University of Alberta

Additive Abstraction-based Heuristics

by

Fan Yang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Spring 2011
Edmonton, Alberta

Examining Committee

Joseph Culberson, Department of Computing Science

Robert Holte, Department of Computing Science

Jonathan Schaeffer, Department of Computing Science

Michael Buro, Department of Computing Science

Terry Gannon, Department of Mathematical Sciences

Eric Hansen, Department of Computer Science and Engineering, Mississippi State University

I dedicate this work to my loving parents who passed on a love of reading and respect for knowledge.

Abstract

In this thesis, we study theoretically and empirically the additive abstraction-based heuristics. First we present formal general definitions for abstractions that extend to general additive abstractions. We show that the general definition makes proofs of admissibility, consistency, and additivity easier, by proving that several previous methods for defining abstractions and additivity satisfy three simple conditions. Then we investigate two general methods for defining additive abstractions and run experiments to determine the effectiveness of these methods for two benchmark state spaces: TopSpin and the Pancake puzzle. Third, we propose that the accuracy of the heuristics generated by abstraction can be improved by checking for infeasibility. The theory and experiments demonstrate the approach to detect infeasibility and the application of this technique to different domains. Finally, we explore the applications of additive abstraction-based heuristics in two state spaces with non-uniform edge costs: the Sequential Ordering Problem (SOP) and the weighted Pancake puzzle. We formalize a novel way of generating additive and non-additive heuristics for these state spaces. Furthermore, we investigate the key concepts to generate good additive and non-additive abstractions. Experiments show that compared to some simple alternative heuristics, well chosen abstractions can enhance the quality of suboptimal solutions for large SOP instances and reduce search time for the weighted Pancake problems.

Acknowledgements

First of all, I would like to thank my supervisors, Dr. Joseph Culberson and Dr. Robert Holte, for their advice, inspiration, and encouragement over the years. I thank their patience while I was at a loss in my first two years as an international student.

I also want to thank my other committee members, Dr. Jonathan Schaeffer, Dr. Michael Buro, Dr. Terry Gannon and Dr. Eric Hansen for reading my proposal and thesis draft. Their suggestions and criticisms during my candidacy exam are an invaluable source of inspiration during the preparation of this thesis.

I am grateful to many people in this department, in particularly to Dr. Lorna Stewart, Dr. Ehab Elmallah, Dr. Ken Wong, Dr Li-Yan Yuan, Ms. Edith Drummond, Neil Burch, Steve Sutphen and Nathan Sturtevant. Studying and working with them gives me an opportunity to know a variety of research fields and research perspectives which otherwise will take forever for me to learn by myself. Thanks are also devoted to Dr. Yong Gao of UBC-Okanagan who kindly provided me encouragement and helpful advices.

Finally, I appreciate my loving family and parents for their patience and support through these years.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

In its purest form, single-agent heuristic search is concerned with the problem of finding a least-cost path between two states (*start* and *goal*) in a state space given a heuristic function. Throughout the thesis the heuristic function $h(t, g)$ specifically refers to the estimation of the cost to reach the state g from any state t, although in some literature the heuristic can be defined by other methods. Standard algorithms for single-agent heuristic search such as *IDA** [65] are guaranteed to find optimal paths if $h(t, g)$ is *admissible*, i.e., never overestimates the actual cost to the goal state from t, and their efficiency is heavily influenced by the accuracy of $h(t, g)$. Considerable research has therefore investigated methods for defining accurate, admissible heuristics.

A common method for defining admissible heuristics, which has led to major advances in solving combinatorial problems [16, 66, 69] and planning applications [22], is to "abstract" the original state space to create a new, smaller state space with the key property that for each path $\vec{p}$ in the original space there is a corresponding abstract path whose cost does not exceed the cost of $\vec{p}$. Given an abstraction, $h(t, g)$ can be defined as the cost of the least-cost abstract path from the abstract state corresponding to t to the abstract state corresponding to g. The best heuristic functions defined by abstraction are typically based on using several abstractions.

Given several abstractions of a state space, the heuristic $h_{max}(t, g)$ can be defined as the maximum of the abstract distances for t given by the abstractions individually. This is the standard method for defining a heuristic function given multiple abstractions [50].

The sum of the costs returned by a set of abstractions is not always admissible. If it is, the set of abstractions is said to be *additive*. One general method defined in [26, 68, 69] creates a set of k additive abstractions for the sliding tile puzzle by partitioning the tiles into k disjoint groups and defining one abstraction for each group by making the tiles in that group distinguished in the abstraction. An important limitation of this and most other existing methods of defining additive abstractions is that they do not apply to spaces in which an operator can move more than one tile at a time, unless there is a way to guarantee that all the tiles that are moved by the operator are in the

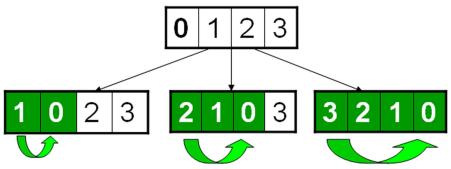


Figure 1.1: In the 4-Pancake puzzle, each state has three successors.

same group.

An example of a state space that has no additive abstractions according to previous definitions is the Pancake puzzle. In the N-Pancake puzzle, a state is a permutation of N tiles $(0, 1, ..., N-1)$ and has $N-1$ successors, with the l^{th} successor formed by reversing the order of the first $l+1$ positions of the permutation $(1 \leq l \leq N-1)$. For example, in the 4-Pancake puzzle shown in Figure 3.4, the state at the top of the figure has three successors, which are formed by reversing the order of the first two tiles, the first three tiles, and all four tiles, respectively. Because the operators move more than one tile and any tile can appear in any location there is no non-trivial way to partition the tiles so that all the tiles moved by an operator are distinguished in just one abstraction.

Other common state spaces that have no additive abstractions according to previous definitions are TopSpin and the Sequential Ordering Problem (SOP).

The SOP is a problem of searching with non-uniform edge costs, and it is a model for several industrial applications, such as the stacker crane application [1] and helicopter routing between oil rigs [85]. Given a graph G with n vertices and directed weighted edges with the start and goal vertices designated, the goal of an SOP instance is to find a minimal cost Hamiltonian path from the start vertex to the goal vertex without violating any constraint. The SOP is a variant of the Asymmetric Traveling Salesman Problem (ATSP) with some precedence constraints. However, a number of ATSP instances with hundreds of vertices can be solved easily, while there are several SOP instances with less than 55 vertices which have not been solved optimally.

As problems scale up, one of the main issues being addressed in the study of heuristic search is to enhance the accuracy of heuristics to speed up the search. Given additional memory, it is also interesting to explore new techniques to enhance the search performance.

1.2 Contributions

This thesis addresses the general definition and applications of additive abstractions. The main contributions of the thesis are summarized as follows.

1. The first contribution of this thesis is to identify general conditions for abstractions to be additive. The new conditions subsume most previous notions of "additive" as special cases. The greater generality allows additive abstractions to be defined for state spaces that had no additive abstractions according to previous definitions, such as TopSpin, the Pancake puzzle, and

related real-world problems such as the genome rearrangement problem [24]. Our definitions are fully formal, enabling rigorous proofs of the admissibility and consistency of the heuristics defined by our abstractions. The importance of our contribution is that it should make future proofs of admissibility, consistency, and additivity easier, because one will only need to show that a particular method for defining abstractions satisfies only three simple conditions.

2. The usefulness of our general definitions is demonstrated experimentally by defining additive abstractions that substantially reduce the CPU time needed to solve TopSpin and the Pancake puzzle. For example, the use of additive abstractions allows the 17-Pancake puzzle to be solved three orders of magnitude faster than previous state-of-the-art methods.

3. We study a technique to increase the abstraction-based heuristic values in some circumstances. The key to the approach is to identify "infeasible" values that cannot possibly be the optimal solution cost. Once identified the infeasible values can be increased to give a better estimate of the solution cost. This approach roughly halved the search time for some domains.

4. The final contribution of this thesis is to introduce methods of defining additive and non-additive abstractions to problems with non-unit edge costs. A novel way of generating additive heuristics is introduced and evaluated for two state spaces with non-uniform edge costs: the SOP and the weighted Pancake puzzle. New indexing schemes for ranking and unranking are designed for storing and obtaining heuristics more efficiently in the state space of the SOP. We analyse and identify some special structural types of problem instances and investigate the design of good abstractions for instances with special properties. Experiments show that well chosen abstractions can enhance the quality of sub-optimal solutions for large SOP instances and reduce search time for the weighted Pancake problems.

1.3 Thesis Outline

Below is a brief summary of the remainder of this thesis.

- Chapter 2: First we describe the state space represented by a weighted directed graph consisting of a set of nodes and edges. Then we briefly introduce and discuss some fundamental search algorithms and their variants. These algorithms differ in the ways they prioritize node expansion, deal with duplicate nodes and reconstruct the solution path.

- Chapter 3: We present formal general definitions for abstractions that extend to general additive abstractions. We provide lemmas proving the admissibility and consistency of both standard and additive heuristics based on these abstractions. This chapter also discusses the relation to previous definitions. We show that the general definition makes proofs of admissibility, consistency, and additivity easier, by proving that several previous methods for defining

abstractions and additivity satisfy three simple conditions. This work was first published in [94].

- Chapter 4: We describe successful applications of additive abstractions to some standard state spaces of combinatorial problems (e.g. (18,4)-TopSpin and the Pancake puzzle). First we investigate two general methods for defining additive abstractions. To illustrate the generality of these methods we define them for the two most common ways of representing states–as a vector of state variables and as a set of logical atoms. Then we run experiments to determine the effectiveness of these methods for two benchmark state spaces: TopSpin and the Pancake puzzle. This work was published in [96].

- Chapter 5: This chapter presents a new way to improve the quality of heuristic values defined by additive abstractions in some circumstances. We show that given additional memory, the new technique to identify infeasibility can be a competitive choice to enhance the search performance. Empirical results show that the technique of identifying infeasibility can also be effective for different domains. This work was published in [93, 95, 96].

- Chapter 6: First we overview the previous work on the Travelling Salesman Problem (TSP) and its variations such as the Asymmetric TSP. Then we formalize the definitions of the SOP state space and abstractions. We investigate the structure of the SOP instances and representative examples are given to illustrate the general situations. Some greedy methods of choosing good abstractions are designed and discussed. The trade-off of the usage of greedy methods is also discussed. We run experiments over a series of TSPLIB instances using heuristics generated from random abstractions and greedy abstractions. The results indicate the benefits of using heuristics based on greedy abstractions for instances of larger size. We also ran experiments over random instances of a new version of the Pancake problem, the weighted Pancake puzzle. The experimental results show that the ideas learned for creating good abstractions for non-unit-cost problems while studying the SOP (and the greedy abstraction methods themselves) can be transferred to another problem with a different search structure.

- Chapter 7: To conclude, we give a brief review of the main contributions of the thesis and discuss future research directions.

Since the discussion of related work occurs throughout each chapter in the thesis, we do not have a separated chapter for related work.

Chapter 2

Background on Search Algorithms

In general, a large variety of classical problems in Artificial Intelligence can be modeled within the following framework. A problem space is a weighted directed graph consisting of a set of nodes and edges. The nodes represent states that are configurations of the problem, and each edge with an associated cost represents an operator that maps one state to another. A problem instance is a problem space together with an initial state and a goal state. Solving a problem instance can be formulated as search in a problem space graph in order to find a sequence of operators that transform the initial state into the goal state. Such a sequence is called a solution path to the problem and the problem is usually called a pathfinding problem.

A search algorithm is a strategy used to decide which node to expand next (i.e., to apply operators to the node to generate its successor nodes). As problems scale up, it is impractical to input the whole graph structure at the beginning of the search algorithm. Typically, the search begins with an implicit graph that specifies the initial node and applicable operators. Then the problem space can be produced partially during the search.

Some fundamental algorithms and their variants will be discussed in the following subsections. Unless otherwise stated, the discussion is restricted to pathfinding problems where the start and goal states are explicitly stated, the goal is to find the optimal solution (i.e., the solution path with the minimum cost), the search algorithm deals with a finite implicit graph, and the number of children for each node (i.e., the branching factor of each node) is a finite number for each node.

2.1 Fundamental Algorithms

Search algorithms can be classified into two categories, best-first search algorithms and depth-first search algorithms. These two algorithms differ in the ways they prioritize node expansion, deal with duplicate nodes, and reconstruct the solution path.

2.1.1 Best-First Search

A standard best-first search algorithm maintains two lists, a CLOSED list and an OPEN list. The CLOSED list is used to store already-expanded nodes, and the OPEN list contains those nodes that have been generated but not yet expanded. The two lists prevent duplicate search effort by detecting nodes that have previously been generated. In addition, maintaining the CLOSED list allows the optimal solution path to be reconstructed after completion of the search by tracing pointers backwards from the goal node to the start node.

At each cycle, best-first search expands a node t of minimum cost $f(t)$ among all nodes that have been generated but not yet expanded, until the goal node is chosen for expansion. Individual best-first search algorithms differ in the cost function $f(t)$. It becomes a breadth-first search algorithm if $f(t)$ is the depth of node t, and it becomes Dijkstra's algorithm [18] if $f(t) = g(t)$, where $g(t)$ is the cost from the initial state to the node t. If $f(t) = g(t) + h(t, goal)$, where $h(t, goal)$ is an admissible heuristic estimate of the cost from node t to a goal node, it is the A* algorithm [37].

Best-first search stores all nodes generated and typically requires space that is exponential in the search depth. Hence it is easy to exhaust the available memory if the problem space scales up.

2.1.2 Depth-First Search

As decribed in [13], "the strategy followed by depth-first search is, as its name implies, to search deeper in the graph whenever possible".

Depth-first search (DFS) stores only the current search path being explored, rather than OPEN and CLOSED lists. Hence the memory requirements of depth-first search are minimal, thereby eliminating the space constraint of best-first search.

As the path is stored on the call stack in a recursive implementation, the solution path is constructed by tracing back up the call stack. However, depth-first search cannot detect duplicated nodes efficiently, so the time complexity is increased inevitably. Moreover, DFS could continue to probe deeper along some fruitless path and would be unable to backup and try a fresh search avenue. For that reason, DFS algorithms are usually equipped with a depth bound in order to guarantee the algorithm to recover from the fruitless path and try other search avenues.

The memory complexity of depth-first search is linear in the maximum search depth, rather than exponential as in best-first search. Therefore, researchers refer to some depth-first search algorithms as *linear-space algorithms*, as classified in [98]. For instance, depth-first iterative-deepening (DFID) [65] simulates breadth-first search using memory that is only linear in the maximum search depth.

DFID is actually a sequence of iterations of depth-bounded DFS. For each iteration, it expands all nodes up to a given depth. Combining the heuristic function and DFID, the successive iterations of IDA* [65] correspond not to the increasing depth of search, but rather to increasing values of the total cost of a path. At each iteration, IDA* expands all the nodes whose total cost $f(t)$ does not exceed a given threshold defined for each iteration. The threshold for the first iteration is the

heuristic value of the root of the search tree and the threshold for the next iteration is the lowest cost of a generated node from the current iteration that exceeded the current threshold. The algorithm halts when the goal node is chosen for expansion. However, IDA* expands some nodes more than once and it does not retain any path information between iterations. In addition, while searching nodes whose costs are less than the current threshold, IDA* just proceeds with a depth-first search, ignoring the information in the values of those nodes.

Another case of depth-first search is Depth-First Branch and Bound (DFBB) [98], which performs a DFS with a global threshold for pruning nodes. DFBB starts at the root node with a global threshold u which is typically set to an easy-to-compute upper bound and may just be infinity in practice. Each time a goal node is reached whose cost is less than u, u is revised to the cost of this new goal node. The DFS process is continued until a better solution is found. DFBB always selects a most recently generated node, or a deepest node to expand next. When we select a node for expansion whose cost is greater than or equal to u, we prune this node. DFBB uses space that is linear in the search depth. The penalty for DFBB to run in linear space is that it expands some nodes not explored by best-first search, i.e. some nodes whose costs are greater than the optimal goal cost.

2.2 Some Variants of Fundamental Algorithms

Based on best-first search and depth-first search algorithms, researchers developed several algorithms for the intelligent use of available memory. These algorithms differ in the way they perform the different search activities such as node duplicate detection, prioritizing node expansion, evaluation cost calculation and search direction (uni-directional, bi-directional). The following subsections take a close look at some of these algorithms.

2.2.1 Memory-bounded Search

As the capacity of memory increased, various search algorithms were developed to use the additional memory to store more information. Reinefeld and Marsland [86] used additional memory for storing transposition tables and they successfully enhanced iterative-deepening searches. MREC [91], MA* [8] and SMA* [90] execute A* but differ in the ways they proceed when the memory is full. MREC begins to execute IDA* on the leaves of the search tree which is currently stored in memory, while MA* and SMA* delete the least promising nodes from the OPEN list to recover the memory for new nodes. SMA* is a simplified version of MA* and it improves MA* in some details, such as changing the data structure of the OPEN list, decreasing the number of cost values stored for each node, etc.

Algorithms like MREC and MA* still maintain OPEN and CLOSED lists, and select the best node from OPEN list for expansion and the worst node for pruning. So even if they generate fewer nodes than IDA*, they do not always run faster than IDA*. Like IDA*, *Iterative Threshold Search* (ITS) [31] employs a fast node generation scheme, while like MA*, ITS makes dynamic use of

memory. Therefore, in most practical search problems where the node-generation time is high, ITS can provide more significant time savings than MREC and MA*. Both ITS [31] and SMA* [90] represent attempts to significantly reduce the constant factor overhead of MA*.

2.2.2 Removing Closed Nodes from Memory

Instead of using the traceback method of solution reconstruction, it is possible to remove the CLOSED list from memory and use a divide-and-conquer method for solution reconstruction. In this way, memory is saved by storing one node in the middle of each generated path rather than the complete path. The midpoint node is used to divide the original problem into two sub-problems. Each subproblem is solved recursively by the same algorithm until all nodes on the optimal path are identified.

Korf et al. [67, 70, 71] presented a class of best-first search algorithms that reduce the space complexity. They called this technique *frontier search* [71]. The key idea is to store only the OPEN list, and not the CLOSED list. Nodes that have been expanded are prevented from being regenerated by storing a list of forbidden operators in each node. Korf showed in [67] that the algorithm dramatically reduces the memory required to store the CLOSED list. The solution path is recovered by a divide-and-conquer technique, either as a bidirectional or unidirectional search.

Considering it is not necessary to remove the entire CLOSED list, Zhou and Hansen proposed some algorithms that only select some nodes in the CLOSED list to be eliminated. In [99] they proposed a different technique for preventing node re-generation that does not require forbidden operators as in [70]. They associated with each node a counter that is initially set to the number of potential parents of a node. Each time a node is expanded, the counter of each of its child nodes is decremented. CLOSED nodes are not removed immediately. Instead, nodes in the CLOSED list can be removed from memory when their counter is zero.

