implies that $g_t(n_{j'}) \geq g_t(n_i)$ since $C([n_i, ..., n_{j'}]) = 0$ as just shown. However, the g-cost of $n_{j'}$ was not changed by the $t + 1$-st expansion (since n_i was

the only node on P whose g-cost was changed), and so $g_{t+1}(n_{j'}) = g_t(n_{j'})$. This means that $g_{t+1}(n_{j'}) \geq g_t(n_i)$ since $g_t(n_{j'}) \geq g_t(n_i)$, which in turn implies that $g_{t+1}(n_i) \geq g_t(n_i)$ since $g_{t+1}(n_i) = g_{t+1}(n_{j'})$ as shown above. This is a contradiction of the fact that the g-cost of n_i was updated by the $t+1$-st expansion, and so $g_{t+1}(n_i) < g_t(n_i)$. Therefore $n_j \neq n_{j'}$ and the statement holds.

Having handled all cases, the statement is true. □

For the next step in the proof of the existence and uniqueness of the path from n_{init} to n that is maintained implicitly using parent pointers, recall that a function, `ReconstructPath`, was introduced which returned such a path. We will now use Lemma A.1.5 to show that `ReconstructPath` is correct and will terminate. We will then show that this function that we want, and then show that this path satisfies the desired properties.

As detailed in Algorithm 2, `ReconstructPath` uses a recursive function with the name of `ReconstructPathRecursive` that takes as its parameters a node and a path. The initial call starts with an empty path and the node n. On a given call to this recursive function, it checks if $n = NONE$. This is the base case for the function. If it is, it returns the given path $[n_0, ..., n_k]$. If not, it recursively calls the function with $parent(n)$ as the node and $[n, n_0, ..., n_k]$ as the path.

Before proceeding with the proof regarding the existence and uniqueness of a path from n_{init} to n, and that $g(n)$ is an upper bound on this path, we will first prove the following useful lemma about the `ReconstructPathRecursive` function.

Lemma A.1.6. *Suppose that* `ReconstructPathRecursive` *is called after t expansions of an OCL algorithm with $P = [n_0, ..., n_k]$ as its path parameter where $P \neq []$ and n as its node parameter such that the following conditions are true:*

1. $\forall 0 \leq i \leq k$, $n_i \in \text{OPEN}_t \cup \text{CLOSED}_t$.
2. $\forall 1 \leq i \leq k$, $parent_t(n_i) = n_{i-1}$.
3. $\forall 0 \leq i < j \leq k$, $n_i \neq n_j$.
4. *Either* $n = NONE$, *or* $n \in \text{OPEN}_t \cup \text{CLOSED}_t$ *and* $n = parent_t(n_0)$.

Then `ReconstructPathRecursive` *will either immediately return* P *or make a recursive call on a path* P' *and a node* n' *such that conditions* 1 *through* 5 *are also true of* P' *and* n'.

Proof. Notice that if $n = NONE$, P will be immediately returned by the definition of `ReconstructPathRecursive`. Therefore the statement holds if $n = NONE$. For the remainder of the proof, we will therefore assume that $n \neq NONE$. This means that a recursive call will necessarily be made with node $n' = parent(n)$ and path $P' = [n, n_0, ..., n_k]$.

Let us first show that n' satisfies condition 4. Since $n' = parent(n)$ and n is the first node on P', we are only left with showing that $n' = NONE$ or n' is on either the open or closed list. If $n' \neq NONE$, n' can only have been set as the parent pointer of n if n was generated by the expansion of n'. This means that n' was previously on the open list and so it to must have been previously generated. Since a node is on the open or closed list for the remainder of the search once it is generated for the first time, this implies that n' is on the open or closed list after t expansions. Therefore, condition 4 is satisfied.

Let us now show that conditions 1 through 3 all hold for P'. First, recall that we assumed that all nodes on P are in the open or closed list by the assumption of condition 1. Since n is also in the open or closed list by the condition 4, all nodes in P' are in the open or closed list. Therefore P' satisfies condition 1. This same approach can be taken for showing condition 2 for P' since P' is just P with the additional edge of (n, n_0) added and $n = parent(n_0)$ by condition 4. Therefore P' satisfies condition 2. Finally, notice that $P' = [n, n_0, ..., n_k]$ satisfies the conditions of Lemma A.1.5, and so we are guaranteed that none of the nodes on this path are the same. As such, P' satisfies condition 3. Therefore, the statement is true. □

Let us now show that `ReconstructPath` is guaranteed to terminate with a path $P = [n_0, ..., n_k]$ from $n_{init} = n_0$ to the given node $n = n_k$ with the path that we want. This is done by the following lemma:

Lemma A.1.7. *If* `ReconstructPath` *is called after* t *node expansions of an OCL algorithm, given a node* n *that is in the open or closed list, then the function*

will terminate and return the path $P = [n_0, ..., n_k]$ *which satisfies the following conditions:*

1. $\forall 0 \leq i \leq k$, n_i *is in either the open or closed list after* t *expansions.*
2. $\forall 1 \leq i \leq k$, $parent_t(n_i) = n_{i-1}$.
3. $\forall 0 \leq i < j \leq k$, $n_i \neq n_j$.
4. $n_0 = n_{init}$ *and* $n_k = n$.

Proof. Given a node n that is on the open or closed list of an OCL algorithm after t expansions, `ReconstructPath` will call `ReconstructPathRecursive` with [] as the path parameter and n as the node parameter. Since n is on the open or closed list, $n \neq NONE$ and so a recursive call will be made with $parent(n)$ for the node and $P = [n]$ as the path. Since P satisfies conditions 1 through 3 of Lemma A.1.6, and $parent(n)$ satisfies condition 4, this means that this subsequent call will either immediately return $[n]$ or it will make a further recursive call on a path and node that satisfy these properties. Clearly, this will be inductively true of any recursion call that is farther down in the recursion stack.

