Revisiting the Theory and Practice of Bidirectional and Suboptimal Heuristic Search Algorithms

by

Jingwei Chen

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 is a general problem-solving method widely used in artificial intelligence (AI). This thesis presents contributions to heuristic search, including contributions to bidirectional optimal search and unidirectional suboptimal search.

For bidirectional optimal search, this thesis presents fundamental theory for the analysis of necessary expansions and the minimum possible number of node expansions needed to solve a given problem in front-to-end heuristic search. A new front-to-end heuristic search algorithm, `NBS`, which has a worst case guarantee for the number of node expansions, is also presented in this thesis.

For unidirectional suboptimal search, this thesis presents the theory of best-first bounded-suboptimal search using priority functions that do not need to perform state re-expansions as long as the search heuristic is consistent. Also, particular priority functions, such as piecewise linear functions are presented in this document. Several new priority functions can significantly outperform existing approaches according to empirical results.

Preface

This thesis is an original work by Jingwei Chen. Part of this thesis has been previously published.

Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this research, and without whom I would not have made it through my PhD program!

My supervisor Dr. Nathan R. Sturtevant, whose insight and knowledge into the subject matter steered me through my research career. The faculty and staff that helped me in University of Denver and University of Alberta. And special thanks to Robert Holte, who provides extra guidance and help in my bidirectional search research.

NSF and NSERC, who funded me during my project.

And thanks to my family for all the support you have shown me through this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A heuristic search problem, defined on a given *state space*, is to find a path from an initial state (*start*) to a goal state (*goal*).

One direct application of such problems is to model a routing problem in a road network system [3, 7, 38] or a path planning problem for robots [2, 12, 52, 68].

While it is less obvious, many other practical problems can also be modeled as a heuristic search problem.

One example is puzzle-solving. More specifically, consider the task of restoring a Rubik's cube to its original state. Once we view the given configuration as the start state, the goal state as the original state, the problem is to find a sequence of operations that leads us from the start state to the goal state. A legal operation in playing this puzzle is to turn a face 90, 180, or 270 degrees. If we let the actions be the legal operations, this task can be solved as a heuristic search problem.

Another example is natural language processing (NLP) tasks, such as caption generation and machine translation, which involve generating sequences of words. The target is to generate a full sentence. If we view an empty sentence as a start state, and a complete sentence – a sequence of tokens with an"end of sentence" token – as a goal state, then such a problem can be modeled naturally as a heuristic search problem. Beam search is a popular heuristic search algorithm that is widely used in NLP [18, 32, 93].

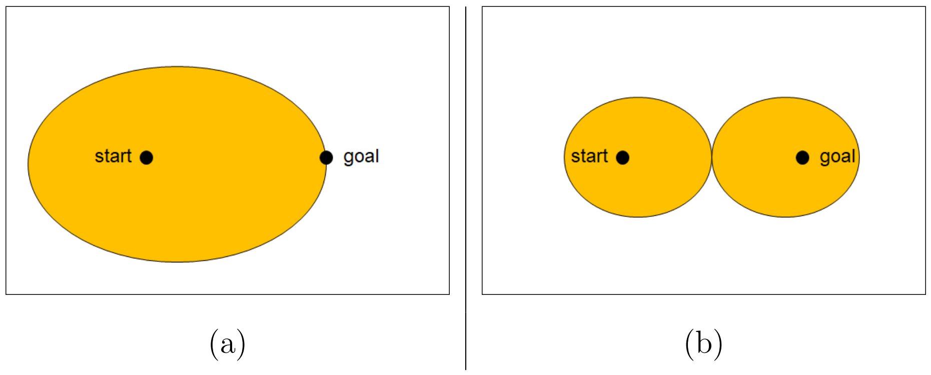


Figure 1.1: Illustration for (a) unidirectional search (b) bidirectional search. The rectangle represents the whole state space, while the yellow zone represents the part that the algorithm explores.

Heuristic search problems are optimization problems. The target is to minimize the search time or the total number of node expansions for a given problem instance. For most algorithms, these two metrics are positively correlated, i.e., fewer total node expansions implies less search time.

The most important constraint is the quality of the path. Problems can be divided into two categories according to this: optimal search problems, where the goal is to find the shortest path, and suboptimal search problems, which trade solution quality for speed.

There are other constraints in practice, such as memory limits, but these will not be addressed in this thesis.

Solvers designed for solving heuristic search problems are called heuristic search algorithms. According to the behavior of the algorithm, we can divide them into two classes: unidirectional algorithms, which explore the state space from start to goal, and bidirectional algorithms, which explore the state space in both directions and try to connect the two frontiers to get a complete path. As illustrated in Figure 1.1 (a) and (b), usually unidirectional algorithms and bidirectional algorithms visit different parts of the state space, resulting in a difference in performance.

Naturally, we can design unidirectional search algorithms to solve optimal search problems or suboptimal search problems specifically. Also, we can have specific bidirectional search algorithms designed for problems of each type. In total, there are four types of algorithms: unidirectional optimal search algorithms, bidirectional optimal search algorithms, unidirectional suboptimal search algorithms, and bidirectional suboptimal search algorithms.

Unidirectional optimal search algorithms is solved, in the sense that A* is optimal among all unidirectional algorithms [20, 42] given a few assumptions; this thesis mainly presents contributions to bidirectional optimal search (in Chapter 3) and unidirectional suboptimal search (in Chapter 4). The discussion for bidirectional suboptimal search algorithms is presented in Chapter 5

1.1 Motivation of this Thesis

The ultimate motivation is to make search algorithms more efficient. The metrics of measuring efficiency include the time and storage usage. In this thesis, the major metric will be the time usage. We would like an algorithm to return an acceptable solution as soon as possible.

1.1.1 Bidirectional Search

When doing optimal search, intuitively, by putting effort from both directions, the total amount of work should be less, since the total number of nodes on two shallower search trees is less than that on one large search tree.

However, although early research on bidirectional heuristic search dates back to the 1960s [35], empirically, algorithms had not demonstrated significant improvements over unidirectional algorithms until 2015.

In the history of bidirectional search, several theories were proposed to explain the poor performance of bidirectional search.

BHPA [65] was proposed in the early 1970s. By looking into the behavior of that algorithm, the first major theory to explain the poor performance is that the frontiers miss [59]. Later, BS* [48] was proposed, which can guarantee that the frontiers will never miss. Unfortunately, the performance of BS* is still disappointing.

Another explanation of the poor performance is that for bidirectional search the solution is found early and most time is spent on proving optimality [44].

However, none of these efforts give bidirectional algorithms desirable performance compared to unidirectional search algorithms. It has even been suggested that bidirectional heuristic search algorithms should never be used [5].

Note that unidirectional search has a solid theory of necessary expansions, i.e., the minimum work to prove the optimality of the solution. In contrast, bidirectional search algorithms were not built upon a theory of necessary expansions. That could be a candidate explanation of the poor performance of bidirectional search: if an algorithm does extra work which is not necessary, it is not strange that the algorithm has poor performance. Also, it seems to be trivial for us to build an optimal bidirectional search algorithm once we have that theory: just do the necessary expansions plus some work to find a optimal solution. Such an algorithm would be optimal among all bidirectional algorithms, just like A* is optimal among all unidirectional algorithms [20].

Thus, this thesis starts by studying the necessary expansions for bidirectional search. From this, we can then produce a bidirectional algorithm, `NBS`, which minimizes the worst case necessary expansions.

1.1.2 Unidirectional Suboptimal Search

Unidirectional suboptimal search relaxes the constraint of the solution quality, so algorithms can have more freedom when exploring a state space. We found it difficult to quantify the necessary expansions that all algorithms must perform. The approach we use to improve suboptimal search is to avoid reopenings and to prioritize the states in a smart way so that, empirically, fewer nodes are expanded.

When Weighted A* [64] was first introduced, updating the cost of states already generated on the *Open* list was always performed. At that time, it was not clear whether such actions are necessary or not. Pearl et al. [62] suggest that reopening and reexpanding states should always be performed once shorter paths are found. However, later work [23, 50] clarifies that it is not necessary for Weighted A* to re-open states to guarantee the found solutions are bounded-optimal.

In this thesis we present the general necessary and sufficient conditions for best-first search to avoid reopenings while still finding feasible solutions. That is, we use priority functions that are more general than Weighted A* (WA*) algorithm. Also, we provide a few priority functions that work well empirically.

1.2 Thesis Outline

The two major bodies of work in this thesis will be addressed in two separate chapters. For each chapter, we provide the problem definition and related work, the contribution to the theory and the contribution to the algorithms.

Contributions include: (1) A theory of front-to-end bidirectional search [24]; (2) The `NBS` algorithm [13]; (3) Theory of the cost of re-expansions [16]; (4) Necessary and sufficient conditions for avoiding reexpansion in suboptimal search [14, 15]; (5) The Improved Optimistic Search (IOS) [16] algorithm.

Chapter 2

Brief Introduction to Search

Search algorithms are a class of solvers designed to solve a type of problem, which we call search problems. Search problems are problems whose start state and goal state (or goal check) are provided, but the correct sequence of steps that transforms the start state to goal state is not known a priori and is what we are pursuing. A sequence from start to goal is called a path. According to the cost of the path, the problems can be categorized into a few types such as shortest path problems, satisficing search, longest path problems, etc.

For different types of problems, there are different solvers designed for them. In the following chapters, we will address those types and solvers in detail.

2.1 Assumptions about Search Algorithms

In the following context, we assume the algorithms are deterministic, expansion-based and black box (DXBB) [24].

Algorithm 1 is the pseudo-code for state expansion. An expansion is the procedure of moving a state from the *Open* list to the *Closed* list, and then adding or updating its successors' information on the *Open* list.

Deterministic The algorithm will behave exactly the same each time on the same instance I. The algorithm does not make any random choices.

Expansion-based The only way for the algorithm to figure out the topology of

Algorithm 1 Expand(n)

```
1: Move $n$ from $Open$ to $Closed$
2: for each $w \in successor(n)$ do
3:     if $w \notin Open \cup Closed$ then
4:         Add $w$ to $Open$
5:     else// $w$ is already generated
6:         if $g(w) \leq g(n) + d(n, w)$ then
7:             Continue for loop // discard $w$
8:         else
9:             $g(w) = g(n) + d(n, w)$
10:        end if
11:    end if
12: end for
```

the graph is expansion.

Black Box The algorithm only has black box access to the graph information such as the cost function and heuristic functions.

It does not mean that there cannot be other types of algorithms. There are many randomized algorithms, for example ones that break ties randomly [1, 49, 92] or that use randomized methods to diversify the kind of nodes selected for expansion such as Monte Carlo tree search (MCTS) [11], which is widely used in reinforcement learning.

Also, there are algorithms that are not expansion-based, such as SAT-based solvers [26, 56] and Integer Linear Programming [31, 70].

2.2 Heuristic Types

In general, a heuristic function $h(u, v)$ is a function that provides an estimate of the optimal path cost from u to v. It maps a pair of states to a real number, Which is called the heuristic value. Sometimes we only care about the cost from one state to an end point (in unidirectional search, the end point mentioned here is the goal). Such a heuristic, $h(n, goal)$, is usually simplified to $h(n)$.

For pruning purposes, we prefer heuristic functions with particular properties. A heuristic function is called admissible if $\forall n, h(n) \leq d(n, goal)$, where $d(u, v)$ is the cost of the shortest path from u to v.

A heuristic function is called consistent if $\forall m, n, h(m) \leq d(m, n) + h(n)$. Note that when a consistent heuristic function satisfies $h(goal) = 0$, it is also admissible.

These are the general definitions of admissibility and consistency. In Section 4.1, we will introduce definitions of strongly and weakly consistent heuristics.

2.3 Search Domains

The primary purpose of this section is to briefly introduce the domains on which the experiments are run in later chapters. For each domain, the definition of states, size of the state space, start and goal states, legal actions, and common heuristic functions will be covered.

2.3.1 Grid Map Pathfinding

The grid map pathfinding problems are benchmarks by Sturtevant [81]. These are actual maps adapted from well-known video games, including Starcraft (SC) and Dragon Age: Origins (DAO).

In different games, the terrain is different, thus the openness of the map and the connectivity of areas is very different in each test set.

State A state on 2D grid maps is the current location, represented by a pair of integers referring to its coordinate.

Size of State Space The size of the state space depends on the size of the map. For a 512x512 map, there are approximately 2.5×10^4 states in total.

Start State and Goal State The start state and goal state are usually randomly picked legal states.

Action The maps can be 4-connected or 8-connected. In this document, unless otherwise specified, the maps are 8-connected. In this case, an action is to move from the current cell to one of its legal eight adjacent cells. The definition of 4-connected graph is analogous: an action is a move to one of the four neighbors. Figure 2.1 is an example of an 8-connected graph. For state S, moving to any the cells surrounding it labelled from S_1 to S_8 are legal actions [1].

Heuristic The heuristic used in this document is octile distance. On 8-connected graphs, when the cost of a diagonal move is cheaper than the sum of a vertical move and a horizontal move, then the shortest path should build upon as many as possible diagonal moves. Figure 2.2 shows an example where the h-cost is $2D + 1$, where D is the cost of a diagonal move the cost of cardinal move is 1. In general, the octile heuristic between two states whose coordinates are (x_1, y_1) and (x_2, y_2) is $min\{|x_1 - x_2|, |y_1 - y_2|\} \times D + (max\{|x_1 - x_2|, |y_1 - y_2|\} - min\{|x_1 - x_2|, |y_1 - y_2|\})$. The octile heuristic is admissible and consistent. There are more advanced heuristics, such as differential heuristics [34, 57], but they are not used in our experiments.

[1] A diagonal move is legal only if both associated cardinal moves are legal

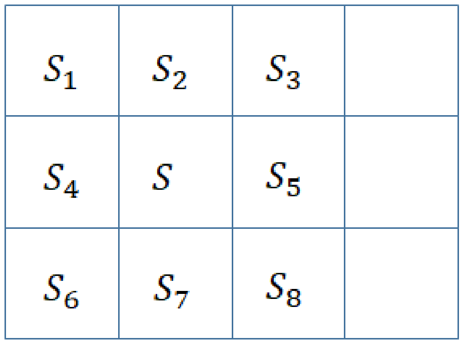


Figure 2.1: An octile grid example.

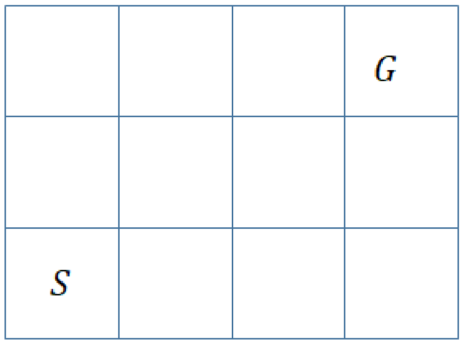


Figure 2.2: An octile heuristic calculation example. The octile heurisic between S and G is 4, given the cost of cardinal move is 1 and the cost of a diagonal move is 1.5.

2.3.2 Sliding-tile Puzzles (STP)

Sliding-tile puzzles includes the 4 x 4 Fifteen Puzzle, and its smaller relative the 3 x 3 Eight Puzzle. There are puzzles of even larger size, such as the 5 x 5 Twenty-Four Puzzle.

For those N-puzzles, there are N 1x1 tiles that are numbered from 1 to N. There are some other variants which employ letters or use larger tiles. Figure 2.3 is an example of a variant called Klotski puzzle, or Huarong Dao.

State A state is the arrangement of current tiles.

Size of State Space For a standard N-puzzle, the size of the state space is $(N + 1)!/2$. For example, for the fifteen-puzzle, the size is $16!/2$ which is about 1×10^{10}.

Start State and Goal State An example of a solved 15-puzzle as a board game is shown in Figure 2.4(a). The goal state is where each tile sits on its own cell. For example, tile 1 in cell 1, tile 2 in cell 2, ... etc. However, computer scientists prefer to use an alternative representation, which is shown in Figure 2.4(b). In such representation, the empty cell is at index 0, tile 1 is at index 1, ... etc.

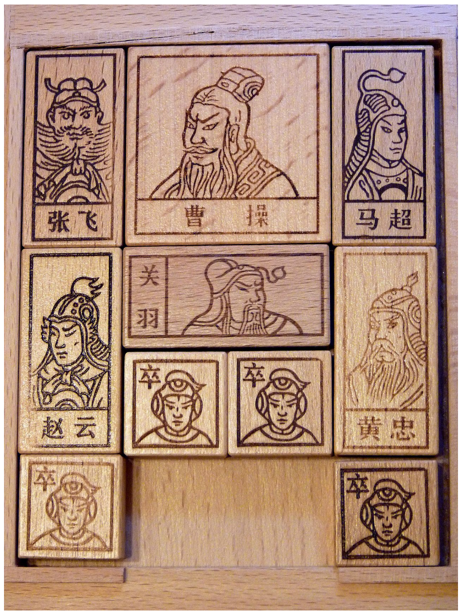


Figure 2.3: A Klotski puzzle [43].

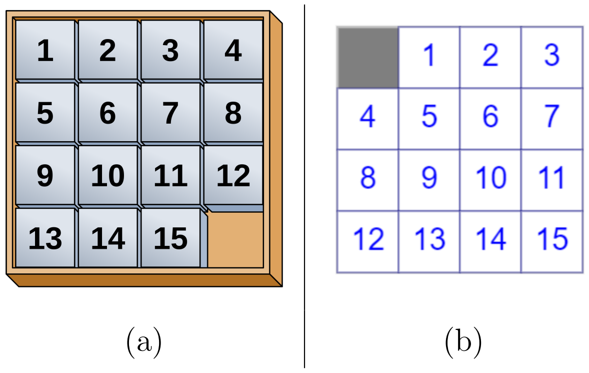


Figure 2.4: Illustration for a solved 15-puzzle (a) standard representation used in board game, with the empty cell in bottom right corner (b) standard representation used by computer scientists, with the empty cell in the top left corner.

The start state is usually generated by applying random actions to the goal state. For example, the Korf instances [46] were generated by applying 100 random moves.

Action The legal moves of a state are to swap the empty slot with an adjacent tile.

Heuristic There are two types of heuristics that are widely used in this domain, Manhattan distance(MD) and pattern databases(PDBs).

Manhattan distance: for each tile, count its Manhattan distance from its current location to target location, and then sum the MD of all tiles up. This heuristic is admissible and consistent.

Pattern databases: Generic Pattern database (PDB) heuristics [17] are lookup tables containing the distance to the goal in an abstracted state space. Such heuristics are admissible and consistent. There is a trade-off between the heuristic strength and the memory usage: the larger the PDB, the stronger the heuristic. Usually the size of the lookup table is chosen carefully to not exceed the size of main memory. But there are approaches for using PDBs which cannot fit in memory. An example technique is to compress the values to fit the PDBs into memory [84].

Specifically, in the sliding tile puzzle, the PDB heuristic is generated through a relaxed problem by unlabling some of the tiles, solving the relaxed problem using breadth-first search, and then using that value as the heuristic for a given state.

2.3.3 Rubik's Cube

Rubik's Cube was invented in 1974 by Erno Rubik of Hungary. The standard version consists of a 3 x 3 x 3 cube, with different colored stickers on each of the exposed squares of the subcubes.

State A configuration of the cubes.

Size of State Space For a 3 x 3 x 3 cube, the size of the state space is approximately 4.3×10^{19}.

Start State and Goal State The goal state is a cube whose every face is the same color. The start state is usually generated by applying random moves to the goal state.

Action The actions allowed in solving this puzzle are rotations of 90, 180, or 270 degrees of a face relative to the rest of the cube.

Although it was proven that every puzzle an be solved in at most 20 moves, its large state space makes it difficult to find an optimal solution for a given configuration in a brute force way. An algorithm introduced in Chapter 3, PEMM, uses bidirectional brute force search solved a 20-move puzzle for the first time [82].

Heuristic In this domain, pattern databases (PDBs) heuristics are effective. Rep-

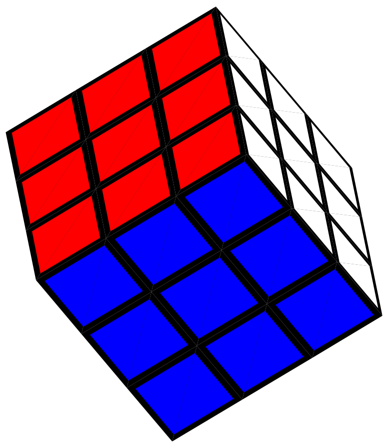
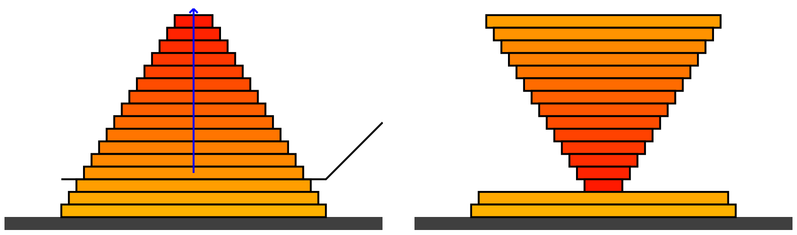


Figure 2.5: A 3 x 3 x 3 Rubik's Cube in original state.



(a) (b)

Figure 2.6: The pancake puzzle states (a) before (b) and after a flip action.

resentative PDBs include the Korf 1997 PDB which computes the moves of 8 corners, 6 edges and other 6 edges (8-6-6) [47], and 12-edge heuristics [83].

2.3.4 4-peg Tower of Hanoi (4-peg TOH)

Although the three-peg version of TOH has a simple recursive solution which has long been known, the 4-peg variant is a non-trivial problem.

State The state is current status of the disks and the pegs.

Start State and Goal State The start state is a state where the disks are in several pegs, while the goal state is all the disks are in another peg.

Action A legal action is to move a disk that is on the top of one peg to another peg, whose top disk is larger.

Heuristic In this domain, the heuristics used are PDBs.

2.3.5 Pancake Puzzle

The pancake puzzle is a problem of sorting a disordered stack of pancakes using a spatula that can be inserted at anywhere in the stack to flip all pancakes above it. Figure 2.6 demonstrate how a flip changes a pancake puzzle state.

It has been shown that the maximum steps for solving any pancake puzzle instance is approximately between $1.07n$ and $1.64n$, where n is the number of pancakes [30]. However, the exact number for solving a particular problem is unknown and needs a solver to find.

State Current stacks of pancakes.

Size of State Space For the pancake puzzle of size n, the size of the state space is $n!$.

Start State and Goal State The start state is a random permutation of pancakes, while the goal state is the pancakes are in sorted order.

Action An action in pancake puzzle is to flip the top k pancakes, where $k \leq n$. Figure 2.6 shows the stacks before and after a flip.

Heuristic The heuristic in this domain is the GAP heuristic [36]. The GAP heuristic is calculated by adding 1 for every two adjacent numbers that are not consecutive.

There is also a weighted version of this problem, which is harder to solve [33]. In regular pancake puzzle, the cost of an action, i.e., a flip, is of cost 1. In this weighted version, the cost is the larger one of the two end points. i.e., flipping a prefix of $V[1], V[2], \cdots, V[i+1]$ pancakes would cost $max\{V[1], V[i+1]\}$.

2.3.6 Planning Domains

For planning domains, we used the problem sets from the international planning competition (IPC) 2018. Since our algorithm requires the heuristic to be consistent, we have to use the FF heuristic [39], which is not the state-of-the-art heuristic. Details are addressed in Chapter 4.

Chapter 3

Bidirectional Optimal Search

3.1 Background and Introduction

Dijkstra's algorithm [21] is designed to solve shortest path problems. This algorithm is guided only by the information of g-cost, where $g(n)$ is the cost of the cheapest known path from start to node n. Such uninformed search algorithms are called unidirectional brute-force search, denoted Uni-BS.

A few years after the invention of Dijkstra's algorithm, research on bidirectional search started in 1966 [22, 58]. Bidirectional search algorithms interleave two separate searches, a forward search from the start and a search backwards from the goal. We can define g-cost in a similar way as uni-BS. For the backward search, the g-cost, $g_B(n)$ is the cost of a path from n to $goal$. Bidirectional algorithms that only exploit g-cost are called bidirectional brute-force search (Bi-BS). Here is the intuition why Bi-BS should outperform Uni-BS: suppose the state space is 2-dimensional, i.e., the number of states whose optimal g-cost is less than d is roughly proportional to d^2. Then solving a problem of cost C^* using Uni-BS will have to expand all the states with g-cost less than C^*, which means the total number of expansions will be $k \times {C^*}^2$, where k is a constant. While, if we use a Bi-BS that meets at the half way point, i.e. in each direction the Bi-BS will not expand states with g-cost greater than C^*, then in each direction it will only have to do $k \times (C^*/2)^2$ expansions. Since $2k \times (C^*/2)^2$ is smaller than $k \times {C^*}^2$, we could expect some benefit from using Bi-BS instead of

Uni-BS. This simple argument is that the area of two smaller circles, whose radius are half of the larger one, is less than the area of one large circle.