For the special case of the multiple sequence alignment application Zhou and Hansen [100] proposed an even simpler technique. Because the search graph of the multiple sequence alignment problem is a lattice, it can be decomposed into a sequence of layers such that each node in a layer can only have successors in the current layer or the next layer, but not in any previous layer. If all nodes in one layer are expanded before the next layer is considered, then all previously-expanded layers can be removed from memory without risking node re-generation.

Zhou and Hansen [101] also presented a family of algorithms based on breadth-first search and heuristic search. The OPEN and CLOSED lists are indexed by layers. By deleting previous layers, memory is recovered. This technique is called *breadth-first heuristic search (BFHS)*. They showed that when using divide-and-conquer solution reconstruction, BFHS outperforms Divide-and-Conquer frontier search (DCFA*) [70] and sparse-memory A* [99] on the 15-puzzle.

2.2.3 Bidirectional Search

Bidirectional search is another branch in the development of search algorithms. Most unidirectional search algorithms may have a bidirectional counterpart, such as the bidirectional A^* algorithm [55]. The traditional version of this general technique is to perform forward and backward searches alternately. It includes both algorithms performing *front-to-end* and others performing *front-to-front* evaluations.

The first proposed algorithm on bidirectional heuristic search called BHPA [83] performed *front-to-end* evaluations which can be viewed as a combination of two traditional best-first searches in opposite directions, as Kaindl and Kainz discussed in [58]. Given a start state s and a goal state g, it employs evaluation functions $h_1(n, g)$ and $h_2(s, n)$, where $h_1(n, g)$ estimates the cost of an optimal path from n to g in the forward search, and $h_2(s, n)$ from s to n in the backward search. However, as shown in [83], BHPA's results were less efficient than those of its unidirectional counterpart for finding optimal solutions.

In order to improve the evaluation accuracy, *front-to-front* evaluations were proposed in algorithms (e.g. BHFFA [17]). Given a start state s and a goal state g, these algorithms employ more accurate dynamic heuristic evaluation functions $H_1(n)$ and $H_2(n)$. Take the search in forward direction for example. When the nodes B_i on the backward search are available in the OPEN list, $H_1(n) = max(h_1(n, g), min_i(h(n, B_i) + k_2(B_i, g)))$, where $h_1(n, g)$ is the original estimate from n to g, $h(n, B_i)$ is the estimated cost from n to B_i, and $k_2(B_i, g)$ is the known cost from B_i to the goal node g. These dynamic evaluations are more accurate than the static *front-to-end* evaluations, therefore algorithms performing *front-to-front* evaluations can be efficient in terms of the number of nodes generated [17]. But they need excessive computation effort especially when the number of nodes B_i increases.

Hence, the traditional approaches did not succeed to improve on unidirectional search for finding and guaranteeing optimal solutions [17, 58, 83]. This is mainly because most traditional bidirectional algorithms are based on traditional best-first search that has exponential storage requirements. Instead of executing two best-first searches alternately, it is possible to search in one direction and store a reasonable number of nodes, then to search in the other direction. *Perimeter search* [19] is one case of such algorithms working as follows: a breadth-first search starts from the goal to all nodes at a predetermined depth d. The final frontier of this breadth-first search is called the *perimeter* P. The search then proceeds from the start state, targeting all of the perimeter nodes and using the heuristic function $h_P(n) = min_{m \in P}[H(n, m) + H^*(m)]$, where $H(n, m)$ is the estimated cost from node n to node m and $H^*(m)$ is the true minimum cost of a path from m to the goal.

In fact, according to [19], any set of nodes which encompass the goal will work as a perimeter. Then an alternative method to generate perimeter nodes would be to use A* with all resulting nodes on the perimeter would have approximately the same cost (f value). If the perimeter is generated by a breadth-first search it is called a constant depth perimeter and it is a constant evaluation perimeter

if it is generated by a heuristic search.

Perimeter search algorithms still require a much larger number of heuristic evaluations. Therefore, perimeter search algorithms are effective only if the cost of computing the heuristic values is small compared to the cost of generating a new node.

Manzini [76] proposed an improved perimeter search algorithm called *BIDA**. By using a more efficient technique for pruning the nonoptimal paths, BIDA* can generate fewer nodes and execute fewer heuristic evaluations than IDA*. However, the improvement is still based on an additional amount of memory space for storing the perimeter.

Chapter 3

A General Theory for Additive Abstractions

The use of abstraction to create heuristics began in the late 1970s [30, 34]. Over the past decade, there has been tremendous progress on defining abstraction and additivity for specific ways of representing states and transition functions. In this chapter, we give formal definitions related to state spaces, abstractions, and the heuristics defined by them, and discuss their meanings and relation to previous work. This work was published in [94, 96].

3.1 Heuristics defined by abstraction

To illustrate the idea of abstraction and how it is used to define heuristics, consider the well-known 8-puzzle (the 3×3 sliding tile puzzle). In this puzzle there are 9 locations in the form of a 3×3 grid and 8 tiles, numbered 1–8, with the 9^{th} location being empty (or blank). A tile that is adjacent to the empty location can be moved into the empty location; every move has a cost of 1. The most common way of abstracting this state space is to treat several of the tiles as if they were indistinguishable instead of being distinct [15]. An extreme version of this type of abstraction is shown in Figure 3.1. Here the tiles are all indistinguishable from each other, so an abstract state is entirely defined by the position of the blank. There are therefore only 9 abstract states, connected as shown in Figure 3.1. The goal state in the original puzzle has the blank in the upper left corner, so the abstract goal is state z shown at the top of the figure. The number beside each abstract state is the distance from the abstract state to the abstract goal. For example, in Figure 3.1, abstract state e is 2 moves from the abstract goal. A heuristic function $h(t, g)$ for the distance from state t to g in the original space is computed in two steps: (1) compute the abstract state corresponding to t (in this example, this is done by determining the location of the blank in state t); and then (2) determine the distance from that abstract state to the abstract goal. The calculation of the abstract distance can either be done in a preprocessing step to create a heuristic lookup table called a *pattern database* [15, 14] or at the time it is needed [25, 54, 51, 73].

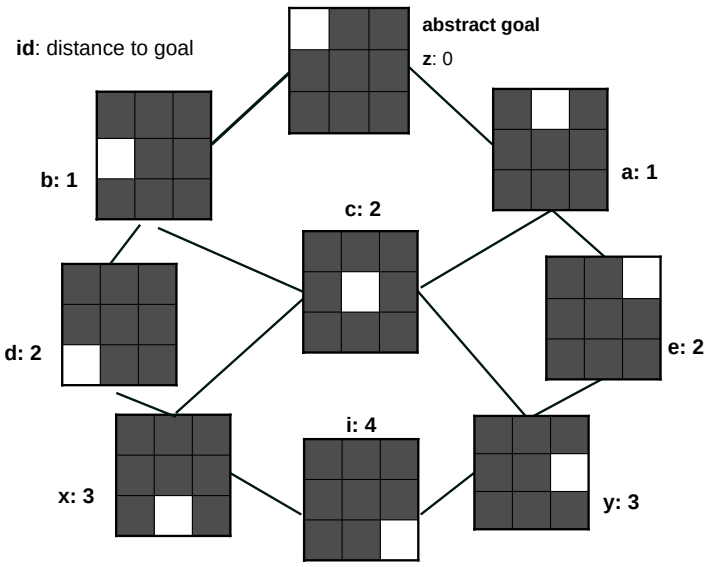


Figure 3.1: An abstraction of the 8-puzzle. The white square in each state is the blank and the non-white squares are the tiles, which are all indistinguishable from each other in this abstraction.

3.1.1 Maximum-based Method for Combining Abstractions

Given several abstractions of a state space, the heuristic $h_{max}(t, g)$ can be defined as the maximum of the abstract distances from t to g given by the abstractions individually. This is the standard method for defining a heuristic function given multiple abstractions [50]. For example, consider state A of the 3×3 sliding tile puzzle shown in the top left of Figure 3.2 and the goal state shown below it. The middle column shows an abstraction of these two states (A_1 and g_1) in which tiles 1, 3, 5, and 7, and the blank, are distinct while the other tiles are indistinguishable from each other. We refer to the distinct tiles as "distinguished tiles" and the indistinguishable tiles as "don't care" tiles. The right column shows the complementary abstraction, in which tiles 1, 3, 5, and 7 are the "don't cares" and tiles 2, 4, 6, and 8 are distinguished. The arrows in the figure trace out a least-cost path to reach the abstract goal g_i from state A_i in each abstraction. The cost of solving A_1 is 16 and the cost of solving A_2 is 12. Therefore, $h_{max}(A, g)$ is 16, the maximum of these two abstract distances.

3.1.2 Additive Abstractions

Figure 3.3 illustrates how additive abstractions can be defined for the sliding tile puzzle [26, 68, 69]. State A and the abstractions are the same as in Figure 3.2, but the costs of the operators in the abstract spaces are defined differently. Instead of all abstract operators having a cost of 1, as was the case previously, an operator only has a cost of 1 if it moves a distinguished tile; such moves are called "distinguished moves" and are shown as solid arrows in Figures 3.2 and 3.3. An operator that moves a "don't care" tile (a "don't care" move) has a cost of 0 and is shown as a dashed arrow in the figures. Least-cost paths in abstract spaces defined this way therefore minimize the number

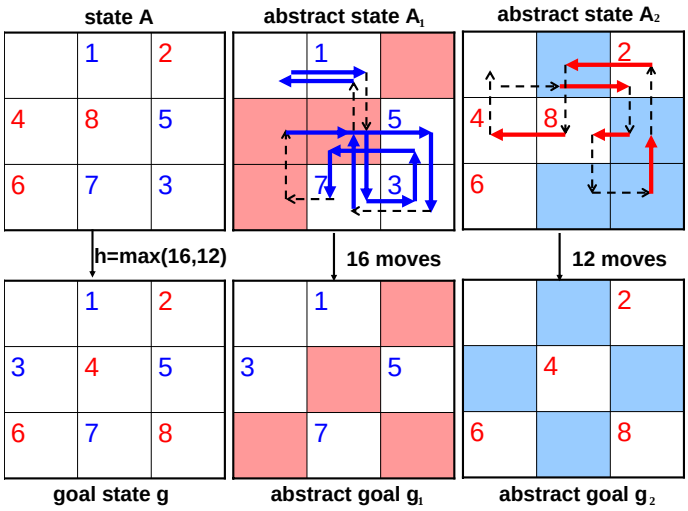


Figure 3.2: Computation of $h_{max}(A, g)$, the standard, maximum-based heuristic value for state A (top left) using the two abstractions shown in the middle and right columns. Solid arrows denote distinguished moves, dashed arrows denote "don't care" moves.

of distinguished moves without considering how many "don't care" moves are made. For example, the least-cost path for A_1 in Figure 3.3 contains fewer distinguished moves (9 compared to 10) than the least-cost path for A_1 in Figure 3.2—and is therefore lower cost according to the cost function just described—but contains more moves in total (18 compared to 16) because it has more "don't care" moves (9 compared to 6). As Figure 3.3 shows, 9 distinguished moves are needed to solve A_1 and 5 distinguished moves are needed to solve A_2. Because no tile is distinguished in both abstractions, a move that has a cost of 1 in one space has a cost of 0 in the other space, and it is therefore admissible to add the two distances. The heuristic calculated using additive abstractions is referred to as h_{add}; in this example, $h_{add}(A, g) = 9 + 5 = 14$. Note that $h_{add}(A, g)$ is less than $h_{max}(A, g)$ in this example, showing that heuristics based on additive abstractions are not always superior to the standard, maximum-based method of combining multiple abstractions even though in general they have proven very effective on the sliding tile puzzles [26, 68, 69].

The general method defined in [26, 68, 69] creates a set of k additive abstractions based on a division of the tiles into k disjoint groups and defining one abstraction for each group by making the tiles in that group distinguished in the abstraction. An important limitation of this method of defining additive abstractions is that it does not apply to spaces in which an operator can move more than one tile at a time, unless there is a way to guarantee that all the tiles that are moved by the operator are in the same group.

An example of a state space that has no additive abstractions according to previous definitions was given in Chapter 1, the Pancake puzzle. Another common state space that has no additive abstractions according to previous definitions—for similar reasons—is TopSpin.

The general definition of additive abstractions presented in the next section overcomes the lim-

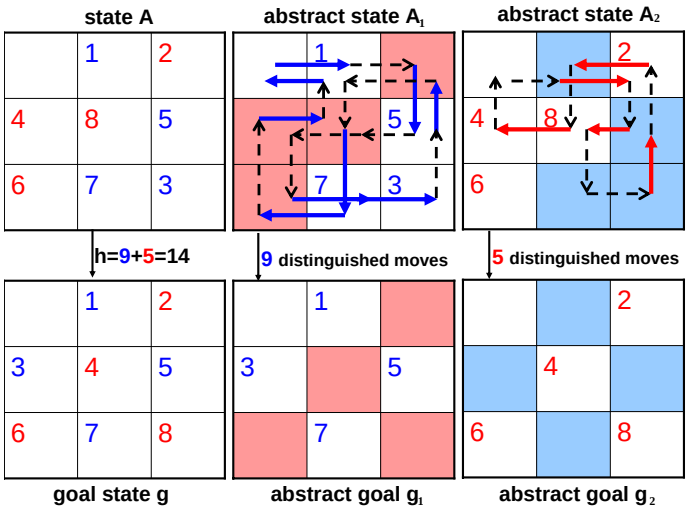


Figure 3.3: Computation of $h_{add}(A, g)$, the additive heuristic value for state A. Solid arrows denote distinguished moves, dashed arrows denote "don't care" moves.

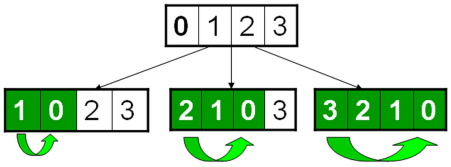


Figure 3.4: In the 4-Pancake puzzle, each state has three successors.

itations of previous definitions. Intuitively, abstractions will be additive provided that the cost of each operator is divided among the abstract spaces. Our definition provides a formal basis for this intuition. There are numerous ways to do this even when operators move many tiles (or, in other words, make changes to many state variables). For example, the operator cost might be divided proportionally across the abstractions based on the percentage of the tiles moved by the operator that are distinguished in each abstraction. We call this method of defining abstract costs "cost-splitting". For example, consider two abstractions of the 4-Pancake puzzle, one in which tiles 0 and 1 are distinguished, the other in which tiles 2 and 3 are distinguished. Then the middle operator in Figure 3.4 would have a cost of $\frac{2}{3}$ in the first abstract space and $\frac{1}{3}$ in the second abstract space, because of the three tiles this operator moves, two are distinguished in the first abstraction and one is distinguished in the second abstraction.

A different method for dividing operator costs among abstractions focuses on a specific location (or locations) in the puzzle and assigns the full cost of the operator to the abstraction in which the tile that moves into this location is distinguished. We call this a "location-based" cost definition. In the Pancake puzzle it is natural to use the leftmost location as the special location since every operator changes the tile in this location. The middle operator in Figure 3.4 would have a cost of 0 in the abstract space in which tiles 0 and 1 are distinguished and a cost of 1 in the abstract space in

which tiles 2 and 3 are distinguished because the operator moves tile 2 into the leftmost location.

Both these methods apply to the Pancake puzzle and TopSpin, and many other state spaces, but the h_{add} heuristics they produce are not always superior to the h_{max} heuristics based on the same abstractions. The theory in the remainder of the chapter and the experiments in the following chapter shed some light on the general question of when h_{add} is preferable to h_{max}.

3.2 Formal Theory of Additive Abstractions

In this section, we give formal definitions and lemmas related to state spaces, abstractions, and the heuristics defined by them, and discuss their meanings and relation to previous work. The definitions in Section 3.2.1 are standard. The definition of state space abstraction in Section 3.2.2 differs from previous definitions only in one important detail: each state transition in an abstract space has two costs associated with it instead of just one. The main new contribution is the definition of additive abstractions in Section 3.2.3.

The underlying structure of our abstraction definition is a directed graph (digraph) homomorphism. For easy reference, we quote here standard definitions of digraph and digraph homomorphism [43].

Definition 3.2.1. *A* digraph *G is a finite set $V = V(G)$ of vertices, together with a binary relation $E = E(G)$ on V. The elements (u, v) of E are called the* arcs *of G.*

Definition 3.2.2. *Let G and H be any digraphs. A* homomorphism *of G to H, written as $\psi : G \rightarrow H$, is a mapping ψ: $V(G) \rightarrow V(H)$ such that $(\psi(u), \psi(v)) \in E(H)$ whenever $(u, v) \in E(G)$.*

Note that the digraphs G and H may have self-loops, (u, u), and a homomorphism is not required to be surjective in either vertices or arcs. We typically refer to arcs as edges, but it should be kept in mind that, in general, they are directed edges, or ordered pairs.

3.2.1 State Space

Definition 3.2.3. *A* state space *is a weighted directed graph $\mathcal{S} = \langle T, \Pi, C \rangle$ where T is a finite set of states, $\Pi \subseteq T \times T$ is a set of directed edges (ordered pairs of states) representing state transitions, and $C : \Pi \longrightarrow \mathcal{R}^*$ (i.e., non-negative real values) is the edge cost function.*

In typical practice, $\mathcal{S}$ is defined implicitly. Usually each distinct state in T corresponds to an assignment of values to a set of state variables. Π and C derive from a successor function, or a set of operators. Although most of our examples are on integer domains, real-valued edge costs are permissible provided conditions are imposed to ensure that the set of paths connecting any given pair of states has a well-defined minimum cost.

In some cases, T is restricted to the set of states from which the goal can be reached. For example, in the 8-puzzle, the set of edges Π is defined by the rule "a tile that is adjacent to the empty

location can be moved into the empty location", and the set of states T is defined in one of two ways: either as the set of states from which the goal can be reached, or as the set of permutations of the tiles and the blank, in which case T consists of two components that are not connected to one another. The standard cost function C for the 8-puzzle assigns a cost of 1 to all edges, but it is easy to imagine cost functions for the 8-puzzle that depend on the tile being moved or the locations involved in the move.

A *path* from state t to state g is a sequence of edges beginning at t and ending at g. Formally, $\vec{p}$ is a path from state t to state g if $\vec{p} = \langle \pi^1, \ldots, \pi^n \rangle, \pi^j \in \Pi$ where $\pi^j = (t^{j-1}, t^j), j \in \{1, \ldots, n\}$ and $t^0 = t, t^n = g$. Note the use of superscripts rather than subscripts to distinguish states and edges within a state space. The *length* of $\vec{p}$ is the number of edges n and its *cost* is $C(\vec{p}) = \sum_{j=1}^{n} C(\pi^j)$. We use $Paths(\mathcal{S}, t, g)$ to denote the set of all paths from t to g in $\mathcal{S}$.