Let us first show that if it terminates, it will terminate with the correct path. First, notice that `ReconstructPathRecursive` will only terminate if some recursive call immediately returns the given path P, in which case P will be returned, unchanged, by each of the shallower recursive calls until it is returned by `ReconstructPath`. Now notice that a recursive call can only immediately return if the node given as a parameter is equal to $NONE$. As described above, it can be shown inductively that P will satisfy conditions 1 through 3 of Lemma A.1.6, and so it also satisfies conditions 1 to 3 of the statement we are currently proving since these are the same three conditions. Furthermore, $n_k = n$ since n was the first node put on P and all other nodes that were added to P were placed shallower than all those nodes previously added to P. It must also be the case that $n_0 = n_{init}$ since $parent(n_0) = NONE$, and the only node that has its parent pointer set as n_0 is necessarily n_{init}. Therefore, if the algorithm ever terminates, it will return a path that satisfies all of conditions 1 through 4.

Now, let us show that this process is guaranteed to terminate. To do so, notice that since n_{init} is the only node with its parent pointer set as $NONE$, the algorithm will terminate if and only if the second to last recursive call has n_{init} as its node parameter. We can therefore show that it will terminate if it can make at most a finite number of recursive calls before it makes one with n_{init} as its node parameter.

For this purpose, recall that `ReconstructPath` was called after t node expansions. Since t is finite and the number of successors of any node is finite, then the number of unique nodes that have thus far been generated must also be finite. As such, the number of nodes in either the open or closed list is also finite. Now notice that each time a recursive call is made, a new node is put on the path that was not previously there. This is guaranteed by condition 3 of Lemma A.1.6. Therefore, it is not possible that an infinite number of recursive calls can be made without one being called with n_{init} as its parameter. Therefore, the algorithm terminates. □

With the correctness and termination of `ReconstructPath` having just been shown, we can now show the existence and uniqueness of a path from n_{init} to any n on either the open or closed list such that the parent pointers of all of the nodes along this path follow the path itself. Note that saying that the path $P = [n_0, ..., n_k]$ from $n_{init} = n_0$ to $n = n_k$ is unique, we mean that if $P' = [n'_0, ..., n'_j]$ is a path from $n_{init} = n'_0$ to $n = n'_j$ such that for all $0 \leq i \leq k$, n_i is in either the open or closed list and for all $0 < i \leq j$, $parent(n'_i) = n'_{i-1}$, then $k = j$ and for all $0 \leq i \leq k$, $n_i = n'_i$. This is proved in the following:

Theorem 2.2.5. *Suppose that there have been t node expansions of an OCL algorithm. If n is a node such that $n \in \text{OPEN}_t \cup \text{CLOSED}_t$, then there exists a unique path $P = [n_0, ..., n_k]$ from $n_{init} = n_0$ to $n = n_k$ such that the following are true:*

1. $\forall 0 \leq i \leq k$, $n_i \in \text{OPEN}_t \cup \text{CLOSED}_t$.
2. $\forall 1 \leq i \leq k$, $parent_t(n_i) = n_{i-1}$.
3. $\forall 0 \leq i < j \leq k$, $n_i \neq n_j$.
4. $g_t(n) \geq C(P)$.

Proof. Let $P = [n_0, ..., n_k]$ be the path returned by `ReconstructPath`. By Lemma A.1.7, $n_0 = n_{init}$, $n_k = n$ and conditions 1 through 3 are satisfied. We now show that condition 4 holds before then showing uniqueness.

So as to show that P satisfies condition 4, notice that P satisfies the needed conditions for Lemma A.1.5. As such, $g_t(n) \geq g_t(n_0) + C(P)$. Since $n_0 = n_{init}$ and $g(n_{init}) = 0$, this means that $g_t(n) \geq C(P)$.

The proof that P is unique will be by contradiction. In this end, suppose there is another such path $[n'_0, ..., n'_j]$ from $n_{init} = n'_0$ to $n = n'_j$. Consider traversing these paths from the back ($n = n_k = n'_j$) to front ($n_{init} = n_0 = n'_0$) one node at a time and in parallel, and comparing the nodes encountered on each path. For example, we will compare n_k to n'_j, then n_{k-1} to n'_{j-1}, and so on. Suppose n_i and $n'_{i'}$ are the first nodes that disagree. Since $n_k = n'_j$, this means that $i < k$ and $i' < j$. This means that $n_{i+1} = n'_{i'+1}$, which requires in turn means that $parent(n_{i+1}) = parent(n'_{i'+1})$. Of course, $parent(n_{i+1}) = n_i$ and $parent(n'_{i'+1}) = n'_{i'}$, and so $n_i = ni''$, but this contradicts these nodes as the first that disagree. Therefore, the path found by `ReconstructPath` is the unique path from n_{init} to n of those consisting of nodes in only the open list or closed list such that the parent pointers of the nodes point along the path. □

A.1.4 OCL Algorithms Progressing Along Candidate Paths

In this section, we will show that OCL algorithms continue to make progress along candidate paths, and when using a full expansion policy they will make this progress while maintaining the optimal g-cost when considering only optimal paths. We begin by considering the case in which we assume that the algorithm is using a full re-expansion policy:

Theorem 2.2.6. *Let $P \in \Pi_{init}$ be a candidate path for a given planning task Γ such that $P = [n_0, ..., n_k]$, P is an optimal path from n_{init} to n where $n = n_k$, and $\forall n_i \in P$, $H(n_i) \neq \infty$. Then after t iterations of a rOCL algorithm on Γ, one of the following will be true:*

1. *$\forall n \in P$, $n \in \text{CLOSED}_t$ and $g_t(n) = g^*(n)$.*

or

2. $\exists n_i \in P$ *such that* $n_i \in \text{OPEN}_t$, $g_t(n_i) = g^*(n_i)$, *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$ *and* $g_t(n_j) = g^*(n_j)$.