In fact, for even higher dimensions or exponential domains, the improvement should be even more significant. If the state space is exponential, and the branching factor is b, then in solving a problem with $C^* = d$, Uni-BS has to do b^d expansion, while Bi-BS could reduce the effort to $2b^{(d/2)}$.

Almost at the same time Bi-BS algorithms were proposed, in 1968, a unidirectional heuristic search (Uni-HS) algorithm, A* was invented [35]. The idea is to prioritize the states by $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function that provides an estimate of the remaining path cost from n to goal.

A* is successful since the heuristic function can guide the search in a goal-oriented way, thus reducing the node expansions.

Since then, researchers have been proposing bidirectional heuristic search (Bi-HS) algorithms, hoping that such algorithms can benefit from heuristics and bidirectionality. However, the performance of such algorithms rarely met expectations.

There are two types of heuristic functions used in bidirectional search. One is called front-to-front heuristic function, $h(u, v)$, which estimates the distance between any two states (u and v). The other is called front-to-end heuristic function, which estimates the cost from some state to an end state. Front-to-end heuristics include forward heuristics and backward heuristics. A forward heuristic function $h_F(n)$ provides an estimate of the cost from n to *goal*, while a backward heuristic function $h_B(n)$ estimates the cost from *start* to n.

Examples of front-to-front algorithms include BHFFA2 [10], SFBDS [29, 51], and BIDA* [53].

Algorithm 2 is the pseudo-code of generic bidirectional search. Line 5 is the condition check for the main loop. If one of the *Open* lists becomes empty during the search, the state space is guaranteed to be fully explored in that direction and we can safely terminate. There are mainly three facts that make bidirectional search different from and harder than unidirectional search:

(1)***Direction selection*** (in line 6). Unidirectional search algorithms always do the forward expansion, while bidirectional algorithms have the freedom of choosing which direction to expand.

(2)***Termination*** (in line 7). As long as the heuristic function is admissible, the solution of cost C found by A* can be proven to be optimal once all the paths with cost $< C$ are expanded. Thus, A* can terminate with cost C once all the paths with cost $< C$ are expanded. However, the situation in bidirectional search is trickier. First of all, the first solution found is not guaranteed to be optimal, even if the heuristic is admissible. Thus, bidirectional search algorithms need some termination condition check to guarantee that the solution they return is optimal, i.e., any possible solution found afterwards will not be better than current solution. There is a lot more information that can help us to prove the optimality of the current solution, including the forward f-cost and the backward f-cost. What we pursue is

Algorithm 2 Pseudo-code for Generic Bidirectional Search Algorithm

```
1: $g_F(start) = g_B(goal) = 0$;
2: $C \leftarrow \infty$
3: $Open_F \leftarrow start$; $Open_B \leftarrow goal$
4: $Closed_F \leftarrow \emptyset$ ; $Closed_B \leftarrow \emptyset$
5: while $Open_F \neq \emptyset$ and $Open_B \neq \emptyset$ do
6:     Choose $dir \in \{forward, backward\}$
7:     if TerminationCondition is met then
8:         return $C$
9:     end if
10:    if $dir == foward$ then
11:        Choose $best \in Open_F$
12:        if $best$ is on the other frontier then
13:            Update $C$
14:        end if
15:        Forward-Expand($best$)
16:    else//do the expansion in the backward direction, analogously
17:    end if
18: end while
19: return $C$
```

the termination condition can help the algorithm terminate as early as possible.

(3)***Solution Detection*** (in line 12). For unidirectional search algorithms, the solution detection is simple: just check if the current state is the goal state. While in bidirectional search, a candidate solution is found when the two frontiers meet. Therefore, for bidirectional search algorithms, the solution detection is to check whether the current state is directly connected to the frontier of the other side. If the frontiers are stored in memory, then the algorithms can use hash tables to perform solution detection; however, if the frontiers are stored in external memory or not even stored, then solution detection is challenging.

Pohl proposed BHPA [65], which keeps expanding states until its termination condition is met. BHPA sorts states by f-cost in each direction and thus keeps track of the minimum f-cost of each direction. The termination condition used in BHPA is f_{min}: if the f_{min} of one direction is greater or equal to the cost of the current solution, the algorithm can safely terminate. The direction selection uses a cardinality criterion: the direction with fewer states on its *Open* list is chosen. Once the direction is determined, a state with minimum f-cost in that direction will be picked for expansion. In his experiments, however, Pohl found BHPA is worse than A* [1]. His explanation is that the shape of the expanded area for BHPA is not circles but cones. In practice, the cones did not meet each other near the middle of their separation but near the endpoints, causing almost twice the work of that of Uni-HS. Such an explanation is called"frontier missing". For a period, techniques such as wave shaping (front-to-front) were developed for pushing the search frontiers towards each other to solve this issue. BHFFA [79] and BHFFA2 [19] are algorithms of this family.

However, Kwa [48] observed that the frontiers weren't missing; they were actually going through each other, called "frontier crossing". If frontier crossing happens, then it means that there exist some states that get expanded in both directions. The algorithm he proposed, BS*, fixed the issue of frontier crossing by nipping and

[1]In the paper he call it HPA, which is exactly what we call A* nowadays

pruning techniques, which rely on the consistency of the heuristic. The direction selection strategy and the termination condition used in BS* is identical to BHPA. The performance of BS* is improved compared to BHPA, by up to 30%; however the algorithm is still slightly worse than Uni-HS in most domains.

Later, Kaindl and Kainz [44] suggested that BHPA and BS* actually find solutions early but spend most of the effort on verifying optimality. They proposed Switch-A*, hoping to prove the optimality of the solution more efficiently [45]. The idea of Switch-A* is to use bidirectional search to find a solution with cost U, and then switch to unidirectional search to exhaust all the states with f-cost less than U to prove the optimality of the solution.

Although the theories above were widely accepted, there was not an algorithm proposed that could overcome all the issues. In 2015, Barker and Korf suggested that Bi-HS is never beneficial [4]: 1. If more than half of the nodes expanded by Uni-HS have $g(n) \leq \frac{1}{2}C^*$, then Uni-HS will beat Bi-HS. 2. Otherwise, if more than half of the nodes expanded by Uni-HS have $g(n) > \frac{1}{2}C^*$, then the heuristic is so weak that Bi-HS won't do any better than Bi-BS.

Such claims drew people's attention. Soon people notice that previous Bi-HS algorithms do not have a guarantee on the meeting point of the search, while Barker and Korf's conjectures assume Bi-HS meets at $\frac{1}{2}C^*$. A new Bi-HS algorithm, `MM`, was proposed to fix that issue [40]. `MM`, which stands for meeting in the middle, has a strong theoretical guarantee that no states with $g(n) > \frac{1}{2}C^*$ will be expanded, i.e., the search frontiers will meet exactly halfway in terms of cost. Also, by defining the regions and analyzing the work done by each type of algorithm, Holte et. al [40] proposed general rules for predicting when Bi-HS will outperform Uni-HS and vice versa. Figure 3.1 illustrates how the g-cost and f-cost divide the states into different regions. N stands for "near", F stands for "far" and R stands for "remote". If the distance from start to a state n is $\leq \frac{1}{2}C^*$, it is called "near" the start; if the distance from start to a state n is between $(\frac{1}{2}C^*, C^*]$, it is called "far" from start; if the distance from start to a state n is $> C^*$, it is called "remote" from start. The distance to goal is defined analogously. Thus, there are 9 regions in total, NN, NF, RN, etc. States in NF regions are "near" the start but "far" from the goal.

A* will expand states in NF, FF, NN, FN; wheveras `MM` will expands states in NF, NR, NN, FN, RN. Note that FF is expanded by A* but not `MM`, while NR and RN are expanded by `MM` but not A*. Qualitatively, if FF is large and NR/NR is relatively small, then `MM` is promising to outperform A*. Unfortunately, the sizes of these regions may not be known a priori.

Algorithm `MM` assigns each state u on $Open_F$ a priority value $Pr_F(u) = max\{f_F(u), 2g_F(u)\}$ and each state v on $Open_B$ with a priority value $Pr_B(v) = max\{f_B(v), 2g_B(v)\}$.

Algorithm `MM` still falls into the framework of Algorithm 2. It chooses the direction with smaller Pr_{min} to expand. The termination condition is revolutionary: it is the first algorithm that exploits the $g_F + g_B$ – all previous algorithms only used f-cost for termination condition. `MM` terminates when $U \leq max\{C, fmin_F, fmin_B, gmin_F + gmin_B\}$, where $C = min\{Pr_{min_F}, Pr_{min_B}\}$.

The solution detection of a variant of MM using external memory, PEMM [82], is also innovative. As is mentioned before, if all the states on the *Open* list can fit in memory, then data structures such as a hash table can be used to perform solution detection efficiently. However, when the state space is large and the problem is difficult, it could be entirely possible that the *Open* lists cannot fit in memory. Since random access to external memory such as disk is a few magnitude slower than access to RAM, the performance of such an algorithm will become very poor. In the PEMM paper, to resolve this issue, Sturtevant suggested that this can be solved by storing states in buckets and performing delayed solution detection (DSD). Once the states with same g-cost and same hash value are stored in the same buckets, we can perform solution detection on a bucket of states instead of a single state. Thus, the access to external memory is batched and the amortized time for accessing the external memory is significantly reduced.

MM itself still has a few limitations. First, empirically, it is still worse than A* in most benchmarks [40]. It remains unclear whether we can design a Bi-HS algorithm that universally outperforms A*. Second, it is not justified that meeting in the middle is the best choice. Later studies [73] show that we can choose anywhere we want to meet. It seems that if the state space is unbalanced, meeting in some fraction instead of the middle will result in fewer node expansions.

However, MM did achieve success to some extent. Its termination condition inspired the study of the sufficient condition in front-to-end bidirectional search, which will be addressed in the next section. Also, it laid the foundation for fMM [73], which is an optimal off-line bidirectional algorithm and will be addressed in the end of this chapter.

Looking at those limitations discussed above, we realized that one piece was missed in Bi-HS's study: the theory of necessary expansions.

In 1985, A* was proven to be "optimally efficient" [20], i.e., under certain assumptions, A* is doing the minimum work among all possible Uni-HS algorithms. The argument is that there exist a set of states that must be expanded by all admis-

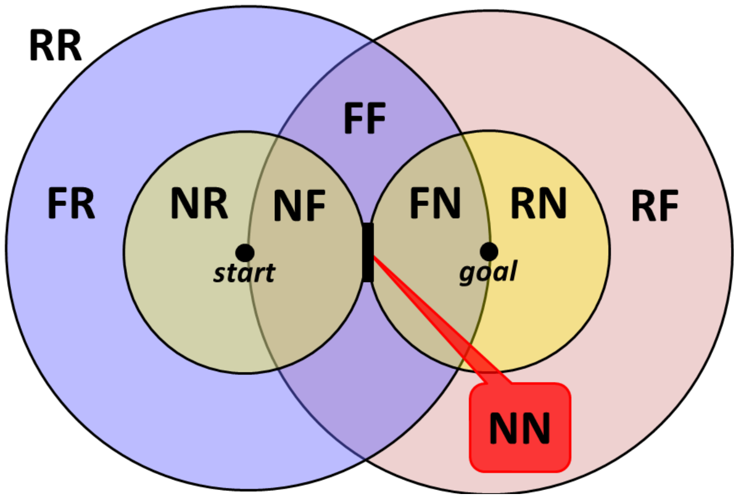


Figure 3.1: MM-region analysis. N=Near, F=Far, R=Remote [40].

sible algorithms and A* only expands those states. For bidirectional front-to-end search, there has been no such theory that defines necessary expansion. As a result, it was impossible to benchmark a Bi-HS algorithm: we do not know exactly how its performance is compared to the true minimum. (There is a theory for front-to-front heuristic search [24, 25]. The statement is very similar to the form of that of front-to-end search [24].)

In the following sections, we will develop theories that lay the fundamental basis for understanding Bi-HS.

(1) The sufficient condition for node expansions. Unlike Uni-HS theory, which provides conditions as a set of necessary states, Bi-HS theory provides conditions as necessary pairs of states. Based on this theory, we then show that no single state is surely expanded by all Bi-HS algorithms. This fact makes it tricky to analyze the minimum amount of work to solve a Bi-HS problem.

(2) We show that we can still find the theoretical minimum number of expansions by reducing the search problem to a vertex cover problem; all algorithms must do at least $|VC|$ node expansions, where VC is a minimum vertex cover of a particular bipartite graph.

(3) We show that no online algorithm can be guaranteed to always do fewer than $2|VC|$ node expansions. Suppose the behavior of the algorithm allows it to return a solution with less than $2|VC|$ node expansions in some instances, then there must exist some other instances on which the algorithm needs at least $2|VC|$ node expansions.

Based on these theories, we build an algorithm, `NBS`, which has a 2x guarantee on the worse case node expansions.

There are a few recent related studies that are built on these theories. Details can be found in section 3.6.

3.2 Assumptions and Terminology

We are trying to solve the front-to-end bidirectional search problem, which is defined by a 5-tuple $\{G, start, goal, h_F, h_B\}$.

A state space G is a finite directed graph whose vertices are states and whose edges are pairs of states.[2] Each edge (u, v) has a cost $c(u, v) \geq 0$. A forward path in G is a finite sequence $U = (U_0, \ldots, U_n)$ of states in G where (U_i, U_{i+1}) is an edge in G for $0 \leq i < n$. We say that forward path U contains edge (u, v) if $U_i = u$ and $U_{i+1} = v$ for some i. Likewise, a backward path is a finite sequence $V = (V_0, \ldots, V_m)$ of states where (V_i, V_{i+1}) is a "reverse" edge, i.e. (V_{i+1}, V_i) is an edge in G for $0 \leq i < m$. Backward path V contains reverse edge (u, v) if $V_i = u$ and $V_{i+1} = v$ for some i. The reverse of path $V = (V_0, \ldots, V_m)$ is $V^{-1} = (V_m, \ldots, V_0)$. The cost of a reverse edge equals the cost of the corresponding original edge. A path pair (U, V) has a forward path (U) as its first component and a backward path (V) as its second component.

If U is a path (forward or backward), $|U|$ is the number of edges in U, $c(U)$ is the cost of U (the sum of the costs of all the edges in U), U_i is the i^{th} state in U $(0 \leq i \leq |U|)$, and $c(U, i)$ is the sum of the costs of the first i edges in U (the cost to reach state U_i via path U). $U_{|U|}$ is the last state in path U, which we also denote $end(U)$. $\lambda_F = (start)$ and $\lambda_B = (goal)$ are the empty forward and backward paths from *start* and *goal*, respectively. Note that $end(\lambda_F) = start$ while $end(\lambda_B) = goal$. Both λ_F and λ_B have a cost of 0. Forward (backward, resp.) path U is optimal if there is no cheaper forward (backward, resp.) path from U_0 to $end(U)$. $d(u, v)$ is the distance from state u to state v, i.e., the cost of the cheapest forward path from u to v. If there is no forward path from u to v then $d(u, v) = \infty$.

Given two states in G, *start* and *goal*, a solution path is a forward path from *start* to *goal*. $C^* = d(start, goal)$ is the cost of the cheapest solution path.

h_F and h_B are the front-to-end heuristic functions. h_F is used to guide the forward

[2] If G has multiple edges from state u to state v, we ignore all but one cheapest of them.

search and h_B is used to guide the backward search. The following is the definition of a front-to-end heuristic function:

Definition 1 *A* front-to-end heuristic *maps an individual state in G to a non-negative real number or to ∞. Front-to-end heuristic h_F is* forward admissible *iff $h_F(u) \leq d(u, goal)$ for all u in G and is* forward consistent *iff $h_F(u) \leq d(u, u') + h_F(u')$ for all u and u' in G. Front-to-end heuristic h_B is* backward admissible *iff $h_B(v) \leq d(start, v)$ for all v in G and is* backward consistent *iff $h_B(v) \leq d(v', v) + h_B(v')$ for all v and v' in G.*

A problem instance is solvable if there is a forward path in G from *start* to *goal*. I_{AD} is the set of solvable problem instances in which h_F is forward admissible and h_B is backward admissible. I_{CON} is the subset of I_{AD} in which h_F is forward consistent and h_B is backward consistent. A search algorithm is I_{AD}-*admissible* iff it is guaranteed to return an optimal solution for any problem instance in I_{AD}.

We assume the algorithms are Deterministic, Expansion-based and Black Box (DXBB). Details of the definition of those assumptions can be found in Chapter 2; the following is a brief overview of the concept.

Deterministic The algorithm will behave exactly the same each time on the same instance I. The algorithm does not make any randomized choices of the path to expand.

Expansion-based The only way for the algorithm to figure out the topology of the graph is by expansion. Before expansion, the algorithm cannot know the number or other properties of the neighbors of a state.

Black Box The algorithm only has black box access to the cost function and heuristic functions. Without making a query to the cost function and the heuristic functions, the algorithm cannot know the exact costs or heuristic values.

3.3 Theory of Minimum Number of Expansions in Front-to-end Bidirectional Search

3.3.1 A Sufficient Condition for Node Expansion for Front-to-End Bidirectional Search

For any forward path U with $U_0 = start$ define

$$f_F(U) = c(U) + h_F(end(U))\,,$$

and for any backward path V with $V_0 = goal$ define

$$f_B(V) = c(V) + h_B(end(V))\,.$$

Definition 2 *For path pair (U, V) define*

$$lb(U, V) = \max\{f_F(U), f_B(V), c(U) + c(V)\}\,.$$

Note that when h_F is forward admissible and h_B is backward admissible, $lb(U, V)$ is a lower bound on the cost of a solution path of the form UZV^{-1}, where Z is a forward path from $end(U)$ to $end(V)$.

Theorem 3 *Let a problem instance $I = (G, start, goal, h_F, h_B) \in I_{CON}$ have an optimal solution cost of C^*. If U is an optimal forward path and V is an optimal backward path such that $U_0 = start$, $V_0 = goal$, and:*

$lb(U, V) < C^*$

then, in solving problem instance I, any admissible DXBB bidirectional front-to-end search algorithm must expand $(end(U), end(V))$.

Proof. We prove the contrapositive. Suppose I, U, and V satisfy the premises of the theorem, and that B is a DXBB bidirectional front-to-end search algorithm that solves I correctly (returns a path $B(I)$ costing C^*) without forward expanding $end(U)$ or backward expanding $end(V)$. Then a new problem instance $I' = (G', h'_F, h'_B) \in I_{AD}$ can be constructed having an optimal solution strictly cheaper than C^* on which B

also returns path $B(I)$ (costing C^*), thereby showing that B is not an admissible algorithm.

$I' = (G', h'_F, h'_B)$ is defined as follows: $h'_F = h_F$, $h'_B = h_B$, and G' has all the vertices in G and all the edges in G except the edge, if there is one, from $end(U)$ to $end(V)$, plus one new edge e from $end(U)$ to $end(V)$ costing

$$\begin{aligned} c(e) = \max\{ \quad & h_F(end(U)) - c(V), \\ & h_B(end(V)) - c(U), \\ & \tfrac{1}{2}(C^* - (c(U) + c(V))) \quad \} \end{aligned}$$

$c(e)$ is positive because $C^* > c(U)+c(V)$. This new edge creates a solution path UV^{-1} whose total cost is $c(U) + c(V) + c(e)$, which is equal to $\max(f_F(U), f_B(V), \frac{1}{2}(C^* + c(U)+c(V)))$. According to the theorem's premises, $lb(U,V) < C^*$, which is equivalent to $\max(f_F(U), f_B(V), c(U) + c(V)) < C^*$, we know $\max(f_F(U), f_B(V), \frac{1}{2}(C^* + c(U) + c(V)))$ is strictly less than C^*. Thus the new edge is an essential part of any optimal solution to I'.

We begin by proving that $I' \in I_{AD}$, i.e. that h_F is forward admissible on G' and h_B is backward admissible on G'. We give the proof for h_F, the proof for h_B is analogous. Let x be any state in G' and let W be any acyclic forward path in G' from x to *goal*. We claim that $h_F(x) \leq c(W)$. If W does not contain the new edge e, the claim trivially follows from the forward admissibility of h_F on G. Hence, assume W contains e, i.e. $W = YZ$ for some forward paths Y, from x to $end(U)$, and Z, from $end(V)$ to *goal*. Using $d_G(u, v)$ to denote the distance from u to v in G, we have

$$
\begin{aligned}
h_F(x) &\leq d_G(x, end(U)) + h_F(end(U)) \\
&\qquad \text{(because } h_F \text{ is forward consistent on } G\text{)} \\
&\leq c(Y) + h_F(end(U)) \\
&\qquad \text{(because } Y \text{ is a path from } x \text{ to } end(U)\text{)} \\
&\leq c(Y) + c(e) + c(V) \\
&\qquad \text{(by definition, } c(e) \geq h_F(end(U)) - c(V)\text{)} \\
&= c(Y) + c(e) + c(V^{-1}) \\
&\qquad \text{(because } c(V^{-1}) = c(V)\text{)} \\
&\leq c(Y) + c(e) + c(Z) \\
&\qquad \text{(by optimality of } V^{-1}\text{)} \\
&= c(W)\,.
\end{aligned}
$$

Hence h_F is forward admissible on G'. By an analogous proof, h_B is backward admissible on G' and thus $I' \in I_{AD}$.

Because B is DXBB it will behave the same on I' as it did on I. In particular it will neither forward expand $end(U)$ nor backward expand $end(V)$, will thus not discover the edge e, and will incorrectly return $B(I)$ as an optimal solution for I'. Hence, B is not an admissible search algorithm. ■

Once we have Theorem 3, we can define surely-expanded pairs in front-to-end bidirectional search as follows:

Definition 4 *For a pair of states (u, v) on instance $I \in I_{CON}$, if U is an optimal path from start to u and V is an optimal path from v to goal, and $lb(U, V) < C^*$, then we say (u, v) is a surely-expanded pair.*

Dechter & Pearl [20] proved (their Theorem 8) that every admissible unidirectional search algorithm must expand every state surely expanded by A* when the given heuristic is consistent. We show that for bidirectional search algorithms, there are no surely expanded states; instead, we have surely expanded pairs. For each of these pairs of states, the algorithm has to expand at least one of them.

Unidirectional heuristic search algorithms are a special case of front-to-end bidirectional heuristic search algorithms in which $h_B(u) = 0$ for all u and where the algorithm always chooses the forward direction to expand. Note that in this case, in $Open_B$, we will only have one path with cost= 0, h-cost= 0, the lb of all possible pairs are determined by f_F. Thus, Dechter & Pearl's results of surely expanded states can be viewed as a special case of Theorem 3:

Theorem 5 *Let a problem instance $I = (G, start, goal, h_F, 0) \in I_{CON}$ have an optimal solution cost of C^*. If U is an optimal forward path such that $U_0 = start$ and $f_F(U) < C^*$ then, in solving problem instance I, any admissible DXBB unidirectional search algorithm must forward expand $end(U)$.*

3.3.2 The Must-Expand Graph $G_{MX}(I)$: Reduction from Bidirectional Search to the Vertex Cover Problem

Definition 6 *The Must-Expand Graph $G_{MX}(I)$ of problem instance $I = (G, start, goal, h_F, h_B) \in I_{CON}$ is an undirected, unweighted bipartite graph defined as follows. For each state $u \in G$, there are two vertices in $G_{MX}(I)$, the left vertex u_F and right vertex u_B. For each pair of states $u, v \in G$, there is an edge in $G_{MX}(I)$ between u_F and v_B iff there exist an optimal forward path U with $U_0 = start$ and $end(U) = u$ and an optimal backward path V with $V_0 = goal$ and $end(V) = v$ such that $lb(U, V) < C^*$. Thus, there is an edge in $G_{MX}(I)$ between u_F and v_B if and only if Theorem 3 requires the state pair (u, v) to be expanded.*

We illustrate this in Figures 3.2 and 3.3. Figure 3.2 shows a problem instance $I = (G, start, goal, h_F, h_B) \in I_{CON}$. In this example a is the start state, f is the goal, and $C^* = 3$. Figure 3.3 shows $G_{MX}(I)$, where d refers to the cost of the shortest path to each state and f refers to the f-cost of that path.