Definition 3.2.4. *The* optimal (minimum) cost *of a path from state t to state g in $\mathcal{S}$ is defined by*

$$\text{OPT}(t, g) = \min_{\vec{p} \in Paths(\mathcal{S}, t, g)} C(\vec{p})$$

A *pathfinding problem* is a triple $\langle \mathcal{S}, t, g \rangle$, where $\mathcal{S}$ is a state space and $t, g \in T$, with the objective of finding the minimum cost of a path from t to g, or in some cases finding a minimum cost path $\vec{p} \in Paths(\mathcal{S}, t, g)$ such that $C(\vec{p}) = \text{OPT}(t, g)$. Having just one goal state may seem restrictive, but problems having a set of goal states can be accommodated with this definition by adding a virtual goal state to the state space with zero-cost edges from the actual goal states to the virtual goal state.

3.2.2 State Space Abstraction

Definition 3.2.5. *An* Abstraction System *is a pair $\langle \mathcal{S}, \aleph \rangle$ where $\mathcal{S} = \langle T, \Pi, C \rangle$ is a state space and $\aleph = \{\langle \mathcal{A}_i, \psi_i \rangle \mid \psi_i : \mathcal{S} \to \mathcal{A}_i, 1 \leq i \leq k\}$ is a set of* abstractions*, where each abstraction is a pair consisting of an* abstract state space *and an* abstraction mapping*, where "abstract state space" and "abstraction mapping" are defined below.*

Note that these abstractions are not intended to form a hierarchy and should be considered a set of independent abstractions.

Definition 3.2.6. *An* abstract state space *is a directed graph with two weights per edge, defined by a four-tuple $\mathcal{A}_i = \langle T_i, \Pi_i, C_i, R_i \rangle$.*

T_i is the set of abstract states and Π_i is the set of abstract edges, as in the definition of a state space. In an abstract space there are two costs associated with each $\pi_i \in \Pi_i$, the *primary cost* $C_i : \Pi_i \longrightarrow \mathcal{R}^*$ and the *residual cost* $R_i : \Pi_i \longrightarrow \mathcal{R}^*$. The idea of having two costs per abstract edge, instead of just one, is inspired by the practice, illustrated in Figure 3.3, of having two types of edges in the abstract space and counting distinguished moves differently than "don't care" moves. In that example, our primary cost is the cost associated with the distinguished moves, and our residual

cost is the cost associated with the "don't care" moves. The usefulness of considering the cost of "don't care" moves arises when the abstraction system is additive, as suggested by Lemmas 3.2.6 and 5.2.2 below. These indicate when the additive heuristic is infeasible and can be improved, the effectiveness of which will become apparent in the experiments reported in Chapter 5.

Like edges, each abstract path $\vec{p}_i = \langle \pi_i^1, \ldots, \pi_i^n \rangle$ in $\mathcal{A}_i$ has a primary and residual cost: $C_i(\vec{p}_i) = \sum_{j=1}^{n} C_i(\pi_i^j)$, and $R_i(\vec{p}_i) = \sum_{j=1}^{n} R_i(\pi_i^j)$.

Definition 3.2.7. *An* abstraction mapping $\psi_i : \mathcal{S} \longrightarrow \mathcal{A}_i$ *between state space* $\mathcal{S}$ *and abstract state space* $\mathcal{A}_i$ *is defined by a mapping between the states of* $\mathcal{S}$ *and the states of* $\mathcal{A}_i$*,* $\psi_i : T \rightarrow T_i$*, that satisfies the two following conditions.*

The first condition is that the mapping is a homomorphism and thus connectivity in the original space is preserved, i.e.,

$$\forall (u, v) \in \Pi, (\psi_i(u), \psi_i(v)) \in \Pi_i \tag{3.1}$$

In other words, for each edge in the original space $\mathcal{S}$ there is a corresponding edge in the abstract space $\mathcal{A}_i$. Note that if $u \neq v$ and $\psi_i(u) = \psi_i(v)$ then a non-identity edge in $\mathcal{S}$ gets mapped to an identity edge (self-loop) in $\mathcal{A}_i$. We use the shorthand notation $t_i^j = \psi_i(t^j)$ for the abstract state in T_i corresponding to $t^j \in T$, and $\pi_i^j = \psi_i(\pi^j) = (\psi_i(u^j), \psi_i(v^j))$ for the abstract edge in Π_i corresponding to $\pi^j = (u^j, v^j) \in \Pi$.

The second condition that the state mapping must satisfy is that abstract edges must not cost more than any of the edges they correspond to in the original state space, i.e.,

$$\forall \pi \in \Pi, C_i(\pi_i) + R_i(\pi_i) \leq C(\pi) \tag{3.2}$$

As a consequence, if multiple edges in the original space map to the same abstract edge $\rho \in \Pi_i$, as is usually the case, $C_i(\rho) + R_i(\rho)$ must be less than or equal to all of them, i.e.,

$$\forall \rho \in \Pi_i,\ C_i(\rho) + R_i(\rho) \leq \min_{\pi \in \Pi, \psi_i(\pi) = \rho} C(\pi) \tag{3.3}$$

Note that if no edge maps to an edge in the abstract space, then no bound on the cost of that edge is imposed.

For example, the state mapping used to define the abstraction in the middle column of Figure 3.3 maps an 8-puzzle state to an abstract state by renaming tiles 2, 4, 6, and 8 to "don't care". This mapping satisfies condition (1) because "don't care" tiles can be exchanged with the blank whenever regular tiles can. It satisfies condition (2) because each move is either a distinguished move ($C_i(\pi_i) = 1$ and $R_i(\pi_i) = 0$) or a "don't care" move ($C_i(\pi_i) = 0$ and $R_i(\pi_i) = 1$) and in both cases $C_i(\pi_i) + R_i(\pi_i) = 1$, the cost of the edge π in the original space.

The set of abstract states T_i is usually equal to $\psi_i(T) = \{\psi_i(t) \mid t \in T\}$, but it can be a superset, in which case the abstraction is said to be *non-surjective* [47]. Likewise, the set of abstract edges

Π_i is usually equal to $\psi_i(\Pi) = \{\psi_i(\pi) \mid \pi \in \Pi\}$ but it can be a superset even if $T_i = \psi_i(T)$. In some cases, one deliberately chooses an abstract space that has states or edges that have no counterpart in the original space. For example, the methods that define abstractions by dropping operator preconditions must, by their very design, create abstract spaces that have edges that do not correspond to any edge in the original space (*e.g.* [82]). In other cases, non-surjectivity is an inadvertent consequence of the abstract space being defined implicitly as the set of states reachable from the abstract goal state by applying operator inverses. For example, if a tile in the 2×2 sliding tile puzzle is mapped to the blank in the abstract space, the puzzle now has two blanks and states are reachable in the abstract space that have no counterpart in the original space [47]. For additional examples and an extensive discussion of non-surjectivity see [52, 102].

All the lemmas and definitions that follow assume an abstraction system $\langle \mathcal{S}, \aleph \rangle$ containing k abstractions has been given. Conditions (3.1) and (3.2) guarantee the following.

Lemma 3.2.1. *For any path $\vec{p} \in Paths(\mathcal{S}, u^1, u^2)$ in $\mathcal{S}$, there is a corresponding abstract path $\psi_i(\vec{p})$ from u_i^1 to u_i^2 in $\mathcal{A}_i$ and $C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \leq C(\vec{p})$.*

Proof. By definition, $\vec{p} \in Paths(\mathcal{S}, u^1, u^2)$ in $\mathcal{S}$ is a sequence of edges $\langle \pi^1, \ldots, \pi^n \rangle, \pi^j \in \Pi$ where $\pi^j = (t^{j-1}, t^j), j \in \{1, \ldots, n\}$ and $t^0 = u^1, t^n = u^2$. Because $\Pi_i \supseteq \psi_i(\Pi)$, each of the corresponding abstract edges exists ($\pi_i^j \in \Pi_i$). Because $\pi_i^1 = (u_i^1, t_i^1)$ and $\pi_i^n = (t_i^{n-1}, u_i^2)$, the sequence $\psi_i(\vec{p}) = \langle \pi_i^1, \ldots, \pi_i^n \rangle$ is a path from u_i^1 to u_i^2.

By definition, $C(\vec{p}) = \sum_{j=1}^n C(\pi^j)$. For each π^j, Condition (2) ensures that $C(\pi^j) \geq C_i(\pi_i^j) + R_i(\pi_i^j)$, and therefore $C(\vec{p}) \geq \sum_{j=1}^n (C_i(\pi_i^j) + R_i(\pi_i^j)) = \sum_{j=1}^n C_i(\pi_i^j) + \sum_{j=1}^n R_i(\pi_i^j) = C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$. □

For example, consider state A and goal g in Figure 3.3. Because of condition (3.1), any path from state A to g in the original space is also a path from abstract state A_1 to abstract goal state g_1 and from abstract state A_2 to g_2 in the abstract spaces. Because of condition (3.2), the cost of the path in the original space is greater than or equal to the sum of the primary cost and the residual cost of the corresponding abstract path in each abstract space.

We use $Paths(\mathcal{A}_i, u, v)$ to mean the set of all paths from u to v in space $\mathcal{A}_i$.

Definition 3.2.8. *The* optimal abstract cost *from abstract state u to abstract state v in $\mathcal{A}_i$ is defined as*

$$\text{OPT}_i(u, v) = \min_{\vec{q} \in Paths(\mathcal{A}_i, u, v)} C_i(\vec{q}) + R_i(\vec{q})$$

Definition 3.2.9. *We define the heuristic value obtained from abstract space $\mathcal{A}_i$ for the cost from state t to g as*

$$h_i(t, g) = \text{OPT}_i(t_i, g_i).$$

Note that in these definitions, the path minimizing the cost is not required to be the image, $\psi_i(\vec{p})$, of a path $\vec{p}$ in $\mathcal{S}$.

The following prove that the heuristic generated by each individual abstraction is admissible (Lemma 3.2.2) and consistent (Lemma 3.2.3).

Lemma 3.2.2. $h_i(t,g) \leq \text{OPT}(t,g)$ *for all* $t, g \in T$ *and all* $i \in \{1,\ldots,k\}$.

Proof. By Lemma 3.2.1, $C(\vec{p}) \geq C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$, and therefore

$$\min_{\vec{p} \in Paths(\mathcal{S},t,g)} C(\vec{p}) \geq \min_{\vec{p} \in Paths(\mathcal{S},t,g)} C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})).$$

The left hand side of this inequality is $\text{OPT}(t,g)$ by definition, and the right hand side is proved in the following Claim 3.2.2.1 to be greater than or equal to $h_i(t,g)$. Therefore, $\text{OPT}(t,g) \geq h_i(t,g)$.

Claim 3.2.2.1 $\min_{\vec{p} \in Paths(\mathcal{S},t,g)} C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \geq h_i(t,g)$ for all $t, g \in T$.

Proof of Claim 3.2.2.1: By Lemma 3.2.1 for every path $\vec{p}$ there is a corresponding abstract path. There may also be additional paths in the abstract space, that is, $\{\psi_i(\vec{p}) \mid \vec{p} \in Paths(\mathcal{S},t,g)\} \subseteq Paths(\mathcal{A}_i,t_i,g_i)$. It follows that $\{C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \mid \vec{p} \in Paths(\mathcal{S},t,g)\} \subseteq \{C_i(\vec{q}) + R_i(\vec{q}) \mid \vec{q} \in Paths(\mathcal{A}_i,t_i,g_i)\}$. Therefore,

$$\min_{\vec{p} \in Paths(\mathcal{S},t,g)} C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \geq \min_{\vec{q} \in Paths(\mathcal{A}_i,t_i,g_i)} C_i(\vec{q}) + R_i(\vec{q}) = \text{OPT}_i(t_i,g_i) = h_i(t,g)$$

□

Lemma 3.2.3. $h_i(t^1,g) \leq \text{OPT}(t^1,t^2) + h_i(t^2,g)$ *for all* $t^1, t^2, g \in T$ *and all* $i \in \{1,\ldots,k\}$.

Proof. By the definition of OPT_i as a minimization and the definition of $h_i(t,g)$, it follows that $h_i(t^1,g) = \text{OPT}_i(t_i^1,g_i) \leq \text{OPT}_i(t_i^1,t_i^2) + \text{OPT}_i(t_i^2,g_i) = \text{OPT}_i(t_i^1,t_i^2) + h_i(t^2,g)$.

To complete the proof, we observe that by Lemma 3.2.2, $\text{OPT}(t^1,t^2) \geq h_i(t^1,t^2) = \text{OPT}_i(t_i^1,t_i^2)$.

□

Definition 3.2.10. *The* h_{max} *heuristic from state* t *to state* g *defined by an abstraction system* $\langle \mathcal{S}, \aleph \rangle$ *is*

$$h_{max}(t,g) = \max_{i=1}^{k} h_i(t,g)$$

From Lemmas 3.2.2 and 3.2.3 it immediately follows that h_{max} is admissible and consistent.

3.2.3 Additive Abstractions

In this section, we formalize the notion of "additive abstraction" that was introduced intuitively in Section 3.1.2. The example there showed that $h_{add}(t,g)$, the sum of the heuristics for state t defined by multiple abstractions, was admissible provided the cost functions in the abstract spaces only counted the "distinguished moves". In our formal framework, the "cost of distinguished moves" is captured by the notion of primary cost.

Definition 3.2.11. *For any pair of states $t, g \in T$ the* additive heuristic *given an abstraction system is defined to be*

$$h_{add}(t, g) = \sum_{i=1}^{k} C_i^*(t_i, g_i).$$

where

$$C_i^*(t_i, g_i) = \min_{\vec{q} \in Paths(\mathcal{A}_i, t_i, g_i)} C_i(\vec{q})$$

is the minimum primary cost of a path in the abstract space from t_i to g_i.

In Figure 3.3, for example, $C_1^*(A_1, g_1) = 9$ and $C_2^*(A_2, g_2) = 5$ because the minimum number of distinguished moves to reach g_1 from A_1 is 9 and the minimum number of distinguished moves to reach g_2 from A_2 is 5.

Intuitively, h_{add} will be admissible if the cost of edge π in the original space is divided among the abstract edges π_i that correspond to π, as is done by the "cost-splitting" and "location-based" methods for defining abstract costs that were introduced at the end of Section 3.1.2. This leads to the following formal definition.

Definition 3.2.12. *An abstraction system $\langle \mathcal{S}, \aleph \rangle$ is* additive *if* $\forall \pi \in \Pi, \sum_{i=1}^{k} C_i(\pi_i) \leq C(\pi)$.

The following lemmas prove that h_{add} is admissible (Lemma 3.2.4) and consistent (Lemma 3.2.5) when the abstraction system $\langle \mathcal{S}, \aleph \rangle$ is *additive*.

Lemma 3.2.4. *If $\langle \mathcal{S}, \aleph \rangle$ is additive then $h_{add}(t, g) \leq \text{OPT}(t, g)$ for all $t, g \in T$.*

Proof. Assume that $\text{OPT}(t, g) = C(\vec{p})$, where $\vec{p} = \langle \pi^1, \ldots, \pi^n \rangle \in Paths(\mathcal{S}, t, g)$. Therefore, $\text{OPT}(t, g) = \sum_{j=1}^{n} C(\pi^j)$. Since $\langle \mathcal{S}, \aleph \rangle$ is additive, it follows by definition that

$$\begin{aligned} \sum_{j=1}^{n} C(\pi^j) &\geq \sum_{j=1}^{n} \sum_{i=1}^{k} C_i(\pi_i^j) = \sum_{i=1}^{k} \sum_{j=1}^{n} C_i(\pi_i^j) \\ &\geq \sum_{i=1}^{k} C_i^*(t_i, g_i) = h_{add}(t, g) \end{aligned}$$

where the last line follows from the definitions of C_i^* and h_{add}. □

Lemma 3.2.5. *If $\langle \mathcal{S}, \aleph \rangle$ is additive then $h_{add}(t^1, g) \leq \text{OPT}(t^1, t^2) + h_{add}(t^2, g)$ for all $t^1, t^2, g \in T$.*

Proof. $C_i^*(t_i^1, g_i)$ obeys the triangle inequality: $C_i^*(t_i^1, g_i) \leq C_i^*(t_i^1, t_i^2) + C_i^*(t_i^2, g_i)$ for all $t^1, t^2, g \in T$. It follows that $\sum_{i=1}^{k} C_i^*(t_i^1, g_i) \leq \sum_{i=1}^{k} C_i^*(t_i^1, t_i^2) + \sum_{i=1}^{k} C_i^*(t_i^2, g_i)$.

Because $\sum_{i=1}^{k} C_i^*(t_i^1, g_i) = h_{add}(t^1, g)$ and $\sum_{i=1}^{k} C_i^*(t_i^2, g_i) = h_{add}(t^2, g)$, it follows that $h_{add}(t^1, g) \leq \sum_{i=1}^{k} C_i^*(t_i^1, t_i^2) + h_{add}(t^2, g)$.

Since $\langle \mathcal{S}, \aleph \rangle$ is additive, by Lemma 3.2.4, $\text{OPT}(t^1, t^2) \geq \sum_{i=1}^{k} C_i^*(t_i^1, t_i^2)$.

Hence $h_{add}(t^1, g) \leq \text{OPT}(t^1, t^2) + h_{add}(t^2, g)$ for all $t^1, t^2, g \in T$. □

We now develop a simple test that has important consequences for additive heuristics. Define $\vec{P}_i(t_i, g_i) = \{\vec{q} \mid \vec{q} \in Paths(\mathcal{A}_i, t_i, g_i)$ and $C_i(\vec{q}) = C_i^*(t_i, g_i)\}$, the set of abstract paths from t_i to g_i whose primary cost is minimal.

Definition 3.2.13. *The* conditional optimal residual cost *is the minimum residual cost among the paths in* $\vec{P}_i(t_i, g_i)$*:*

$$R_i^*(t_i, g_i) = \min_{\vec{q} \in \vec{P}_i(t_i, g_i)} R_i(\vec{q})$$

Note that the value of $(C_i^*(t_i, g_i) + R_i^*(t_i, g_i))$ is sometimes, but not always, equal to the optimal abstract cost $\text{OPT}_i(t_i, g_i)$. In Figure 3.3, for example, $\text{OPT}_1(A_1, g_1) = 16$ (a path with this cost is shown in Figure 3.2) and $C_1^*(A_1, g_1) + R_1^*(A_1, g_1) = 18$, while $C_2^*(A_2, g_2) + R_2^*(A_2, g_2) = \text{OPT}_2(A_2, g_2) = 12$. As the following lemmas show, it is possible to draw important conclusions about h_{add} by comparing its value to $(C_i^*(t_i, g_i) + R_i^*(t_i, g_i))$.