Proof. Suppose there have been t node expansions of an OCL algorithm on task Γ. Let $P \in \Pi_{init}$ such that $P = [n_0, ..., s_k]$ and for any $n_i \in P$, $H(n_i) \neq \infty$. If $n_{init} = n_0$ is in the open list, then condition two clearly holds since there are no nodes that are shallower than n_0 on P. Since $n_{init} = n_0$ must always be in either the open or closed list, we now consider the case in which it is in the closed list. Let n_i be the deepest node from P such that n_i is in the closed list, $g_t(n_i) = g^*(n_i)$, and these conditions also hold for any ancestor of n_i along P. Such a node is guaranteed to exist since n_{init} satisfies these conditions. If $n_i = n_k$, then condition 1 holds and the statement is true. Otherwise $i < k$, and notice that $g_t^\delta(n) = 0$ since $g_t(n_i) = g^*(n_i)$. Since the OCL algorithm being used is the full re-expansion policy, the g-cost error of n_{i-1} must be no larger than the g-cost error of n_i by Lemma 2.2.4. Therefore, $g_t(n_{i+1}) = g^*(n_{i+1})$ and n_{i+1} must be on the open list, as if it is on the closed list it would contradict the selection of n_i. As all the ancestors of n_{i+1} along P are in the closed list and have a g-cost error of 0, condition 2 holds, and so the statement is true. □

The following then represents a weaker statement that holds if nothing is assumed by the re-expansion policy being used:

Theorem 2.2.7. *Let* $P \in \Pi_{init}$ *be a candidate path for a given planning task* Γ *such that* $P = [n_0, ..., n_k]$ *and* $\forall n_i \in P$, $H(n_i) \neq \infty$. *Then after* t *iterations of an OCL algorithm on* Γ, *one of the following will be true:*

1. $\forall n \in P$, $n \in \text{CLOSED}_t$.

or

2. $\exists n_i \in P$ *such that* $n_i \in \text{OPEN}_t$, *and* $\forall n_j \in P$ *where* $0 \leq j < i$, $n_j \in \text{CLOSED}_t$.

Proof. This proof is almost identical to the proof to Theorem 2.2.6. We again start by supposing that there have been t node expansions of an OCL algorithm on task Γ. Let $P \in \Pi_{init}$ such that $P = [n_0, ..., s_k]$ and for any $n_i \in P$, $H(n_i) \neq \infty$. If $n_{init} = n_0$ is in the open list, then condition two clearly holds. Let us now assume that $n_{init} = n_0$ is in the closed list. Let n_i be the deepest node on P which is in the closed list. Such a node must exist since n_0 is in the closed list. Since n_i is the deepest such node, the result is that $\forall j \leq i$, n_j is in the closed list. This means that if $n_i = n_k$ then all the nodes on P are in the closed list and condition 1 holds. If $i < k$, then n_i must have been expanded, n_{i+1} must have been previously generated, and since $H(n_{i+1}) \neq \infty$, n_{i+1} must be in either the closed or open list. By the selection of n_i, n_{i+1} cannot be in the closed list and so it must be in the open list. As all the ancestors of n_{i+1} along P are in the closed list, condition 2 holds, and the statement holds true. □

A.1.5 A Lower Bound on C^*

In this section, we show how the nodes that have been explored by an OCL algorithm can be used to generate a lower bound on C^* when parent pointers are always updated. The theorem is given as follows where $UNOPENED$ contains the set of nodes on the closed list whose g-cost is updated, but which are not moved back to the open list.

Theorem 2.2.11. *Let h be an admissible heuristic and suppose that there have been t iterations of an OCL algorithm that always performs parent pointer updates, such that the search has yet to expand a goal node. Then the following is true:*

$$C^* \geq \min_{n \in \text{OPEN}_t \cup \text{UNOPENED}_t} g_t(n) + h(n) \ .$$

Proof. Let O denote the OCL algorithm in use on the given planning task. This proof works as follows. We will define a new rOCL algorithm O' which will always select the same node for expansion as O, but for which the nodes that O would have in the unopened list, O' will put back in its open list. The result will be that at any time, the open list of O' will be given by the union of the open and unopened list

of O. This will then allow us to use Theorem 2.2.10 to get the bound given by the statement.

Let us define O' in more detail. O' will have two types of nodes in its open list: marked nodes and unmarked nodes. All nodes are set as unmarked when they are generated for the first time. A node n will only be marked if it is in the closed list, its g-cost and parent pointer is updated, and the `ShouldOpenNode` function of O returns $False$. In this case, O' will move n back to the open list, but n will be marked.

A marked node n can become unmarked if a lower g-cost path is found to it, and the `ShouldOpenNode` function of O returns $True$. This case corresponds to a node being moved back from the closed list to the open list during the execution of O.

The policy used by O' to select nodes from the open list is the same as O, except O' only uses this policy to select from the unmarked nodes. This means that if the set of unmarked nodes is the exact same set as the open list in O (with g-cost values and parent pointers also being the same), then both O and O' will select the same node for expansions.

By definition, it is clear that on the t-th iteration of O', the unmarked nodes of O' will correspond exactly to the open list of O, the marked nodes of O' will correspond exactly to the nodes on the unopened list of O, and the closed list of O' will correspond exactly to the remainder of the closed list of O when the unopened nodes are removed. By Theorem 2.2.10, the minimum of $g(n) + h(n)$ over every node n on the open list of O' will be a lower bound on C^*. Since the open list of O' is given by the union of the open and unopened lists of O, this means that the minimum of $g(n) + h(n)$ over every node n in the open and unopened lists of O is also a lower bound on C^*. Therefore, the statement is proven. □

A.2 ϵ-Greedy Node Selection on Problem Graph

In this section, we will show that for any $\epsilon > 0$, a GBFS enhanced with ϵ-greedy node selection will have a finite expected runtime for any ϵ where $0 < \epsilon < 1$ on the

graphs defined in Section 4.1 on which GBFS will never find a solution. This will hold regardless of the branching factor of the graphs.

Let us begin by recalling the definition of these types of graphs as trees with three types of nodes. These types are denoted as T_0, T_1, and T_g. Every type T_0 node has b successors for some constant $b > 0$. $b - 1$ of these successors are T_0 nodes and the remaining successor is a type T_1 node. Type T_1 nodes have only a single successor: a type T_2 node. The heuristic value of type T_0, T_1, and T_g nodes will be denoted as H_0, H_1, and H_g, respectively, where $H_g < H_0 < H_1$.