By construction, the edges in $G_{MX}(I)$ exactly correspond to the state pairs that must be expanded according to Theorem 3, and therefore any vertex cover for $G_{MX}(I)$ will, by definition, represent a set of expansions that covers all the required state pairs. For example, one possible vertex cover includes exactly the vertices in the left side with at least one edge–$\{a_F, c_F, d_F, e_F\}$. This represents expanding all the required state pairs in the forward direction. This requires four expansions and is not optimal because the required state pairs can be covered with just three expansions: a and c in the forward direction and f in the backward direction. This corresponds to a minimum vertex cover of $G_{MX}(I) : \{a_F, c_F, f_B\}$.

Theorem 3 can be interpreted as: every algorithm must expand states which form a vertex cover on $G_{MX}(I)$. Then, the size of the minimum vertex cover is clearly a lower bound on the minimum number of node expansions:

Theorem 7 *Let $I \in I_{CON}$. Let A be an admissible DXBB bidirectional front-to-end search algorithm, and S_F (resp. S_B) be the set of states expanded by A on input I in the forward (resp. backward) direction. Together, S_F and S_B correspond to a*

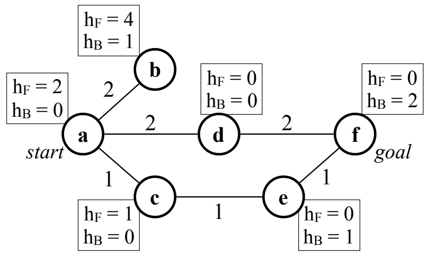


Figure 3.2: A sample bidirectional search problem instance.

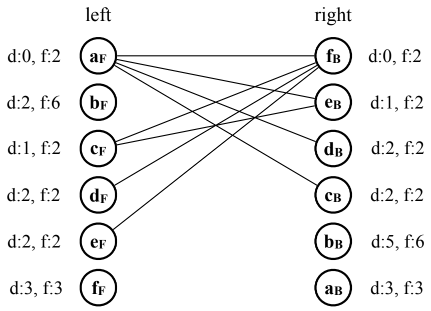


Figure 3.3: The Must-Expand Graph for Figure 3.2, where C^*=3.

vertex cover for $G_{MX}(I)$. In particular, $|S_F| + |S_B|$ is lower-bounded by the size of the smallest vertex cover for $G_{MX}(I)$.

Proof. Let (u_F, v_B) be an edge in $G_{MX}(I)$. Then Theorem 3 requires the state pair (u, v) to be expanded by A on input I, i.e., A must expand u in the forward direction or v in the backward direction. Thus the set of states expanded by A on input I corresponds to a vertex cover of $G_{MX}(I)$. ■

Shaham et al. proved that the bound is tight for off-line algorithms, i.e. there exists an off-line algorithm, which is guarantee to find the minimum vertex cover with proper parameters [73]. However, in general, those parameters can not be determined before solving the problem. Thus, such algorithms are not practical tools to do the search.

How about online algorithms? If the algorithm has a guarantee on the number of node expansions with respect to the size of the minimum vertex cover, what could be the best guarantee? The best guarantee is 2x the size of minimum vertex cover.

Theorem 8 *Let A be any admissible DXBB front-to-end bidirectional search algorithm. Then, there exists a problem instance $I \in I_{CON}$ and a DXBB front-to-end bidirectional search algorithm B such that A expands at least twice as many states in solving I as B expands in solving I.*

Proof. Consider the two problem instances I_1 and I_2 in Figure 3.4 and Figure 3.5 In these instances $h_F(n) = h_B(n) = 0$ for all states, S is the start and G is the goal. There is an edge between S and G with cost $2k + 2$, where k is a positive integer. The cost of the edge from S to A_1 and the cost of the edge from B_1 to G is $\frac{1}{2}$; all other edges have cost 1.

Assume A is given the first instance. Then it must expand some states in the forward direction and some states in the backward direction, to prove that $2k + 2$ is the optimal cost. Assume it expands up to A_m in the forward direction and B_n in the backward direction. For state A_{m+1}, its forward g-cost is $m + \frac{1}{2}$. Likewise, $g_B(B_{n+1}) = n + \frac{1}{2}$. Since the h-cost is 0 everywhere, $lb(A_{m+1}, B_{n+1}) = g_F(A_{m+1}) +$

$g_B(B_{n+1}) = m + n + 1$. Then $m + n + 1 \geq 2k + 2$, otherwise (A_{m+1}, B_{n+1}) is a sufficient pair: $lb(A_{m+1}, B_{n+1}) = max\{f_F(A_{m+1}), f_B(B_{n+1}), g_F(A_{m+1}) + g_B(B_{n+1})\} = g_F(A_{m+1}) + g_B(B_{n+1}) = m + n + 1 < 2k + 2$.

Without loss of generality, let $m \geq n$. There are two cases: (1) $m = n$ and (2) $m > n$.

We analyze each case individually. (1) In this case, A expands the same number of nodes in both directions on instance I_1. There must be an order of expanding states, either expanding A_m before B_n, or expanding B_n before A_m. Without loss of generality, we assume it expands A_m before B_n.

Then consider instance I_2. Since A is a DXBB algorithm, it initially cannot distinguish I_1 from I_2 and it must initially behave the same on both instances. Hence, on both instances, A will still expand m states in the forward direction. Note that the algorithm expands B_n after A_n, it will not be able to detect that B_n is the end-point, and thus it will expand n states in the backward direction. By comparison, let algorithm B expand all states in backward direction. B would only expands n states in the backward direction and then be able to terminate since it exhausted all the states in the backward direction. On I_2, algorithm A is doing $m+n = 2n$ expansions, which is two times worse than algorithm B.

(2) In this case, the argument is analogous. We can make an instance I_2 such that Algorithm A does $m + n$ expansions while Algorithm B only expands n states, causing A to be more than two times worse than algorithm B. ■

Once we view bidirectional search problems as vertex cover problems, we can do the search in a vertex-cover way. The algorithm introduced in the next section, `NBS`, and other algorithms such as `DVCBS` [74], are using a vertex cover solver at the top level, i.e., they are trying to minimize the total number of expansions by finding a minimal vertex cover.

3.4 The Near-Optimal Bidirectional Search (NBS) Algorithm

3.4.1 High Level Idea

In this section, we present a new admissible front-to-end bidirectional heuristic search algorithm, Near-Optimal Bidirectional Search (NBS), that is guaranteed to do no more than 2|VC| expansions.

In Section 3.3.2, we have already shown that $2|VC|$ is the best theoretical guar-

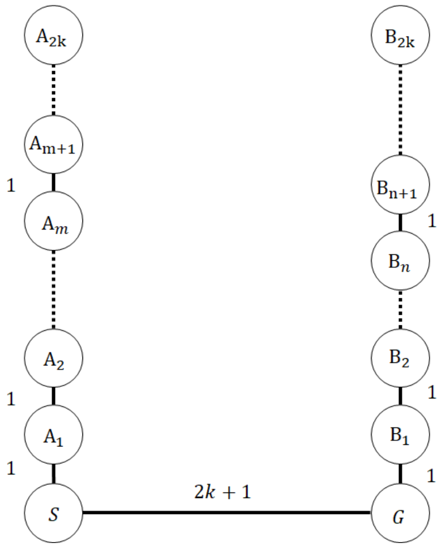


Figure 3.4: A problem instance with $C^* = 2k + 1$.

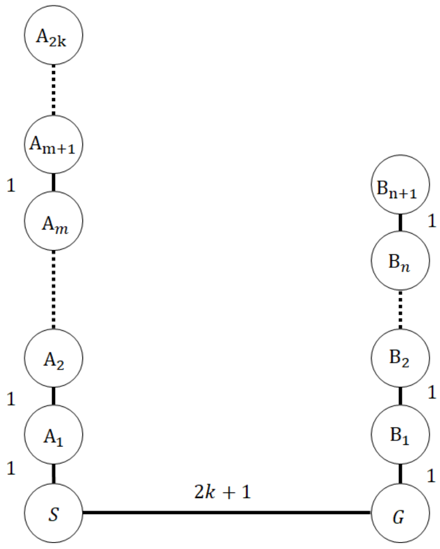


Figure 3.5: A problem instance with $C^* = 2k + 1$.

antee. That is, it is not possible to design an algorithm that is guaranteed to always do less than 2|VC| expansions. Therefore, NBS is near-optimal with respect to its worst-case guarantee on necessary node expansions.

The high level idea of NBS can be described with pseudo-code Algorithm 3. The main idea of NBS is to expand a pair with minimum *lb* until either a solution is found, or the state space is exhausted. The pseudo-code for a single expansion procedure is shown in Algorithm 4.

For a given surely-expanded pair of states, where expanding either one would be sufficient, NBS expands both. This approach is adapted from the greedy algorithm for finding a vertex cover [61] that selects an edge whose two endpoints are uncovered and then adds both vertices to the vertex cover.

This way of making node selection is very different from traditional Bi-HS algorithms. Previous algorithms usually pick a search direction and then choose the best state; NBS, however, is trying to do a vertex cover.

Note that the size of the vertex cover found by the greedy vertex cover algorithm is guaranteed to be no more than twice the minimum vertex cover; therefore, the size of the set of states expanded by NBS will not be more than 2|VC|, as long as each state is expanded in each direction at most once. However, if NBS has to re-expand states,

Algorithm 3 NBS outline

```
1: $C \leftarrow \infty$
2: $Open_F \leftarrow start$; $Open_B \leftarrow goal$
3: $Closed_F \leftarrow \emptyset$ ; $Closed_B \leftarrow \emptyset$
4: while $Open_F \neq \emptyset$ and $Open_B \neq \emptyset$ do
5:     $Pairs \leftarrow Open_F \times Open_B$
6:     $lbmin \leftarrow \min\{lb(X,Y) \mid (X,Y) \in Pairs\}$
7:     if $lbmin \geq C$ then  return $C$
8:     end if
9:     Choose $(U,V) \in pairs$ such that $lb(U,V) = lbmin$
10:    Forward-Expand($U$)
11:    Backward-Expand($V$)
12: end while
13: return $C$
```

Algorithm 4 `NBS`: Forward-Expand(U)

```
1: Move $U$ from $Open_F$ to $Closed_F$
2: for each $W \in expand_F(U)$ do
3:     if $\exists Y \in Open_B$ with $end(Y) = end(W)$ then
4:         $C = \min(C, c(W) + c(Y))$
5:     end if
6:     if $\exists X \in Open_F \cup Closed_F$ with $end(X) = end(W)$ then
7:         if $c(X) \leq c(W)$ then
8:             Continue for loop // discard $W$
9:         else
10:            remove $X$ from $Open_F / Closed_F$
11:        end if
12:        Add $W$ to $Open_F$
13:    end if
14: end for
```

then the total number of expansions will be more than 2|VC|. This corresponds to the tie-breaking strategy in Algorithm 3 Line 9. When choosing a pair of (U, V), we have to guarantee that they have not only the minimum *lb*, but also optimal g-cost.

Conceptually, one tie-breaking strategy that can be used is presented in Algorithm 5. The idea is to first find all pairs for which *lb* is smallest. Among these pairs, it then chooses the pairs (U, V) with smallest cost $c(U)$ and finally, among those, the ones with smallest cost $c(V)$. Any pair (U, V) from the remaining candidates is a valid choice. As long as the heuristics are consistent, such a tie-breaking strategy guarantees that when the pair is picked, both states will have optimal g-cost, i.e., no re-expansion is needed.

Algorithm 5 `NBS` outline sub-routine: choosing (U, V)

```
1: $minset \leftarrow \{(X, Y) \in Pairs \mid lb(X, Y) = lbmin\}$
2: $Uset \leftarrow \{X \mid \exists Y (X, Y) \in minset\}$
3: $Umin \leftarrow \min\{c(X) \mid X \in Uset\}$
4: Choose any $U \in Uset$ such that $c(U) = Umin$
5: $Vset \leftarrow \{Y \mid (U, Y) \in minset\}$
6: $Vmin \leftarrow \min\{c(Y) \mid Y \in Vset\}$
7: Choose any $V \in Vset$ such that $c(V) = Vmin$
```

3.4.2 Properties of `NBS`

In this section we first prove that, for all problem instances in I_{AD}, `NBS`

- terminates, and
- returns a solution with cost C^* if one exists. Otherwise it will return ∞.

We also prove that, for all problem instances in I_{CON}, `NBS` never expands the same state twice in the same search direction.

Theorem 9 *For any finite state space S with non-negative edge costs* `NBS` *halts for any start and goal states in S.*

Proof. Since there are no negative edge costs, there will not be any negative-cost cycles. Thus, `NBS` never expands a node via a path containing a cycle. i.e., `NBS` only expands a node via an acyclic path. In a finite state space there are a finite number of acyclic paths to each state. Therefore each state can only be expanded a finite number of times in each search direction while reducing the g-cost of the state. Thus, after a finite number of iterations all the reachable nodes will be exhausted and at least one of the open lists will be empty, causing `NBS` to halt. ■

Definition 10 *For any optimal path $P = s_0, s_1, \ldots s_n$ from start (s_0) to goal (s_n), let i be the largest index such that $s_k \in Closed_F$ $\forall k \in [0, i-1]$, and let j be the smallest index such that $s_k \in Closed_B$ $\forall k \in [j+1, n]$. We say that P "has not been found" if $i < j$ and that P "has been found" otherwise $(i \geq j)$.*

In the following, we show that given an admissible heuristic, if no optimal path has been found yet, then $lbmin < C^*$.

Definition 11 *Node n is called "optimally expanded in the forward search direction" if $n \in Closed_F$ and $g_F(n) = d(start, n)$. Likewise, n is called "optimally expanded in the backward search direction" if $n \in Closed_B$ and $g_B(n) = d(n, goal)$.*

Lemma 12 *Let $P = s_0, s_1, \dots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not optimally expanded in the forward direction and s_{n-1} is optimally expanded in the forward direction, then $s_n \in Open_F$ and $g_F(s_n) = d(start, s_n)$. Analogously, let $P = s_0, s_1, \dots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not optimally expanded in the backward direction and s_1 is optimally expanded in the backward direction, then $s_0 \in Open_B$ and $g_B(s_0) = d(s_0, goal)$.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. If $n = 0$, $s_0 = start$ has not been closed in the forward direction and the lemma is true because line 2 of Algorithm 3 puts $start \in Open_F$ with $g_F(start) = d(start, start) = 0$. Suppose $n > 0$. When s_{n-1} was expanded to become optimally expanded in the forward direction, s_n was generated via an optimal path (in Algorithm 4 lines 6 and 12, $g_F(n) + cost(n, c) = d(start, s_{n-1}) + cost(s_{n-1}, s_n) = d(start, s_n)$).

There could be three possibilities for s_n's status when s_{n-1} is expanded to become optimally expanded: (1) s_n is not generated in the forward direction yet; (2) s_n is on $Closed_F$; (3) s_n is on $Open_F$. In cases (1) and (2), s_n will be put onto $Open_F$ with cost $g_F(s_n) = g_F(s_{n-1}) + cost(s_{n-1}, s_n)$, which is optimal; in case (3), the cost of s_n gets updated to the optimal cost $g_F(s_n) = g_F(s_{n-1}) + cost(s_{n-1}, s_n)$. ■

Lemma 13 *Let $P = s_0, s_1, \dots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not optimally expanded in the forward direction then there exists an i ($0 \leq i \leq n$) such that $s_i \in Open_F$ and $g_F(s_i) = d(start, s_i)$. Let i_{min} be the smallest such i and define n_F (for path P) to be $s_{i_{min}}$. Analogously, let $P = s_0, s_1, \dots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not optimally expanded in the backward direction then there exists a j ($0 \leq j \leq n$) such that $s_j \in Open_B$ and $g_B(s_j) = d(s_j, goal)$. Let j_{max} be the largest such j and define n_B (for path P) to be $s_{j_{max}}$.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. If $start \notin Closed_F$ then $i = 0$ has the required properties ($start \in Open_F$ and $g_F(start) = d(start, start) = 0$, because of line 2 of Algorithm 3.

Suppose $start \in Closed_F$. Let k ($0 \leq k < n$) be the largest index such that s_k is optimally expanded. Such a k must exist because $start$ ($k = 0$) is optimally expanded. By Lemma 12 $s_{k+1} \in Open_F$ and $g(s_{k+1}) = d(start, s_{k+1})$. Therefore $i = k + 1$ has the required properties. ■

Lemma 14 *When we run* `NBS` *on an instance $I \in I_{AD}$, if $C > C^*$, then there exist a pair (U, V) on open such that $lb(U, V) \leq C^*$. (C is the value stored in Algorithm 3 as the best solution cost found so far)*

Proof. Let $P = s_0, s_1, \ldots s_n$ be an optimal path from $start$ (s_0) to goal state s_n. Since $C > C^*$, we know that P has not been found. Then (1) s_n is not optimally expanded in the forward direction, thus there exists an i ($0 \leq i \leq n$) such that $s_i \in Open_F$ and $g_F(s_i) = d(start, s_i)$. Let i_{min} be the smallest such i and define n_F (for path P) to be $s_{i_{min}}$. (2) s_0 is not optimally expanded in the backward direction thus there exists a j ($0 \leq j \leq n$) such that $s_j \in Open_B$ and $g_B(s_j) = d(s_j, goal)$. Let j_{max} be the largest such j and define n_B (for path P) to be $s_{j_{max}}$. (3) $i_{min} < j_{max}$, otherwise P has been found.

Based on those facts, we can let $U = s_0, s_1, ...s_{i_{min}}$, $V = s_{j_{max}}, ..., s_{n-1}, s_n$.

Since the forward and backward heuristics are admissible, $f_F(U) \leq C^*$, $f_B(V) \leq C^*$. Combining the fact that $g_F(U) + g_B(V) \leq C(P) = C^*$, $lb(U, V) \leq C^*$. ■

Theorem 15 *For any $(G, h_F, h_B) \in I_{AD}$,* `NBS` *returns C^*.*

Proof. Proof by contradiction. Suppose the solution returned C is greater than C^*. According to Algorithm 3 Line 7, $lbmin \geq C > C^*$. However, according to Lemma 14, there exists a pair (U, V) such that $lb(U, V) \leq C^*$. Thus, $lbmin \leq lb(U, V) \leq C^*$. This contradicts our previous observation that $lbmin > C^*$. ■

Now we have proven that for I_{AD}, `NBS` will terminate with the optimal cost. Therefore, `NBS` is optimal.

In the following, we show that when the heuristic is consistent, whenever `NBS` expands a state, the state will be expanded with optimal cost. Therefore, `NBS` only needs to expand a state once in a particular direction.

Lemma 16 *Let $I = (G, start, goal, h_F, h_B) \in I_{CON}$, U any forward path, and W any forward path such that $f_F(U) \geq f_F(W)$ and $c(U) \geq c(W)$. Then $lb(U, V) \geq lb(W, V)$ for all backward paths V. Similarly, if V is any backward path, and W any backward path such that $f_B(V) \geq f_B(W)$ and $c(V) \geq c(W)$, then $lb(U, V) \geq lb(U, W)$ for all forward paths U.*

Proof.

This is the proof for forward direction. The proof for backwards direction is analogous.

$lb(U, V) = \max\{f_F(U), f_B(V), c(U) + c(V)\}$

$\geq \max\{f_F(W), f_B(V), c(W) + c(V)\} = lb(W, V)$. ■

Corollary 17 *Let $I = (G, start, goal, h_F, h_B) \in I_{CON}$, U any forward path, and V is any backward path. Then $lb(U, V) \geq lb(W, V)$ for any prefix W of U and $lb(U, V) \geq lb(U, Y)$ for any prefix Y of V.*

Proof. This is the proof for forward direction. The proof for backwards direction is analogous.

This follows from Lemma 16, noting that $f_F(U) \geq f_F(W)$ (because h_F is forward consistent) and $c(U) \geq c(W)$ (because edge costs are non-negative). ■

Lemma 18 *Let $I = (G, start, goal, h_F, h_B) \in I_{CON}$, If pair (U, V) is chosen for expansion by* `NBS`*, then $c(U) = d(start, end(U))$, $c(V) = d(end(V), goal)$,*

Proof. Proof by contradiction. If $c(U) > d(start, end(U))$, according to Lemma 14, there exists a prefix of U, W, on $Open_F$. According to Corollary 17, $lb(U, V) \geq lb(W, V)$. Therefore, according to the tie-breaking strategy in Algorithm 3 Line 9, pair (W, V) instead of (U, V) would be chosen, which contradicts our assumption. ■

Theorem 19 `NBS` *never expands the same state twice in the same search direction.*

Proof. According to Lemma 18, if a path is closed, it must be closed with optimal cost. Therefore, once a path is closed, no shorter path will be found to the end point of that path. As a consequence, no reopenings or re-expansions will occur. ■

It is possible for `NBS` to expand the same state twice by expanding it once in each direction. However, because state spaces are finite and no state is expanded more than twice by `NBS`, we get the following result. i.e. `NBS` is complete.

Theorem 20 *Let $I \in I_{CON}$, let $G_{MX}(I)$ be the Must-Expand Graph of I, and let $VC(I)$ be the size of the smallest vertex cover of $G_{MX}(I)$. Then* `NBS` *does no more than $2\,VC(I)$ state expansions on $G_{MX}(I)$ to cover its surely expanded pairs.*

Proof. If (u, v) is a surely expanded pair then $lb(U, V) < C^*$ for every optimal forward path U from *start* to u and every optimal backward path from *goal* to v. `NBS` will select exactly one such (U, V) pair for expansion and expand both $end(U) = u$ and $end(V) = v$.

A minimum vertex cover for I might require only one of them to be expanded, so for each expansion required by a minimum vertex cover, `NBS` might do two. ■

3.4.3 Efficient Selection of State Pairs for Expansion

Algorithm 5 shows the conceptual principle for choosing pairs. A naive implementation is to look at all pairs from $Open_F \times Open_B$, and select the best pair. Such approach will require $O(n^2)$ time to find a pair, where n is the size of one *Open* list.

In this section, we will show an efficient data structure that returns such pairs in amortized $O(\log n)$ time.

Algorithm 6 NBS pseudo-code for selecting the best pair from *Open* list. C_{lb} is set to 0 when the search begins.

```
1: procedure PREPAREBEST
2:     while min $f$ in $waiting_D < C_{lb}$ do
3:         move best node from $waiting_D$ to $ready_D$
4:     end while
5:     while true do
6:         if $ready_D \cup waiting_D = \emptyset$ then return false
7:         end if
8:         if $ready_F.c + ready_B.c \leq C_{lb}$ then return true
9:         end if
10:        if $waiting_D.f \leq C_{lb}$ then
11:            move best node from $waiting_D$ to $ready_D$
12:        else
13:            $C_{lb} = \min(waiting_F.f,\ waiting_B.f,\ ready_F.c{+}ready_B.c)$
14:        end if
15:    end while
16: end procedure
```

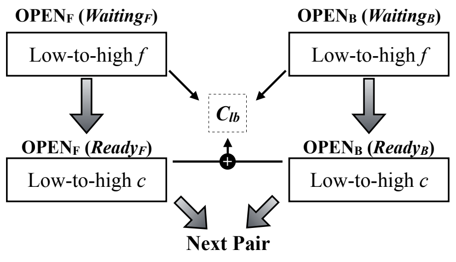


Figure 3.6: The open list data structure used in NBS.

Pseudo-code for the pair selection procedure is shown in Algorithm 6. The idea is to introduce two priority queues, *waiting* and *ready* for each direction.

Here is the intuition why it will work: When a node is generated, it is put into *waiting*; when moving states from *waiting* to *ready*, we have already filtered out those nodes with high f-cost. Thus, when we pick a pair from $ready_F \times ready_B$ whose sum of g-cost is minimum, that pair must have minimum lb, because lb comes from $max\{f_F, f_B, g_F + g_B\}$.

The details of how the procedure works are explained as follows. First, paths with f-cost lower than C_{lb} must immediately be moved to *ready* (line 2). If $ready_D$ and $waiting_D$ are jointly empty in either direction, the procedure is halted and the search will terminate (line 6). If the best state pairs in *ready* have $c(U) + c(V) \leq C_{lb}$, the procedure completes; these paths will be expanded next (line 8).