Lemma 3.2.6. *Let* $\langle \mathcal{S}, \aleph \rangle$ *be any additive abstraction system and let* $t, g \in T$ *be any states. If* $h_{add}(t, g) \geq C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$ *for all* $j \in \{1, \ldots, k\}$*, then* $h_{add}(t, g) \geq h_{max}(t, g)$*.*

Proof. By the definition of $\text{OPT}_i(t_i, g_i)$, $\forall j \in \{1, \ldots, k\}, C_j^*(t_j, g_j) + R_j^*(t_j, g_j) \geq \text{OPT}_j(t_j, g_j)$. Therefore, $\forall j \in \{1, \ldots, k\}, h_{add}(t, g) \geq C_j^*(t_j, g_j) + R_j^*(t_j, g_j) \geq \text{OPT}_j(t_j, g_j) \Rightarrow h_{add}(t, g) \geq \max_{1 \leq i \leq k} \text{OPT}_i(t_i, g_i) = h_{max}(t, g)$. $\square$

Lemma 3.2.6 gives a condition under which h_{add} is guaranteed to be at least as large as h_{max} for a specific states t and g. If this condition holds for a large fraction of the state space T, one would expect that search using h_{add} to be at least as fast as, and possibly faster than, search using h_{max}. This will be seen in the experiments reported in Chapter 4. The opposite is not true in general, i.e., failing this condition does not imply that h_{max} will result in faster search than h_{add}.

Lemma 3.2.7. *For an additive* $\langle \mathcal{S}, \aleph \rangle$ *and path* $\vec{p} \in Paths(\mathcal{S}, t, g)$ *with* $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$*,* $C_j(\psi_j(\vec{p})) = C_j^*(t_j, g_j)$ *for all* $j \in \{1, \ldots, k\}$*.*

Proof. Suppose for a contradiction that there exists some i_1, such that $C_{i_1}(\psi_{i_1}(\vec{p})) > C_{i_1}^*(t_{i_1}, g_{i_1})$. Then because $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$, there must exist some i_2, such that $C_{i_2}(\psi_{i_2}(\vec{p})) < C_{i_2}^*(t_{i_2}, g_{i_2})$, which contradicts the definition of C_i^*. Therefore, such an i_1 does not exist and $C_j(\psi_j(\vec{p})) = C_j^*(t_j, g_j)$ for all $j \in \{1, \ldots, k\}$. $\square$

Lemma 3.2.8. *For an additive* $\langle \mathcal{S}, \aleph \rangle$ *and a path* $\vec{p} \in Paths(\mathcal{S}, t, g)$ *with* $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$*, then* $R_i(\psi_i(\vec{p})) \geq R_i^*(t_i, g_i)$ *for all* $i \in \{1, \ldots, k\}$*.*

Proof. Following Lemma 3.2.7 and the definition of $\vec{P}_i(t_i, g_i)$, $\psi_i(\vec{p}) \in \vec{P}_i(t_i, g_i)$ for all $i \in \{1, \ldots, k\}$. Because $R_i^*(t_i, g_i)$ is the smallest residual cost of paths in $\vec{P}_i(t_i, g_i)$, it follows that $R_i(\psi_i(\vec{p})) \geq R_i^*(t_i, g_i)$. $\square$

3.3 Relation to Previous Work

The aim of the preceding formal definitions is to identify fundamental properties that guarantee that abstractions will give rise to admissible, consistent heuristics. We have shown that the following two conditions guarantee that the heuristic defined by an abstraction is admissible and consistent

$$(P1) \qquad \forall (u, v) \in \Pi, (\psi_i(u), \psi_i(v)) \in \Pi_i$$

$$(P2) \qquad \forall \pi \in \Pi, C(\pi) \geq C_i(\pi_i) + R_i(\pi_i)$$

and that a third condition

$$(P3) \qquad \forall \pi \in \Pi, C(\pi) \geq \sum_{i=1}^{k} C_i(\pi_i)$$

guarantees that $h_{add}(t, g)$ is admissible and consistent.

Previous work has focused on defining abstraction and additivity for specific ways of representing states and transition functions. These are important contributions because ultimately one needs computationally effective ways of defining the abstract state spaces, abstraction mappings, and cost functions that our theory takes as given. The importance of our contribution is that it should make future proofs of admissibility, consistency, and additivity easier, because one will only need to show that a particular method for defining abstractions satisfies the three preceding conditions. These are generally very simple conditions to demonstrate, as we will now do for several methods for defining abstractions and additivity that currently exist in the literature.

Previous Definitions of Abstraction

The use of abstraction to create heuristics began in the late 1970s and was popularized in Pearl's landmark book on heuristics [82]. Two abstraction methods were identified at that time: "relaxing" a state space definition by dropping operator preconditions [30, 34, 82, 92], and "homomorphic" abstractions [4, 64]. These early notions of abstraction were unified and extended in [78, 84], which give a formal definition that is the same as ours in all important respects except for the concept of "residual cost" that we have introduced.[1]

Today's two most commonly used abstraction methods are among the ones implemented in Prieditis's Absolver II system [84]. The first is "domain abstraction", which was independently introduced in the seminal work on pattern databases [14, 16] and then generalized in [47]. It assumes a state is represented by a set of state variables, each of which has a set of possible values called its domain. An abstraction on states is defined by specifying a mapping from the original domains to new, smaller domains. For example, an 8-puzzle state is typically represented by 9 variables, one for each location in the puzzle, each with the same domain of 9 elements, one for each tile and one more for the blank. A domain abstraction that maps all the elements representing the tiles to the

[1] Prieditis's definition allows an abstraction to expand the set of goals. This can be achieved in our definition by mapping non-goal states in the original space to the same abstract state as the goal.

same new element ("don't care") and the blank to a different element would produce the abstract space shown in Figure 3.1. The reason this particular example satisfies property (P1) is explained in Section 3.2.2. In general, a domain abstraction will satisfy property (P1) as long as the conditions that define when state transitions occur (*e.g.,* operator preconditions) are guaranteed to be satisfied by the "don't care" symbol whenever they are satisfied by one or more of the domain elements that map to "don't care". Property (P2) follows immediately from the fact that all state transitions in the original and abstract spaces have a primary cost of 1.

The other major type of abstraction used today, called "drop" in [84], was independently introduced for abstracting planning domains represented by grounded (or propositional) STRIPS operators [22]. In a STRIPS representation, a state is represented by the set of logical atoms that are true in that state, and the directed edges between states are represented by a set of operators, where each operator a is described by three sets of atoms, $P(a)$, $A(a)$, and $D(a)$. $P(a)$ lists a's preconditions: a can be applied to state t only if all the atoms in $P(a)$ are true in t (*i.e.*, $P(a) \subseteq t$). $A(a)$ and $D(a)$ specify the effects of operator a, with $A(a)$ listing the atoms that become true when a is applied (the "add" list) and $D(a)$ listing the atoms that become false when a is applied (the "delete" list). Hence if operator a is applicable to state t, the state $u = a(t)$ it produces when applied to t is the set of atoms $u = (t - D(a)) \cup A(a)$.

In this setting, Edelkamp [22] defined an abstraction of a given state space by specifying a subset of the atoms and restricting the abstract state descriptions and operator definitions to include only atoms in the subset. Suppose V_i is the subset of the atoms underlying abstraction mapping $\psi_i : \mathcal{S} \longrightarrow \mathcal{A}_i$, where $\mathcal{S}$ is the original state space and $\mathcal{A}_i$ is the abstract state space based on V_i. Two states in $\mathcal{S}$ will be mapped to the same abstract state if and only if they contain the same subset of atoms in V_i, *i.e.*, $\psi_i(t) = \psi_i(u)$ iff $t \cap V_i = u \cap V_i$. This satisfies property (P1) because operator a being applicable to state t ($P(a) \subseteq t$) implies abstract operator $a_i = \psi_i(a)$ is applicable to abstract state t_i ($P(a) \cap V_i \subseteq t \cap V_i$) and the resulting state $a(t) = (t - D(a)) \cup A(a)$ is mapped by ψ_i to $a_i(\psi_i(t))$ because set intersection distributes across set subtraction and union ($V_i \cap ((t - D(a)) \cup A(a)) = ((V_i \cap t) - (V_i \cap D(a))) \cup (V_i \cap A(a))$). Again, property (P2) follows immediately from the fact that all operators in the original and abstract spaces have a primary cost of 1.

Helmert et al. [45] described a more general approach to defining abstractions for planning based on "transition graph abstractions". A transition graph is a directed graph in which the arcs have labels, and a transition graph abstraction is a directed graph homomorphism that preserves the labels.[2] Hence, Helmert et al.'s method is a restricted version of our definition of abstraction and therefore satisfies properties (P1) and (P2). Helmert et al. make the following interesting observations that are true of our more general definition of abstractions:

[2]"Homomorphism" here means the standard definition of a digraph homomorphism (Definition 3.2.2), which permits non-surjectivity (as discussed in Section 3.2.2), as opposed to Helmert et al.'s definition of "homomorphism", which does not allow non-surjectivity.

- the composition of two abstractions is an abstraction. In other words, if $\psi : \mathcal{S} \longrightarrow \mathbf{A}$ is an abstraction of $\mathcal{S}$ and $\phi : \mathbf{A} \longrightarrow \mathbf{B}$ is an abstraction of $\mathbf{A}$, then $(\phi \circ \psi) : \mathcal{S} \longrightarrow \mathbf{B}$ is an abstraction of $\mathcal{S}$. This property of abstractions was exploited by Prieditis [84].

- the "product" $\mathcal{A}_1 \times \mathcal{A}_2$ of two abstractions, $\mathcal{A}_1$ and $\mathcal{A}_2$, of $\mathcal{S}$ is an abstraction of $\mathcal{S}$, where the state space of the product is the Cartesian product of the two abstract state spaces, and there is an edge $\pi_{1\times2}$ in the product space from state (t_1, t_2) to state (u_1, u_2) if there is an edge π_1 from t_1 to u_1 in $\mathcal{A}_1$ and there is an edge π_2 from t_2 to u_2 in $\mathcal{A}_2$. The primary cost of $\pi_{1\times2}$ is the minimum of $C_1(\pi_1)$ and $C_2(\pi_2)$ and the residual cost of $\pi_{1\times2}$ is taken from the same space as the primary cost. Because they are working with labelled edges Helmert et al. require the edge connecting t_1 to u_1 to have the same label as the edge connecting t_2 to u_2; this is called a "synchronized" product and is denoted $\mathcal{A}_1 \otimes \mathcal{A}_2$ (refer to Definition 6 in [45] for the exact definition of synchronized product).

Figure 3.5 shows the synchronized product, B, of two abstractions, $\mathcal{A}_1$ and $\mathcal{A}_2$, of the 3-state space $\mathcal{S}$ in which the edge labels are a and b. $\mathcal{A}_1$ is derived from $\mathcal{S}$ by mapping states s_1 and s_2 to the same state ($s_{1,2}$), and $\mathcal{A}_2$ is derived from $\mathcal{S}$ by mapping states s_2 and s_3 to the same state ($s_{2,3}$). Note that B contains four states, more than the original space. It is an abstraction of $\mathcal{S}$ because the mapping of original states s_1, s_2, and s_3 to states $(s_{1,2}, s_1)$ $(s_{1,2}, s_{2,3})$ and $(s_3, s_{2,3})$, respectively, satisfies property (P1), and property (P2) is satisfied automatically because all edges have a cost of 1. From this point of view the fourth state in B, (s_3, s_1), is redundant with state $(s_{1,2}, s_1)$. Nevertheless it is a distinct state in the product space.

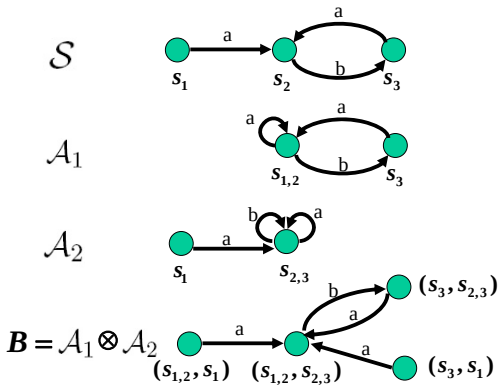


Figure 3.5: $\mathcal{S}$ is the original state space. $\mathcal{A}_1$ and $\mathcal{A}_2$ are abstractions of $\mathcal{S}$. $B = \mathcal{A}_1 \otimes \mathcal{A}_2$ is the synchronized product of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Haslum et al. [38] introduce a family of heuristics, called h^m (for any fixed $m \in \{1, 2, ...\}$), that is based on the idea of "critical paths" as classified in the recent literatures [20, 44], which are not covered by our definition because the value of the heuristic for state t, $h^m(t, g)$, is not defined as the

distance from the abstraction of t to the abstract goal state g. Instead it takes advantage of a special monotonicity property of costs in planning problems: the cost of achieving a subset of the atoms defining the goal is a lower bound on the cost of achieving the goal. When searching backwards from the goal g to the start state s, as Haslum et al. do, this allows an admissible heuristic to be defined in the following recursive minimax fashion ($|t|$ denotes the number of atoms in state t):

$$h^m(s,t) = \begin{cases} 0, & t \subseteq start \\ \min\limits_{(t',t)\in\Pi} C(t',t) + h^m(t'), & |t| \leq m \\ \max\limits_{t'\subseteq t, |t'|\leq m} h^m(t'), & |t| > m \end{cases}$$

The first two lines of this definition are the standard method for calculating the cost of a least-cost path. It is the third line that uses the fact that the cost of achieving any subset of the atoms in t is a lower bound on the cost of achieving the entire set of atoms t. The recursive calculation alternates between the min and max calculation depending on the number of atoms in the state currently being considered in the recursive calculation, and is therefore different than a shortest path calculation or taking the maximum of a set of shortest path calculations.

Recently, Helmert and Röger [46] introduced a new way of defining abstractions for the pancake problem called *relative-order* abstractions. Unlike the abstraction considered in previous work [27, 96], *relative-order* abstractions map two states s_1 and s_2 in the original state space to the same abstract state s' if the relative order of the distinguished tiles in the abstraction is the same in both s_1 and s_2 (refer to Definition 7 in [46] for the exact definition of *relative-order* abstractions). The abstraction induced by relative orders still preserves connectivity in the original state space (refer to Theorem 1 in [46] for the proof) and hence it satisfies property (P1). Property (P2) follows immediately from the fact that all state transitions in the original and abstract state spaces have a primary cost of one.

Previous definitions of additive abstractions

Prieditis [84] included a method ("Factor") in his Absolver II system for creating additive abstractions, but did not present any formal definitions or theory.

The first thorough discussion of additive abstractions is due to Korf and Taylor [69]. They observed that the sliding tile puzzle's Manhattan Distance heuristic, and several of its enhancements, were the sum of the distances in a set of abstract spaces in which a small number of tiles were "distinguished". As explained in Section 3.1.2, what allowed the abstract distances to be added and still be a lower bound on distances in the original space is that only the moves of the distinguished tiles counted towards the abstract distance and no tile was distinguished in more than one abstraction. This idea was later developed in a series of papers [26, 68], which extended its application to other domains, such as the 4-peg Towers of Hanoi puzzle.

In the planning literature, the same idea was proposed by Haslum et al. [38], who described it as partitioning the operators into disjoint sets $B_1, ...B_k$ and counting the cost of operators in set B_i

only in abstract space A_i. The example they give is that in the Blocks World operators that move block i would all be in set B_i, effectively defining a set of additive abstractions for the Blocks World exactly analogous to the Korf and Taylor abstractions that define Manhattan Distance for the sliding tile puzzle.

Edelkamp [22] took a different approach to defining additive abstractions for STRIPS planning representations. His method involves partitioning the atoms into disjoint sets $V_1, ...V_k$ such that no operator changes atoms in more than one group. If abstract space A_i retains only the atoms in set V_i then the operators that do not affect atoms in V_i will have no effect at all in abstract space A_i and will naturally have a cost of 0 in A_i. Since no operator affects atoms in more than one group, no operator has a non-zero cost in more than one abstract space and distances in the abstract spaces can safely be added. Haslum et al. [39] extended this idea to representations in which state variables could have multiple values. In a subsequent paper Edelkamp [23] remarks that if there is no partitioning of atoms that induces a partitioning of the operators as just described, additivity could be "enforced" by assigning an operator a cost of zero in all but one of the abstract spaces—a return to the Korf and Taylor idea.

All the methods just described might be called "all-or-nothing" methods of defining abstract costs, because the cost of each edge $C(\pi)$ is fully assigned to be the cost of the corresponding abstract edge $C_i(\pi_i)$ in one of the abstractions and the corresponding edges in all the other abstractions are assigned a cost of zero. Any such method obviously satisfies property (P3) and is therefore additive.

Our theory of additivity does not require abstract methods to be defined in an all-or-nothing manner, it allows $C(\pi)$ to be divided in any way whatsoever among the abstractions as long as property (P3) is satisfied. This possibility has been recognized in some recent publications [61, 62, 63]. This generalization is important because it eliminates the requirement that operators must move only one "tile" or change atoms/variables in one "group", and the related requirement that tiles/atoms be distinguished/represented in exactly one of the abstract spaces. This requirement restricted the application of previous methods for defining additive abstractions, precluding their application to state spaces such as the Pancake puzzle and TopSpin. As the following chapter shows, with our definition, additive abstractions can be defined for any state space, including the two just mentioned.

Helmert et al. [45] showed that the synchronized product of additive abstractions produces a heuristic h_{sprod} that dominates h_{add}, in the sense that $h_{sprod}(s) \geq h_{add}(s)$ for all states s. This happens because the synchronized product forces the same path to be used in all the abstract spaces, whereas the calculation of each C_i^* in h_{add} can be based on a different path. The discussion of the negative results and infeasibility in the following chapters highlight the problems that can arise because each C_i^* is calculated independently.

In the planning literature, Haslum et al. [38] first defined a h_{max} over a collection of h_{add} heuristics. In order to exploit the different strengths of different heuristics, recently Coles et al.

[12] introduced a new structure for combining heuristics for planning which is a framework called the Additive-Disjunctive Heuristic Graph (ADHG). This framework applies a directed acyclic graph consisting of vertices and edges representing the dependencies of heuristics (refer to Definition 4.2 in [12] for the exact definition of ADHG Heuristics), and it allows h_{max} and h_{add} to be combined more flexibly. Coles et al. [12] defined and proved that safely additive ADHG heuristics (refer to Definition 4.4 in [12] for the definition of safely additive ADHG heuristics) are admissible whose proof is a straightforward generalization of property (P3) in our definitions.

3.4 Chapter Summary

In this chapter we have presented a formal, general definition of additive abstractions that removes the restrictions of most previous definitions, thereby enabling additive abstractions to be defined for any state space.

We have proven that heuristics based on additive abstractions are consistent as well as admissible. Our definition formalizes the intuitive idea that abstractions will be additive provided the cost of each operator is divided among the abstract spaces.

A distinctive feature of our definition is that each edge in an abstract space has two costs instead of just one. This was inspired by previous definitions treating "distinguished" moves differently than "don't care" moves in calculating least-cost abstract paths. Formalizing this idea with two costs per edge has enable us to develop a way of testing if the heuristic value returned by additive abstractions is provably too low. Chapter 5 explores this in detail.

The importance of our contribution is that it should make future proofs of admissibility, consistency, and additivity easier, because one will only need to show that a particular method for defining abstractions satisfies the three preceding conditions. These are generally very simple conditions to demonstrate, as we did for several previous methods for defining abstractions and additivity that currently exist in the literature.

Chapter 4

Additive Abstractions for Combinatorial Puzzles

The aim of the formal definitions in the preceding chapter is to identify fundamental properties that guarantee that abstractions will give rise to admissible, consistent heuristics. This chapter reports the results of applying the general definition of additive abstraction given in the preceding chapter to two benchmark state spaces: TopSpin and the Pancake puzzle. This work was published in [94, 96] where a few additional experimental results may be found.