Since the type T_0 nodes form an infinite plateau, GBFS will never expand any node outside of this plateau once a first node from this plateau is expanded. In contrast, we will show that when using ϵ-greedy node selection, a type T_g node will be expanded with a finite expected runtime. For this proof, we will first show that a type T_g node will be generated (or equivalently, that a type T_1 node will be expanded) with a finite expected runtime. Then, we will show that once a type T_g node is generated, it will expanded with a finite expected runtime. Together, these will prove the desired statement.

We begin by showing that a type T_1 node will be expanded with a finite expected runtime. In particular, where $P_1(t = i)$ is the probability that a type T_1 node is expanded for the first time by the i-th node expansion, this means that we must show that the following converges:

$$\sum_{i=2}^{\infty} i \cdot P_1(t = i) \tag{A.11}$$

Notice that this summation starts at 2 since the first node expanded, the initial node, is necessarily a type T_0 node, and so a type T_1 can be expanded at earliest as the second node.

We now turn to calculating $P_1(t = k)$. For this purpose, we will let $K_0(t)$ and $K_1(t)$ be the number of type T_0 and T_1 nodes, respectively, in the open list after t expansions, both of which we will need to calculate. Notice that if all of the first t node expansions are type T_0 nodes, then there will be t type T_1 nodes in OPEN_t and so $K_1(t) = t$. This is because every expansion will add a single type T_1 node into the open list.

Now consider $K_0(t)$. If $t = 1$, then there will be $b - 1$ nodes of type T_0 in the open list, and so $K_0(1) = b - 1$. If $t = 2$, then there will be $2 \cdot (b - 1) - 1$ node on the open list, since the search started with one type T_0 node in the open list, $2 \cdot (b - 1)$ were added into the open list by the expansions, and 2 type T_0 nodes were expanded and therefore removed from the open list. This means that and so $K_0(2) = 2 \cdot (b - 1) - 1$. This argument can be extended to show that after $t > 1$ such expansions, the number of T_0 nodes in the open list will be as follows:

$$K_0(t) = t \cdot (b - 1) - (t - 1) \ .$$

$P_1(t = k)$ will then be given by the product of the probability that the first $t - 1$ expansions were all type T_0 nodes and the probability that the t-th node expansion is a type T_1 node. For calculating this latter quantity, notice that since $K_0(t - 1) \geq 0$, a type T_0 node will be selected for expansion if a greedy node selection is made. If an exploratory action is made, the probability that a type T_1 node is selected for expansion is given by $K_0(t - 1)/[K_0(t - 1) + K_1(t - 1)]$. As such, for $k \geq 1$, $P_1(t = k)$ is given as follows:

$$P_1(t = k) = \epsilon \cdot \frac{K_0(k - 1)}{K_0(k - 1) + K_1(k - 1)} \cdot \prod_{i=1}^{k-2} \left[1 - \epsilon \cdot \frac{K_0(i)}{K_0(i) + K_1(i)} \right] \quad \text{(A.12)}$$

$$= \epsilon \cdot \frac{k - 1}{(k - 1) \cdot (b - 1) + 1} \cdot \prod_{i=1}^{k-2} \left[1 - \epsilon \cdot \frac{i}{i \cdot (b - 1) + 1} \right] \quad \text{(A.13)}$$

This can now be substituted into equation A.11 to show that the first type T_1 node will be expanded with an expected runtime as follows:

$$\sum_{k=2}^{\infty} \left[k \cdot \epsilon \cdot \frac{k - 1}{(k - 1) \cdot (b - 1) + 1} \cdot \prod_{i=1}^{k-2} \left[1 - \epsilon \cdot \frac{i}{i \cdot (b - 1) + 1} \right] \right] \quad \text{(A.14)}$$

To show that this converges, we will use the ratio test which states that if a_j is the j-th term of an infinite series, the series converges if $a_{j+1}/a_j \rightarrow z$ in the limit as $j \rightarrow \infty$ where $0 \leq z < 1$. After cancelling out the shared portions of the products in a_j and a_{j+1}, this means we must show that the following is a constant less than 1.

$$\lim_{j \rightarrow \infty} \frac{\epsilon \cdot (j + 1) \cdot j/[j \cdot (b - 1) + 1]}{\epsilon \cdot j \cdot (j - 1)/[(j - 1) \cdot (b - 1) + 1]} \cdot \left[1 - \epsilon \cdot \frac{j - 1}{(j - 1) \cdot (b - 1) + 1} \right] \quad \text{(A.15)}$$

The left component of this product approaches $(b-1)/(b-1) = 1$ in the limit, while the right component approaches $1-\epsilon/(b-1)$. Therefore, the whole expression approaches $1-\epsilon/(b-1)$ and so this limit converges for any $\epsilon > 0$. This proves that a type T_1 node will be expanded with a finite expected runtime, which equivalently means that a type T_g node will be expanded with a finite expected runtime.

Now let us show that after the first node expansion, a type T_g node will be expanded with a finite expected runtime. Where $P_g(t = k)$ is the probability that a node of type T_g is expanded for the first time k expansions after a node of type T_g was first generated, the expected runtime is given by the following:

$$\sum_{i=1}^{\infty} i \cdot P_g(t = i) \tag{A.16}$$

In this case, it is difficult to calculate $P_g(t = k)$ exactly as their are many possible combinations for the way nodes are expanded. However, it is well-known that if an infinite series of the form $a_1 + a_2 + ...$ converges then so too will $a'_1 + a'_2 + ...$ if $a_j \geq a'_j$ for all j. Therefore, we will show this sequence converges by identifying a $P'_g(t = i)$ where $P_g(t = i) \leq P'_g(t = i)$ such that the following converges:

$$\sum_{i=1}^{\infty} i \cdot P'_g(t = i) \tag{A.17}$$

This essentially means that we need to find a function that provides an upper bound on the probability of the algorithm expanding a node of type T_g after the first such node is generated. For this upper bound notice that because nodes of type T_g have the lowest heuristic value of any node, once such a node is in the open list it will be expanded the first time the algorithm expands nodes according to the heuristic. This means that the probability that a node of type T_g will not be expanded on any iteration will be at most ϵ. Therefore, the probability that a node of type T_g is not expanded in the first t iterations after a node of type T_g is generated for the first time is at most ϵ^t.