If the $ready_D$ queue is empty in either direction, any paths with $f = C_{lb}$ can be moved to *ready* (line 10). While we could, in theory, move all such paths to *ready* in one step, doing so incrementally allows us to break ties on equal f towards higher c first, which slightly improves performance [78] [3]. If there are no paths with $f \leq C_{lb}$ in *waiting* and in *ready* with $c(U) + c(V) \leq C_{lb}$, then the C_{lb} estimate is too low, and C_{lb} must be increased (line 13).

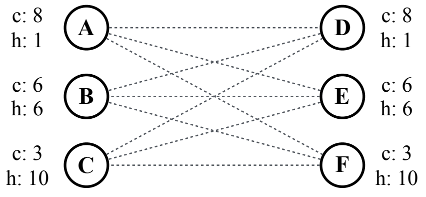
We illustrate this on an artificial example from Figure 3.7.[4] To begin, $C_{lb} \leftarrow 0$ and we assume that (A, B, C) are on $waiting_F$ and (D, E, F) are on $waiting_B$. First, C_{lb} is set to 9 (line 13). Then, A and D can be added to *ready* because they have lowest f-cost, and $C_{lb} = 9$. However, $c(A) + c(D) = 16 > C_{lb} = 9$, so we cannot expand A and D. Instead, we increase C_{lb} to 12 and then add B and E to ready. Now, because $c(B) + c(E) = 12 \leq C_{lb}$ we can expand B and E.

We can prove that the amortized run time over a sequence of Insertion and PrepareBest operations of our data structure is $O(\log(N))$, where N is the total number of such operations.

Theorem 21 *Let N be the total number of calls of Insertion and PrepareBest made by* `NBS` *during search. Then the total time is $N \log N$.*

[3]In that paper, they renamed the `NBS` algorithm in this document as $\texttt{NBS}_F$. The modified version, which eagerly moves all paths with $f = C_{lb}$ from *waiting* to *ready*, is called $\texttt{NBS}_A$. Table 1 of their paper indicates that $\texttt{NBS}_A$ outperforms $\texttt{NBS}_F$ in terms of node expansions on many domains.

[4]This example also illustrates why we cannot just sort by minimum f_F or c when performing expansions.



Lower Bound	
AD	16
AE	14
AF	13
BD	14
BE	12
BF	13
CD	13
CE	13
CF	13

Figure 3.7: A sample state space to illustrate the priorities of state pairs calculated when running `NBS`.

Proof. (Amortized analysis. Proof by potential method.)

Let the number of states at a particular time t on $Open_F \cup Open_B$ be n, and the number of states on $ready_F \cup ready_B$ be m.

Let the potential function $P = 2(n - m)\log N$. Initially, $n = m = 0$, we have $P = 0$; since $m \leq n$ at all times, $P \geq 0$ during the course of computation. Now we can start our analysis.

The amortized cost is defined as $\hat{c} = c + \Delta P$, where c is the actual cost of an operation and ΔP is the change in the potential function introduced by the operation.

(1) For a single Insertion operation, we will only add the state to *waiting*. $c = \log(n - m)$. $\Delta P = 2((n + 1) - m)\log N - 2(n - m)\log N = 2\log N$. Thus, in this case, $\hat{c} = c + \Delta P = \log(n - m) + 2\log N \leq 3\log N$.

Therefore, for Insertion operation, $\hat{c} = O(\log N)$.

(2) For a PrepareBest operation, we might need to extract multiple states from *waiting* and insert them to *ready*, and finally pop the front from *ready*. Let the total number of states moved from *waiting* to *ready* be k.

Then $c \leq k\log(n - m) + k\log(m + k) + \log(m + k) \leq (2k + 1)\log N$. $\Delta P = 2((n - 1) - (m - k + 1))\log N - 2(n - m)\log N = -2k\log N$.

Thus, in this case, $\hat{c} = c + \Delta P \leq (2k + 1)\log N - 2k\log N = \log N$.

Therefore, for PrepareBest operation, $\hat{c} = O(\log N)$. ■

Theorem 21 guarantees that even though the data structure looks complicated and a single PrepareBest operation can be expensive, the amortized time of PrepareBest operation over a sequence of operations is still inexpensive – the same as the data structures typically used in A*. Note that there are decrease key operations which will occur when we find a shorter path to a certain states on *Open*. The overhead of that operation is similar to data structures in A*, thus we do not explicit discuss it here.

Table 3.1: Average state expansions for unidirectional (A*) and bidirectional search across domains. Problems marked as unsolvable are due to memory limit.

Domain	A*	BS*	MMe	NBS	MM_0
DAO	**9,646**	11,501	13,013	12,085	17,634
Mazes	64,002	42,164	51,074	**34,474**	51,075
4 Peg TOH (12+2)	1,437,644	**1,106,189**	1,741,480	1,420,554	12,644,722
4 Peg TOH (10+4)	19,340,099	8,679,443	11,499,867	**6,283,143**	12,644,722
16 Pancake(GAP)	**125**	339	283	335	*unsolvable*
16 Pancake($GAP\backslash 2$)	1,254,082	947,545	**578,283**	625,900	*unsolvable*
16 Pancake($GAP\backslash 3$)	*unsolvable*	29,040,138	7,100,998	**6,682,497**	*unsolvable*
15 puzzle	15,549,689	**12,001,024**	13,162,312	12,851,889	*unsolvable*

3.5 Results and Discussion

In Section 3.4, we have proven the worst case performance of NBS is no more than 2x the theoretical minimum necessary node expansions. In other words, in theory, there is no algorithm that will terminate with less than half the necessary expansions of NBS. But the theory does not tell us about what happens in practice. Does NBS tend to be close to minimum node expansions, or 2x the minimum? Also, since NBS introduces more complicated data structures, how is it performing time-wise?

Experiments are designed to answer these questions.

In Table 3.1 we present results on problems from four different domains, including grid-based pathfinding problems [81] ('brc' maps from Dragon Age: Origins (DAO)), random 4-peg Tower of Hanoi (TOH) problems, random pancake puzzles, and the standard 15 puzzle instances [46]. In this table, MMe [75] is a variant of MM and MM_0 is MM with the zero-heuristic.

The canonical goal state is used for all puzzle problems. Such a choice makes it more efficient to search backwards to solve the 15-puzzle and TOH because of the lower branching factor. For example, for a random 15-puzzle state whose empty cell is in the center, the branching factor is 4, while branching factor of the goal state is only 2. In our experiments, when we collect the results of unidirectional search algorithms, we search forward to the standard goal states.

On each of these domains we use standard heuristics of different strength. More specifically, we use octile distance for grid maps, the additive pattern databases heuristics [28] for TOH, GAP [37] for the 16 pancake and Manhattan Distance for the 15-puzzle.

In grid maps we varied the difficulty of the problem by changing the map type-/topology. In TOH and the pancake puzzle we varied the strength of the heuristic. The $GAP\backslash K$ heuristic is the same as the GAP heuristic, except that gaps involving the first k pancakes are not added to the heuristic. The general trend is that with very strong heuristics, A* has the best performance. As heuristics get weaker, or the problems get harder, the bidirectional approaches improve relative to A*. `NBS` is never far from the best algorithm, and on some problems, such as TOH, it has significantly better performance than all other approaches. Runtime and node expansions/second are found in Table 3.2. `NBS` is 30% slower than A* on the DAO problems, but competitive on other problems. `NBS` is slower than BS*, but this is often compensated for by performing fewer node expansions.

In Table 3.3 we look at the percentage of total nodes on closed compared to total expansions with $f = C^*$ by each algorithm in each domain. For the majority of domain and heuristic combinations there are very few expansions with $f = C^*$. The exception is the pancake puzzle with the GAP heuristic. On random instances this heuristic is often perfect, so all states expanded have $f = C^*$. This is why `NBS` does more than twice the number of expansions as A* on these problems—these expansions are not accounted for in our theoretical analysis. BS* puts nodes on closed that it does not expand, which is why it has a negative percentage.

Table 3.4 is the results from Sturtevant et. al. [85]. These results clearly show where `NBS` stands.

Column B presents the heuristics used in the experiments. As for the pancake puzzle problems, the $GAP\backslash K$ heuristic is computed by ignoring the top K pancake and calculating GAP heuristic on the remaining pancakes. $GAP - K$ heuristic is computed by subtracting K from the GAP heuristic of the given state. $GAP \times K$ is computed by multiplying a factor K ($K < 1$) to the GAP heuristic of the given state. Those heuristics are weaker than GAP heuristic.

Column C is the percentage of the problems whose vertex cover are bidirectional. Column D and E are the average size of the forward and backward minimum vertex cover, respectively. A forward minimum vertex cover are all the states n with $f_F(n) < C^*$, and a backward minimum vertex cover is defined analogously. Column F is the average size of the unidirectional vertex cover, i.e., the smaller one between forward vertex cover and backward vertex cover. Column G is the average size of the vertex cover.

The size of minimum vertex cover is in column MVC, and minimum possible node expansions by a unidirectional search (the smaller one of forward search and backward search) is in column UVC.

The general trend is that when the heuristic is weak, then the minimum vertex cover is likely to be bidirectional and `NBS` tends to perform well. The ratio of total expansion by `NBS` over the MVC is closer to 1 in that case. When the heuristic is strong and the minimum vertex cover is likely to be unidirectional, then the ratio of total expansion by `NBS` over the MVC is closer to 2. That ratio exceeds 2 only for easy problems where a lot of pairs with $lb = C^*$ are expanded.

3.6 Conclusions and Later Work

This thesis (1) provides sufficient conditions for expansions in bidirectional search. It turns out that no single state is surely expanded when an algorithm is guaranteed to find optimal solution; however, in terms of pairs of states, if a pair meets the sufficient condition for expansion, at least one of the states must be expanded.

(2) Based on the sufficient condition, this thesis shows that we can build G_{MX}, which can link the bidirectional search problem to the vertex cover problem.

(3) Finally, it presents the first front-to-end heuristic search algorithm that is near-optimal in necessary node expansions. It thus addresses questions dating back to Pohl's work on the applicability of bidirectional heuristic search [65]. When the heuristic is not strong, `NBS` provides a compelling alternative to A*.

Once we have `NBS`, it seems to be more reasonable to use $\texttt{NBS}_0$ instead of $\texttt{MM}_0$ to conduct Bi-BS. When the forward state space and the backward state space is symmetric, i.e., the branching factor is close, then the performance of $\texttt{NBS}_0$ is identical to $\texttt{MM}_0$. However, if the state space is unbalanced, i.e., the branching factor of one direction is much larger than the other direction, then doing $\texttt{MM}_0$ could be arbitrarily worse, while $\texttt{NBS}_0$ still has the 2x guarantee. In fact, in road routing systems, bidirectional Dijkstra's algorithm with an alternative back and forth policy is widely used [60, 67, 88], which is functionally identical to $\texttt{NBS}_0$, since $\texttt{NBS}_0$ will expand the state with lowest g-cost in each direction at each step.

There has been some follow-up research focusing on how to further improve bidirectional search based on the G_{MX} theory.

Shaham et al.[73] [5] suggests that the minimum vertex cover of G_{MX} is restrained. That is, there exist a real number m, such that the union of all the forward states satisfying $\{u|f_F(u) < C^*, g_F(u) < m\}$ and all the backwards states satisfying $\{v|f_B(v) < C^*, g_B(v) < C^* - m\}$ form a minimum vertex cover.

In that paper, a generalization of the `MM` algorithm is also proposed by introducing one more free parameter p, which is called `fMM`.

The behavior of `fMM` is identical to `MM`, except for its priority computation. It assigns each state u on $Open_F$ with a value $Pr_F(u) = max\{f_F(u), g_F(u)/p\}$ and assigns each state v on $Open_B$ with a value $Pr_B(v) = max\{f_B(v), g_B(v)/(1-p)\}$, where $0 \leq p \leq 1$. Basically, p is a parameter that determines the meeting point.

[5] I am one of the co-authors of this work.

When $p = \frac{1}{2}$, `fMM` becomes `MM`.

An observation is that `fMM` with the proper parameter will be able to do the exact minimum vertex cover. `fMM` with a fixed p is not guaranteed to do minimum node expansions all the time. In fact, p is a per instance parameter and cannot be computed in advance. Therefore, in general, `fMM` is not guaranteed to provide a worse-case guarantee in node expansions, let alone to find solution with minimum node expansions.

General breadth-first heuristic search (GBFHS, [6]) is another algorithm that has the control on meeting point. GBFHS expands states by g-cost layers and uses a parameter called split function to determine the depth of search on each direction. If the parameters are chosen carefully, the performance of GBFHS and `fMM` can be identical [77].

Shperberg et. al. also proposed the algorithm iterative-deepening bidirectional heuristic search, or IDBiHS [76]. IDBiHS is to GBFHS what IDA* is to A*. IDBiHS uses linear memory (with respect to the solution depth), which is memory efficient and is very useful in solving large scale problems.

There are other studies focused on vertex cover based search.

A representative algorithm is `DVCBS` [78], which selects a node for expansion in a greedy way to find a dynamic vertex cover on G_{MX}. Although it does not have the worst-case guarantee, it is empirically better than `NBS` in many domains.

An open question is whether `NBS` can be implemented efficiently in front-to-front bidirectional heuristic search.

The idea of `NBS` is to pick a pair of states with minimum lb and expand both states. What makes it practical and successful is the data structure that allows the efficient pair selection. If there exists a data structure that can return the pair with minimum lb under the front-to-front heuristics, then there will be no obstacle in building the front-to-front `NBS`. However, personally, I think the task is challenging, especially when the heuristic calculation is purely front-to-front, since in general it cannot be avoided to look up every pair of states from the frontiers.

It is possible, however, when the front-to-front heuristic is generated from a consistent front-to-end heuristic, there could exist such data structures. For example, we can create a front-to-front heuristic $h(u, v) = max\{0, |h_F(u) - h_F(v)|, |h_B(u) - h_B(v)|\}$. Consistency will limit the change around the frontiers, thus it is possible we do not have to compute the f-cost of all pairs in a brute-force way to find the best pair.

Another open question is whether the theory and algorithms can be further improved when we know the heuristic is consistent. The sufficient condition assumes that the heuristic is admissible but not consistent. However, if the heuristic is consistent and the DXBB solver knows that, then it could exploit that information and further reduce the expansions.

Table 3.2: Average running time and expansions per second for unidirectional (A*) and bidirectional search across domains.

Average Running Time (in seconds)					
Domain	h	A*	BS*	`MMe`	`NBS`
DAO	Octile	0.005	0.006	0.015	0.007
Mazes	Octile	0.035	0.022	0.060	0.019
TOH4	12+2	3.23	2.44	4.17	3.54
TOH4	10+4	52.08	23.06	30.64	16.60
Pancake	GAP	0.00	0.00	0.00	0.00
Pancake	$GAP\backslash 2$	14.16	4.91	5.25	5.23
Pancake	$GAP\backslash 3$	N/A	212.33	72.13	77.17
15 puzzle	MD	47.68	29.59	41.38	37.67
Expansion Rate ($\times 10^3$ nodes per seconds)					
DAO	Octile	1,896	1,912	851	1,662
Mazes	Octile	2,225	2,366	848	2,290
TOH4	12+2	444	453	418	401
TOH4	10+4	371	376	375	379
Pancake	GAP	156	564	564	153
Pancake	$GAP\backslash 2$	89	193	109	120
Pancake	$GAP\backslash 3$	N/A	137	98	87
15 puzzle	MD	326	406	318	338

Table 3.3: Percent of expansions with (f-cost $= C^*$) for each algorithm/domains.

Domain	Heuristic	A*	BS*	`MMe`	`NBS`
DAO	Octile	1.3%	0.6%	0.7%	1.2%
Mazes	Octile	0.0%	0.0%	0.0%	0.0%
TOH4	12+2 PDB	0.0%	0.0%	0.0%	0.0%
TOH4	10+4 PDB	0.0%	0.0%	0.0%	0.0%
Pancake	GAP	60.7%	81.6%	76.2%	81.0%
Pancake	GAP-2	0.2%	1.6%	6.0%	0.0%
Pancake	GAP-3	N/A	-0.6%	5.7%	0.0%
15 puzzle	MD	5.5%	0.5%	0.6%	0.3%

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**
Domain	Heuristic	BMVC	FVC	BVC	Min UVC	MVC	A*	Rev-A*	NBS$_\epsilon$	DVCBS$_\epsilon$
12-Pancake	GAP	0%	10	10	**8**	8	30	**28**	101	54
	GAP\1	64%	1,102	1,233	954	**832**	1,124	1,267	864	**841**
	GAP\2	96%	30,794	39,135	28,371	**12,466**	32,134	40,876	9,528	**8,011**
	GAP\3	100%	367,374	454,242	322,294	**48,994**	379,684	477,463	32,802	**25,080**
	GAP−1	74%	228	236	205	**187**	229	237	241	**174**
	GAP−2	100%	3,062	3,159	2,895	**1,643**	3,085	3,183	1,353	**1,269**
	GAP−3	100%	28,853	29,366	27,844	**8,406**	29,209	29,717	6,441	**5,290**
	GAP×0.9	56%	248	255	225	**216**	249	256	229	**204**
	GAP×0.8	48%	2,149	2,212	2,056	**1,904**	2,150	2,213	1,755	**1,712**
	GAP×0.7	88%	16,736	16,985	16,347	**11,744**	16,737	16,986	9,708	**9,330**
TOH4(12)	10+2	0%	64,264	68,149	**59,153**	59,153	**64,334**	68,173	100,080	69,010
	8+4	34%	456,156	456,439	420,749	**382,390**	457,401	459,373	**411,085**	434,347
	6+6	86%	772,889	796,639	697,965	**463,586**	789,603	806,767	**446,603**	525,811
	4+8	66%	530,936	547,752	480,027	**406,480**	548,850	568,615	**411,212**	427,702
	2+10	2%	162,656	174,659	152,421	**152,384**	**172,088**	190,364	199,880	192,271
	Zero	100%	8,262,691	8,560,419	7,826,880	**476,455**	8,262,691	8,560,419	450,539	**425,578**
Grids	Octile	1%	9,525	9,222	7,785	**7,754**	9,646	**9,339**	12,136	9,815
DAO	Zero	100%	19,448	19,935	17,760	**15,167**	19,466	19,955	16,819	**15,946**
Grids	Octile	90%	63,657	63,694	49,257	**31,193**	63,660	63,698	**36,029**	49,298
Mazes[1]	Zero	100%	71,455	71,571	57,961	**31,263**	71,474	71,590	36,034	**31,840**
Grids	Octile	69%	97,215	97,072	77,765	**62,449**	97,228	97,084	**70,186**	90,301
Mazes[4]	Zero	100%	114,084	114,111	94,678	**63,540**	114,105	114,133	70,259	**65,306**
Grids	Octile	29%	125,496	126,236	108,282	**103,841**	125,538	126,278	128,572	**124,038**
Mazes[16]	Zero	100%	144,773	145,445	126,898	**109,795**	144,801	145,473	129,310	**118,757**
Road Maps	ED	16%	72,119	69,461	47,702	**47,666**	72,119	69,461	**69,110**	69,461
Distance	Zero	100%	226,041	229,400	184,419	**129,484**	226,041	229,400	**143,428**	160,940
Road Maps	ED / speed	95%	133,147	128,165	93,645	**80,456**	133,148	128,166	**97,889**	128,165
Time	Zero	100%	230,075	227,785	181,105	**104,952**	230,076	227,786	119,873	**111,329**

Table 3.4: This is the result from Sturtevant's work [85] evaluating the critical state measurements and algorithm performance on a variety of domains. Measures C-G are theoretical measures reporting necessary node expansions, while algorithmic results (H-K) report all node expansions.

Chapter 4

Unidirectional Suboptimal Search

4.1 Background and Introduction

Optimal heuristic search algorithms, whether unidirectional or bidirectional, are able to find the shortest paths between a given start and goal state. A* and `NBS` from the previous chapter are examples of this kind of algorithm.

However, there are scenarios where a suboptimal solution found quickly is preferred over a optimal solution found slowly. In particular, time-sensitive applications such as embedded systems [8] or video games [9] favor such solutions. This motivates the study of suboptimal search algorithms.

Suboptimal search algorithms can be classified into three categories. The first category is satisficing search, which aims at returning a solution that is "good enough" by some metric. In practice, such search algorithms usually just return the first solution found, regardless of the solution quality, such as Greedy Best First Search (GBFS) [69]. The second category is bounded cost search (BCS) [80], which aims at finding a solution whose cost is less or equal to C, where C is a given desired bound. The third one, which we will focus on in this chapter, is bounded suboptimal search (BSS), which has a guarantee on the cost of the returned solution.

4.1.1 Assumptions in Bounded Suboptimal Search

Most assumptions we make here are analogous to those in the bidirectional optimal search chapter.

A bounded suboptimal search (BSS) problem is defined by a 5-tuple $\{G, start, goal, h, B\}$. The state space G is a finite directed graph whose vertices are states and whose edges are pairs of states.

Each edge $(u, v) \in G$ has a cost $c(u, v)$. We assume there is at most one edge between each pair of states, and the edge costs are non-negative. A path in G is a finite sequence $U = (u_0, \ldots, u_n)$ of states in G where (u_i, u_{i+1}) is an edge in G for $0 \leq i < n$.

Let $d(u, v)$ be the cost of the cheapest path from state u to state v in G. If there is no path from u to v then $d(u, v) = \infty$. We let $g^*(u) = d(start, u)$.

The shortest path from start to goal is the optimal solution cost $C^* = d(start, goal)$.

In suboptimal search, the currently found path to a state is not necessarily the shortest path. Let the g-cost of a state be the cost of the current best path from start to that state. Thus, $g(n) \geq d(start, n)$

Heuristic function h maps a state to a real number. A heuristic is admissible if $\forall n, h(n) \leq d(n, goal)$.

Definition 22 *A heuristic is consistent on an undirected graph if* $\forall m, n, |h(n) - h(m)| \leq d(n, m)$*. If this property holds on a directed graph, we say the heuristic is* strongly consistent*. A heuristic is* weakly consistent *if* $\forall m, n, h(n) \leq d(n, m) + h(m)$*.*

A weakly consistent heuristic only requires that the h-cost does not decrease more than the edge cost, but can go up along an edge as much as it wants. A strongly consistent heuristic limits the heuristic change in both directions, whether increasing or decreasing, to not exceed the edge cost.

The BSS problem is to find a solution path with cost $\leq B(C^*)$, where $B : \mathbb{R} \to \mathbb{R}$ is a given bounding function that satisfies $\forall x \geq 0, B(x) \geq x$.

We assume that search algorithms only have black box access to the state space (the same as that in the previous chapter). That is, they are only allowed to explore the state space by node expansions, and they can only prioritize states based on the g-cost and h-cost of a state.

There are constraints on solution quality for BSS problem solvers. Basically, we need the algorithm to be B-admissible. The formal definition of B-admissible algorithm is as following:

Definition 23 *If an algorithm A is guaranteed to return a solution with cost $\leq B(C^*)$ for a set of instances I, where B is a given bounding function, then A is* B-admissible *on I.*

The most popular bounding function is $B_w(x) = w \cdot x$, which means when the optimal solution is C^*, the returned solution cost is no more than wC^*. Representative algorithms for solving this problem include weighted A* (WA*) [64], A$^*_\epsilon$ [63], Optimistic Search [86], Explicit Estimation Search (EES) [87], and Dynamic Potential Search (DPS) [33]. There are other types of bounding functions, such as the additive bounding function $B_\gamma(x) = x + \gamma$, where γ is a given constant [89]. BFS^{F_γ} [89] is a representative solver for this additive bounding function.

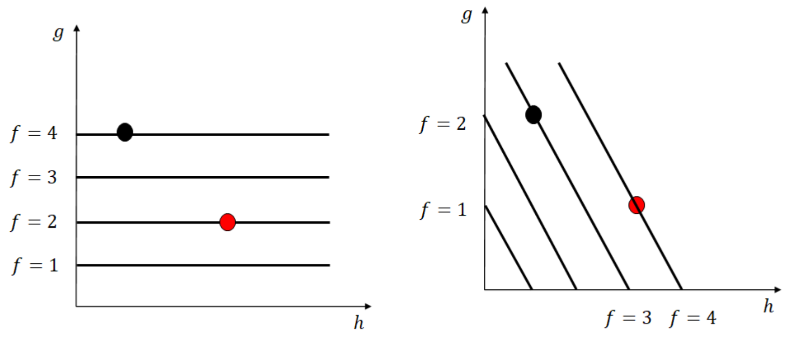
In the following, we give a brief introduction to a few representative BSS algorithms. Before that, we introduce generic best-first search, since this is the foundation framework that is used by many BSS algorithms.