4.1 Defining Costs

We will investigate two general methods for defining the primary cost of an abstract state transition $C_i(\pi_i)$, which we call "cost-splitting" and "location-based" costs. To illustrate the generality of these methods we will define them for the two most common ways of representing states—as a vector of state variables, which is the method we implemented in our experiments, and as a set of logical atoms as in the STRIPS representation for planning problems.

In a state variable representation a state t is represented by a vector of m state variables, each having its own domain of possible values D_j, *i.e.*, $t =< t(0), ..., t(m-1) >$, where $t(j) \in D_j$ is the value assigned to the j^{th} state variable in state t. For example, in puzzles such as the Pancake puzzle and the sliding tile puzzles, there is typically one variable for each physical location in the puzzle, and the value of $t(j)$ indicates which "tile" is in location j in state t. In this case the domain for all the variables is the same. State space abstractions are defined by abstracting the domains. In particular, in this setting domain abstraction ψ_i will leave specific domain values unchanged (the "distinguished" values according to ψ_i) and map all the rest to the same special value, "don't care". The abstract state corresponding to t according to ψ_i is $t_i =< t_i(0), ..., t_i(m-1) >$ with $t_i(j) = \psi_i(t(j))$. As in previous research with these state spaces a set of abstractions is based on a division of the domain values into disjoint sets $E_1, ..., E_k$ with E_i being the set of distinguished values in abstraction i. Note that the theory developed in Chapter 3 does not require the distinguished

values in different abstractions to be mutually exclusive; it allows a value to be distinguished in any number of abstract spaces provided abstract costs are defined appropriately.

As mentioned previously, in a STRIPS representation a state is represented by the set of logical atoms that are true in the state. A state variable representation can be converted to a STRIPS representation in a variety of ways, the simplest being to define an atom for each possible variable-value combination. If state variable j has value v in the state variable representation of state t then the atom $variable\text{-}j\text{-}has\text{-}value\text{-}v$ is true in the STRIPS representation of t. The exact equivalent of domain abstraction can be achieved by defining V_i, the set of atoms to be used in abstraction i, to be all the atoms $variable\text{-}j\text{-}has\text{-}value\text{-}v$ in which $v \in E_i$, the set of distinguished values for domain abstraction i.

4.1.1 Cost-splitting

In a state variable representation, the cost-splitting method of defining primary costs works as follows. A state transition π that changes b^π state variables has its cost, $C(\pi)$, split among the corresponding abstract state transitions $\pi_1, \ldots, \pi_k$ in proportion to the number of distinguished values they assign to the variables, *i.e.,* in abstraction i

$$C_i(\pi_i) = \min_{\pi, \psi_i(\pi)=\pi_i} \frac{b_i^\pi * C(\pi)}{b^\pi}$$

if π changes b^π variables and b_i^π of them are assigned distinguished values by π_i. For example, the 4-Pancake puzzle is composed of 4 tiles. If a particular state transition reverses the order of all four tiles (i.e., $b^\pi = 4$) and two of them are distinguished according to abstraction ψ_i (i.e., $b_i^\pi = 2$), the corresponding abstract state transition, π_i, would cost $\frac{2}{4}$.

If each domain value is distinguished in at most one abstraction (*e.g.* if the abstractions are defined by partitioning the domain values) cost-splitting produces additive abstractions, *i.e.*, $C(\pi) \geq \sum_{i=1}^k C_i(\pi_i)$ for all $\pi \in \Pi$. Because $C(\pi)$ is known to be an integer, h_{add} can be defined to be the ceiling of the sum of the abstract distances, $\lceil \sum_{i=1}^k C_i(\pi_i) \rceil$, instead of just the sum. Meanwhile, we must be careful if an inequality arises due to round off errors. In our implementation we avoid this possibility by simulating the rational numbers using integers obtained by scaling by the Least Common Multiple of the denominators.

With a STRIPS representation, cost-splitting could be defined identically, with b^π being the number of atoms changed (added or deleted) by operator π in the original space and b_i^π being the number of atoms changed by the corresponding operator in abstraction i.

4.1.2 Location-based Costs

In a location-based cost definition for a state variable representation, a state variable loc_π is associated with state transition π and π's full cost $C(\pi)$ is assigned to abstract state transition π_i if π_i changes the value of variable loc_π to a value that is distinguished according to ψ_i. Formally:

$$C_i(\pi_i) = \begin{cases} \min_{\pi, \psi_i(\pi)=\pi_i} C(\pi), & \text{if } \pi_i = (t_i^1, t_i^2), t_i^1(loc^\pi) \neq t_i^2(loc^\pi), \text{ and} \\ & t_i^1(loc^\pi) \text{ is a distinguished value according to } \psi_i. \\ 0, & \text{otherwise.} \end{cases}$$

Instead of focusing on the value that is assigned to variable loc_π, location-based costs can be defined equally well on the value that variable loc_π had before it was changed. In either case, if each domain value is distinguished in at most one abstraction location-based costs produce additive abstractions. The name "location-based" is based on the typical representations used for puzzles, in which there is a state variable for each physical location in the puzzle.

For example, for a STRIPS representation of states location-based costs can be defined by choosing an atom a in the Add list for each operator π and assigning the full cost $C(\pi)$ to abstraction i if a appears in the Add list of π_i. If atoms are partitioned so that each atom appears in at most one abstraction, this method will define additive costs.

Although the cost-splitting and location-based methods for defining costs can be applied to a wide range of state spaces, they are not guaranteed to define heuristics that are superior to other heuristics for a given state space. We determined experimentally that heuristics based on cost-splitting substantially improve performance for sufficiently large versions of TopSpin and that heuristics based on location-based costs vastly improve the state of the art for the 17-Pancake puzzle.[1]

The following subsections describe the positive results in detail. The negative results are discussed in Section 4.2.2.

4.2 Experimental Results

In all our experiments all edges in the original state spaces have a cost of 1 and we define $R_i(\pi_i) = 1 - C_i(\pi_i)$, its maximum permitted value when edges cost 1. The heuristic search algorithm is *IDA** since it reduces the memory requirement compared to *A**. Algorithms are coded in C on a machine with an AMD Athlon(tm) 64 Processor 3700+ with a 2.4 GHz clock rate and 2GB main memory. We use pattern databases to store the heuristic values. The pre-processing time required to compute the pattern databases is excluded from the times reported in the results, because the PDB needs to be calculated only once and this overhead is amortized over the solving of many problem instances.

4.2.1 Positive Results

In this section we experimentally show that heuristics based on cost-splitting substantially improve performance for sufficiently large versions of TopSpin and that heuristics based on location-based costs vastly improve the state of the art for the 17-Pancake puzzle.

[1] This claim was true when this work was published. Recently Helmert and Röger [46] have improved the previous results published in [27].

TopSpin with Cost-Splitting

In the (N, K)-TopSpin puzzle (see Figure 4.1) there are N tiles (numbered $1, \ldots, N$) arranged on a circular track, and two physical movements are possible: (1) the entire set of tiles may be rotated around the track, and (2) a segment consisting of K adjacent tiles in the track may be reversed. As in previous formulations of this puzzle as a state space [28, 51, 53], we do not represent the first physical movement as an operator, but instead designate one of the tiles (tile 1) as a reference tile with the goal being to get the other tiles in increasing order starting from this tile (regardless of its position). The state space therefore has N operators (numbered $1, \ldots, N$), with operator a reversing the segment of length K starting at position a relative to the current position of tile 1. For certain combinations of N and K all possible permutations can be generated from the standard goal state by these operators, but in general the space consists of connected components and so not all states are reachable [9]. In the experiments in this section, $K = 4$ and N is varied.

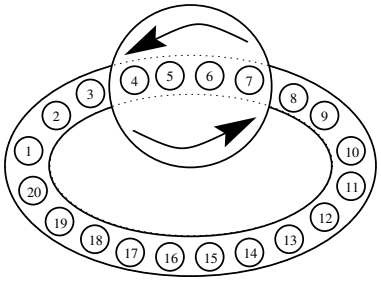


Figure 4.1: The TopSpin puzzle.

The sets of abstractions used in these experiments are described using a tuple written as a_1–a_2–…–a_M, indicating that the set contains M abstractions, with tiles $1 \ldots (a_1)$ distinguished in the first abstraction, tiles $(a_1 + 1) \ldots (a_1 + a_2)$ distinguished in the second abstraction, and so on. For example, 6-6-6 denotes a set of three abstractions in which the distinguished tiles are $(1 \ldots 6)$, $(7 \ldots 12)$, and $(13 \ldots 18)$ respectively.

The experiments compare h_{add}, the additive use of a set of abstractions, with h_{max}, the standard use of the same abstractions, in which as described in Section 3.1 the full cost of each state transition is counted in each abstraction and the heuristic returns the maximum distance to goal returned by the different abstractions. Cost-splitting is used to define operator costs in the abstract spaces for h_{add}. Because $K = 4$, each operator moves 4 tiles. If b_i of these are distinguished tiles when operator op is applied to state s_i in abstraction i, applying op to s_i has a primary cost of $\frac{b_i}{4}$ in abstraction i.

In these experiments the heuristic defined by each abstraction is stored in a pattern database (PDB). Each abstraction would normally be used to define its own PDB, so that a set of M abstractions would require M PDBs. However, for TopSpin, if two (or more) abstractions have the same number of distinguished tiles and the distinguished tiles are all adjacent, one PDB can be used for all of them by suitably renaming the tiles before doing the PDB lookup. For the 6-6-6 abstractions, for

example, only one PDB is needed, but three lookups would be done in it, one for each abstraction. Because the position of tile 1 is effectively fixed, this PDB is N times smaller than it would normally be. For example, with $N = 18$, the PDB for the 6-6-6 abstractions contains $17 \times 16 \times \ldots \times 13$ entries. The memory needed for each entry in the h_{add} PDBs is twice the memory needed for an entry in the h_{max} PDBs because of the need to represent fractional values.

We ran experiments for the values of N and sets of abstractions shown in the first two columns of Table 4.1. Start states were generated by a random walk of 150 moves from the goal state. There were 1000, 50 and 20 start states for $N = 12, 16$ and 18, respectively. The average solution length for these start states is shown in the third column of Table 4.1. The average number of nodes generated and the average CPU time (in seconds) for *IDA** to solve the given start states is shown in the **Nodes** and **Time** columns for each of h_{max} and h_{add}. The **Nodes Ratio** column gives the ratio of **Nodes** using h_{add} to **Nodes** using h_{max}. A ratio less than one (highlighted in bold) indicates that h_{add}, the heuristic based on additive abstractions with cost-splitting, is superior to h_{max}, the standard heuristic using the same set of abstractions.

N	Abs	Average Solution Length	h_{max}		h_{add} based on cost-splitting		Nodes Ratio
			Nodes	Time	Nodes	Time	
12	6-6	9.138	14,821	0.05	53,460	0.16	3.60
12	4-4-4	9.138	269,974	1.10	346,446	1.33	1.28
12	3-3-3-3	9.138	1,762,262	8.16	1,388,183	6.44	**0.78**
16	8-8	14.040	1,361,042	3.42	2,137,740	4.74	1.57
16	4-4-4-4	14.040	4,494,414,929	13,575.00	251,946,069	851.00	**0.056**
18	9-9	17.000	38,646,344	165.42	21,285,298	91.76	**0.55**
18	6-6-6	17.000	18,438,031,512	108,155.00	879,249,695	4,713.00	**0.04**

Table 4.1: $(N, 4)$-TopSpin results using cost-splitting.

When $N = 12$ and $N = 16$ the best performance is achieved by h_{max} based on a pair of abstractions each having $\frac{N}{2}$ distinguished tiles. As N increases the advantage of h_{max} decreases and, when $N = 18$, h_{add} outperforms h_{max} for all abstractions used. Moreover, even for the smaller values of N h_{add} outperforms h_{max} when a set of four abstractions with $\frac{N}{4}$ distinguished tiles each is used. This is important because as N increases, memory limitations will preclude using abstractions with $\frac{N}{2}$ distinguished tiles and the only option will be to use more abstractions with fewer distinguished tiles each. The results in Table 4.1 show that h_{add} will be the method of choice in this situation.

The Pancake Puzzle with Location-based Costs

In this section, we present the experimental results on the 17-Pancake puzzle using location-based costs. The same notation as in the previous section is used to denote sets of abstractions, *e.g.* 5-6-6 denotes a set of three abstractions, with the first having tiles $(0 \ldots 4)$ as its distinguished tiles,

the second having tiles $(5 \ldots 10)$ as its distinguished tiles, and the third having tiles $(11 \ldots 16)$ as its distinguished tiles. Also as before, the heuristic for each abstraction is precomputed and stored in a pattern database. Unlike TopSpin, there are no symmetries in the Pancake puzzle that enable different abstractions to make use of the same PDB, so a set of M abstractions for the Pancake puzzle requires M different PDBs.

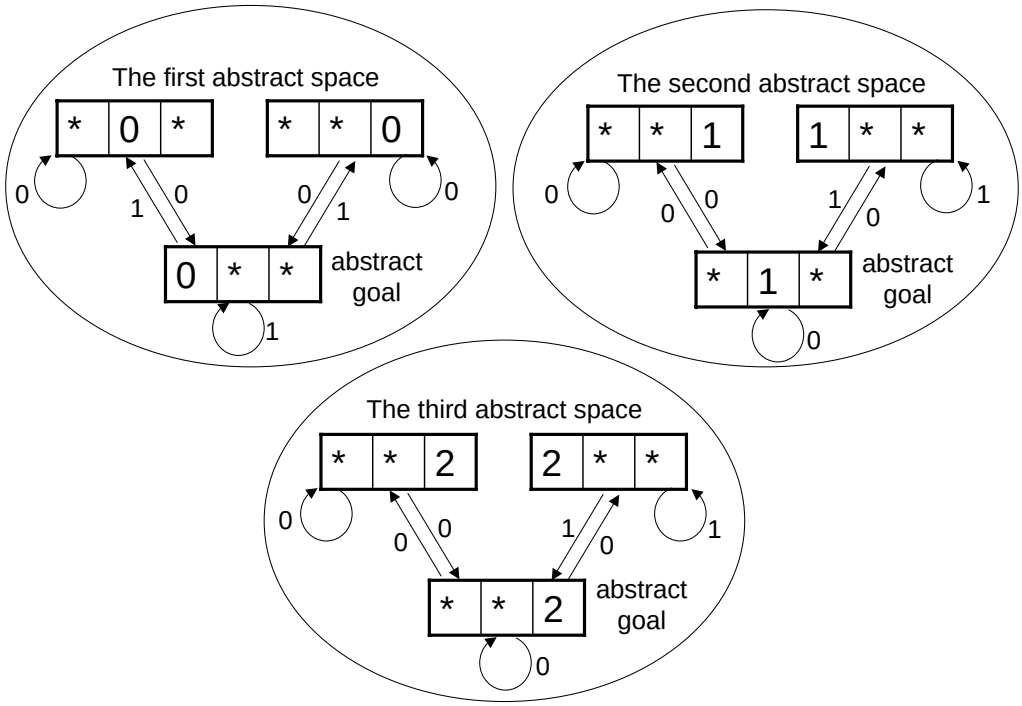


Figure 4.2: The additive abstraction defined by the location-based method for the 3-Pancake puzzle.

Additive abstractions are defined using the location-based method with just one reference location, the leftmost position. This position was chosen because the tile in this position changes whenever any operator is applied to any state in the original state space. As defined in Section 4.1.2, this means that every edge cost in the original space will be fully counted in some abstract space as long as each tile is a distinguished tile in some abstraction. Take the 3-Pancake puzzle for example. In Figure 4.2, there are three abstractions, each of which has only one distinguished tile. The primary cost of an edge (t_i^1, t_i^2) is 1 in the abstract space i if the tile in the leftmost position of the state t_i^1 is distinguished according to abstraction ψ_i and 0 otherwise. Consider state $t =$ | 2 | 1 | 0 |. The corresponding abstract states in the first, second and third abstract spaces in Figure 4.2 are | * | * | 0 |, | * | 1 | * | and | 2 | * | * |, respectively. Therefore the minimum primary cost from t_i to the abstract goal in each abstract state is 0, 0 and 1, respectively. As before, we use h_{add} to denote the heuristic defined by adding the values returned by the individual additive abstractions. For this example, the heuristic $h_{add}(t, g) = 0 + 0 + 1 = 1$.

Our first experiment compares *IDA** using h_{add} with the results[1] for the 17-Pancake puzzle published in [27] (shown in Table 4.2), which were obtained using a single abstraction having the rightmost seven tiles (10–16) as its distinguished tiles and an advanced search technique called Dual *IDA** (*DIDA**).[2] *DIDA** is an extension of *IDA** that exploits the fact that, when states are permutations of tiles as in the Pancake puzzle, each state s has an easily computable "dual state" s^d with the special property that inverses of paths from s to the goal are paths from s^d to the goal. If paths and their inverses cost the same, *DIDA** defines the heuristic value for state s to reach the goal t, as the maximum of $h(s,t)$ and $h(s^d,t)$, and sometimes will decide to search for a least-cost path from s^d to goal when it is looking for a path from s to goal.

The results of this experiment are shown in the top three rows of Table 4.3. The **Algorithm** column indicates the heuristic search algorithm. The **Abs** column shows the set of abstractions used to generate heuristics. The **Nodes** column shows the average number of nodes generated in solving 1000 randomly generated start states. These start states have an average solution length of 15.77. The **Time** column gives the average number of CPU seconds needed to solve these start states on an AMD Athlon(tm) 64 Processor 3700+ with 2.4 GHz clock rate and 1GB memory. The **Memory** column indicates the total size of each set of PDBs.

N	Algorithm	Abs	Average Solution Length	h based on a single large PDB		
				Nodes	Time	Memory
17	*DIDA**	rightmost-7	15.77	124,198,462	37.713	98,017,920

Table 4.2: The results[1] for the 17-Pancake puzzle [27], which were obtained using a single abstraction having the rightmost seven tiles $(10-16)$ as its distinguished tiles and an advanced search technique called Dual *IDA** (*DIDA**).

N	Algorithm	Abs	Average Solution Length	h_{add} based on Location-based Costs		
				Nodes	Time	Memory
17	*IDA**	4-4-4-5	15.77	14,610,039	4.302	913,920
17	*IDA**	5-6-6	15.77	1,064,108	0.342	18,564,000
17	*IDA**	3-7-7	15.77	1,061,383	0.383	196,039,920
17	*DIDA**	4-4-4-5	15.77	368,925	0.195	913,920
17	*DIDA**	5-6-6	15.77	44,618	0.028	18,564,000
17	*DIDA**	3-7-7	15.77	37,155	0.026	196,039,920

Table 4.3: 17-Pancake puzzle results using h_{add} based on location-based costs.

Clearly, the use of h_{add} based on location-based costs results in a very significant reduction in nodes generated compared to using a single large PDB, even when the latter has the advantage of

[1] Recently Helmert and Röger [46] improved the best results in [27].