Now recall that the probability that a node of type T_g is expanded for the first time on the t-th expansion after a node of type T_g is first generated is given by probability that no node of type T_g was expanded for the first $t-1$ expansions after a node of type T_g is first expanded, multiplied by the probability that it is expanded

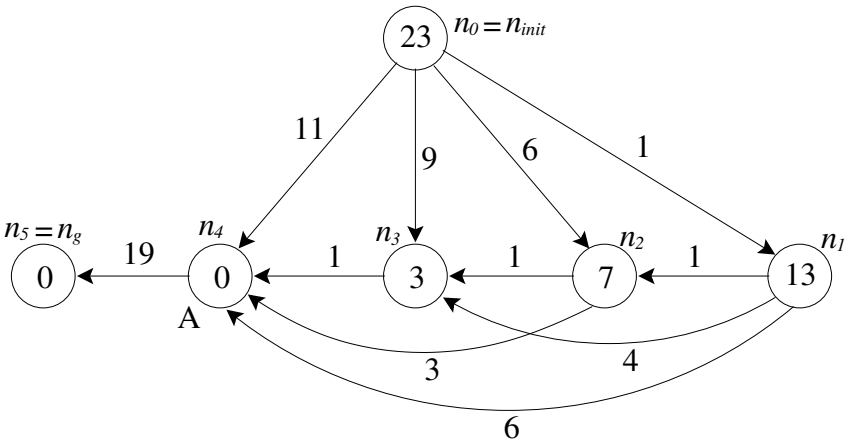


Figure A.1: The 6 node graph in Martelli's graph family.

by the t-th expansion Since the probability that a node of type T_g is expanded on any particular iteration can be at most 1, this means that we can set $P'_g(t = i) = \epsilon^{i-1} \cdot 1 = \epsilon^{i-1}$ as our upper bound on $P_g(t = i)$.

With this upper bound, we can again use the ratio test to show that equation A.17 converges. In this case, the ratio of consecutive elements converges to ϵ. Therefore, the expected runtime of equation A.17 converges if $\epsilon < 1$, and so too does equation A.16 since A.17 is an upper bound on this equation. Therefore, once a node of type T_g is generated for the first time, it will be expanded in a finite expected amount of time if $\epsilon < 1$. When this is combined with the above, it means that a node of type T_g will be expanded with an expected runtime if ϵ satisfies the condition that $0 < \epsilon < 1$.

A.3 nrBFS Performance on Martelli's Graphs

In this section, we will formally demonstrate that nrBFS will find solutions no more than 1.5 times larger than the optimal solution cost in Martelli's family of graphs. We begin with a description of this family.

A.3.1 Formally Defining Martelli's Graphs

For any $M \geq 3$, there is a member of this family, denoted G_M, which has $M + 1$ nodes denoted as $n_0, ..., n_M$. For example, the 6-node member of this family, G_5 is

shown in Figure A.1. n_0 is the initial node in G_M and n_M is the goal node. For any i, j where $0 \leq i < j < M$ there is an edge from n_i to n_j. The last edge is given by (n_{M-1}, n_M). The cost of these edges are as follows:[1]

- $\kappa(n_{M-1}, n_M) = 2^{M-1} + M - 2$.
- Where $0 \leq i < j < M$, $\kappa(n_i, n_j) = 2^{M-i-2} - i - 2^{M-j-1} + j$.

By this definition, the cost of edge (n_i, n_{i+1}) is 1 for any $i < M - 1$. As such, the optimal solution path is given by $[n_0, n_1, ..., n_{M-1}, n_M]$. As all these edges have a cost of 1 except for (n_{M-1}, n_M), the cost of this optimal path is $(M-1) + (2^{M-1} + M - 2) = 2^{M-1} + 2M - 3$.

The admissible heuristic function h used in G_M is defined as follows:

- $h(n_{M-1}) = h(n_M) = 0$.
- For $1 \leq i < M - 1$, $h(n_i) = 2^{M-i-1} + 2(M - i) - 3$.

A.3.2 Using nrBFS on Martelli's graph

As shown by Martelli [61], this definition of the heuristic function will mean that the second node expanded by A* — after only the initial node n_0 — is n_{M-1}. This will also be true of nrBFS since no transposition will have been found by this point. Since the only way to get to the goal node n_M is through n_{M-1}, this means that the solution found by a nrBFS^{g+h} that does not perform parent pointer updates will be $[n_0, n_{M-1}, n_M]$.[2] The cost of this suboptimal path is calculated as follows:

$$\begin{aligned} C([n_0, n_{M-1}, n_M]) &= \kappa(n_0, n_{M-1}) + \kappa(n_{M-1}, n_M) \\ &= (2^{M-2} - 0 - 2^{M-(M-1)-1} + M - 1) + (2^{M-1} + M - 2) \\ &= 2^{M-2} - 1 + M - 1 + 2^{M-1} + M - 2 \\ &= 2^{M-1} + 2^{M-2} + 2M - 4 \end{aligned}$$

[1] This definition differs from the standard definition given by Martelli [61] in which n_0 is the goal node and n_M would be the initial node. This change was made to be consistent with the notation used in the remainder of this thesis.

[2] If parent pointer updates are used, the optimal solution path will be returned even when using nrBFS$g + H$. However, if an edge is added from n_0 to n_M which has the same cost as the path $[n_0, n_{M-1}, n_M]$, then a suboptimal solution will be found with a cost of the path $[n_0, n_{M-1}, n_M]$ regardless of whether parent pointer updating is performed or not.