Algorithm 7 is the pseudo-code for best-first search. When we change the way we calculate f-cost, the priorities of states change, thus the order we expand changes and we get a different algorithm. For example, when we use $f = g$, we get Dijkstra's algorithm; when we use $f = h + g$, we get A*.

Details of best-first search, including how we should choose a proper f-cost function in suboptimal search will be addressed in Section 4.3.

Algorithm 7 Best-First Search

```
Input: $start$, $goal$,$G$,$h$
1: Push($start$, $Open$)
2: while $Open$ not $empty$ do
3:     Remove state $s$ with minimum $f$ from $Open$
4:     if $s == goal$ then return $success$
5:     end if
6:     Expand $s$
7: end while
       return $failure$
```



(a) (b)

Figure 4.1: The priority of states in (a) Dijkstra's algorithm (b) WA*.

4.1.2 Representative BSS Algorithms

Weighted A* (WA*) WA* [64] is a best-first search algorithm that prioritizes nodes by $f(n) = g(n) + w \cdot h(n)$, where $g(n)$ is the known least cost from start to n, and w is a chosen constant which satisfies $w > 1$. This is equivalent to prioritizing by $f(n) = h(n)+g(n)/w$. The solution returned by WA* is upper-bounded by wC^* [66]. It is proven that the solution is bounded even if WA* does not reopen states [23]. This bound is loose: in practice, the quality of the returned solution is often much better than this theoretical guarantee. Details will be addressed in Section 4.4

The implementation of WA* is relatively simple. Applying a small modification to the A* priority function can result in WA*, without having to change the data structures or other parts of the algorithm.

Figure 4.1 illustrates how the expansion order changes when we change the way we compute f-cost. In both examples, the red state has $g = 2$, $h = 3$, while the black state has $g = 4$, $h = 1$.

In Dijkstra's algorithm, the f-cost of the red state is 2 and that of the black state is 4, therefore the algorithm expands the red state before the black state. However, in weighted A* with $w = 2$, we can see that $f(red) = 4$ and $f(black) = 3$ and the algorithm expands the black state first.

A*$_\epsilon$ A*$_\epsilon$ [63] was the first focal search algorithm. The idea of focal search is to create a subset of the *Open* list as the focal list and only choose candidate states from the focal list to expand during the search. States are placed on the focal list if their cost satisfies $f(n) \leq w \cdot min(f_{Open})$. A separate priority function f_{focal} is used to sort all the states in focal. It can be proven that if a state on the focal list is expanded and a solution is found, then the solution will be w-admissible. The pseudo-code for focal search is Algorithm 8.

Different focal search algorithms use different strategies to prioritize the focal list. A*$_\epsilon$ selects the state with minimum $\hat{d}(n)$, where $\hat{d}(n)$ is an inadmissible distance-to-go estimate, since that node appears to be the closest to a goal. If A*$_\epsilon$ does not reopen states, then the solution is not bounded by wC^* [23].

Explicit Estimation Search (EES) EES is another focal search algorithm. It uses three queues called the *Open* list, the focal list, and the cleanup list. The *Open* list sorts states by $\hat{f}$ which uses a cost-to-go estimate [1], while the focal list sorts states by $\hat{d}$ which uses a distance-to-go estimate [2]. The f used in the cleanup list is the regular f-cost.

The node selection strategy of EES works as follows:

1. if $\hat{f}(best_{\hat{d}}) \leq w \cdot f(best_f)$ then $best_{\hat{d}}$
2. else if $\hat{f}(best_{\hat{f}}) \leq w \cdot f(bestf)$ then $best_{\hat{f}}$

[1]A cost-to-go estimate is what we call a heuristic function, estimating the remaining path cost.
[2]A distance-to-go estimate is the estimated number of edges in the remaining path.

Algorithm 8 Focal Search

```
Input: start, goal, G, h
1: Push(start, Open)
2: Push(start, FOCAL)
3: while FOCAL not empty do
4:     best ← ChooseNode(FOCAL)
5:     Remove best from Open
6:     Remove best from FOCAL
7:     if best == goal then return success
8:     end if
9:     if $f_{min}$ increased then
10:        FixFocal()
11:    end if
12:    for each successor n of best do
13:        Add n to Open
14:        if $f(n) \leq Bf_{min}$ then
15:            Add n to FOCAL
16:        end if
17:    end for
18: end while
    return failure
```

3. else $best_f$

EES must reopen states to guarantee a B-admissible solution.

Optimistic Search Optimistic Search exploits the fact that the solution found by WA* is usually much better than its guarantee. It uses WA* with higher weights $(2w-1)$ to search for a solution, and then uses A* to improve the found solution until the incumbent solution meets the solution quality requirement. Optimistic Search reopens states both in the search stage and the proving stage. In Section 4.4 we will revisit this algorithm and show an improved version of this algorithm.

Dynamic Potential Search (DPS) DPS is another focal search algorithm. In its focal list, it chooses the states with maximum $\frac{w \cdot f_{min} - g(n)}{h(n)}$ to expand, where f_{min} comes from the *Open* list.

Note that $\frac{w \cdot f_{min} - g(n)}{0 - h(n)}$ is the slope of a line that goes through the point $(h(n), g(n))$ and the point $(0, w \cdot f_{min})$ in the $h - o - g$ plane. Then if we take a second look at the DPS priority function, we will notice that $\frac{w \cdot f_{min} - g(n)}{h(n)}$ is the absolute value of that

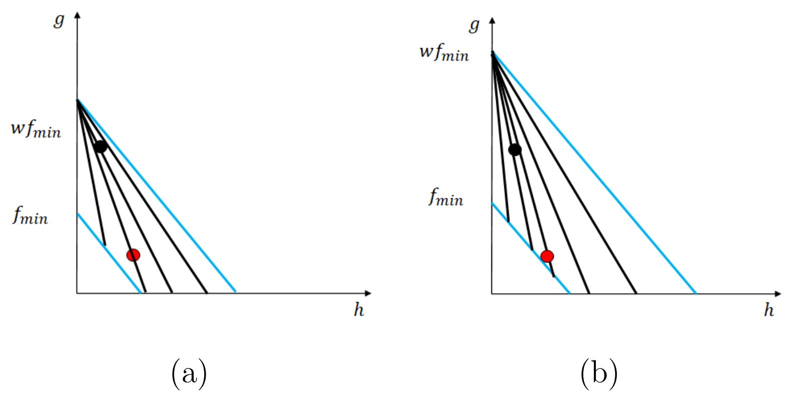


Figure 4.2: The priority of states during a DPS search procedure (a) with smaller f_{min} (b) with larger f_{min}.

slope. Thus the priority function for DPS can be viewed as the absolute value of a slope of a line that goes through the point $(h(n), g(n))$ and the point $(0, w \cdot f_{min})$ in the $h - o - g$ plane. We can call the reference point $(0, w \cdot f_{min})$ the "viewpoint".

Figure 4.2 provides an illustration of the priorities. Whenever f_{min} changes, the "viewpoint" is changed, thus all the slopes are changed. For example, the order of the priority of the red state and the black state changes in Figure 4.2 (a) and (b). Therefore, the focal list needs to be resorted, which is computationally expensive. Also, DPS needs to reopen states during the search to maintain B-admissible solutions.

The original DPS work proves that although conceptually two lists are used, DPS can be implemented with only one list [33].

Above is the brief introduction to the representative BSS algorithms. We often classify them into two categories: WA* and other algorithms. WA* is special because: (1) WA* is a simple best-first search, which is easy to understand and implement; (2) when the heuristic is consistent, it can find a bounded solution without re-openings. As mentioned above, all other algorithms, such as EES and DPS, need to reopen states to guarantee that the solution returned is bounded-suboptimal.

Re-openings, as we will show in the following section, can be harmful to suboptimal search algorithms.

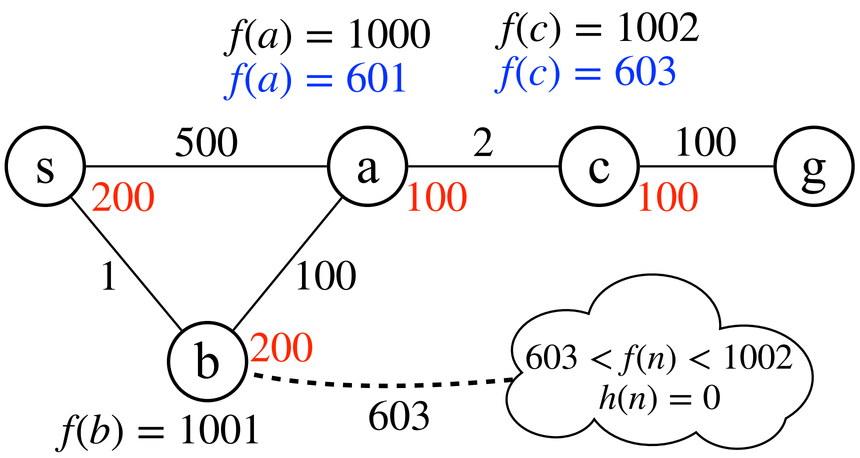


Figure 4.3: Best-case from performing re-openings in an example with $w = 5$.

Also, A$^*_\epsilon$ and EES require extra information (a cost-to-go estimate, which is different from normal the heuristic function), which is not always available.

These facts lead to a question: is there a way to keep the search as simple as possible but still get improved performance? Our answer is, yes, by using BFS-NR with Φ, which will be addressed in detail in Section 4.3.

4.2 Reopening in Bounded Suboptimal Search

In heuristic search, reopening, or re-expanding, means taking a state from the *Closed* list and placing it back onto the *Open* list.

We will demonstrate that node re-openings can have a significant negative or positive impact on performance. Previous work has studied the influence of re-openings on the solution quality in WA* [90], but we are not aware of any work that shows worst-case bounds for the total number of expansions. In the following we show our worst-case analysis of the impact of reopenings in WA* when the underlying heuristics are consistent. The examples are inspired by the analysis of inconsistent heuristics and BPMX [27]. In that paper, Δ is defined to be the greatest common divisor of all the non-zero edge weights. The critical observation they made is that each time a

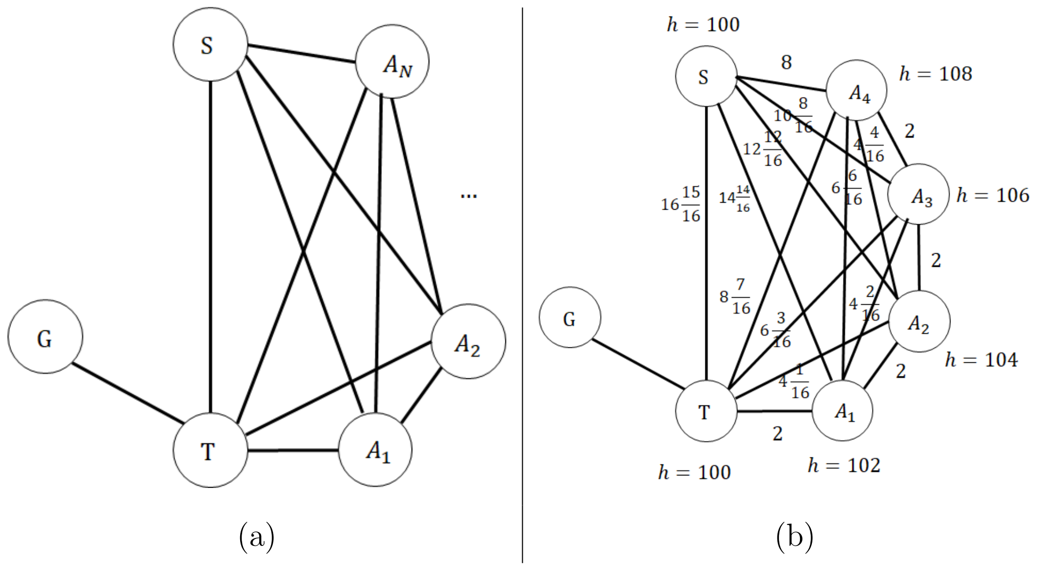


Figure 4.4: (a) Generic example where re-openings require $O(2^N)$ re-expansions. (b) Specific instance of part (a) for $N = 4$ and $w = 2$.

node n is reopened, $g(n)$ is increased by at least Δ. Therefore, they concluded that if A* performs $M > N$ node expansions then there must be a edge with cost of at least $\Delta \times \lceil (M - N)/N \rceil$.

The first example, in Figure 4.3, shows how performing re-openings can provide arbitrarily large savings. In this figure edges are labeled with their cost in black. Nodes are labeled with their h-cost in red. If re-openings are not allowed, a WA* search with $w = 5$ will expand s, followed by a with $f(a) = 1000$ and b with $f(b) =$

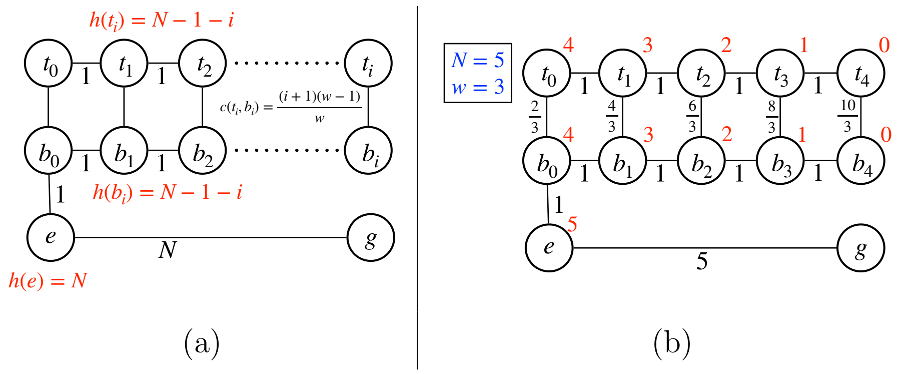


Figure 4.5: (a) Generic example where re-openings require $O(N^2)$ re-expansions. (b) Specific instance of part (a) for $N = 5$ and $w = 3$.

Table 4.1: The order of expansions in Figure 4.4(b)

Order	State	f-cost	Order	State	f-cost
1	T	$216\frac{15}{16}$	17	T	$216\frac{7}{16}$
2	A_1	$218\frac{14}{16}$	18	A_1	$218\frac{6}{16}$
3	T	$216\frac{14}{16}$	19	T	$216\frac{6}{16}$
4	A_2	$220\frac{12}{16}$	20	A_2	$220\frac{4}{16}$
5	T	$216\frac{13}{16}$	21	T	$216\frac{5}{16}$
6	A_1	$218\frac{12}{16}$	22	A_1	$218\frac{4}{16}$
7	T	$216\frac{12}{16}$	23	T	$216\frac{4}{16}$
8	A_3	$222\frac{8}{16}$	24	A_3	222
9	T	$216\frac{11}{16}$	25	T	$216\frac{3}{16}$
10	A_1	$218\frac{10}{16}$	26	A_1	$218\frac{2}{16}$
11	T	$216\frac{10}{16}$	27	T	$216\frac{2}{16}$
12	A_2	$220\frac{8}{16}$	28	A_2	220
13	T	$216\frac{9}{16}$	29	T	$216\frac{1}{16}$
14	A_1	$218\frac{8}{16}$	30	A_1	218
15	T	$216\frac{8}{16}$	31	T	216
16	A_4	224			

1001. Then, because a cannot be re-opened, the cloud of nodes at the bottom, which can be arbitrarily large, will be expanded with $f(n) > 603$. However, if re-openings are allowed, then a will be re-opened with $f(a) = 601$, shown in blue. This leads to c being expanded with $f(c) = 603$, followed by the goal with $f(g) = 602$. With re-openings, only 6 expansions are required, but without re-openings, an arbitrarily large number are needed.

In the second example in Figure 4.4(a) we show that WA* can do exponentially worse if there are exponentially many edge costs. The example can be viewed as Martelli's family [54], where the number of edge weights grows exponentially with the graph size [94].

Table 4.2: The order of expansions in Figure 4.5(b)

Order	State	f-cost	Order	State	f-cost
1	t_0	12.00	12	b_1	11.33
2	t_1	10.00	13	b_2	9.33
3	t_2	8.00	14	b_3	7.33
4	t_3	6.00	15	b_4	5.33
5	t_4	4.00	16	b_0	12.67
6	b_4	7.33	17	b_1	10.67
7	b_3	8.67	18	b_2	8.67
8	b_4	6.67	19	b_3	6.67
9	b_2	10.00	20	b_4	4.67
10	b_3	8.00	21	e	16.67
11	b_4	6.00	22	g	6.67

In this graph, $h(S) = h(T) = 100$, $h(A_i) = 100 + 2xi$, $c(T, A1) = c(A1, A2) = ...c(A_{N-1}, A_N) = 2x$, where x is a real number such that $x > \frac{1}{2w-2}\frac{1}{2^N}$. Note that there are 2^N paths from S to T in the following format: $S, A_{k,k}, A_{k,k-1}, ...A_{k,1}, T$, where $0 \leq k \leq N$ and $k, j > k, i$ if $j > i$.

For each path, we can make a one-to-one mapping of that path to a n-bit binary string $b_N, b_{N-1}, ..., b_1$ in this way: for each i, $b_i = 1$ if A_i is not part of the path. Note that is is equivalent to assigning an integer in the range $[0, 2^N)$ to a path.

Once we assign the edge costs carefully, as above, we can verify that the first time the algorithm will find a path of cost $4x + \frac{2^N-1}{2^N}$, and then a better path of cost $4x + \frac{2^N-2}{2^N}$, and then a better path of cost $4x + \frac{2^N-3}{2^N}$, ..., following the pattern all the way to the final path of cost $4x$. Since each time we strictly decrease the cost by $\frac{1}{2^N}$, we can guarantee that state T will be expanded 2^N times.

A concrete instance of the exponential worst case is shown in Figure 4.4(b) for $N = 5$ and $w = 2$. Table 4.1 shows the order of node expansions in this example.

The third example in Figure 4.5(a) shows the potential cost of performing re-openings when we have polynomial many edge costs. While this is an artificial example, the example is reflective of similar real-world problems that arise in grid pathfinding problems. Figure 4.5(a) is a scalable example that works for any weight $w > 1$ and for the problem size, N (the total number of states in the state space is $2N + 2$). The start state is t_0 and the goal is g. There is a top row of states t_i and a bottom row of states b_i with identical heuristics and unit edge costs between states in the same row. The cost of the edges between the top and bottom paths gradually increases along the path. After expanding the top row, the states going across the bottom row from left-to-right have decreasing f-costs. Thus, the bottom nodes will be expanded right-to-left. After expanding each node, all subsequent nodes to the right in the row will be re-expanded, as the g-cost has decreased.

We provide a concrete instance of the general graph in Figure 4.5(b) for $N = 5$ and $w = 3$. Table 4.2 shows the order of node expansions in this example. There are 22 total node expansions; b_4 is expanded $N = 5$ times, b_3 is expanded 4 times, and so on.

In general, the top nodes (t_i) will each be expanded once. The bottom nodes (b_i) will have $\frac{N(N+1)}{2}$ cumulative expansions. The last two nodes (e and g) will each be expanded once. Cumulatively there are $2N+2$ which is $\Theta(N)$ nodes in the graph and there will be $\frac{N(N+1)}{2} + N + 2$ which is $\Theta(N^2)$ total expansions. This is equivalent to the worst case performance of the A* variants B and B' with inconsistent heuristics [27, 54, 55].

Summarizing the results here, we provided examples showing that re-openings in WA* can result in arbitrarily large savings or $O(N^2)$ total expansions in a problem with $O(N)$ states. Thus, the choice of whether to use re-openings is going to depend on the properties of a domain. If there are many transpositions, such as in grid maps, the overhead of re-openings can be expensive.

Reopening is not mandatory for suboptimal search. In next section, section 4.3, we

Algorithm 9 Best-First Search Guided by Φ

Input: *start*, *goal*,G,h,Φ,RP

```
1: Push(start, Open)
2: while Open not empty do
3:     Remove state $s$ with minimum $\Phi(h(s), g(s))$ from Open
4:     if $s$ == goal then return success
5:     end if
6:     Move $s$ to Closed
7:     for each successor $s_i$ of $s$ do
8:         if $s_i$ on Open then
9:             Update $g(s_i)$ of $s_i$ on Open if shorter
10:        else
11:            if $s_i$ not on Closed then
12:                Add $s_i$ to Open
13:            else
14:                if RP == True then
15:                    reopen $s_i$ if shorter
16:                end if
17:            end if
18:        end if
19:    end for
20: end while
       return failure
```

will show that suboptimal search algorithms can be complete or even B-admissible even if they do not reopen states.

4.3 Algorithm: BFS-NR Guided by Φ

Best-first search with a priority function is a general algorithm; its pseudo-code is shown in Algorithm 9. The algorithm keeps expanding the state with minimum f-cost according to the provided Φ function until the goal state is removed from *Open*, when it terminates.

We introduce the notion of a Φ function which determines how potential suboptimality is distributed across a path, potentially allowing more suboptimality at the beginning or end of the search, i.e. where the portion of g-cost is small or large. We assume $\Phi(x, y)$ is a function $\mathbb{R}^2 \to \mathbb{R}$. $\Phi(x, y)$ is continuous, but not necessarily differentiable. Given a state u in *Open* reached with cost $g(u)$, the priority of u is $f(u) = \Phi(h(u), g(u))$.

Once a state is expanded, it is removed from the *Open* list and put onto the *Closed* list, while its successors are put on *Open* or updated to a lower cost if they already are on *Open* with higher cost. If a shorter path is found to a state on *Closed*, it may optionally be reopened.

When we do best-first search, one could try different reopening policies, such as always re-expand (AR) and never re-expand (NR) [72]. Once we apply the NR policy to best-first search, we get Best-First Search with the NR policy (BFS-NR) [91]. In Algorithm 9, by simply setting the parameter RP to be $False$, we can get BFS-NR.

The first interesting fact is that BFS-NR is complete.

Theorem 24 *In a finite state space, a best-first search with any priority function* Φ *is complete (finds a solution if one exists) even if it does not re-open states.*

Proof.

Assume that the optimal path from *start* to *goal* is $p_0, p_1, p_2, ...p_n$, where $p_0 = start$, $p_n = goal$. If *goal* is not expanded, then prior to each expansion there always exists at least one "frontline" state p_i. A "frontline" state p_i is a state such that $0 \leq i \leq n$ and p_i is on *Open* while none of $p_{i+1}, ...p_n$ is closed. This implies that the search always makes progress on exploring the optimal path (even if the costs used to explore the path are not optimal), and thus will eventually expand the goal and complete. We prove this by induction.

Base case: At the very beginning, p_0, which is the *start* state, is on *Open*, meeting the requirement for this lemma.

Inductive step: Assume at some point there exists some p_i which meets this condition. If the next state that is chosen for expansion is not from $p_i, p_{i+1}, ...p_n$, then p_i is still a "frontline" state after the next expansion. Otherwise, suppose we choose to expand p_k which is one of $p_i, p_{i+1}, ...p_n$. Then p_{k+1} will be placed on open and meet the condition. ■

According to Section 4.2, we know reopening can be bad; according to Theorem 24, we are guaranteed to find a solution if we use NR policy. Then can we just use NR policy? Unfortunately, for bounded suboptimal search, we not only want a solution but also care about the solution quality; we want the algorithm to be B-admissible.

BSS algorithms such as A^*_ϵ [63], EES [87], and DPS [33] are forced to reopen states in order to be B-admissible.

Thus, in the following context, we discuss how to find bounded suboptimal solutions using the NR policy. More specifically, BFS-NR algorithm. When we apply the NR policy to best-first search, we get Best-First Search with the NR policy (BFS-NR) [91]. In Algorithm 9, by simply setting the parameter RP to be $False$, we can get BFS-NR.

4.3.1 Generalized Conditions Making BFS-NR Bounded

We assume the underlying heuristic is consistent. In the following text, we will show that when we make a few fair assumptions, it follows that a consistent heuristic is a necessary condition.

Different Φ will result in different behaviors of the algorithm. What we are interested in are the Φ functions that make BFS-NR a B-admissible algorithm.