[2] In particular, *DIDA** with the "*jump if larger*" (JIL) policy and the bidirectional pathmax method (BPMX) to propagate the inconsistent heuristic values that arise during dual search. See [27] for details. BPMX was first introduced in [28].

being used by a more sophisticated search algorithm. Note that the total memory needed for the 4-4-4-5 PDBs is only one percent of the memory needed for the rightmost-7 PDB, and yet *IDA** with 4-4-4-5 generates 8.5 times fewer nodes than *DIDA**with the rightmost-7 PDB. Getting excellent search performance from a very small PDB is especially important in situations where the cost of computing the PDBs must be taken into account in addition to the cost of problem-solving [51].

The memory requirements increase significantly when abstractions contain more distinguished tiles, but in this experiment the improvement of the running time does not increase accordingly. For example, the 3-7-7 PDBs use ten times more memory than the 5-6-6 PDBs, but the running time is almost the same. This is because the 5-6-6 PDBs are so accurate there is little room to improve them. The average heuristic value on the start states using the 5-6-6 PDBs is 13.594, only 2.2 less than the actual average solution length. The average heuristic value using the 3-7-7 PDBs is only slightly higher (13.628).

The last three rows in Table 4.3 show the results when h_{add} with location-based costs is used in conjunction with *DIDA**. These results show that combining our additive abstractions with state-of-the-art search techniques results in further significant reductions in nodes generated and CPU time. For example, the 5-6-6 PDBs use only 1/5 of the memory of the rightmost-7 PDB but reduce the number of nodes generated by *DIDA** by a factor of 2783 and the CPU time by a factor of 1347.

To compare h_{add} to h_{max} we ran plain *IDA** with h_{max} on the same 1000 start states, with a time limit for each start state ten times greater than the time needed to solve the start state using h_{add}. With this time limit only 63 of the 1000 start states could be solved with h_{max} using the 3-7-7 abstraction, only 5 could be solved with h_{max} using the 5-6-6 abstraction, and only 3 could be solved with h_{max} using the 4-4-4-5 abstraction. To determine if h_{add}'s superiority over h_{max} for location-based costs on this puzzle could have been predicted using Lemma 3.2.6, we generated 100 million random 17-Pancake puzzle states and tested how many satisfied the requirements of Lemma 3.2.6. Over 98% of the states satisfied those requirements for the 3-7-7 abstraction, and over 99.8% of the states satisfied its requirements for the 5-6-6 and 4-4-4-5 abstractions.

4.2.2 Negative Results

Not all of our experiments yielded positive results. Here we explore some trials where our additive approaches did not perform as well. By examining some of these cases closely, we shed light on the conditions which might indicate when these approaches will be useful.

TopSpin with Location-Based Costs

In this experiment, we used the 6-6-6 abstraction of $(18, 4)$-TopSpin as in Section 4.2.1 but with location-based costs instead of cost-splitting. The primary cost of operator a, the operator that reverses the segment consisting of locations a to $a + 3$ ($modulo$ 18), is 1 in abstract space i if the tile in location a before the operator is applied is distinguished according to abstraction ψ_i and 0

otherwise.

This definition of costs was disastrous, resulting in $C_i^*(t_i, g_i) = 0$ for all abstract states in all abstractions. In other words, in finding a least-cost path it was never necessary to use operator a when there was a distinguished tile in location a. It was always possible to move towards the goal by applying another operator, a', with a primary cost of 0. To illustrate how this is possible, consider state | 0 | 4 | 5 | 6 | 3 | 2 | 1 | of $(7, 4)$-$TopSpin$. This state can be transformed into the goal in a single move: the operator that reverses the four tiles starting with tile 3 produces the state | 3 | 4 | 5 | 6 | 0 | 1 | 2 | which is equal to the goal state when it is cyclically shifted to put 0 into the leftmost position. With the 4-3 abstraction this move has a primary cost of 0 in the abstract space based on tiles 4...6, but it would have a primary cost of 1 in the abstract space based on tiles 0...3 (because tile 3 is in the leftmost location changed by the operator). However the following sequence maps tiles 0...3 to their goal locations and has a primary cost of 0 in this abstract space (because a "don't care" tile is always moved from the reference location):

0	*	*	*	3	2	1
0	*	*	1	2	3	*
0	*	3	2	1	*	*
0	1	2	3	*	*	*

The Pancake Puzzle with Cost-Splitting

Table 4.4 compares h_{add} and h_{max} on the 13-Pancake puzzle when costs are defined using cost-splitting. The memory is greater for h_{add} than h_{max} because the fractional entries that cost-splitting produces require more bits per entry than the small integer values stored in the h_{max} PDB. In terms of both run-time and number of nodes generated, h_{add} is inferior to h_{max} for these costs, the opposite of what was seen in Section 4.2.1 using location-based costs.

N	Abs	Average Solution Length	h_{max}		h_{add} based on costing-splitting	
			Nodes	Time	Nodes	Time
13	6-7	11.791	166,479	0.0466	1,218,903	0.3622

Table 4.4: h_{add} vs. h_{max} on the 13-Pancake puzzle.

Cost-splitting, as we have defined it for the Pancake puzzle, adversely affects h_{add} because it enables each individual abstraction to get artificially low estimates of the cost of solving its distinguished tiles by increasing the number of "don't care" tiles that are moved. For example, with cost-splitting the least-cost sequence of operators to get tile "0" into its goal position from abstract state | * | 0 | * | * | * | is not the obvious single move of reversing the first two positions. That move costs $\frac{1}{2}$, whereas the 2-move sequence that reverses the entire state and then reverses the first four positions costs only $\frac{1}{5} + \frac{1}{4}$.

As a specific example, consider state | 7 | 4 | 5 | 6 | 3 | 8 | 0 | 10 | 9 | 2 | 1 | 11 | of the 12-Pancake puzzle. Using the 6-6 abstractions, the minimum number of moves to get tiles 0–5 into their goal positions is 8, and for 6–11 it is 7, where in each case we ignore the final locations of the other tiles. Thus, h_{max} is 8. By contrast, h_{add} is 6.918, which is less than even the smaller of the two numbers used to define h_{max}. The two move sequences whose costs are added to compute h_{add} for this state each have slightly more moves than the corresponding sequences on which h_{max} is based (10 and 9 compared to 8 and 7), but involve more than twice as many "don't care" tiles (45 and 44 compared to 11 and 17) and so are less costly.

There is hope that this pathological situation can be detected, at least sometimes, by inspecting the residual costs. If the residual costs are defined to be complementary to the primary costs (*i.e.* $R_i(\pi_i) = C(\pi) - C_i(\pi_i)$), as we have done, then decreasing the primary cost increases the residual cost. If the residual cost is sufficiently large in one of the abstract spaces the conditions of Lemma 5.2.2 will be satisfied, signalling that the value returned by h_{add} is provably too low. This is the subject of Chapter 5, on "infeasibility".

4.3 Chapter Summary

In this chapter, we have presented two specific, practical methods for defining abstract costs, cost-splitting and location-based costs.

These methods were applied to standard state spaces that did not have additive abstractions according to previous definitions: TopSpin, and the Pancake puzzle. Additive abstractions using cost-splitting reduce search time substantially for (18,4)-TopSpin and additive abstractions using location-based costs vastly reduce the node generations for the 17-Pancake puzzle over the state of the art.[3]

We also report negative results, demonstrating that additive abstractions are not always superior to the standard, maximum-based method for combining multiple abstractions. By exploring some trials and investigating the results, we shed light on the contributions which might indicate when these methods will be useful.

[3]Note that our best results are still superior to the best results known for the 17-Pancake puzzle in the recent publication [46].

Chapter 5

Infeasibility

This chapter explores a new way to improve the quality of heuristic values defined by additive abstractions in some circumstances. More memory is needed to store extra information when applying this technique. Given additional memory, is it a good choice to check for infeasibility? What is infeasibility? How can we identify infeasibility? Is this technique effective for different problem domains? These questions guide us to explore infeasibility further. Comparative experimental results show the potential benefits of this technique. The work has been published in [93, 95, 96]

5.1 What is Infeasibility?

Given a state t and a goal g, the heuristic value $h(t, g)$ is infeasible if it is proved that the cost of a solution for t cannot be $h(t, g)$. An example of infeasibility occurs with the Manhattan Distance (*MD*) heuristic for the sliding tile puzzle. It is well-known that the parity of $MD(t)$ is the same as the parity of the optimal solution cost for state t. If some other heuristic for the sliding tile puzzle returns a value of the opposite parity, it can safely be increased until it has the correct parity.

5.2 The Approach to Identify Infeasibility

The key to the approach is to identify "infeasible" values—ones that cannot possibly be the optimal solution cost. The example in Section 5.1 relies on specific properties of the *MD* heuristic and the puzzle. Following the notations in Section 3.2, $C_i^*(t_i, g_i)$ represents the minimum primary cost of a path in the abstract space from t_i to g_i, $\vec{P}_i(t_i, g_i)$ is the set of abstract paths from t_i to g_i whose primary cost is minimum, and $R_j^*(t_j, g_j)$ is the minimum residual cost among the paths in $\vec{P}_i(t_i, g_i)$. The following lemmas give a problem-independent method for testing infeasibility.

Lemma 5.2.1. *Given an additive $\langle \mathcal{S}, \aleph \rangle$ and a path $\vec{p} \in Paths(\mathcal{S}, t, g)$ with $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$, then $\sum_{i=1}^{k} C_i^*(t_i, g_i) \geq C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$ for all $j \in \{1, \ldots, k\}$.*

Proof. By Lemma 3.2.1, $C(\vec{p}) \geq C_j(\psi_j(\vec{p})) + R_j(\psi_j(\vec{p}))$ for all $j \in \{1, \ldots, k\}$. By Lemma 3.2.7 $C_j(\psi_j(\vec{p})) = C_j^*(t_j, g_j)$, and by Lemma 3.2.8 $R_j(\psi_j(\vec{p})) \geq R_j^*(t_j, g_j)$. Therefore $C(\vec{p}) \geq$

$C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$, and the lemma follows from the premise that $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$. □

Lemma 5.2.2. *Let $\langle \mathcal{S}, \aleph \rangle$ be any additive abstraction system and let $t, g \in T$ be any states. If $h_{add}(t, g) < C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$ for some $j \in \{1, \ldots, k\}$, then $h_{add}(t, g) \neq OPT(t, g)$.*

Proof. This lemma follows directly as the contrapositive of Lemma 5.2.1. □

As Lemma 5.2.2 shows, there is an interesting consequence when this condition fails for state t: we know that the value returned by h_{add} for t is not the true cost to reach the goal from t. Detecting this is useful because it allows the heuristic value to be increased without risking it becoming inadmissible.

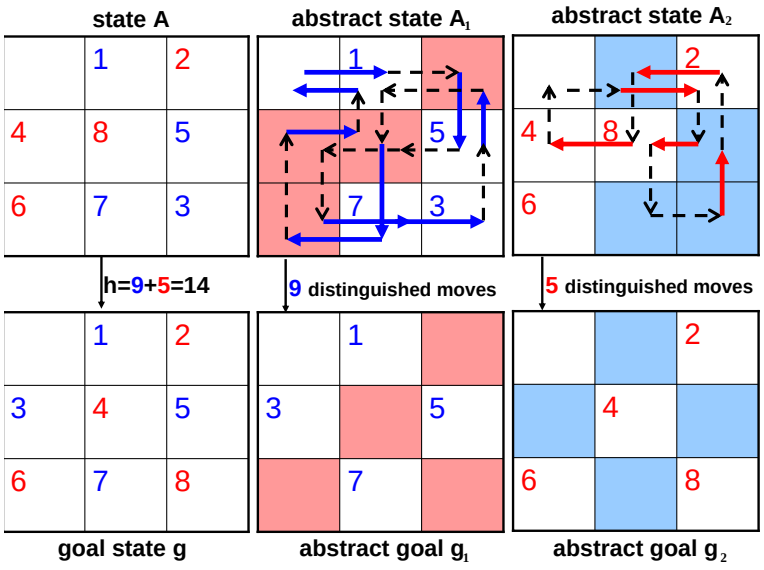


Figure 5.1: Computation of the additive heuristic value for state A. Solid arrows denote distinguished moves, and dashed arrows denote "don't care" moves.

To illustrate how infeasibility can be detected using Lemma 5.2.2 consider state A of the 3×3 sliding tile puzzle shown in the top left of Figure 5.1 and the goal state g shown below it. The middle column shows an abstraction of these two states (A_1 and g_1) in which tiles 1, 3, 5, and 7 are distinguished tiles while the other tiles are "don't care" tiles. The right column shows the complementary abstraction, in which tiles 1, 3, 5, and 7 are the "don't care" tiles and tiles 2, 4, 6, and 8 are distinguished.

The arrows in the figure trace out a least-cost path with the conditional optimal residual cost to reach the abstract goal g_i from state A_i in each abstraction.

As shown in the middle column of the figure it takes at least 9 distinguished moves from the abstract state A_1 to the goal state g, so $C_1^*(A_1, g) = 9$. The abstract paths that solve the problem with 9 distinguished moves require, at a minimum, 9 "don't care" moves, so $R_1^*(A_1, g) = 9$. A

similar calculation for the abstract state A_2 on the right of the figure yields $C_2^*(A_2, g) = 5$ and $R_2^*(A_2, g) = 7$. The value of $h_{add}(A, g)$ is therefore $C_1^*(A_1, g) + C_2^*(A_2, g) = 9 + 5 = 14$. This value is based on the assumption that there is a path in the original space that makes 9 moves of tiles 1, 3, 5, and 7, and 5 moves of the other tiles. However, the value of $R_1^*(A_1, g)$ tells us that any path that uses only 9 moves of tiles 1, 3, 5, and 7 to put them into their goal locations must make at least 9 moves of the other tiles, it cannot possibly make just 5 moves. Therefore there does not exist a solution costing as little as $C_1^*(A_1) + C_2^*(A_2) = 14$. In short, $(C_1^*, R_1^*) = (9, 9), (C_2^*, R_2^*) = (5, 7)$. $h_{add} = \sum_{i=1}^{2} C_i^* < (C_1^* + R_1^*)$. So h_{add}=14 is an infeasible heuristic value and the heuristic value can be increased by one without risking it becoming inadmissible.

5.3 Improving Infeasible Heuristic Values

Given an additive abstraction system, the key to identifying infeasibility is to check whether there exists some j ($1 \leq j \leq k$) such that $h_{add}(t, g) < C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$. If there exists such a j, then $h_{add}(t, g)$ is an infeasible heuristic value. Once identified the infeasible values can be increased to give a better estimate of the solution cost. Formally, the heuristic $h_{add-check}$ is defined by $h_{add-check}(t, g) =$

$$\begin{cases} h_{add}(t, g) + \varepsilon, & \text{If } h_{add}(t, g) \text{ is identified to be infeasible.} \\ h_{add}(t, g), & \text{Otherwise.} \end{cases}$$

Generally, ε is assigned to be one for a state space with unit edge costs, or more according to some special structural property.

5.4 Experimental Results

To illustrate the potential of this method for improving additive heuristics, this section reports the results with infeasibility checking for the sliding tile puzzle, TopSpin and the pancake puzzle.

All experiments in this chapter store the values of C^* and R^* defined by each abstraction in a lookup table in the form of a pattern database, and we perform *IDA** as the heuristic search algorithm. Algorithms are coded in C on a machine with an AMD Athlon(tm) 64 Processor 3700+ with 2.4 GHz clock rate and 1GB memory.

In Tables 5.1-5.3, the **N** column shows the size of the problem. The **Abs** column shows the set of abstractions used to generate heuristics. The "No Infeasibility Check" columns show the results of h_{add} and the "Infeasibility Check" columns present the results of $h_{add-check}$. The **Nodes** column shows the average number of nodes generated in solving 1000 randomly generated start states. The **Time** column gives the average number of CPU seconds needed to solve these start states.

5.4.1 The Sliding Tile Puzzle

We experimented with the standard 15-puzzle and the glued 15-puzzle [73] which is a variation of the traditional 15-Puzzle. In the glued 15-Puzzle, a particular tile is glued to the board in its goal position. During the search the glued tile is fixed and only 14 tiles can be moved. Therefore the solution path must work its way around an extra constraint.

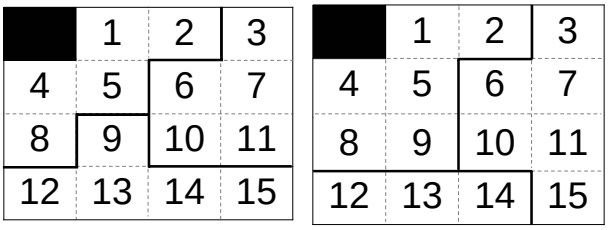


Figure 5.2: Different tile partitionings for the 15-puzzle. Left: 5-5-5 partitioning. Right: 6-6-3 partitioning.

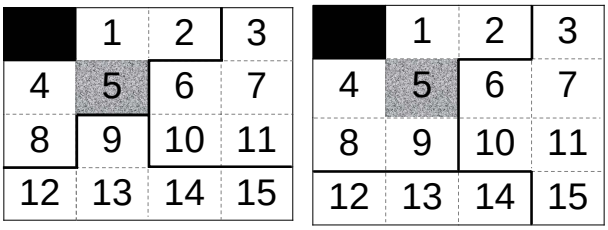


Figure 5.3: Different tile partitionings for a glued 15-puzzle. Left: 4-5-5 partitioning. Right: 5-6-3 partitioning. The shaded square indicates that Tile 5 is glued to the board in its goal position.

N	Abs	Average Solution Length	h_{add} based on zero-one cost-splitting			
			No Infeasibility Check		Infeasibility Check	
			Nodes	Time	Nodes	Time
15	5-5-5	52.522	2,237,899	0.552	912,661	0.340
15	6-6-3	52.522	1,261,566	0.336	479,781	0.173
14	4-5-5	53.280	2,945,864	0.594	996,210	0.230
14	5-6-3	53.280	2,185,207	0.457	739,990	0.183

Table 5.1: The effect of infeasibility checking on the 15-puzzle (N=15) and the glued 15-puzzle (N=14).

Table 5.1 presents average results of *IDA** solving 1000 test instances of the 15-puzzle and a glued 15-puzzle using different tile partitionings (shown in Figures 5.2-5.3) and costs defined by the method described in Section 4.1.

In Table 5.1 the first two rows are results on the 15-puzzle and the last two rows are results on the glued 15-puzzle. As can be seen infeasibility checking reduces the number of nodes generated and the CPU time by over a factor of 2. However, there is a space penalty for this improvement,

because the R^* values must be stored in the pattern database in addition to the normal C^* values. This increases the amount of memory required, and it is not clear if storing R^* is the best way to use this extra memory. This experiment merely shows that infeasibility checking is one way to use extra memory to speed up search for some problems.

The blank is always regarded as a distinguished tile for each abstraction since there is sufficient memory to store these pattern databases. It leads to the results that the performance of original h_{add} reported in Table 5.1 is better than that reported in the previous work which compressed at the blank [26, 96].