Notice that $C - C^* = 2^{M-2} - 1$. As $2^{M-2} - 1$ is less than 50% larger than C^* for $M \geq 3$, this means that any solution found is no larger than 1.5 times optimal. In fact, the following shows that $C/C^* \rightarrow 1.5$ as $M \rightarrow \infty$:

$$\lim_{M\to\infty} C/C^* = \lim_{M\to\infty} \frac{2^{M-1} + 2^{M-2} + 2M - 4}{2^{M-1} + 2M - 3} \tag{A.18}$$

$$= \lim_{M\to\infty} \frac{2^{M-1}}{2^{M-1}} \cdot \frac{1 + 1/2 + 2M/2^{M-1} - 42^{M-1}}{1 + 2M/2^{M-1} - 3/2^{M-1}} \tag{A.19}$$

$$= 1.5 \tag{A.20}$$

Now let us consider the inconsistency along the optimal solution path. For this calculation, first notice that since $h(n_{M-1}) = h(n_M) = 0$, $INC_h(n_{M-1}, n_M) = 0$. This means that when calculating the inconsistency needed for the portion of the optimal solution path that matters for the bounds we only need to consider the inconsistency on $[n_1, ..., n_{M-1}]$ since we can ignore the inconsistency of the first edge (n_0, n_1) and $INC_h(n_{M-1}, n_M) = 0$. Now notice that the following holds for any i where $1 < i < M - 2$:

$$\begin{aligned} h(n_i) - h(n_{i+1}) - \kappa(n_i, n_{i+1}) &= \left[2^{M-i-1} + 2(M-i) - 3\right] \\ &\quad - \left[2^{M-i-2} + 2(M-i-1) - 3\right] - 1 \\ &= 2^{M-i-1} - 2^{M-i-2} + 1 \\ &> 0 \end{aligned}$$

Therefore, $INC_h(n_i, n_{i+1}) > 0$ for any i where $1 < i < M - 2$. By Lemma 6.3.2, this implies the first line of the following derivation which then continues based on the definitions above:

$$\begin{aligned} \sum_{i=1}^{M-2} INC_h(n_i, n_{i+1}) &= h(n_1) - h(n_{M-1}) - g^*(n_1, n_{M-1}) \\ &= \left[2^{M-1-1} + 2(M-1) - 3\right] - 0 - (M-2) \\ &= 2^{M-2} + M - 3 \end{aligned}$$

Therefore, the bound says that any solution found will be no more costly than C^* than $2^{M-2} + M - 3$. Since the actual amount of additive suboptimality is $2^{M-2} - 1$ as shown above, the deviation between the bound and the actual cost of the solution found is $M - 3$. As C^* is $O(2^M)$, this deviation is insignificant.

A.4 Bounding the Performance of BFS$^{g+B(h)}$ when h is Non-Decreasing

In this section, we will show that where h is a consistent heuristic and B is a given bounding function, BFS$^{g+B(h)}$ will satisfy B provided that there exists an optimal solution path along which the h-cost of the nodes is monotonically non-decreasing. This is formalized by the following theorem:

Theorem 6.5.5. *BFS$^{g+B(h)}$ will satisfy B if the following conditions hold:*

1. *h is consistent.*
2. *$\forall x \geq 0, y \geq 0, B(x+y) \geq B(x) + y$.*
3. *There exists an optimal solution path $[n_0, ..., n_k]$ such that $h(n_i) \geq h(n_{i+1})$ for all $0 \leq i < k$.*

Proof. Let h be a consistent heuristic, B be a bounding function that satisfies condition 2 above, and $P_{opt} = [n_0, ..., n_k]$ be an optimal solution path such that $h(n_i) \geq h(n_{i+1})$ for all $0 \leq i < k$. We will now show that the inconsistency along P_{opt} will be at most $B(C^*) - C^*$. This will allow us to Theorem 6.3.7 to prove that any solution returned by BFS$^{g+B(h)}$ will have a cost of at most $B(C^*)$.

To show this bound, we will first show that for any j where $0 \leq j < k$, the following is true:

$$INC_{B(h)}([n_j ..., n_k]) \leq B(h(n_j)) - h(n_j) - B(0) \ .$$

In the base case, $j = k - 1$. When this holds, the path consists solely of the edge (n_{k-1}, n_k). As such, the sum of the inconsistency along this path is given by $INC_H(n_{k-1}, n_k)$. There are now two cases to consider: $INC_{B(h)}(n_{k-1}, n_k) = 0$ and $INC_{B(h)}(n_{k-1}, n_k) > 0$.

Let us begin with the case that $INC_{B(h)}(n_{k-1}, n_k) = 0$. To prove that $INC_{B(h)}(n_{k-1}, n_k)$ is no larger than $B(h(n_{k-1})) - h(n_{k-1}) - B(0)$ in this case merely requires that we show that $B(h(n_{k-1})) - h(n_{k-1}) - B(0) \geq 0$. Notice that this follows immediately by the assumption that B satisfies condition 2, and so the statement holds in this case.

Now suppose that $INC_{B(h)}(n_{k-1}, n_k) > 0$. This implies the first line of the following, which allows for the derivation that proceeds it:

$$INC_{B(h)}(n_{k-1}, n_k) = B(h(n_{k-1})) - B(h(n_k)) - \kappa(n_{k-1}) \quad \text{(A.21)}$$

$$= B(h^*(n_{k-1})) - B(0) - \kappa(n_{k-1}, n_k) \quad \text{(A.22)}$$

$$\leq B(h^*(n_{k-1})) - B(0) - h(n_{k-1}) \quad \text{(A.23)}$$

Line A.22 holds by the fact that $h(n_k) = 0$ since h is admissible and n_k is a goal node. The final line then holds since $h(n_{k-1}) \leq h(n_k) + \kappa(n_{k-1}, n_k)$ by the fact that h is consistent and $h(n_k) = 0$. Therefore, the statement is true in the base case.