Here is the intuition why there exist such Φ functions: when we use WA*, we are forcing each segment to be bounded. i.e., the cost of the path found between any two states s_i and s_j on the closed list is bounded. However, that is stronger than what we need. We only care about the cost of the path between the start and the goal, and we do not necessarily care if the cost to some non-goal state is not bounded. Thus, it is possible that there exist some Φ functions that guarantee the overall solution quality but locally allow paths not to be bounded. In the following context, we will show that such Φ functions exist. There are infinitely many such functions, as long as they satisfy the necessary and sufficient conditions that we will define shortly. We will also provide a few specific functions.

We start with a few intuitive assumption that Φ will satisfy.

Property 1 *For any given $\delta > 0$, $\Phi(x+\delta, y) > \Phi(x, y)$, $\Phi(x, y+\delta) > \Phi(x, y)$.*

Property 2 *For any given $\delta > 0$, $\Phi(x, y+\delta) \leq \Phi(x+\delta, y)$*

Property 1 means that the Φ-value grows monotonically along each axis. In terms of the behavior of the search algorithm, it is equivalent to say that for two states with same h-cost, the one with lower g-cost should be preferred; similarly, for two states with same g-cost, the one with lower h-cost should be preferred.

Property 2 means when there are two states with the same sum of h-cost and g-cost, the one with lower g-cost should not be preferred over the one with higher g-cost, although they could be equally preferred.

Property 3 $\Phi(x, 0) = x$.

The return value of the Φ function provides a partial ordering over states in *Open*. In this sense, there are many possible partial orderings that Φ could use. We assume $\Phi(x, 0) = x$ because (1) it gives Φ a semantic meaning: Φ is a lower bound on the optimal solution cost, not only for the start state, but also for every state expanded; and (2) it will simplify our remaining derivations.

With the first 3 assumptions made, now we can say something interesting about Φ.

Definition 25 *Let I_{AD} be all problem instances with admissible heuristics and I_{CON} be all instances with consistent heuristics. Furthermore, let I_{CON_S} stand for instances with strongly consistent heuristics, while I_{CON_W} stands for instances with weakly consistent heuristics. The instances with inconsistent but admissible heuristics are denoted by I_{INC}.*

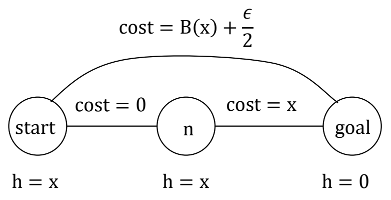


Figure 4.6: Illustration for Theorem 26.

Theorem 26 *If, for a given Φ, there exists an $x \geq 0$ such that $\Phi(x, 0) > \Phi(0, B(x))$, where B is the given bounding function, then BFS-NR using Φ is not guaranteed to be bounded on I_{AD}.*

Proof. Suppose $\Phi(x, 0) > \Phi(0, B(x))$. Then, according to Property 1, there exists an $\epsilon > 0$ such that $\Phi(x, 0) = \Phi(0, B(x) + \epsilon)$.

Then, we can create a problem instance that belongs to I_{AD} with 3 states, *start*, n, *goal*, as illustrated in Figure 4.6. In this example $h(start) = h(n) = x$, $c(start, n) = 0$, $c(start, goal) = B(x) + \frac{\epsilon}{2}$, $c(n, goal) = x$. On this problem BFS-NR will expand the goal and find a solution of cost $B(x) + \frac{\epsilon}{2}$, meaning it is not bounded. ■

We now introduce two properties, the first of which, Property 4, is necessary according to Theorem 26.

Property 4 $\Phi(x, 0) \leq \Phi(0, B(x))$, *where B is the given bounding function.*

Property 5 $\Phi(x, 0) = \Phi(0, B(x))$, *where B is the given bounding function.*

Note that Property 5 is more strict than Property 4. Any Φ that meets Property 5 will also meet Property 4.

Weighted A* meets the stronger Property 5. This seems to be a natural choice, since it makes full use of the allowable suboptimality. But, some algorithms, such as Dynamic WA* [66], have $\Phi(x, 0) < \Phi(0, B(x))$. Φ_{AB}, which will be introduced in next section, satisfies Property 4 everywhere, but does not satisfy Property 5 in one portion of the space.

Given Properties 1–4, we present an important result, the Φ-inequality.

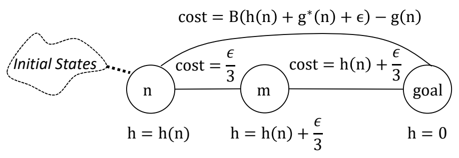


Figure 4.7: Illustration for Theorem 30.

Definition 27 *Suppose BFS-NR expands a state n with cost $g(n)$. The Φ-inequality is $\Phi(h(n), g(n)) \leq g^*(n) + h(n)$*

When the heuristic is consistent, $g^*(n) + h(n)$ is a lower bound on the optimal cost from *start* to *goal* through n. Thus, Φ can be interpreted as an estimate of the optimal path cost through the each state. Note that this contrasts with how f is used in algorithms like WA* – as an estimate of the solution cost that will be found.

In the following, we will show that BFS-NR is bounded if and only if the Φ-inequality holds. As the Φ function determines how potential suboptimality is distributed across a path, the Φ-inequality guarantees that the local suboptimality is bounded such that a bounded suboptimal path to the goal always exists.

Theorem 28 [*Sufficiency*] *Suppose that Φ satisfies Property 1 - 4. Then, if the Φ-inequality holds for every state expanded, BFS-NR is bounded on I_{AD}.*

Proof. Suppose the Φ-inequality holds for every state expanded, then it also holds for *goal*. Therefore, $\Phi(0, g(goal)) \leq g^*(goal) + h(goal) = C^*$. According to Property 4, $C^* \leq \Phi(0, B(C^*))$. According to Property 1, $g(goal) \leq B(C^*)$, therefore the solution is bounded. ■

Since Property 5 is stronger than Property 4, then Theorem 28 also holds for Property 5, which we state below, as Corollary 29.

Corollary 29 *Suppose that Φ satisfies Property 1 - 3 and Property 5. Then, if the Φ-inequality holds for every state expanded, BFS-NR is bounded on I_{AD}.*

Theorem 28 suggests that the Φ-inequality is a sufficient condition for BFS-NR to be bounded. But, is it a necessary condition? The short answer is, yes. The next theorem, Theorem 30, tells us that this condition must hold for every single expansion. Otherwise, there could be some instance $I \in I_{AD}$ on which BFS-NR would not be bounded. This theorem requires Property 5; there is no equivalent theorem when only Property 4 holds.

Theorem 30 [*Necessity*] *Suppose Φ satisfies Property 1 - 3 and Property 5. Then, only if the Φ-inequality holds for every state expanded will BFS-NR be bounded on I_{AD}.*

Proof.

We prove the contrapositive.

Suppose there exists one state n expanded by BFS-NR on a problem $I \in I_{AD}$ that violates the Φ-inequality. That is, $\Phi(h(n), g(n)) > h(n) + g^*(n)$. Then, we can show that BFS-NR is not bounded on I.

Figure 4.7 illustrate an I_{AD} instance with consistent heuristic. Suppose BFS-NR expands a few initial states and comes to the situation illustrated in Figure 4.7, where BFS-NR decides to expand the state n whose $\Phi(h(n), g(n)) > h(n) + g^*(n)$ without knowing which states are after n. Since Φ is continuous, there must exist $\epsilon > 0$ such that $\Phi(h(n), g(n)) = h(n) + g^*(n) + \epsilon$. Now, we can alter the remaining graph such that there are only 2 successors of n: m and *goal*. We set $c(n, goal)$ such that $\Phi(0, g(n) + c(n, goal)) = \Phi(h(n), g(n))$. $h(m) = h(n) + \frac{\epsilon}{3}$, $c(n, m) = \frac{\epsilon}{3}$, $c(m, goal) = h(n) + \frac{\epsilon}{3}$.

Since the algorithm just expanded n, we know n was the state with minimum priority on *Open*. Thus, once we expand n and put m and *goal* on *Open*, the *goal* will be chosen for expansion immediately, since it will be the state with minimum Φ value. In this case, $\Phi(0, g(goal)) = h(n) + g^*(n) + \epsilon = \Phi(0, B(h(n) + g^*(n) + \epsilon))$. However, there exists a path from *start-n-m-goal*, whose cost is $h(n) + g^*(n) + \frac{2\epsilon}{3}$. In this problem, $C^* \leq h(n) + g^*(n) + \frac{2\epsilon}{3}$, while the solution we returned has cost $B(h(n) + g^*(n) + \epsilon)$, which is strictly greater than $B(C^*)$.

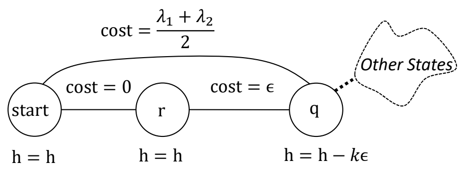


Figure 4.8: Illustration for Theorem 31.

Hence, the BFS-NR is not bounded on I. ■

In short, if BFS-NR expands a state that violates the Φ-inequality, we can construct a new problem instance where BFS-NR will not be bounded.

Thus, to understand when BFS-NR is guaranteed to find bounded-suboptimal solutions, we must study the properties that will cause the Φ-inequality to hold. As we will now show, the Φ-inequality holds for each expansion if and only if we have Properties 1–3, Property 5 and Property 6, and a consistent heuristic. Note that Properties 1–3 and Property 5 are general assumptions and have already been introduced; Property 6 will be introduced in the following context.

Theorem 31 *Suppose that Φ satisfies Property 1 - 3. Then there does not exist a Φ that can guarantee that the Φ-inequality holds for all states expanded on problem instances $I \in I_{INC}$.*

By construction.

We create a parametric example where using any Φ function that meets Property 1 - 3 with an inconsistent heuristic will cause BFS-NR to not be bounded.

As is illustrated in Figure 4.8, create a $I \in I_{INC}$ where *start* has 2 successors, q and r.

$h(start) = h$, $h(q) = h - k\epsilon$, $h(r) = h$, $c(start, r) = 0$, $c(r, q) = \epsilon$. where $k > 1$. Then we can see that the heuristic is inconsistent between q and r.

Since Φ is continuous, we can let $\Phi(h - k\epsilon, \lambda_1) = h - k\epsilon + \epsilon$ and $\Phi(h - k\epsilon, \lambda_2) = h$. According to Property 1, $\lambda_1 < \lambda_2$.

Then set $c(start, q) = \frac{\lambda_1+\lambda_2}{2}$.

We can compute

$$f(q) = \Phi(h(q), g(q)) = \Phi(h - k\epsilon, \frac{\lambda_1 + \lambda_2}{2})$$

$$f(r) = \Phi(h(r), g(r)) = \Phi(h, 0) = h$$

Since $\lambda_1 < \lambda_2$, according to Property 1,

$$\Phi(h - k\epsilon, \frac{\lambda_1 + \lambda_2}{2}) < \Phi(h - k\epsilon, \lambda_2) = h$$

. Therefore $f(q) < f(r)$. Thus we will expand q instead of r.

On the other hand,

$$h(q) + g^*(q) = h(q) + c(p, r) + c(r, q) = h - k\epsilon + \epsilon$$

$$\Phi(h - k\epsilon, \frac{\lambda_1 + \lambda_2}{2}) > \Phi(h - k\epsilon, \lambda_1) = h - k\epsilon + \epsilon$$

which breaks the Φ-inequality on state q. ■

Theorem 32 *Suppose that Φ satisfies Property 1 - 3. Then, there does not exist a Φ that can guarantee that BFS-NR is bounded for instances $I \in I_{INC}$.*

Proof.

Theorem 30 tells us the Φ-inequality must hold for every single expansion. Theorem 31 shows that it will not on problems in I_{INC}. ■

If the heuristic is not consistent, then without further assumptions, we cannot guarantee that the algorithm returns a bounded solution on every instance. It may return bounded solutions for some instances, but not for all. Thus, Theorem 32 indicates that consistent heuristics are a necessary condition to guarantee that BFS-NR is bounded for all instances $I \in I_{CON}$. This is not counter-intuitive; A^* also needs to perform re-expansions to guarantee optimal solutions when the heuristic is inconsistent [27, 54].

Now we will study Φ functions given problems in I_{CON}.

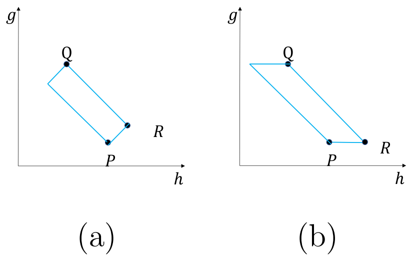


Figure 4.9: The successor parallelogram when the heuristic is (a) strongly consistent (b) weakly consistent.

Definition 33 *Successor Parallelogram.*

Figure 4.9 (a) and (b) illustrate the successor parallelogram for strongly consistent heuristics and weakly consistent heuristics, respectively. Suppose that state q is a descendant of state p. p is an expanded state. The h-cost and g-cost of p are $h(p)$, $g(p)$, respectively. We put a point P in the $h-o-g$ plane, whose coordinates are $(h(p), g(p))$. We put another point Q, with coordinates $(h(q), g(p)+d(p,q))$. This point represents the cost of q if we reach q optimally from p.

After that, we draw out two 45° straight lines from P and Q (45° straight lines correspond to consistency). For a strongly consistent graph, we draw out two -45° straight lines from P and Q; for weakly consistent graph, we draw out two horizontal lines from P and Q.

These four lines will give us a parallelogram in every possible case. Such a parallelogram is called a successor parallelogram.

As the name indicates, we can prove that for any state p_1 on the optimal path from p to q, p_1 must be in the successor parallelogram, given that the heuristic is consistent.

Definition 34 *We define R as the right most corner of successor parallelogram.*

Point R is interesting because that is the point with the maximum Φ value in the parallelogram.

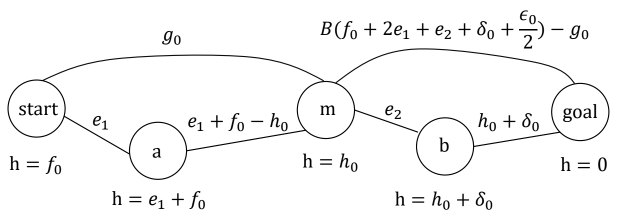


Figure 4.10: The examples for Theorem 37

.

Lemma 35 *Point R is the point with maximum Φ value.*

Proof. According to Property 1, the maximum Φ value is on the line segment QR.

For any given point S in line segment QR, Let $x = x_S$, $y = y_R$, $\delta = x_R - x_S$. According to Property 2, we know $\Phi(x_R, y_R) \geq \Phi(x_S, y_S)$. ■

The next property is the key property needed for avoiding re-expansions. It is a sufficient condition to guarantee the Φ-inequality under Property 4. It is necessary when Φ meets Property 5. There are two versions of the condition depending on the consistency of the heuristic:

Property 6 [*Consistency of* Φ] *For a strongly consistent heuristic: for any given $\delta > 0$, $\Phi(x + \delta, y + \delta) \leq \Phi(x, y) + 2\delta$*

For a weakly consistent heuristic: for any given $\delta > 0$, $\Phi(x + \delta, y) \leq \Phi(x, y) + \delta$

We refer to this property as the Φ function being *consistent*, as it arises from the consistency of the underlying heuristic. This limits a Φ function from changing too quickly.

Theorem 36 *Assume BFS-NR is using a priority function Φ which meets Properties 1 to 4 and 6 on an problem instance $I \in I_{CON}$. Then for all expansions the Φ-inequality holds.*

Proof.

Proof by induction. This is the proof for strongly consistent heuristics; for weakly consistent heuristics, the argument is analogous.

Base case: Initially *start* is chosen for expansion, so $\Phi(h(start), g(start)) = \Phi(h(start), 0) = \Phi(h(start) + g^*(start), 0)$, and the claim holds.

Inductive step: Assume the Φ-inequality holds for all expanded paths and a state q is the next state from *Open* is chosen for expansion. We need to prove that $\Phi(h(q), g(q)) \leq \Phi(h(q) + g^*(q), 0)) = h(p) + g^*(p)$.

Let the node that is closed on an optimal path from *start* to q with highest g^*-value be p. The hypothesis guarantees that:

$$\Phi(h(p), g(p)) \leq \Phi(h(p) + g^*(p), 0) = h(p) + g^*(p) \tag{4.1}$$

Since p is closed, there must exist a direct successor of p, p_x on *Open*, which is on the optimal path from p to q. Examining the successor parallelogram formed around p and q, according to Lemma 35, the right corner point R is of maximum priority, whose coordinate is: $(h(p) + t, g(p) + t)$, where

$$t = \frac{d(p,q) + h(q) - h(p)}{2} \tag{4.2}$$

.

Therefore,

$$\begin{aligned}\Phi(h(p_x), g(p_x)) &= \Phi(h(p_x), g(p) + d(p, p_x)) \\ &\leq \Phi(h(p) + t, g(p) + t)\end{aligned} \tag{4.3}$$

According to Property 6,

$$\begin{aligned}&\Phi(h(p) + t, g(p) + t) \\ &\leq \Phi(h(p), g(p)) + 2t \\ &= \Phi(h(p), g(p)) + d(p,q) + h(q) - h(p)\end{aligned} \tag{4.4}$$

By combining inequalities (4.1), (4.3) and (4.4) we can get

$$\begin{aligned}
&\Phi(h(p_x), g(p_x)) \\
&\leq h(p) + g^*(p) + d(p,q) + h(q) - h(p) \\
&= h(q) + g^*(p) + d(p,q) \qquad (4.5) \\
&= h(q) + g^*(q) \\
&= \Phi(h(q) + g^*(q), 0)
\end{aligned}$$

Since during the search, we chose to expand q instead of p_x, the priority of q will be no more than that of p_x. Thus:

$$\Phi(h(q), g(q)) \leq \Phi(h(p_x), g(p_x)) \qquad (4.6)$$

Combine inequality (4.5) and (4.6) together, we can get:

$$\Phi(h(q), g(q)) \leq \Phi(h(q) + g^*(q), 0)$$

i.e. the Φ-inequality holds for expansion on q.

This proves that the Φ-inequality holds for each expansion. ■

Theorem 37 *Assume BFS-NR is using a priority function Φ with a bounding function $B(x)$ on a problem instance $I \in I_{CON}$. If Φ is not consistent then the Φ-inequality is not always guaranteed to hold. That is, if there exists h_0, g_0, δ_0 such that (1) for a strongly consistent heuristic $\Phi(h_0 + \delta_0, g_0 + \delta_0) > \Phi(h_0, g_0) + 2\delta_0$, or (2) for a weakly consistent heuristic $\Phi(h_0 + \delta_0, g_0) > \Phi(h_0, g_0) + \delta_0$, then the Φ-inequality is not always guaranteed to hold.*

By construction.

We will construct an example where BFS-NR fails if Φ is not consistent. In the following proof, $e_2 = \delta_0$ if the heuristic is strongly consistent, $e_2 = 0$ if the heuristic is weakly consistent.

Let $\Phi(h_0, g_0) = f_0$, $\Phi(h_0 + \delta_0, g_0 + e_2) = \Phi(h_0, g_0) + \delta_0 + e_2 + \epsilon_0$. Figure 4.10 shows the instance where this holds where the h-costs are $h(m) = h_0$, $h(a) = f_0 + e_1$, $h(b) = h_0 + \delta_0$, and the edge costs are $c(start, m) = g_0$, $c(start, a) = e_1$, $c(a, m) = e_1 + f_0 - h_0$, $c(m, goal) = B(f_0 + 2e_1 + e_2 + \delta_0 + \frac{\epsilon_0}{2}) - g_0$, $c(m, b) = e_2$, $c(b, goal) = \delta + h_0$.

Where

$$e_1 \leq \frac{h_0 + g_0 - f_0}{2} \tag{4.7}$$

and

$$e_1 < \frac{\epsilon_0}{4} \tag{4.8}$$

.

(Note that $\Phi(h_0, g_0) = f_0$ and $h_0 + g_0 = \Phi(h_0 + g_0, 0)$. According to Property 2, $\Phi(h_0, g_0) \leq \Phi(h_0 + g_0, 0)$, which guarantees $h_0 + g_0 - f_0 \geq 0$. Therefore, there must exist a non-negative e_1.)

BFS-NR will expand *start* first and put m and a on *Open*. At that point, $f(m) = \Phi(h_0, g_0) = f_0$, $f(a) = \Phi(h(a), g(a)) = \Phi(f_0 + e_1, e_1)$, which means we will expand m before a.

According to Equation 4.7, the cost of *start-a-m* is $2e_1 + f_0 - h_0 \leq (h_0 + g_0 - f_0) + f_0 - h_0 = g_0$, which means *start-a-m* is shorter than *start-m*.

Since we do not re-open nodes, when we reach b, $h(b) = h_0 + \delta_0, g(b) = g(m) + c(m, b) = g_0 + e_2$.

Since $f(b) = \Phi(h(b), g(b))$, we can compute that $f(b) = f_0 + \delta_0 + e_2 + \epsilon_0$.

At the same time,

$f(goal) = \Phi(0, g(goal))$

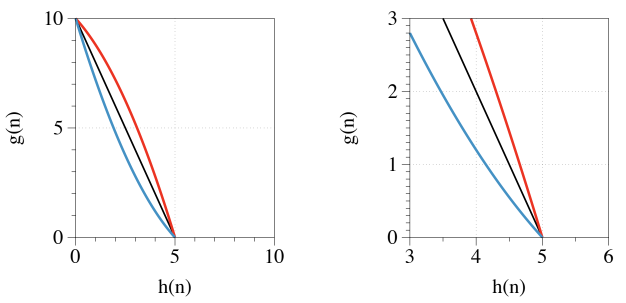
$= \Phi(0, B(f_0 + 2e_1 + \delta_0 + e_2 + \frac{\epsilon_0}{2}))$

$= f_0 + 2e_1 + \delta_0 + e_2 + \frac{\epsilon_0}{2}$

According to Equation 4.8, $f_0 + 2e_1 + \delta_0 + e_2 + \frac{\epsilon_0}{2}$

$< f_0 + 2\frac{\epsilon_0}{4} + \delta_0 + e_2 + \frac{\epsilon_0}{2}$

$= f_0 + \delta_0 + e_2 + \epsilon_0$



(a) (b)

Figure 4.11: Different functions produce isolines with different properties.

Therefore, $f(goal) < f(b)$ and we will expand *goal*. In such a case, the total cost of the path will be $B(f_0 + 2e_1 + e_2 + \delta_0 + \frac{\epsilon_0}{2})$.

However, the shortest path from *start* to *goal* should be *start-a-m-b-goal*, whose cost is $f_0 + 2e_1 + e_2 + \delta_0$. We can see that

$B(f_0 + 2e_1 + e_2 + \delta_0 + \frac{\epsilon_0}{2}) > B(f_0 + 2e_1 + e_2 + \delta_0)$,

which means the expansion of the *goal* breaks the Φ-inequality. ■

To summarize the theoretical results:

(1) On instances in I_{INC}, BFS-NR is not bounded.

(2) On instances in I_{CON}, to make BFS-NR bounded, Φ needs to meet a few properties. We always assume Properties 1 to 3. Then, Property 6 (consistency of Φ) is both necessary and sufficient for BFS-NR to be bounded on I_{CON} when Φ meets Property 5.

If Φ only meets Property 4, then Property 6 is sufficient but not necessary for BFS-NR to be bounded on I_{CON}.

4.3.2 Φ Functions For Linear Bounds

In Section 4.3.1, we present the conditions that allow us to determine whether a Φ function can be used in BFS-NR. In this section, we present specific Φ functions that can be used in search.

Figure 4.12 provides an example showing the limitation of WA*. In this example, we look for a 2-admissible solution. The top path, $start - m - a - goal$ has fewer nodes and higher cost, while the bottom path, $start - m - b - b_1 ... - goal$ has more nodes and optimal cost, 100. Note that the top path has cost 185, which is 2-admissible. However, WA* will not be able to discover that path. It will be forced to expand b_1, b_2,... before expanding a due to their lower f-cost. Thus, WA* cannot solve this problem instance with very few node expansions.

The problem is that WA* implicitly distributes the suboptimally evenly, that is, every segment of the found path must be B-admissible. This is much stronger than what we actually need: a total path that is B-admissible. The following Φ functions we introduce focus on distributing the suboptimality unevenly. In short, Convex Downward Parabola (XDP) and its piecewise version try to search optimally when the g-cost is low and search greedily when the g-cost is high. Convex Upward Parabola (XUP) and its piecewise version does it the other way around. They search greedily when the g-cost is low and search optimally when the g-cost is high.