5.4.2 The TopSpin Puzzle

N	Abs	Average Solution Length	h_{add} based on costing-splitting			
			No Infeasibility Check		Infeasibility Check	
			Nodes	Time	Nodes	Time
12	6-6	9.138	53,460	0.16	20,229	0.07
12	4-4-4	9.138	346,446	1.33	174,293	0.62
12	3-3-3-3	9.138	1,388,183	6.44	1,078,853	4.90
16	8-8	14.040	2,137,740	4.74	705,790	1.80
16	4-4-4-4	14.040	251,946,069	851.00	203,213,736	772.04
18	6-6-6	17.000	879,249,695	4,713.00	508,851,444	2,846.52

Table 5.2: The effect of infeasibility checking on $(N, 4)$-TopSpin using cost-splitting.

Table 5.2 presents the results obtained for the $(N, 4)$-TopSpin puzzle with costs defined by cost-splitting that is described in Section 4.2.1. The "No Infeasibility Check" columns in Table 5.2 are the same as the "h_{add} based on cost-splitting" columns of the corresponding rows in Table 4.1. Comparing these to the "Infeasibility Check" columns shows that in most cases infeasibility checking reduces the number of nodes generated and the CPU time by roughly a factor of 2.

When location-based costs are used with TopSpin infeasibility checking adds one to the heuristic value of almost every state. However, this simply means that most states have a heuristic value of 1 instead of 0 (recall the discussion in Section 4.2.2), which is still a very poor heuristic.

5.4.3 The Pancake Puzzle

Infeasibility checking produces almost no benefit for the 17-Pancake puzzle with location-based costs because the conditions of Lemma 5.2.2 are almost never satisfied. The experiment discussed at the end of Section 4.2.1 showed that fewer than 2% of the states satisfy the conditions of Lemma 5.2.2 for the 3-7-7 abstraction, and fewer than 0.2% of the states satisfy the conditions of Lemma 5.2.2 for the 5-6-6 and 4-4-4-5 abstractions.

Infeasibility checking for the 13-Pancake puzzle with cost-splitting also produces very little benefit, but for a different reason. For example, Table 5.3 shows the effect of infeasibility checking

on the 13-Pancake puzzle; the results shown are averages over 1000 start states. Cost-splitting in this state space produces fractional edge costs that are multiples of $\frac{1}{360360}$ (360360 is the Least Common Multiple of the integers from 1 to 13 and as mentioned in Section 4.1.1 we simulate the rational numbers using integers obtained by scaling by the Least Common Multiple of the denominators to avoid round off errors in our implementation), and therefore if infeasibility is detected the amount added is $\frac{1}{360360}$. But recall that h_{add}, with cost-splitting, is defined as the ceiling of $\sum_{i=1}^{k} C_i(\pi_i)$. The value of h_{add} will therefore be the same, whether $\frac{1}{360360}$ is added or not, unless the sum of the $C_i(\pi_i)$ is exactly an integer. As Table 5.3 shows, this does happen but only rarely.

N	Abs	Average Solution Length	h_{add} based on costing-splitting			
			No Infeasibility Check		Infeasibility Check	
			Nodes	Time	Nodes	Time
13	6-7	11.791	1,218,903	0.3622	1,218,789	0.4453

Table 5.3: The effect of infeasibility checking on the 13-Pancake puzzle using cost-splitting.

5.5 Chapter Summary

The main contribution of this chapter is to show that the accuracy of the heuristics generated by abstraction can be improved by checking for infeasibility. The theory and experiments shed some light on the question of how to detect infeasibility of h_{add} and how to apply this technique to different domains. Given additional memory, the new technique to identify infeasibility can enhance the search performance of heuristic search.

Generally, we can safely add one to an infeasible heuristic value for problems with unit edge cost. But this improvement seems weak when most of the edges cost more than one, and it is also weak with h_{add} based on cost-splitting for the Pancake puzzle.

Chapter 6

Additive Abstractions for Problems with Non-uniform Edge Costs

Our previous results have shown that additive abstraction-based heuristics can reduce search time substantially for some standard state spaces of combinatorial problems (e.g. (18,4)-TopSpin and the 17-Pancake puzzle). In these spaces edge costs were all one. A great number of industry applications can be modelled as problems of searching with non-uniform edge costs. In this chapter we will explore good additive abstractions in the state spaces where the edge cost is non-uniform. The Sequential Ordering Problem (SOP) and the weighted Pancake puzzle are chosen as two testbeds in this chapter. Although these two problems are on integer domains, real-valued edge costs are permissible provided conditions are imposed to ensure that the set of paths connecting any given pair of states has a well-defined minimum cost.

In the remainder of this chapter, we first present the definitions and background of the SOP (see Section 6.1); then we investigate the structure of the SOP instances in Section 6.2 and design greedy methods of choosing good abstractions in Sections 6.3 and 6.4; the experimental results (see Section 6.6) show that well chosen abstractions can enhance the quality of suboptimal solutions for large SOP instances. In Section 6.6 we then design a new weighted problem based on the Pancake puzzle. Recall our additive approaches performed very strongly on the unweighted version. Although this problem has a very different structure than the SOP, we transfer the techniques for the SOP and show that there is some generality to our ideas.

6.1 Definition and Background of the SOP

The Sequential Ordering Problem (SOP) is a problem of searching with non-uniform edge costs, and it is a model for several industrial applications, such as the stacker crane application [1] and helicopter routing between oil rigs [85].

An instance of the SOP is defined by a weighted directed graph $G = \langle V, E, C, P \rangle$ where

- V is a finite set of vertices with the start s and goal g vertices designated. We define the

intermediate vertices $\mathcal{I} = V \setminus \{s, g\}$. We define $n = |V|$.

- $E = (\mathcal{I} \times \mathcal{I}) \cup (\{s\} \times \mathcal{I}) \cup (\mathcal{I} \times \{g\})$ is a set of directed edges (ordered pairs of vertices) representing vertex transitions. The graph is complete, except that in-degree(s)=out-degree(g)=0.
- $C : E \longrightarrow \mathcal{N} = \{0, 1, 2, 3, ...\}$ is the edge cost function.
- P is the set of precedence constraints. A constraint $(u <_p v)$ means that vertex u must precede vertex v in any solution path. We define $P_{sg} = \bigcup_{v \in \mathcal{I}} \{(s <_p v), (v <_p g)\}$ and note that P_{sg} must always be a subset of P. We call $P' = P \setminus P_{sg}$ the *prime* constraints.

The *objective* is to find a minimum cost Hamiltonian path from s to g which does not violate P.

An SOP instance is typically represented [89] by an $n \times n$ cost matrix M, where the entry $M_{i,j}$ is the cost of the edge $i \longrightarrow j$ in G, or it is a symbol to represent the constraint. In this chapter, ∞ is used to represent the constraint, although in some papers, -1 is used.

The SOP is a variant of the Asymmetric Traveling Salesman Problem (ATSP) in which precedence constraints can be specified. However, a number of ATSP instances with hundreds of vertices can be solved easily, while there are several SOP instances with less than 55 vertices which have not been solved optimally in the Traveling Salesman Problem Library (TSPLIB) [87, 89].

In the following sections, first we present the formal definitions for the SOP state space and abstraction, then we discuss both standard heuristics and abstraction-based heuristics. Representative examples are studied to explore the situations when the abstraction-based heuristics are superior (or inferior) to the standard heuristics.

6.1.1 State Space for Standard Search Algorithms

Standard search algorithms (e.g. DFBB, A*, and IDA*) for the SOP problem attempt to construct a path from s to g by adding one vertex at a time. In the underlying state space of such searches, a state on the k^{th} level is a pair $(\overrightarrow{p}, \mathcal{U})$ where $\overrightarrow{p} = \langle v_1, v_2, ..., v_k \rangle$ ($v_1 = s$ and $v_i \in \mathcal{I}, i \in \{2, ..., k\}$) is a sequence of vertices that have been visited, and $\mathcal{U} = \mathcal{I} \setminus \{v_2, ..., v_k\}$ is the set of unvisited intermediate vertices. If and only if $\mathcal{U} = \emptyset$ may we add the goal vertex to the path. The initial and goal states of the space are $(\langle s \rangle, \mathcal{I})$ and $(\langle v_1 = s, v_2, ...v_{n-1}, v_n = g \rangle, \emptyset)$. There are n levels and $(k-1)!\binom{n-2}{k-1}$ intermediate states are on the k^{th} level ($k \in [2, n-1]$) since $\overrightarrow{p}$ may be any permutation of the $(k-1)$ intermediate vertices. However some of the intermediate states and goal states may not be reachable without violating the constraints.

Consider a state $(\langle v_1, v_2, ..., v_k \rangle, \mathcal{U})$. If there exists $i < j$ such that $(v_j <_p v_i) \in P$ then the path represents a *current violation* of the constraints no matter how the continuation is made. The cost of reaching a state $(\overrightarrow{p}, \mathcal{U})$ from s is infinity if the path is in current violation, otherwise the cost is the sum of the edge weights along the path $\overrightarrow{p}$.

During search, we are interested in the "cost to go" of a state, that is we want to know how much it costs to reach a goal state from the current state, in order to choose our next move wisely, or

backtrack and try a different path. A state $(\overrightarrow{p},\mathcal{U})$ represents a *forward violation* if $\exists v \in \overrightarrow{p}, \exists u \in \mathcal{U}$ with $(u <_p v) \in P$. If a state is in forward violation then the cost of reaching g is infinite. Note that this condition can be determined independently of the order of the vertices in $\overrightarrow{p}$. This motivates in part the abstraction we develop in the next section.

6.1.2 The Dynamic Programming State Space

The state space described in this section abstracts the original search space in that various paths may map to the same state in the dynamic programming state space. It is nearly identical to that introduced by [5, 6, 40], and is an intermediate step towards our goal.

In this abstraction, a *state* is a pair $(\ell,\mathcal{U})$ where $\ell \in \mathcal{I}$ is the last vertex visited , and $\mathcal{U} \subseteq \mathcal{I} \setminus \{\ell\}$ is the set of unvisited intermediate vertices; thus the members of $\mathcal{U}$ are candidates for the next vertex to be visited. If and only if $\mathcal{U} = \emptyset$ may we add the goal vertex to the path. The initial and goal states of the space are $(s,\mathcal{I})$ and$(g,\emptyset)$.

Note that this abstraction merges states of the original search space depending only on the last vertex visited and the remaining unvisited vertices, and so in effect merges all original states in which the path ending in ℓ is a permutation of the set $p = \mathcal{I} \setminus (\mathcal{U} \cup \{\ell\})$. We define that a state $(\ell,\mathcal{U})$ represents a *detectable violation* of P if $\exists y \in \{\ell\} \cup \mathcal{U}, \exists x \in \mathcal{I} \setminus \mathcal{U}, (y <_p x) \in P$. Note that a detectable violation may represent a current violation or a forward violation of the original state space.

The forward cost of a state to reach the goal can be defined as

$$cost(\ell,\mathcal{U}) = \begin{cases} \infty, \ (\ell,\mathcal{U}) \text{ represents a detectable violation} \\ M_{\ell,g}, \ \ \mathcal{U} = \emptyset \\ min_{r \in \mathcal{U}}\{M_{\ell,r} + cost(r,\mathcal{U} \setminus \{r\})\}, \text{otherwise} \end{cases}$$

An intermediate state in the dynamic programming state space is a state $(\ell,\mathcal{U})$ when $\ell \in \mathcal{I}$. For intermediate states, since $\ell \in \mathcal{I}$, $\mathcal{U}$ may be any subset of the remaining $n-3$ intermediate states. Thus, there are $(n-2)2^{n-3}$ intermediate states some of which may not be reachable without violating the constraints. Although this represents a significant reduction over the size of the original search space, it nevertheless grows quickly on instances with more than 20 vertices and it is not practical to search this state space exhaustively.

6.1.3 State Space Abstraction

Next we introduce a cost preserving homomorphism to take the abstraction a step further. The method we use for the non-additive abstractions is similar to those used by [48, 49].

In this abstraction, we partition $\mathcal{I}$ into equivalence classes. Vertices in classes with one element are *distinguished vertices*, while vertices in larger classes are *don't-care vertices*. For each abstraction, the *distinguished set* is the set of all distinguished vertices in $\mathcal{I}$, and the *don't-care set* is the set of all vertices in $\mathcal{I}$ that belong to don't-care vertices.

We choose a set $\mathcal{C}_\mathcal{R}$ of canonical representatives, with one element from each equivalence class of the partition. For example, suppose V has 6 elements with $s = 1, g = 6$ and $\mathcal{I} = \{2, 3, 4, 5\}$. If we partition $\mathcal{I}$ as $\{\{2\}, \{3\}, \{4, 5\}\}$, then we " don't care" which of the vertices $4, 5$ are visited, only how many of this subset. If vertex 4 is chosen to be the representative of the class $\{4, 5\}$, $\mathcal{C}_\mathcal{R} = \{2, 3, 4\}$, the distinguished set is $\{2, 3\}$ and the don't-care set is $\{4, 5\}$. For each vertex u, we let Ω_u be the equivalence class containing u. In the above example $\Omega_4 = \{4, 5\}$.

Later we may have multiple abstractions, and so have several such partitions. Supposing this is the i^{th}, we define an abstract mapping $\psi_i(u) = x \in (\Omega_u \cap {\mathcal{C}_\mathcal{R}}^i)$ (i.e., the canonical representative of u in the i^{th} abstraction) with $\psi_i(s) = s$ and $\psi_i(g) = g$.

This mapping induces a new instance with $\psi_i(V) = V^i = {\mathcal{C}_\mathcal{R}}^i \cup \{s, g\}$. For $x, y \in V^i$ we define the abstract edge cost by

$$M^i_{x,y} = \min_{\psi_i(u)=x, \psi_i(v)=y} \{M_{u,v}\}$$

Therefore $M^i_{x,y} = \infty$ only if $M_{u,v} = \infty, \forall u, v$ where $\psi_i(u) = x, \psi_i(v) = y$. Recall that $M^i_{x,y} = \infty$ represents the constraint $(y <_p x)$.

We now apply this abstraction to the dynamic programming state space, that is $\psi_i(\ell, \mathcal{U}) = (\psi_i(\ell), \psi_i(\mathcal{U}))$. Note that $\psi_i(\mathcal{U})$ may contain multiple copies of the same element. For example, a state $(2, \{3, 4, 5\})$ may map to an abstract state $\psi_i(2, \{3, 4, 5\}) = (2, \{3, 4, 4\})$ if we define that the distinguished set is $\{2, 3\}$, the don't-care set is $\{4, 5\}$ and the canonical representative of $\{4, 5\}$ is vertex 4. We define a *distinguished state* as an abstract state $(\psi_i(\ell), \psi_i(\mathcal{U}))$ when the intermediate vertex ℓ is a distinguished vertex in the i^{th} abstraction. A *don't-care state* is an abstract state $(\psi_i(\ell), \psi_i(\mathcal{U}))$ when the intermediate vertex ℓ is a don't-care vertex in the i^{th} abstraction.

We define that an abstract state $\psi_i(\ell, \mathcal{U})$ represents a *detectable violation* if $\exists y \in \psi_i(\{\ell\} \cup \mathcal{U})$, $\exists x \in \psi_i(\mathcal{I} \setminus \mathcal{U})$, $M^i_{x,y} = \infty$ (i.e., $y <_p x$).

This abstraction may hide some precedence constraints, but the heuristic will remain admissible. The vertex cost of the generic abstraction heuristic is then defined as

$$cost(\psi_i(\ell, \mathcal{U})) = \begin{cases} \infty, \ \ \psi_i(\ell, \mathcal{U}) \text{ is in detectable violation} \\ M^i_{\psi_i(\ell), g}, \ \ \psi_i(\mathcal{U}) = \emptyset \\ \min_{u \in \mathcal{U}} \{M^i_{\psi_i(\ell), \psi_i(u)} + cost(\psi_i(u, \mathcal{U} \setminus \{u\})\}, \text{otherwise} \end{cases}$$

Clearly, the heuristic $h_{max}(\ell, \mathcal{U}) = \max_i cost(\psi_i(\ell, \mathcal{U}))$ is an admissible heuristic.

Indexing schemes for rank and unrank

In order to store and get heuristic values more efficiently, it is necessary to define an indexing scheme for abstract states. For indexing purposes, we consider the states of the abstract space in blocks defined by the value of ℓ in the state $(\ell, \mathcal{U})$. Let b_i equal 1 plus the size of the ith equivalence class. Then the number of states in the block defined by ℓ is $B_\ell = (b_\ell - 1) \prod_{i \in \mathcal{C}_\mathcal{R}, i \neq \ell} b_i$ and the total number of intermediate abstract states is $\sum_{\ell \in \mathcal{C}_\mathcal{R}} B_\ell$. We lexigraphically sort states in the same

ℓ	$\mathcal{U}$	rank($\ell, \mathcal{U}$)
2	$\{\}$	0
2	$\{3\}$	1
2	$\{4\}$	2
2	$\{3,4\}$	3
2	$\{4,4\}$	4
2	$\{3,4,4\}$	5
3	$\{\}$	6
3	$\{2\}$	7
3	$\{4\}$	8
3	$\{2,4\}$	9
3	$\{4,4\}$	10
3	$\{2,4,4\}$	11
4	$\{\}$	12
4	$\{2\}$	13
4	$\{3\}$	14
4	$\{2,3\}$	15
4	$\{4\}$	16
4	$\{2,4\}$	17
4	$\{3,4\}$	18
4	$\{2,3,4\}$	19

Table 6.1: An example to rank the intermediate states in the abstract state space.

block B_ℓ and sort blocks in order of the increasing value of ℓ. These lead to our indexing schemes for rank and unrank.

We refer to the example at the beginning of Section 6.1.3, $n = 6$, $\mathcal{I} = \{\{2\},\{3\},\{4,5\}\}$, and $\mathcal{C}_\mathcal{R} = \{2,3,4\}$. Table 6.1 shows the method used to order all the intermediate states. In this example, the total number of states $\sum_{\ell \in \mathcal{C}_\mathcal{R}} B_\ell = B_2 + B_3 + B_4 = (b_2 - 1) \times b_3 \times b_4 + (b_3 - 1) \times b_2 \times b_4 + (b_4 - 1) \times b_2 \times b_3$. Since $b_2 = 2$, $b_3 = 2$ and $b_4 = 3$, $\sum_{\ell \in \mathcal{C}_\mathcal{R}} B_\ell = 20$ and the rank ranges from 0 to 19.

As in the example, in this chapter we will have k distinguished vertices, and the remaining $n - 2 - k$ vertices in $\mathcal{I}$ will be in a single don't care class. The total number of distinguished states is $k(n - 2 - k + 1)2^{k-1}$ and the total number of don't-care states is $(n - 3 - k + 1)2^k$. Thus, the total number of abstract states is $k(n - 1 - k)2^{k-1} + (n - 2 - k)2^k$.