Now suppose that the statement is true for $[n_j, ..., n_k]$ for some $j > 0$ and consider $INC_{B(h)}([n_{j-1}, n_j, ..., n_k])$. One again, we need to handle the cases in which $INC_{B(h)}(n_{j-1}, n_j) = 0$ and when $INC_{B(h)}(n_{j-1}, n_j) > 0$. Let us first assume that $INC_{B(h)}(n_{j-1}, n_j) = 0$, in which case the inconsistency of $B(h)$ along $[n_{j-1}, n_j, ..., n_k]$ is equal to the inconsistency of $B(h)$ along $[n_j, ..., n_k]$. By the induction hypothesis, this inconsistency is bound by $B(h(n_j)) - h(n_j) - B(0)$. To show the statement is true when $INC_{B(h)}(n_{j-1}, n_j) = 0$, we can therefore show that $B(h(n_{j-1})) - h(n_{j-1}) - B(0)$ is at least as large as $B(h(n_j)) - h(n_j) - B(0)$. In this end, notice that $h(n_{j-1}) \geq h(n_j)$ by the assumption that condition 3 holds for P_{opt}. This means that for some $\mu \geq 0$, $h(n_{j-1}) = h(n_j) + \mu$. This allows for the following derivation:

$$B(h(n_{j-1})) - h(n_{j-1}) - B(0) = B(h(n_j) + \mu) - h(n_j) - \mu - B(0) \quad \text{(A.24)}$$

$$\geq B(h(n_j)) + \mu - h(n_j) - \mu - B(0) \quad \text{(A.25)}$$

$$\geq B(h(n_j)) - h(n_j) - B(0) \quad \text{(A.26)}$$

The first line is achieved by substitution. The second line then follows by the assumption of condition 2 on B. By cancelling out the μ terms we are left with the final line and so the statement is true when $INC_{B(h)}(n_{j-1}, n_j) = 0$.

Now assume that $INC_{B(h)}(n_{j-1}, n_j) > 0$. In this case, we can perform the

following derivation:

$$
\begin{aligned}
&INC_{B(h)}([n_{j-1}, n_j,...., n_k]) \\
&\qquad = INC_{B(h)}(n_{j-1}, n_j) + INC_{B(h)}([n_j, ..., n_k]) && \text{(A.27)} \\
&\qquad = B(h(n_{j-1})) - B(h(n_j)) - \kappa(n_{j-1}, n_j) \\
&\qquad\qquad + INC_{B(h)}([n_j, ..., n_k]) && \text{(A.28)} \\
&\qquad \leq B(h(n_{j-1})) - B(h(n_j)) - \kappa(n_{j-1}, n_j) \\
&\qquad\qquad + B(h(n_j)) - h(n_j) - B(0) && \text{(A.29)} \\
&\qquad \leq B(h(n_{j-1})) - h(n_{j+1}) - B(0) && \text{(A.30)}
\end{aligned}
$$

Line A.27 is a result of expanding the summation. The fact that $INC_{B(h)}(n_{j-1}, n_j)$ is positive then implies line A.28. Line A.29 then holds by the bound given in the induction hypothesis. The last line follows by cancelling out the $B(h(n_j))$ terms and the fact that h is consistent implies that $h(n_{j+1}) \leq h(n_j) + \kappa(n_{j-1}, n_j)$. Therefore, the statement holds in the case that $INC_{B(h)}(n_{j-1}, n_j) > 0$.

Having handled all cases, this means that for any j where $0 \leq j < k$, the following is true:

$$INC_{B(h)}([n_j..., n_k]) \leq B(h(n_j)) - h(n_j) - B(0) \ .$$

In the case that $j = 0$, this means that the inconsistency of H along P_{opt} is no larger than $B(h(n_0)) - h(n_0) - B(0)$. Now recall that $h(n_0) \leq h^*(n_0) \leq C^*$ since $n_0 = n_{init}$. This means that for some $\mu \geq 0$, $C^* = h(n_0) + \mu$. Using the same argument as was used for proving the base case above, this means that $B(C^*) - C^* - B(0)$ is an upper bound on the inconsistency of H along P_{opt}, as is $B(C^*) - C^*$ since $B(0) \geq 0$. As this is also necessarily an upper bound on the inconsistency of H along the portion of P_{opt} from n_1 to n_k, this means that any solutions returned by BFS^{g+H} will have a cost of no more than $B(C^*) - C^* + C^* = B(C^*)$ by Theorem 6.3.7. □

Appendix B

Historical Notes and Additional Information

In this chapter, we will provide further detail in cases where our definitions or notation differ from those existing in the literature, and provide any additional technical details regarding results in the main body of the thesis as needed.

B.1 Nodes, States, and Vertices

Despite the fact that heuristic search research has been conducted since at least the 1960s, the term "node" continues to be used to mean different things by different authors. In this thesis, the term is used to refer to the vertices of a graph representation of a state-space. "Node" and "state" will also often be used interchangeably due to the strong correspondence between these objects. This is also the approach taken in several textbooks including those by Pearl [68], and the textbook from Edelkamp and Schrödl [16].

However, the term "node" has also been used in reference to other objects. For example, Lelis [55] uses the term "node" to refer to the vertices in the **search tree** generated from a state-space problem. The search tree is similar to the graph representation of a state-space problem, except that if there are multiple paths from the initial state to the some state s, each of these paths will end in a unique "node" in the search tree. This means that there may be multiple vertices in the search tree that correspond to the same state.

A similar approach is taken by Thayer [86], who defines a "node" as referring

to the collection of a state and a unique path to a state. This was also the approach taken in my Master's thesis [93], in which "node" refers to the collection of a state and a parent pointer, since the use of parent pointers is typically the way that the path to a state is stored in practice.

Given the multitude of "node" definitions, we use it to refer to a vertex in a graph representation of a state-space, as this definition was found to be the most convenient for our purposes.