The formulas and derivations are shown below.

Convex Downward Parabola

Our first priority function has isolines similar to the bottom parabola shown in blue in Figure 4.11. A parabola is a function is of the form $y = ax^2 + bx + c$, where a, b, c are to be determined. Next, we will show how to solve a, b, c and get the equation for Φ. Assume that parabola goes through the points $(0, wU)$ and $(U, 0)$ and has slope -1 at $(U, 0)$. The slope of -1 at $(U, 0)$ means that paths with low g-cost must be

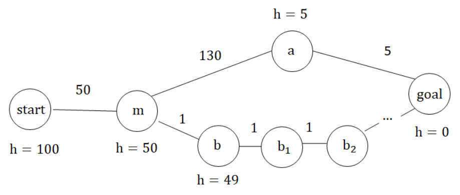


Figure 4.12: An example to show the limitation of the WA* priority function.

near-optimal. This results in the following equation set:

$$\begin{cases} c = wU \\ a \cdot U^2 + b \cdot U + c = 0 \\ 2a \cdot U + b = -1 \end{cases} \tag{4.9}$$

Solving for a, b and c, the equation of the parabola is

$$y = \frac{w-1}{U} \cdot x^2 + (1-2w) \cdot x + wU \tag{4.10}$$

We can rewrite equation (4.10) as an equation of U:

$$wU^2 + (x - 2wx - y)U + wx^2 - x^2 = 0 \tag{4.11}$$

The Φ function we are looking for is in proportion to the larger root of equation (4.11):

$$\Phi_{XDP}(x, y) = \frac{1}{2w}[y + (2w-1)x + \sqrt{(y-x)^2 + 4wyx}] \tag{4.12}$$

Convex Upward Parabola

Our second function corresponds to the top parabola shown in red in Figure 4.11. This parabola goes through points $(0, wU)$ and $(U, 0)$, has slope -1 at point $(0, wU)$. Because the slope is -1 near $(0, wU)$ it means that near the goal the path found must be near-optimal.

$$\begin{cases} c = wU \\ a \cdot U^2 + b \cdot U + c = 0 \\ b = -1 \end{cases} \tag{4.13}$$

Following the same steps as before we get the priority function:

$$\Phi_{XUP}(x, y) = \frac{1}{2w}(y + x + \sqrt{(y+x)^2 + 4w(w-1)x^2}) \tag{4.14}$$

This function is a convex upward parabola (XUP) and also meets properties 1 – 6.

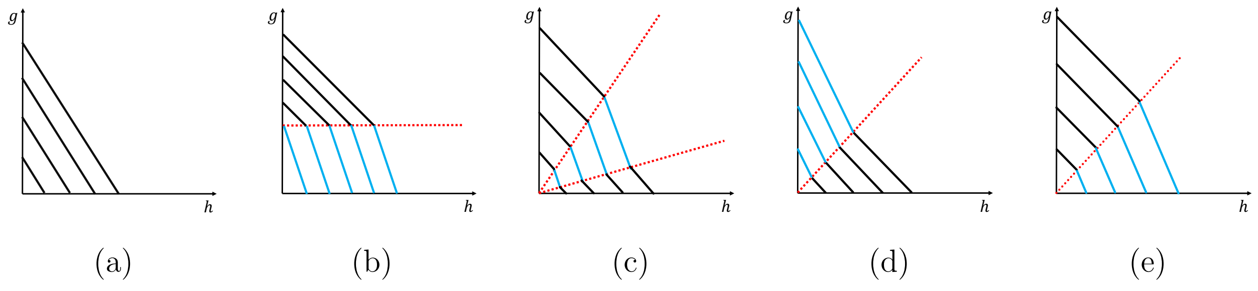


Figure 4.13: The contour plots of (a) Φ_{WA^*} (b) Φ_{AB} (c) Φ_z (d) Φ_{pwXD} (e) Φ_{pwXU}. Red lines indicate free parameters.

Piecewise Functions

A general piecewise function for $B_w(x)$ is:

$$\Phi_{B_w}(x,y) = \begin{cases} x+y & y < K_1 x \\ A(y+Bx) & K_1 x \le y < K_2 x \\ \frac{1}{w}(x+y) & y \ge K_2 x \end{cases} \tag{4.15}$$

Figures 4.13(c)-(d) illustrate several possible contour plots for this priority function, where (c) is the most general form. The behavior of the most general Φ function will perform optimally near *start* and *goal* with suboptimal search in the middle region (where $K_1 x \le y < K_2 x$). Note that equation set has 4 parameters, K_1,K_2, A and B. However, there are only 2 free variables. Once we choose K_1, A is determined. As for the remaining variables, if we fix one, the other is determined. We manually choose K_1 and $B = (2w-1)$, and then compute K_2 and A.

Intuitively, there are a few advantages of this approach over previous *XUP* and *XDP* functions. Firstly, *XDP* and *XUP* are quadratic functions and are more complicated to implement and analyze, involving a square root operation. The piecewise curves are slightly easier to implement in A*, although both approaches only require small changes to A* compared to standard algorithms. These new priority functions also give us more degrees of freedom to distribute the suboptimality. When we are near *start*, we usually need to expand a few states optimally to validate the final solution; when we are close to *goal*, the heuristics are usually perfect, which means that the path we find is typically optimal in those portions. It is in the middle where we actually have the freedom to find suboptimal paths - and we can use a higher degree of suboptimality there, instead of evenly distributing it across the solution path, as WA* does. Φ_{z1} implements this approach, as shown in Figure 4.13(c).

$$\Phi_{z1} = \begin{cases} y + x & y < \frac{1}{w}x \\ \frac{w+1}{2w^2-w+1}(y + (2w-1)x) & \frac{1}{w}x \le y < \frac{2w^2+w+1}{w-1}x \\ \frac{1}{w}(y+x) & \frac{2w^2+w+1}{w-1}x \le y \end{cases} \tag{4.16}$$

We can get simpler forms by setting $K_1 = 0$ or $K_2 = \infty$ resulting in a 2-part piecewise function. We call these the piecewise Convex Downward (pwXD) function Figure 4.13(d) and piecewise Convex Upward (pwXU) Figure 4.13(e). Each function has one free parameter, K. In our experiments, we always use $K = 2w - 1$ giving:

$$\Phi_{pwXD} = \begin{cases} y + x & y < \frac{K-w}{w-1}x \\ \frac{1}{w}(y + Kx) & \frac{K-w}{w-1}x \le y \end{cases} \tag{4.17}$$

$$\Phi_{pwXU} = \begin{cases} \frac{1}{K}y + x & y < \frac{K(w-1)}{K-w}x \\ \frac{1}{w}(y + x) & \frac{K(w-1)}{K-w}x \le y \end{cases} \tag{4.18}$$

Not that for Equation (4.17), if $K > 2w - 1$, then property 6 is violated.

4.3.3 Φ Functions For Additive Bounds

This subsection discusses additive suboptimality bounds. For reference, the Φ function used by WA* with weight w corresponds to straight contour plots with a slope of $-w$, as shown in Figure 4.13(a). We can achieve constant suboptimality with Φ_{AB} and a parameter K, $K \ge \gamma$ that can be tuned.

$$\Phi_{AB}(x, y) = \begin{cases} x + \frac{K-\gamma}{K}y & y < K \\ x + y - \gamma & y \ge K \end{cases} \tag{4.19}$$

Figure 4.13(b) illustrates Φ_{AB}, a priority function that allows a BFS-NR to find a solution with cost at most $C^* + \gamma$. The behavior of this algorithm is easy to understand: Φ_{AB} searches with WA* and then switches to A*. There is one parameter K, which defines the size of the region where WA* is performed. A larger K performs

WA* for longer distances; as a trade off, the weight of WA*, $\frac{K}{K-\gamma}$, must be smaller. We can verify that all the required properties are satisfied. Note that when $x < K-\gamma$, only Property 4 holds because $\Phi_{AB}(x, 0) < \Phi_{AB}(0, B(x))$; when $x \geq K-\gamma$, Property 5 holds because $\Phi_{AB}(x, 0) = \Phi_{AB}(0, B(x))$.

An interesting fact is that reversing the approach by doing A* and then switching to WA* won't work. The following Φ is not guaranteed to find bounded suboptimal solutions because when $x < b$, $\Phi(x+\delta, y+\delta) > 2\delta$.

$$\Phi(x,y) = \begin{cases} \frac{b+\gamma}{b}x + y - \gamma & x < b \\ x + y & x \geq b \end{cases}$$

4.4 Improved Optimistic Search (IOS)

The theoretical guarantee of WA* is w-admissible, i.e., $C/C^* \leq w$, where C is the cost of the solution found by WA* with weight w. It is believed that the bound is loose. Studies show that in practice, C/C^* grows in proportion to $\sqrt{w}$ [41].

Then comes the research question: how can we leverage that empirical observation? Thayer and Ruml proposed Optimistic Search to exploit that property [86]. The idea is to split the search procedure into two phases, the searching phase and the proving phase. In the searching phase, run WA* search with higher weights ($2w-1$ in their experiments) for an incumbent solution. While in the proving phase, run an A* search to re-expand some states until the solution is shortened enough to be proven w-admissible. Since the solution quality is usually much better than $C/C^* \leq 2w-1$, there is usually very little work in phase two.

We combine BFS-NR with Φ and Optimistic Search. A straightforward approach is to try alternative priority functions in the searching phase, since we already know other priority functions could outperform WA* in many cases.

There are other improvements, which will be addressed in detail below, that can be integrated into Optimistic Search, together creating a new algorithm, Improved

Algorithm 10 Improved Optimistic Search

```
1: procedure IMPROVED OPTIMISTIC SEARCH($start$, $goal$, $w$)
2:     Push($start$, $Open$)
3:     Push($start$, FOCAL)
4:     $I \leftarrow \varnothing$    [ $c(I) = \infty$) ]
5:     while $c(I)$ not $w$-admissible do
6:         if est. path length of best on FOCAL $< c(I)$ then
7:             Expand best from FOCAL
8:             if best == $goal$ then
9:                 $I \leftarrow path(best)$
10:            end if
11:        else
12:            Expand best from $Open$
13:            if child $s$ has shorter path to $s$ on FOCAL then
14:                // Choose one of the following policies:
15:                (a) Update cost of $s$ on FOCAL // Update
16:                (b) Re-open $s$ on FOCAL // Re-open
17:                if $s \in I$ then
18:                    (c) update cost of $I$ // Solution-update
19:                end if
20:            end if
21:        end if
22:    end while
23:    return $failure$
24: end procedure
```

Optimistic Search (IOS).

The pseudo-code for IOS is shown in Algorithm 10.

Despite the re-opening policies (lines 13-18), the approach is relatively simple. IOS expands the best state on FOCAL until an incumbent solution I is found (line 6), and stores the value $c(I)$, where $c(I)$ is the cost of the solution I. Then it expands the best state on *Open* until w-admissibility is proven (line 12). If re-openings are allowed, states re-opened on FOCAL in line 16 will then be re-expanded in line 6. Because some priority functions such as Φ_{XUP} do not directly estimate the length of the solution that will be found, line 6 explicitly uses the estimated path length.

Termination and Proving Bounds

Existing algorithms, such as Optimistic search and EES, use the minimum f-cost of a state on *Open*, $f_{\min}$, to prove the optimality of the solution. In particular, $f_{\min}$

on *Open* is a lower bound on the optimal solution cost. This holds because, with a consistent heuristic, the minimum f-cost in *Open* never decreases, and at the goal the f-cost is equivalent to the solution cost. Thus, a solution found in FOCAL with cost less than or equal to $w \cdot f_{\min}$ is guaranteed to be w-admissible. If the solution cannot immediately be proven to be optimal, states on *Open* are expanded until the bound on the solution quality is proven.

For example, if a solution is found with cost 100, and w is 1.5, those algorithms are required to expand the states until the minimum f in *Open* is raised to $100/1.5 = 66.67$.

While IOS uses this termination condition ($c(I) \leq wf_{\min}$), it can also use a second termination condition. It uses the maximum f-cost seen during the search, since we proved that value is a lower bound of the cost of an optimal path. As long as the maximum f-cost is found at a state other than the start state, this approach will be able to terminate faster.

Re-expansion Policies in IOS

Given the potential impact of re-expansions, several policies for dealing with re-expansions are studied. Because re-expansions never occur with a consistent heuristic in *Open*, there are only two contexts in which re-openings can occur: (1) When expanding states in FOCAL, a shorter path to a state in FOCAL is discovered. (2) When expanding states in *Open*, a shorter path to a state in FOCAL is discovered.

When a shorter path is found there are three simple policies that can be followed in either context, although more complex policies have been studied [72]. The *re-open* policy always moves states from *Closed* back to FOCAL when a shorter path is found. The *update* policy updates the g-cost and parent pointers of a state when a shorter path is found, but does not re-open the state by placing it back on FOCAL. The *ignore* policy ignores states that are already in *Closed*.

An additional policy can be used when a state from *Open* leads to a shorter path in FOCAL. This policy relies on the fact that IOS has an incumbent solution when expanding *Open*. If all states on the incumbent solution path are marked, the search will know when it has updated (reduced) the cost of the incumbent solution by reducing the g-cost of one of the states on the path. In this case, the cost of the incumbent solution can be reduced without re-expanding the path to update all g-costs. The search must simply take note of the reduction in g-cost, and reduce the stored incumbent solution cost by the same amount. This policy is called the *solution-update* policy. If shorter paths are found to several different states on the incumbent solution path, only the maximum improvement can be used for updating the incumbent solution cost. Thus, in Algorithm 10, one of the three policies following line 13 should be used.

4.5 Results and Discussion

In the following, we run experiments on a server with 16 GB RAM and 24 processors 6-core Intel Xeon CPU E5-2630 (2.30GHz). We tested on the 15-puzzle with unit edge costs and the heavy tile setting, where the cost of moving tile X is X. Manhattan Distance (MD) and PDB heuristics are used as heuristic for regular tiles, and the modified MD [87] is used for heavy tiles. The instances are the standard 100 Korf instances [46]. We also tested on 1098 problems with solution length $[128 - 132)$ from the Dragon Age: Origins benchmark set [81]. Further experiments evaluated performance on a heavy variant of the pancake puzzle [33], where the cost of flipping a prefix $(V[1] \cdots V[i+1])$ is $max(V[1]; V[i+1])$. We also use their $HGAP$ heuristic [33]. The problem set consists of 50 randomly generated 12-pancake instances. "-" means the algorithm is not able to solve all instances due to running out of memory.

Results on linear bounds in Table 4.3 show that Φ_{pwXD} and Φ_{z1} typically perform well. For instance, on the heavy pancake puzzle with $w = 1.5$, Φ_{pwXD} is 8x better than WA* and 2.48x better than Φ_{XDP}. Note that Φ_{pwXD} doesn't depend on w for states with $g < h$, after which it searches with weight of $2w - 1$. Thus, if the number of states with $g < h$ is small, then Φ_{pwXD} with w will perform similarly to

		Suboptimality Bound/ (w)			
Domain	Φ	1.5	2.0	3.0	10.0
15-Puzzle	Φ_{WA^*}	273,101	40,544	11,600	3,758
	Φ_{XDP}	166,447	21,338	7,550	3,586
	Φ_{XUP}	373,023	71,014	16,934	3,859
	Φ_{pwXD}	70,799	**11,230**	**4,978**	4,621
	Φ_{pwXU}	-	2,792,255	823,029	67,065
	Φ_{z1}	**43,009**	12,323	7,397	**3,321**
Heavy 15-Puzzle	Φ_{WA^*}	333,320	*114,848*	57,778	44,207
	Φ_{XDP}	200,318	82,295	**48,203**	43,141
	Φ_{XUP}	702,468	161,126	82,916	34,065
	Φ_{pwXD}	*101,498*	**52,386**	50,664	**30,438**
	Φ_{pwXU}	-	4,335,932	1,970,396	125,473
	Φ_{z1}	**95,674**	53,274	54,537	34,315
Heavy Pancake Puzzle	Φ_{WA^*}	8,498,635	*973,556*	22,732	33
	Φ_{XDP}	2,460,235	93,355	1,123	**20**
	Φ_{XUP}	-	7,072,634	404,622	318
	Φ_{pwXD}	***988,899***	**23,718**	**367**	29
	Φ_{pwXU}	51,245,052	-	11,244,399	24,377
	Φ_{z1}	3,323,705	535,430	98,353	1,087

Table 4.3: Average state expansions for each priority functions on different domains.

Φ_{WA^*} with weight $2w-1$. We italicize entries Φ_{XDP} with $w = 1.5$ and Φ_{WA^*} with $w = 1.5 * 2 - 1 = 2$ to highlight this in Table 4.3 on the heavy pancake puzzle and heavy 15-puzzle. The cost for the reduced node expansions is slightly worse solution quality. However, under the problem definition, the task is to find a solution within the bound, not to optimize solution quality.

Results on additive suboptimality bounds are presented in Table 4.10 for the sliding tile puzzle and Table 4.11 for grid maps. In our experiments, we tried 3 different K: $\gamma+1$, 2γ and $h_0 = max\{h(start), \gamma+1\}$, and compared against BFS^{F_γ} [89]. In most cases $K = h_0$ gives the best performance. On grid maps, the poor performance of BFS^{F_γ} is fully due to reopening states.

We also run experiments in FastDownward with the CEGAR heuristic [71] on plan-

		Suboptimality Bound/ (w)			
Domain	Φ	1.5	2.0	3.0	10.0
15-Puzzle	Φ_{WA^*}	56.37	63.71	77.99	118.87
	Φ_{XDP}	56.79	64.35	78.65	121.07
	Φ_{XUP}	55.87	63.73	77.69	121.99
	Φ_{pwXD}	66.83	75.23	89.33	119.57
Heavy	Φ_{WA^*}	442.34	509.10	619.62	930.06
15-Puzzle	Φ_{XDP}	454.28	516.78	620.98	933.08
	Φ_{XUP}	438.34	502.56	619.58	988.26
	Φ_{pwXD}	519.14	586.26	685.66	1038.12
Heavy	Φ_{WA^*}	81.18	82.78	88.94	105.16
Pancake	Φ_{XDP}	81.56	84.34	92.78	106.26
	Φ_{XUP}	80.21	82.08	86.70	102.30
	Φ_{pwXD}	82.22	86.36	95.74	104.90

Table 4.4: Average path costs for each priority functions on different domains.

		Suboptimality Bound/ (w)		
Domain	Φ	2.0	3.0	10.0
Data-network	Φ_{WA^*}	15	14	16
	Φ_{pwXD}	16	16	16
Spider	Φ_{WA^*}	13	17	20
	Φ_{pwXD}	18	18	18
Termes	Φ_{WA^*}	11	12	16
	Φ_{pwXD}	12	13	15
Total of	Φ_{WA^*}	69	72	77
all 8 domians	Φ_{pwXD}	74	79	74

Table 4.5: IPC Problems solved (each domain contains 20 problems).

ning domains from IPC 2018 problem sets. Table 4.5 shows the number of problems WA* and piecewise XD can solve; while Table 4.6 gives the average number of nodes expanded for solving the common set. These problems were run with a 15 minute timeout on a cluster with 2.1 GhZ Intel Xeon E5-2683 CPUs where each job was given 4 GB RAM.

In the final portion of this study, we look at the impact of the best previous enhancements with different priority functions for FOCAL that avoid node re-expansions. Results are presented across all domains. No algorithm performs re-openings, but so-

		Suboptimality Bound/ (w)		
Domain	Φ	2.0	3.0	10.0
Data-network	Φ_{WA^*}	302,558	996,910	827,652
	Φ_{pwXD}	328,049	668,416	524,842
Spider	Φ_{WA^*}	44,904	30,017	48,977
	Φ_{pwXD}	12,650	37,827	112,482
Termes	Φ_{WA^*}	21,742,548	18,605,174	12,429,415
	Φ_{pwXD}	13,941,669	6,327,308	7,023,560

Table 4.6: Average expansions on commonly solved problems

Table 4.7: Node reductions in IOS using the improved termination condition.

Bound	With Bound	No Bound	Gain
1.25	15,402	16,802	9.1%
1.50	10,797	11,219	3.9%
2.00	7,348	7,842	6.7%
3.00	5,425	5,750	6.0%

lution updating and the improved termination conditions are enabled.

The complete results are in Table 4.12. For each domain, three different priority functions are used along with 4 different weights. All approaches are able to solve all problems except in the Heavy Pancake puzzle, where some algorithms could not solve problems with lower weights.

IOS with Φ_{XDP} has the best performance in 11 of the 15 weight/domain combi-

Table 4.8: Node reductions in IOS using solution updating.

Bound	Updating	No Updating	Gain
1.25	11,423	15,402	34.8%
1.50	8,715	10,797	23.9%
2.00	6,573	7,348	11.8%
3.00	5,386	5,425	0.7%

	DAO Grids			15 Puzzle			Heavy 15-Puzzle			Heavy Pancake		
Bound	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}
1.25	**10,940**	11,423	11,695	**264,112**	350,976	521,722	668,032	**397,710**	741,031	-	-	-
1.50	**8,533**	8,715	8,573	**35,280**	47,659	76,756	427,210	**152,310**	174,369	**211,783**	1,074,523	-
2.00	6,829	6,573	**6,266**	**12,050**	14,240	17,397	**64,128**	71,672	85,007	**3,412**	23,567	405,185
3.00	5,818	5,386	**5,060**	**4,468**	4,633	6,627	**48,226**	52,868	48,949	**63**	382	10,342

Table 4.9: Average performance for IOS with WA*, XDP and XUP as priority functions for Focal.

		Additive Suboptimality Bound/ (γ)				
Algorithm	Parameter	0	4	16	64	256
BFS^{F_γ}		12,325	7,494	**878**	**356**	235
Φ_{AB}	$K = \gamma + 1$	12,325	12,265	7,186	446	**204**
Φ_{AB}	$K = 2\gamma$	12,325	12,010	1,972	399	399
Φ_{AB}	$K = h_0$	12,325	**6,273**	1,160	446	**204**

Table 4.10: Average state expansions for algorithm on STP for additive bound ($h_0 = max\{h(start), \gamma + 1\}$.)

nations tested. The only exception is the Heavy 15-puzzle, where there are a few problems where Φ_{XDP} has very poor performance. We are continuing to study this domain to better understand the performance here.

4.6 Conclusions

In this chapter, we show that our research on suboptimal search studies the following question: (1) The impact of reopenings in suboptimal search; (2) the general properties of Φ which guarantee BFS-NR will find bounded suboptimal solutions, providing

		Additive Suboptimality Bound/ (γ)				
Algorithm	Parameter	0	4	16	64	256
BFS^{F_γ}		1,240	1,182	1,219	986	1,078
Φ_{AB}	$K = \gamma + 1$	1,240	1,237	1,169	711	**368**
Φ_{AB}	$K = 2\gamma$	1,240	1,233	1,107	**485**	481
Φ_{AB}	$K = h_0$	1,240	**1,137**	**929**	518	**368**

Table 4.11: Average state expansions for algorithm on DAO for additive bound ($h_0 = max\{h(start), \gamma + 1\}$.)

	DAO Grids			15 Puzzle			Heavy 15-Puzzle			Heavy Pancake		
Bound	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}	Φ_{XDP}	WA*(w_f)	Φ_{XUP}
1.25	**10,940**	11,423	11,695	**264,112**	350,976	521,722	668,032	**397,710**	741,031	-	-	-
1.50	**8,533**	8,715	8,573	**35,280**	47,659	76,756	427,210	**152,310**	174,369	**211,783**	1,074,523	-
2.00	6,829	6,573	**6,266**	**12,050**	14,240	17,397	**64,128**	71,672	85,007	**3,412**	23,567	405,185
3.00	5,818	5,386	**5,060**	**4,468**	4,633	6,627	**48,226**	52,868	48,949	**63**	382	10,342

Table 4.12: Average performance for IOS with WA*, XDP and XUP as priority functions for FOCAL.

necessary and sufficient conditions on Φ; (3) specific choices for Φ that improve the performance in practice.

A quick summary is that the priority functions generated from our framework show improvements over existing methods for both linear and constant suboptimality bounds. Two priority functions stand out, and we write them in the terms of f, g, and h below:

For additive bounds one simple function is

$$f_{AB} = \begin{cases} h + \frac{h_0 - \gamma}{h_0} g & g < h_0 \\ h + g - \gamma & g \geq h_0 \end{cases}$$

where $h_0 = max\{h(start), \gamma + 1\}$.