B.2 Alternative Best-First Search Definitions

Dechter and Pearl provided the first ever generalized version of best-first search which they referred to as BF* [12]. That definition, which was referred to as BF*, also uses any given evaluation function Φ, but differs from rBFS$^\Phi$ in the way the algorithm decides if re-expansions are to be made. In particular, BF* does not explicitly keep track of the g-cost of a node, only the node's evaluation according to Φ. This means that BF* will move a node back from the closed list to the open list only in the case that the new path results in a lower evaluation of Φ than that being stored. This is often because the new path has a lower g-cost, provided that Φ uses g as part of its evaluation, but the general definition of BF* does not require it to be for that reason.

However, for many of Dechter and Pearl's main results, two key assumptions are made that make the re-expansion policy of BF* equivalent to the policy of rBFS$^\Phi$ by which a node is moved back from the closed list to the open list if the new path has a lower g-cost than the existing one. The first of these assumptions is that Φ is strictly increasing over solution paths. This means that if P_1 and P_2 are solution paths such that $C(P_1) < C(P_2)$, then the evaluation of P_1 by Φ must be less than the evaluation of P_2 (where $C(P_1) = C(P_2)$ implies equality of evaluation as well). The second assumption is that the evaluation function is **order preserving**. For this definition, let $P_1 = [n_0, ..., n_k]$ and $P_2 = [n'_0, ..., n'_j]$ both be paths from $n_0 = n'_0 = n_{init}$ to some node $n = n_k = n'_j$, and continue extending these paths along some common sequence of nodes $[n''_0, ..., n''_i]$ such that $n = n_k = n'_j = n''_0$. The resulting paths

would be $P_1' = [n_0, ..., n_k, n_1'', .., n_i'']$ and $P_2' = [n_0', ..., n_j', n_1'', .., n_i'']$. For Φ to be order preserving it must satisfy the following: if the evaluation of P_1 is no larger than the evaluation of P_2, then the evaluation of P_1' must also be no larger than the evaluation of P_2.

To see how these assumptions make BF* and rBFS$^\Phi$ equivalent along solution paths, consider P_1' and P_2' as defined above and assume these are solution paths such that $C(P_1') < C(P_2')$. This necessarily requires that the evaluation of P_1' is less than the evaluation of P_2', which in turn requires that the evaluation of P_1 is less than P_2 by the fact that Φ is order preserving. As such, if BF* finds path P_1 to n after finding path P_2, the Φ-cost and parent pointer of n will be updated. Of course, this is because $C(P_1) < C(P_2)$ and so if the path along P_2 is found to n before the path along P_1 during BF*, then the Φ-cost of n will be updated. The same will be true in the corresponding rBFS$^\Phi$ search since the g-cost along P_2 will be larger than the g-cost along P_1. The result is that the two algorithms are equivalent under these two assumptions when using the same evaluation function.

B.3 Assuming the Heuristic Value of Goals is 0

In Chapter 6, it was assumed that the heuristic value of any goal node is 0. This assumption is not used except in the proof of Lemma 6.3.3. Where $[n_0, ..., n_k]$ is the optimal path from n_0 to the nearest goal node and H is a heuristic function, this lemma states that the inadmissibility of the heuristic value of n_0 is bound by the inconsistency of H along an optimal path from n to the nearest goal node. If the assumption that all goals have a heuristic value of 0 is removed, then the inadmissibility of $H(n)$ will be increased by $H(n_k)$.

The bound given by Theorem 6.3.7 will similarly increase by $H(n_k)$ if this assumption is removed. When assuming that all goal nodes have a heuristic cost of 0, this theorem states that if $P_{opt} = [n_0, ..., n_k]$ is an optimal solution path then the cost of any solution returned by BFS^{g+H} will have the following as an upper bound:

$$C^* + \sum_{j=1}^{k-1} INC_H(n_j, n_{j+1}) \ . \tag{B.1}$$

If we removed the assumption that the heuristic value of goal nodes is 0, than this upper bound would instead be the following:

$$C^* + H(n_k) + \sum_{j=1}^{k-1} INC_H(n_j, n_{j+1}) \ .$$

However, this increase in the bound only happens if heuristic values of the goal nodes can be assigned arbitrary values. Given some natural restrictions on what the heuristic function H can return for goal nodes, the bound returns to that given in expression B.1 even if these restrictions are weaker than the requirement that the heuristic value of all goal nodes is 0. One such restriction is that all goal nodes have a heuristic value of at least $H(n_k)$ where n_k is the goal node on the optimal solution path P_{opt} from the theorem statement.

To see why the bound given in expression B.1 would still be satisfied under this weaker restriction on H, consider the proof of Theorem 6.3.7. At the time that a goal node n is selected for expansion, the proof identifies that there is a node n_i from P_{opt} that is in the open list. Since n is selected instead of n_i, this means the following is true:

$$\begin{aligned} g(n) + H(n) &\leq g(n_i) + H(n_i) \\ g(n) &\leq g(n_i) + H(n_i) - H(n) \\ &\leq g(n_i) + H(n_i) - H(n_k) \end{aligned}$$

where this last line holds because n is a goal node that is on an optimal path and so $H(n_k) \leq H(n)$. This $-H(n_k)$ term would then cancel out the added $H(n_k)$ term that would appear when $H(n_i)$ is substituted out for the sum of $h^*(n_i)$ and the inadmissibility of $H(n_i)$. The rest of the proof would remain as is and so the same bound holds even if the assumption that the heuristic value of goal nodes is 0 is replaced by the assumption that those goal nodes that lie on an optimal path have the lowest heuristic values of any goal nodes.

B.3.1 Weighting An Admissible Heuristic with a Bounding Function

In Section 6.5 of Chapter 6, we provide upper bounds on the cost of any solution found by BFS$^{g+H_B}$ where $H_B = B(h)$, h is an admissible heuristic, and B is a bounding function. These bounds are also derived using by Theorem 6.3.7.

While the admissibility of h ensures that $h(n) = 0$ for any goal node n, no assumptions were made that would require $B(0)$ to be equal to 0. However, due to the fact that the goal nodes along optimal solution paths will have the lowest H_B-cost of any goal nodes (since $H_B(n) = B(0)$ for all goal nodes), the argument above demonstrates why the proofs of the bounds in Section 6.5 can employ Theorem 6.3.7.