For a linear bounding function, the overall best performance was found with

$$f_{pwXD} = \begin{cases} g + h & g < h \\ \frac{1}{w}(g + (2w - 1)h) & g \geq h \end{cases}$$

However, there are many other possible Φ functions. There is significant potential to study these Φ functions in the context of algorithms such as ARA* [50] which use suboptimal search within a broader search framework.

The major limitation of this work is the dependency on consistent heuristics. Our theory shows that in general, if the heuristic is inconsistent, in an arbitrary way, then the solution might be unbounded; however, if the inconsistency of a heuristic is bounded or can be measured in certain way, then it might be still possible to create a bounded algorithm.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The thesis presents the following contributions to heuristic search:

Contribution 1: Establishes sufficient conditions for states that must be expanded by all front-to-end bidirectional algorithms, including unidirectional search algorithms, to guarantee an optimal solution:

Let $I = (G, start, goal, h_F, h_B) \in I_{AD}$ have an optimal solution cost of C^*. If U is an optimal forward path and V is an optimal backward path such that $U_0 = start$, $V_0 = goal$, and:

$lb(U, V) < C^*$

where $lb(U, V) = \max\{f_F(U), f_B(V), c(U) + c(V)\}$,

when solving problem instance I, any admissible DXBB bidirectional front-to-end search algorithm must expand at least one of the following states: $\{end(U), end(V)\}$. The pair $\{end(U), end(V)\}$ is called a surely-expanded pair in front-to-end bidirectional search.

Such conditions are a generalization of Dechter & Pearl's results on unidirectional search that all states n with $g^*(n) + h(n) < C^*$ must be expanded by unidirectional algorithms (Theorem 8 [20]). Sufficient conditions are fundamental theory which make it possible for analysis of the minimum work to guarantee the optimality of an algorithm.

Contribution 2: Based on the sufficient condition, this thesis shows the key to minimum possible work by analyzing a special bipartite graph, G_{MX}. Unlike unidirectional search, where the sufficient condition directly implies the minimum set of states that must be expanded by all algorithms (the surely expanded states form a unique set), the bidirectional search sufficient condition does not imply such a minimum set of states (two algorithm can expand different sets of states while both expand all surely expanded pairs).

This thesis makes the following observation: the sufficient condition is a binary relation over set S, i.e., a subset of $S \times S$, where S is the set of all states in the state space. Such a binary relation can be represented using the bipartite graph G_{MX}. If an algorithm expands all the surely expanded pairs, then the states it expands must be a vertex cover of G_{MX}. Thus, we have a corollary that the number of state expansions in the search problem is greater or equal to the size of the minimum vertex cover in G_{MX}. Other researchers proved that this bound is tight, i.e., the minimum number of states expansion is strictly equal to the size of the minimum vertex cover in G_{MX} [73].

The link from heuristic search to vertex cover provides a revolutionary point of view for doing bidirectional search. This link (i) allows us to build the theory of when bidirectional search can be beneficial (for example, if the minimum vertex cover is bidirectional, then it is likely that bidirectional search algorithm can do better than A^*) (ii) inspire the invention of bidirectional algorithms that try to minimize the vertex cover, such as `NBS` and `DVCBS`.

Contribution 3: Provides the proof that if a DXBB bidirectional admissible algorithm that has a guaranteed bound λ in state expansions, $\lambda|VC|$, where $|VC|$ is the size of the minimum vertex cover of G_{MX}, then $\lambda \geq 2$. i.e., no algorithm can guarantee that it can always find an optimal solution with $< 2|VC|$ state expansions. Note that A^*, as a special case of a front-to-end search algorithm which only does forward expansions, does not have any guarantees for state expansions – it can perform arbitrarily bad on a bidirectional search problem instance.

Contribution 4: Presents a front-to-end bidirectional search algorithm `NBS`, which returns an optimal solution with a guaranteed bound in node expansions. The guaranteed bound of `NBS` is 2x, which is the best one can achieve.

Contribution 5: Shows analysis of the impact of reopenings in unidirectional suboptimal search. Weighted A* can do exponentially worse when there is no assumption about edge costs. When there are assumptions about edge costs, for example, a very limited number of different edge costs, then reopenings can be polynomially worse, which happens a lot in practice, especially in grid maps. In theory, reopenings can result in unbounded improvements in node expansions.

Contribution 6: Present a novel framework to do suboptimal search: best-first search using a Φ function with a never-reopening policy (BFS-NR with Φ). BFS-NR with Φ is a generalization of Weighted A*. As long as the Φ function satisfies a few conditions, BFS-NR with Φ returns a bounded suboptimal solution. The conditions are presented and proved to be sufficient and necessary in this thesis. Also, several Φ functions, including $pwXD$, are proposed in this document. Empirical evaluation shows that they can outperform previous algorithms.

Contribution 7: Suggest improvements that can be made to Optimistic Search algorithms. Based on those improvements, we develop an new algorithm, Improved Optimistic Search (IOS).

5.2 Future Work

The major limitation of the work presented in this thesis in bidirectional search is that although the guarantee of `NBS` is the best possible, in practice `NBS` does not outperform A* in many domains.

There are many open questions in bidirectional search.

(1) Some studies try to predict when bidirectional search algorithms should be used [85]. However, those studies are closer to a posterior analysis. It is still an open

question whether we can have an algorithm or a portfolio of algorithms that apply to a general graph and do the heuristic search efficiently.

(2) Our theory shows that bidirectional search can be modeled as a vertex cover problem. The tricky part is that the graph, G_{MX} is not known a priori; it is built during the search. Thus, it is worthwhile looking at the state-of-the-art online vertex cover solvers and seeing if any of them can be adapted to search algorithms.

(3) External memory bidirectional search, more specifically, an external memory version of `NBS` or `DVCBS`. Using external memory enables an algorithm to solve harder problems where the open lists can not fit in RAM. The external memory version of the `MM` algoithm, PEMM, is better than external A^* in a few cases [82]. Since `NBS` in general, is better than `MM`, it would be interesting to see if an external memory version of `NBS` can be even more effiecient.

(4) Effective front-to-end heuristics for bidirectional search. In this thesis, the results indicate that when the heuristics are strong, A^* tends to be better. Later studies show that when the heuristics are strong, the vertex cover is unidirectional [85]. In some sense, current front-to-end heuristics do not help bidirectional search as much as they help unidirectional search. Are there techniques that could be beneficial in building front-to-end bidirectional heuristics? Studies indicate that the value compression technique [84] works well with `MM`, but it remains unclear whether it is suitable for general bidirectional algorithms.

(5) Besides all the open questions for front-to-end bidirectional search, there are some related open questions for front-to-front bidirectional search coming from the theory of this thesis. Since there is a similar form of sufficient condition for state expansions in front-to-front bidirectional search, the vertex cover reduction in front-to-end search also applies to front-to-front search. The major challenge for front-to-front search is to find the pair with a minimum lb efficiently. Front-to-end algorithms, such as `NBS`, can reduce the time of selecting a pair to amortized $O(\log n)$, where n is the size of the open lists. In contrast, the worst case for front-to-front pair selection

is $O(n^2)$. However, it is costly in general case does not mean that it is always costly. There could be special cases in front-to-front search where the pair selection can be made efficiently. For example, it looks promising when we know the front-to-front heuristic is created by front-to-end heuristics.

Initially, the study of suboptimal search was trying to answer the following question: can we define a maximal set of states that all algorithms must expand? One trivial set is, when we are looking for solutions that are not more than wC^*, we have to expand all the states n with $d(n) + h(n) < C^*/w$. However, in practice, this set is very small and even can be negligible when w is relatively large, while the number of states expanded to solve the problem could be a few magnitudes larger. i.e., there is a huge gap between the current necessary work vs. the actual work.

One open question follows from that is, is the set of all the states n with $d(n) + h(n) < C^*/w$ the maximal necessary expansions? We could not either prove it to be maximal or give a larger set since suboptimal search algorithms are a huge family. The behaviors of the algorithms can differ widely. For example, some of them only expand states that are provably good, while some others try to find a solution regardless of its quality and then refine it; some of them reopen states, while others do not. The diversity makes it difficult for the analysis of the necessary expansions that all algorithms must do.

When we realized that it is hard to draw conclusions for all algorithms, we started to focus on a specific family of algorithms, BFS-NR. We find the conditions that allow this family of algorithms to return B-admissible solutions, where B is a given bounding function. Nevertheless, we also leave a lot of open questions:

(1) Φ for general bounding functions. In this thesis, we present Φ functions for additive and linear bounding functions. There could be other bounding functions, such as $B(x) = \lambda x \log x$, where λ is a constant. Such a bounding function allows the solution quality to be worse as the C^* grows. The question will be if there is an elegant way of representing the formula of Φ for such bounding functions. If we try

to solve Φ using the method provided in this thesis, the analytical solution can be very complex, or there is no analytical solution.

(2) How to find good parameters for Φ functions? As presented in the thesis, a few free parameters can be tuned in a Φ function. We do not have a systematic way of picking parameters; we choose to run a few samples, picking the best parameters and then fixing them during the remaining experiments. It is possible that a more advanced parameter-picking strategy will further improve BFS-NR with Φ.

(3) How can extra information improve BFS-NR? When we do the analysis, we assume the algorithm prioritizes the states only according to its g-cost and h-cost. However, we could have extra information in practice, and it does not make sense to throw it away and pretend it does not exist. For example, the difference between the h-cost of the current state and start state; an inadmissible estimate, such as the distance-to-go estimate used in EES or the backward h-cost. It is an open question of how to design BFS-NR to use such information.

(4) Minimum work among BFS-NR. Since BFS-NR is a family of algorithms, an interesting question is the theoretical minimum expansions among all possible BFS-NR algorithms. Unfortunately, even though we limit ourselves to a particular family of algorithms, the analysis of necessary work is still non-trivial.

(5) Analysis of Focal-NR. We know that in general, if we use a focal search algorithm and never reopen states, the solution is not B-admissible. Are such solutions not bounded at all, or are they still bounded in some way? If we have a concrete conclusion for the bound of the solution returned by Focal-NR, we would be able to set our mind at rest in many cases when we receive a solution from Focal-NR.

Bibliography

[1] M. Asai and A. Fukunaga, "Tie-breaking strategies for cost-optimal best first search," *Journal of Artificial Intelligence Research*, vol. 58, pp. 67–121, 2017.

[2] A. Athar, A. M. Zafar, R. Asif, A. A. Khan, F. Islam, O. Hasan, *et al.*, "Whole-body motion planning for humanoid robots with heuristic search," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2016, pp. 4720–4727.

[3] M. H. Baaj and H. S. Mahmassani, "An ai-based approach for transit route system planning and design," *Journal of advanced transportation*, vol. 25, no. 2, pp. 187–209, 1991.

[4] J. Barker and R. Korf, "Limitations of front-to-end bidirectional heuristic search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[5] J. K. Barker and R. E. Korf, "Limitations of front-to-end bidirectional heuristic search," in *Proc. 29th AAAI Conference on Artificial Intelligence*, 2015, pp. 1086–1092.

[6] M. Barley, P. Riddle, C. L. López, S. Dobson, and I. Pohl, "Gbfhs: A generalized breadth-first heuristic search algorithm," in *Eleventh Annual Symposium on Combinatorial Search*, 2018.

[7] S. Bate and K. Stanley, "Heuristic route planning: An application to fighter aircraft," in *Proceedings of the IEEE 1988 National Aerospace and Electronics Conference*, IEEE, 1988, pp. 1114–1120.

[8] J Benton, M. Do, and W. Ruml, "A simple testbed for on-line planning," in *Proceedings of the ICAPS Workshop on Moving Planning and Scheduling Systems into the Real World*, Citeseer, 2007.

[9] V. Bulitko, Y. Björnsson, N. R. Sturtevant, and R. Lawrence, "Real-time heuristic search for pathfinding in video games," in *Artificial Intelligence for Computer Games*, Springer, 2011, pp. 1–30.

[10] D. de Champeaux, "Bidirectional heuristic search again," *J. ACM*, vol. 30, no. 1, pp. 22–32, 1983.

[11] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo tree search: A new framework for game AI.," *AIIDE*, vol. 8, pp. 216–217, 2008.

[12] C. Chen, M. Rickert, and A. Knoll, "Kinodynamic motion planning with space-time exploration guided heuristic search for car-like robots in dynamic environments," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, pp. 2666–2671.

[13] J. Chen, R. C. Holte, S. Zilles, and N. R. Sturtevant, "Front-to-end bidirectional heuristic search with near-optimal node expansions," *International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[14] J. Chen and N. R. Sturtevant, "Conditions for avoiding node re-expansions in bounded suboptimal search," *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[15] J. Chen and N. R. Sturtevant, "Necessary and sufficient conditions for avoiding reopenings in best first suboptimal search with general bounding functions," *AAAI Conference on Artificial Intelligence*, 2021.

[16] J. Chen, N. R. Sturtevant, W. Doyle, and W. Ruml, "Revisiting suboptimal search," *Symposium on Combinatorial Search (SoCS)*, pp. 18–25, 2019.

[17] J. C. Culberson and J. Schaeffer, "Searching with pattern databases," in *Conference of the Canadian Society for Computational Studies of Intelligence*, Springer, 1996, pp. 402–416.

[18] D. Dahlmeier and H. T. Ng, "A beam-search decoder for grammatical error correction," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012, pp. 568–578.

[19] D. De Champeaux, "Bidirectional heuristic search again," *Journal of the ACM (JACM)*, vol. 30, no. 1, pp. 22–32, 1983.

[20] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.

[21] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[22] J. E. Doran, "Double tree searching and the Graph Traverser," Dept. of Machine Intelligence and Perception, University of Edinburgh, Tech. Rep. Research Memo EPU-R-22, 1966.

[23] R. Ebendt and R. Drechsler, "Weighted A* search–unifying view and application," *Artificial Intelligence*, vol. 173, no. 14, pp. 1310–1342, 2009.

[24] J. Eckerle, J. Chen, N. Sturtevant, S. Zilles, and R. Holte, "Sufficient conditions for node expansion in bidirectional heuristic search," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

[25] J. Eckerle and T. Ottmann, "An efficient data structure for bidirectional heuristic search," in *ECAI*, 1994, pp. 600–604.

[26] N. Eén and N. Sörensson, "An extensible SAT-solver," in *International conference on theory and applications of satisfiability testing*, Springer, 2003, pp. 502–518.

[27] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang, "Inconsistent heuristics in theory and practice," *Artificial Intelligence (AIJ)*, vol. 175, no. 9-10, pp. 1570–1603, 2011.

[28] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *Journal of Artificial Intelligence Research*, vol. 22, pp. 279–318, 2004.

[29] A. Felner, C. Moldenhauer, N. R. Sturtevant, and J. Schaeffer, "Single-frontier bidirectional search," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[30] G. Fertin, A. Labarre, I. Rusu, S. Vialette, and E. Tannier, *Combinatorics of genome rearrangements*. MIT press, 2009.

[31] C. A. Floudas and X. Lin, "Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications," *Annals of Operations Research*, vol. 139, no. 1, pp. 131–162, 2005.

[32] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," *ACL 2017*, p. 56, 2017.

[33] D. Gilon, A. Felner, and R. Stern, "Dynamic potential search—a new bounded suboptimal search," in *Symposium on Combinatorial Search (SoCS)*, 2016, pp. 36–44.

[34] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory.," in *SODA*, Citeseer, vol. 5, 2005, pp. 156–165.

[35] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[36] M. Helmert, "Landmark heuristics for the pancake problem," in *Third Annual Symposium on Combinatorial Search*, 2010.

[37] M. Helmert, "Landmark heuristics for the pancake problem," in *Proc. 3rd Annual Symposium on Combinatorial Search, (SoCS)*, 2010.

[38] H. Hiraishi, H. Ohwada, and F. Mizoguchi, "Time-constrained heuristic search for practical route finding," in *Pacific Rim International Conference on Artificial Intelligence*, Springer, 1998, pp. 389–398.

[39] J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[40] R. C. Holte, A. Felner, G. Sharon, N. R. Sturtevant, and J. Chen, "MM: A bidirectional search algorithm that is guaranteed to meet in the middle," *Artificial Intelligence*, vol. 252, pp. 232–266, 2017.

[41] R. C. Holte, R. Majadas, A. Pozanco, and D. Borrajo, "Error analysis and correction for weighted A*'s suboptimality.," in *SoCS*, 2019, pp. 135–139.

[42] R. C. Holte and S. Zilles, "On the optimal efficiency of cost-algebraic A*," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 2288–2295.

[43] Horace.wk.chan, *Huarongdao*, https://commons.wikimedia.org/ wiki/File:HuaRongDao.jpg, 2012.

[44] H. Kaindl and G. Kainz, "Bidirectional heuristic search reconsidered," *J. Artificial Intelligence Resesearch (JAIR)*, vol. 7, pp. 283–317, 1997.

[45] H. Kaindl, G. Kainz, R. Steiner, A. Auer, and K. Radda, "Switching from bidirectional to unidirectional search," in *IJCAI*, 1999, pp. 1178–1183.

[46] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[47] R. E. Korf, "Finding optimal solutions to rubik's cube using pattern databases," in *AAAI/IAAI*, 1997, pp. 700–705.

[48] J. B. H. Kwa, "BS*: An admissible bidirectional staged heuristic search algorithm," *Artificial Intelligence*, vol. 38, no. 1, pp. 95–109, 1989.

[49] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[50] M. Likhachev, G. J. Gordon, and S. Thrun, "ARA*: Anytime A* with provable bounds on sub-optimality," *Advances in neural information processing systems*, vol. 16, pp. 767–774, 2003.

[51] M. Lippi, M. Ernandes, and A. Felner, "Efficient single frontier bidirectional search," in *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SoCS*, 2012.

[52] M. Madrid and A. Badan, "Heuristic search method for continuous-path tracking optimization on high-performance industrial robots," *Control Engineering Practice*, vol. 5, no. 9, pp. 1261–1271, 1997.

[53] G. Manzini, "BIDA*: an improved perimeter search algorithm," *Artificial Intelligence*, vol. 75, no. 2, pp. 347–360, 1995.

[54] A. Martelli, "On the complexity of admissible search algorithms," *Artificial Intelligence*, vol. 8, no. 1, pp. 1–13, 1977.

[55] L. Mérő, "A heuristic search algorithm with modifiable estimate," *Artificial Intelligence*, vol. 23, pp. 13–27, 1984.

[56] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 530–535.

[57] T. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, IEEE, vol. 1, 2002, pp. 170–179.

[58] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *The computer journal*, vol. 9, no. 3, pp. 275–280, 1966.

[59] N. J. Nilsson, *Principles of Artificial Intelligence.* Springer, 1982, ISBN: 978-3-540-11340-9.

[60] D. Ostrowski, I. Pozniak-Koszalka, L. Koszalka, and A. Kasprzak, "Comparative analysis of the algorithms for pathfinding in GPS systems," *ICN 2015*, p. 114, 2015.

[61] C. H. Papadimitriou and K. Steiglitz, "Combinatorial optimization: Algorithms and complexity," 1982.

[62] J. Pearl, "Heuristics: Intelligent search strategies for computer problem solving," 1984.

[63] J. Pearl and J. H. Kim, "Studies in semi-admissible heuristics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 4, pp. 392–399, Jul. 1982, ISSN: 0162-8828. DOI: 10.1109/TPAMI.1982.4767270.

[64] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[65] I. Pohl, "Bi-directional search," in *Machine Intelligence*, B. Meltzer and D. Michie, Eds., vol. 6, Edinburgh University Press, 1971, pp. 127–140.

[66] I. Pohl, "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving," in *Proceedings of the 3rd international joint conference on Artificial intelligence*, Morgan Kaufmann Publishers Inc., 1973, pp. 12–17.

[67] I. Rahayuda and N. Santiari, "Dijkstra and bidirectional Dijkstra on determining evacuation routes," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1803, 2021, p. 012 018.

[68] R. Robotin, G. Lazea, and P. Dobra, "Mobile robots path planning with heuristic search," *Journal of Control Engineering and Applied Informatics*, vol. 12, no. 4, pp. 18–23, 2010.

[69] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall, 2009, ISBN: 0136042597, 9780136042594.

[70] A. Schrijver, *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[71] J. Seipp and M. Helmert, "Diverse and additive cartesian abstraction heuristics," in *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[72] V. Sepetnitsky, A. Felner, and R. Stern, "Repair policies for not reopening nodes in different search settings," in *Symposium on Combinatorial Search (SoCS)*, 2016, pp. 81–88.

[73] E. Shaham, A. Felner, J. Chen, and N. R. Sturtevant, "The minimal set of states that must be expanded in a front-to-end bidirectional search.," in *SoCS*, 2017, pp. 82–90.

[74] E. Shaham, A. Felner, N. R. Sturtevant, and J. S. Rosenschein, "Minimizing node expansions in bidirectional search with consistent heuristics," *Symposium on Combinatorial Search (SoCS)*, pp. 81–89, 2018.

[75] G. Sharon, R. C. Holte, A. Felner, and N. R. Sturtevant, "Extended abstract: An improved priority function for bidirectional heuristic search," *Symposium on Combinatorial Search (SoCS)*, pp. 139–140, 2016.

[76] S. Shperberg, S. Danishevski, A. Felner, and N. R. Sturtevant, "Iterative-deepening bidirectional heuristic search with restricted memory," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2021.

[77] S. Shperberg and A. Felner, "On the differences and similarities of fmm and gbfhs," in *International Symposium on Combinatorial Search*, vol. 11, 2020.

[78] S. Shperberg, A. Felner, N. R. Sturtevant, A. Hayoun, and E. S. Shimony, "Enriching non-parametric bidirectional search algorithms," in *AAAI Conference on Artificial Intelligence*, 2019, pp. 2379–2386.

[79] L. Sint and D. de Champeaux, "An improved bidirectional heuristic search algorithm," *Journal of the ACM (JACM)*, vol. 24, no. 2, pp. 177–191, 1977.

[80] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, and K. Goldberg, "Potential-based bounded-cost search and anytime non-parametric A*," *Artificial Intelligence*, vol. 214, pp. 1–25, 2014.

[81] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 –148, 2012.

[82] N. Sturtevant and J. Chen, "External memory bidirectional search," *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 676–682, 2016.

[83] N. Sturtevant, A. Felner, and M. Helmert, "Exploiting the rubik's cube 12-edge pdb by combining partial pattern databases and bloom filters," in *International Symposium on Combinatorial Search*, vol. 5, 2014.

[84] N. Sturtevant, A. Felner, and M. Helmert, "Value compression of pattern databases," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[85] N. R. Sturtevant, S. Shperberg, A. Felner, and J. Chen, "Predicting the effectiveness of bidirectional heuristic search," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 281–290.

[86] J. T. Thayer and W. Ruml, "Faster than weighted A*: An optimistic approach to bounded suboptimal search.," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2008, pp. 355–362.

[87] J. T. Thayer and W. Ruml, "Bounded suboptimal search: A direct approach using inadmissible estimates," in *International Joint Conference on Artifical Intelligence (IJCAI)*, 2011, pp. 674–679.

[88] G. Vaira and O. Kurasova, “Parallel bidirectional Dijkstra’s shortest path algorithm,” *Databases and Information Systems VI, Frontiers in Artificial Intelligence and Applications*, vol. 224, pp. 422–435, 2011.

[89] R. Valenzano, S. J. Arfaee, R. Stern, J. Thayer, and N. Sturtevant, “Using alternative suboptimality bounds in heuristic search,” in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2013, pp. 233–241.

[90] R. Valenzano, N. Sturtevant, and J. Schaeffer, “Worst-case solution quality analysis when not re-expanding nodes in best-first search,” in *AAAI Conference on Artificial Intelligence*, 2014, pp. 885–892.

[91] R. Valenzano, N. Sturtevant, and J. Schaeffer, “Worst-case solution quality analysis when not re-expanding nodes in best-first search,” in *AAAI Conference on Artificial Intelligence*, 2014, pp. 885–892.

[92] R. Valenzano and F. Xie, “On the completeness of best-first search variants that use random exploration,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[93] P. Wang and H. T. Ng, “A beam-search decoder for normalization of social media text with application to machine translation,” in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2013, pp. 471–481.

[94] Z. Zhang, N. R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner, “A* search with inconsistent heuristics,” in *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.