6.1.4 Additive Abstractions

In Chapter 4, one successful method of meeting the conditions of an additive abstraction system was the location-based method. For a combinatorial problem, a special variable in the abstract representation is chosen, and when that value is distinguished, the cost is assigned to the primary cost of the abstract state space, otherwise the primary cost is zero. The obvious choice for the distinguished variable here is ℓ in the state $(\ell, \mathcal{U})$.

We have two obvious choices: either we charge the cost of a move when we step from a distinguished state; or when a distinguished state is reached. Using the first, given $x, y \in V^i$ and the

distinguished set D_i ($D_i \subseteq \mathcal{I}$) in abstraction ψ_i, the primary cost of an edge $x \rightarrow y$ in additive abstraction can be defined as

$$C_i(x,y) = \begin{cases} M^i_{s,y}, & x = s, y \in D_i \\ M^i_{x,y}, & x \in D_i \\ 0 & \text{otherwise} \end{cases}$$

Thus the full cost of an original state transition is assigned to a state transition in the additive abstraction as a primary cost only if we step from a distinguished state, or we start from the start and step to a distinguished state. Although s and g are distinguished in each abstraction, the definition still guarantees a partition of the edges and each edge will be charged at most once over the set of abstractions.

We define the residual cost $R_i(x,y) = M^i_{x,y} - C_i(x,y)$.

As described in Section 6.1.3, we say that an abstract state $\psi_i(\ell,\mathcal{U})$ represents a detectable violation if $\exists y \in \psi_i(\{\ell\} \cup \mathcal{U})$, $\exists x \in \psi_i(\mathcal{I} \setminus \mathcal{U})$, $M^i_{x,y} = \infty$ (i.e., $y <_p x$).

Therefore the primary forward cost of an abstract state is then defined as

$$C_i(\psi_i(\ell,\mathcal{U})) = \begin{cases} \infty, \ \ \psi_i(\ell,\mathcal{U}) \text{ is in detectable violation} \\ C_i(\psi_i(\ell), g), \ \ \psi_i(\mathcal{U}) = \emptyset \\ \min_{u \in \mathcal{U}}\{C_i(\psi_i(\ell), \psi_i(u)) + C_i(\psi_i(u, \mathcal{U} \setminus \{u\})\}, \text{otherwise} \end{cases}$$

The abstractions are chosen so that the sets of distinguished vertices in the different abstractions partition $\mathcal{I}$ (e.g. $D_i \cap D_j = \emptyset, \forall i \neq j$). Therefore each intermediate vertex is distinguished in at most one abstraction and each edge can be charged at most once over the set of abstractions. It is clear that our definitions of costs satisfy properties $(P1)$, $(P2)$ and $(P3)$ of the theory summarized in Section 3.3 and therefore $h_{add}(\ell,\mathcal{U}) = \sum_i C_i(\psi_i(\ell,\mathcal{U}))$ is an admissible heuristic.

6.1.5 Depth-First Branch and Bound Algorithms

In the SOP state space, edges have arbitrary (non-negative) edge costs. IDA* might be inefficient when searching this state space because there might be too many distinct f-values, and A* cannot be used due to the limitation of memory. Hence we apply Depth-First Branch and Bound (DFBB) to the SOP.

Depth-First Branch and Bound (DFBB) has an important feature that it has a solution ready whenever it stops (with possible exception of an initial time period before the first solution is found) and the quality of the solution improves with additional computation time. This feature is very useful for problem-solving under varying or uncertain time constraints. For example, some SOP problems cannot be solved exactly with limited computation time, and in some circumstances we do not need optimal solutions but rather good ones that can be found quickly.

Figure 6.1 gives the pseudocode of DFBB. It starts at the root node with a global upper bound u on the cost of an optimal solution. Whenever a leaf node is reached whose cost is less than u, u is updated to the cost of this new leaf. Whenever a state t is selected for expansion a cost $f(t) = g(t) + h(t, goal)$ is computed, where $g(t)$ is the cost to reach state t, and $h(t, goal)$ is a

Initialized u
DFBB (node t**)**
    IF (t is a goal node)
        IF ($g(t) <$ u)
            $u \longleftarrow g(t)$
        END IF
        RETURN
    END IF
    Generate all n_t children of t
    Evaluate and **Sort** t's children into order $t_1, t_2, ..., t_{n_t}$
    FOR (i from 1 to n_t)
        IF ($f(t_i) < u$)
            DFBB(t_i)
        END IF
    END FOR
    RETURN

Figure 6.1: Depth First Branch and Bound.

lower bound (heuristic) on the cost from state t to a goal state $goal$. When $f(t) \geq u$, this branch is pruned, and thus the search will typically only visit some subset of the underlying space, depending on the accuracy of the heuristic.

Figure 6.2 illustrates DFBB used to solve the SOP. A branch is also pruned when a state represents a violation. Given a state $(\ell, \mathcal{U})$ in the dynamic programming state space, the violation is detected by the following conditions.

- If $\exists x \in \mathcal{I} \setminus (\mathcal{U} \cup \{\ell\})$ with $(\ell <_p x) \in P$, then the state represents a violation. (Note that we only detect a part of *current violations* defined in Section 6.1.1.)
- If $\exists x \in \mathcal{I} \setminus \mathcal{U}$ and $y \in \mathcal{U}$, such that $(y <_p x) \in P$ then the state represents a violation.

The early survey by Lawler and Wood [74] described the essential features of the branch-and-bound (B&B) technique and discussed its applications to many problems in AI. Noticing that various heuristic search procedures applied in AI are considered to be related to branch-and-bound approach, Nau et al. [80] unified the varying conceptions and provided a general formulation for the B&B approach. Therefore a number of heuristic search algorithms can also be considered special cases of the B&B procedure. Labat and Pomerol [72] investigated this topic further and concluded that B&B algorithms also rely on classical ideas of heuristic search. The effectiveness of B&B relies on the quality of heuristics.

DFBB applies heuristics for two main purposes: node pruning and children ordering (i.e. the branch selection). First, the heuristics are used as the lower bound to prune some nodes. Second, the children are usually sorted by their costs in increasing order ($f(t) = g(t) + h(t, goal)$), as mentioned in [97]. Thus the accuracy and the relative order are two critical issues when we consider the quality of heuristics.

Initialized u
SOP-DFBB (node t**)**
    IF (t is a goal node)
        IF ($g(t) <$ u)
            $u \longleftarrow g(t)$
        END IF
        RETURN
    END IF
    Generate all n_t children of t
    Evaluate and **Sort** t's children into order $t_1, t_2, ..., t_{n_t}$
    FOR (i from 1 to n_t)
        IF ($f(t_i) < u$ **AND** t_i does not represent a violation)
            SOP-DFBB(t_i)
        END IF
    END FOR
    RETURN

Figure 6.2: Depth First Branch and Bound for the SOP.

A number of heuristics [3, 21, 33, 49, 56, 88] used by DFBB have been proposed, compared and discussed during the past thirty years for the Traveling Salesman Problem. In addition to the focus on how to improve the accuracy, researchers [7, 32, 74, 75, 77] also recognized the importance of child ordering and concluded in various combinatorial problem domains that different ordering strategies can change the search performance of DFBB dramatically.

6.1.6 Overview of Previous Heuristics

The SOP is a variant of the Traveling Salesman Problem (TSP) which is a classical problem in combinatorial optimization studied in computer science. Given a list of cities and their pairwise distances, a *tour* is a path that starts from a city, visits each city exactly once and returns to the start city. The task of the TSP is to find a cheapest possible tour (i.e., an optimal tour). In this section we overview some basic TSP heuristics and discuss how to adapt them to the SOP.

As described in Section 6.1.5, branch and bound algorithms solve a problem by breaking it up into successively smaller subproblems, calculating bounds on the objective function value over each subproblem, and using these bounds to discard certain subproblems from further consideration. In other words, if the best solution found so far costs less than the lower bound for a subproblem, we need not explore the subproblem at all.

During search the branch and bound algorithm includes and excludes sets of edges, therefore the subproblems are also TSP problems, and the corresponding lower bounds (heuristics) can be obtained by replacing the subproblem with an easier problem.

In the TSP literature, a great number of heuristics have been proposed, improved, classified and discussed [3, 10, 21, 36, 33, 35, 41, 42, 56, 75, 79, 81, 88]. Suppose a TSP instance is associated with a graph $G = (V, E)$ in which each vertex represents a city and the weight of each edge $i \longrightarrow j$

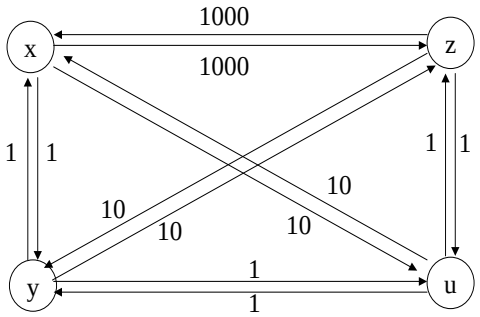
represents the cost from i to j. The goal is to find a Hamiltonian cycle with minimal cost (i.e., an optimal tour).

In the following sections, we briefly describe some basic TSP heuristics that can be used as heuristics for the branch and bound algorithm. Then we discuss how to create a version of additive abstractions related to some previous heuristics, and last we illustrate examples to show how to adapt some TSP heuristics to the SOP.

Neighbourhood Heuristics I

These are perhaps the simplest TSP heuristics. Define $Out(v)$ to be the least edge cost among vertex v's outgoing edges, and $In(v)$ to be the least edge cost among vertex v's incoming edges. Then both $\Sigma_{v \in V} Out(v)$ and $\Sigma_{v \in V} In(v)$ are lower bounds for the TSP. One application of this idea is the Nearest Neighbour (NN) algorithm. The key to this algorithm is to let the salesman start from a random city, and then successively go to the nearest unvisited city (not just the nearest city). This algorithm is easy to implement and executes quickly.

It is likely that there are much better tours if the cost of last few stages of the tour are large. For example in Figure 6.3 the cost of an optimal tour is 22, but if starting from x, NN gets a tour x-y-z-u-x with cost=1+1+1+1000=1003. There exist many specially arranged city distributions which make the NN algorithm give the worst route [36].



If the salesman starts from x, one best tour is x-y-z-u-x, and the minimum cost = 1+10+1+10=22.

But NN will get a tour x-y-z-u-x with cost=1+1+1+1000=1003

Figure 6.3: A simple TSP instance.

Neighbourhood Heuristics II

In this heuristic, two edges connected to a vertex are considered. Let $Sum(v)$ be the minimum sum of two edges connected to the vertex v. Here we require that for a directed graph these two edges

must have only one endpoint in common (i.e., one edge is $v's$ outgoing edge and the other is $v's$ incoming edge). For example, two edges $a \longrightarrow b$ and $b \longrightarrow a$ cannot be chosen at the same time. $Tour(v)$ is defined to be the sum of the two optimal tour edges connected with the vertex v. Noting that $Sum(v) \leq Tour(v)$, researchers [79] proposed a lower bound L for a TSP as follows.

- $L = \frac{1}{2}\Sigma_{v \in V} Sum(v) \leq \frac{1}{2}\Sigma_{v \in V} Tour(v) =$ optimal tour cost.

For example, in Figure 6.3 $L = \frac{1}{2}(11 + 2 + 2 + 11) = 13$. Although it is easy to compute L, there exist several situations where the lower bound may be ineffective (too small). As a result it may not work effectively as a lower bound for some circumstances.

The Assignment Problem (AP)

The assignment problem (AP) [3, 81] is to assign to each city i a city j, with the weight of the edge $i \rightarrow j$ as the cost of this assignment, and it is required that the same city must be assigned only once in such a way that the total cost of all assignments is minimized. A solution to AP provides a lower bound on the cost of the TSP because the assignments need not form a single tour, hence either a single cycle or a collection of disjoint cycles is formed.

We define an *AP-based* heuristic value for a TSP instance to be the solution cost to the corresponding assignment problem.

Researchers [29, 60] noticed that the AP can be stated in the following equivalent form:

- Find a cycle cover of minimum weight in G. Here a cycle cover[1] is a subgraph of G in which each of the n vertices has in-degree 1 and out-degree 1.

The *AP-based* heuristic value can be increased by the patching algorithm that was first described and analysed in [60]. The patching procedure converts the optimal solution of the assignment problem to an overall cycle by a sequence of patching operations, each of which joins two disjoint cycles together by deleting one edge in each cycle and inserting two new edges. Zhang et al. [11, 57, 97] applied this idea to the branch and bound algorithm for the ATSP.

DFBB needs to solve an AP for every node explored to obtain the optimal AP solution serving as a lower bound. The AP is solvable in $O(n^3)$ time [81], hence it is time-consuming to solve an AP at every node when n is large. As a result, the lower bound for a node may reduce the search tree, but if the reduction is too small the overall search time may increase due to the time invested in computing lower bounds.

The Held-Karp Lower Bound (HK)

The Held-Karp (HK) lower bound was first successfully used for the symmetric TSP by Held & Karp [41, 42] and Christofides [10]. This lower bound is evaluated as a 1-tree relaxation, where a 1-tree on an n-city symmetric TSP is defined as follows:

[1] A vertex cycle cover (commonly called simply a cycle cover) of a graph is a set of disjoint cycles which are subgraphs of G and contain all vertices of G.

- A 1-tree is a connected graph $G = (V, E)$ with vertices 1,2,...,n consisting of a tree on the vertices 2,3,...,n together with the additional vertex 1 connected to the tree by two edges.

To obtain a HK lower bound requires solving the minimum cost 1-tree problem which can be decomposable into two independent problems

- To find a Minimum Spanning Tree (MST) on the vertices 2,3,...,n; and
- To find two smallest cost edges among those connected with vertex 1 in G.

Solving the second problem requires $O(|V|)$ comparisons, whereas the first problem can be solved by the algorithm of Prim [13] of complexity of $O(|E| + |V| \text{ lg } |V|)$, or the algorithm of Kruskal[13] of complexity $O(|E| \text{ lg } |V|)$. Therefore, the lower bound for a node may reduce the search tree, but if the reduction is too small the overall search time may increase due to the time invested in computing lower bounds. Held and Karp [42] also proposed an iterative version of the HK lower bound. It involves computing a large number of minimum spanning trees, which is still time-consuming for practical use. A solution to a minimum cost 1-tree problem provides a lower bound on the cost of the TSP because the 1-tree need not form a tour. A tour is simply a 1-tree in which each vertex has degree 2. If a minimum 1-tree is a tour, then it is a tour of minimum cost.

Discussions

Taking the neighbourhood heuristics and the AP-based heuristics for example, first we discuss how to create a version of additive abstractions related to some previous heuristics. Then we illustrate examples to show how to adapt some TSP heuristics to the SOP heuristics.

The neighbourhood heuristics can be regarded as a special version of our abstraction-based heuristics. Given an n-city TSP instance associated with a graph $G = (V, E)$ in which each vertex represents a city and the weight of each edge $i \longrightarrow j$ represents the cost from i to j, vertices are assigned to n abstractions with only one distinguished vertex in each. In each abstraction if we only charge the cost of a move when the salesman steps from a distinguished vertex to other cities, the resulting additive heuristic value will be $\Sigma_{v \in V} Out(v)$. In the same fashion, if we only charge the cost of a step when the salesman moves to a distinguished city, $\Sigma_{v \in V} In(v)$ is our additive heuristic value. Applying the same partitioning, if we only charge half the cost of each move when the salesman steps from or to a distinguished city, then the lower bound L mentioned in Section 6.1.6 is the resulting additive heuristic value. This method for defining primary costs is a kind of cost-splitting.

Regarding the assignment problem, we can construct abstraction-based heuristics according to the vertex cycle cover. We partition the vertices corresponding to the subcycles in the minimum cycle cover for the graph G and assign each set of vertices to be the distinguished set in each abstraction. The sum of the minimum tour value of each abstraction is the additive abstraction-based heuristic value. For most cases this abstraction-based heuristic value is not exactly the same as the AP-based heuristic value which is the solution cost to the corresponding assignment problem. It is because

there are abstract paths between distinguished vertices and don't-care vertices, and sometimes the resulting abstraction-based heuristic value is less than the real AP-based heuristic value.

Some heuristics for the TSP can be adapted to be the heuristics for the SOP. Given an instance of the SOP defined by a weighted directed graph $G = \langle V, E, C, P \rangle$ with the start s and goal g vertices designated, as described in Section 6.1 our goal is to find a minimum cost Hamiltonian path (instead of a Hamiltonian cycle) from s to g which does not violate P. Given an intermediate state $t = (\ell, \mathcal{U})$ and a goal state $goal = (g, \emptyset)$ the neighbourhood heuristics can be adapted to be SOP heuristics as follows.

- $h_{out}(t, goal) = \sum_{v \in \mathcal{U} \cup \{\ell\}} Out(v)$ where $Out(v)$ is the minimum edge cost of v's outgoing edges whose other endpoint is $u, u \in \mathcal{U} \cup \{g\}$.
- $h_{in}(t, goal) = \sum_{v \in \mathcal{U} \cup \{g\}} In(v)$ where $In(v)$ is the minimum edge cost of v's incoming edges whose other endpoint is $u, u \in \mathcal{U} \cup \{\ell\}$.

As for AP-based heuristics for the SOP, we assign each vertex u ($u \in \mathcal{U} \cup \{\ell\}$) to a vertex v ($v \in \mathcal{U} \cup \{g\}$). In addition we require that ℓ must not be assigned to g if $\mathcal{U} \neq \emptyset$, and each vertex must not be assigned to itself. For example, given an intermediate state $t = (1, \{2, 3\})$ and a goal state $goal = (g, \emptyset)$, there are totally six possible assignments $\binom{1\ 2\ 3}{2\ 3\ g}$, $\binom{1\ 2\ 3}{2\ g\ 3}$, $\binom{1\ 2\ 3}{3\ 2\ g}$, $\binom{1\ 2\ 3}{3\ g\ 2}$, $\binom{1\ 2\ 3}{g\ 2\ 3}$, $\binom{1\ 2\ 3}{g\ 3\ 2}$, But only two assignments, $\binom{1\ 2\ 3}{2\ 3\ g}$ and $\binom{1\ 2\ 3}{3\ g\ 2}$, are feasible assignments. The AP-based heuristic value for the SOP is the minimum cost among those of all feasible assignments.

6.2 Designing Good Abstractions

As the effectiveness of admissible heuristics increases with the heuristic accuracy, in this section we consider how to design good abstractions to improve the quality of the heuristics.

6.2.1 Key Concepts

After preliminary experiments, we explore underlying keys for good abstractions considering expensive edges, cheap edges and precedence constraints.

Expensive Edges

The first key is to maximize the use of expensive edges as much as possible in the abstract state space. Take Figure 6.4 for example. Vertex x is connected to each of its neighbours by a very expensive outgoing edge. All the other edges that are not shown in the figure are cheap edges. If x is distinguished in the abstract state space, at least one outgoing expensive edge will be part of the abstract solution path. However, as shown in the right part of Figure 6.4 if x is a don't-care vertex, none of the expensive edges can be used because expensive edges will be replaced by other cheap ones. As a result, the heuristic value is too low to be a good one.

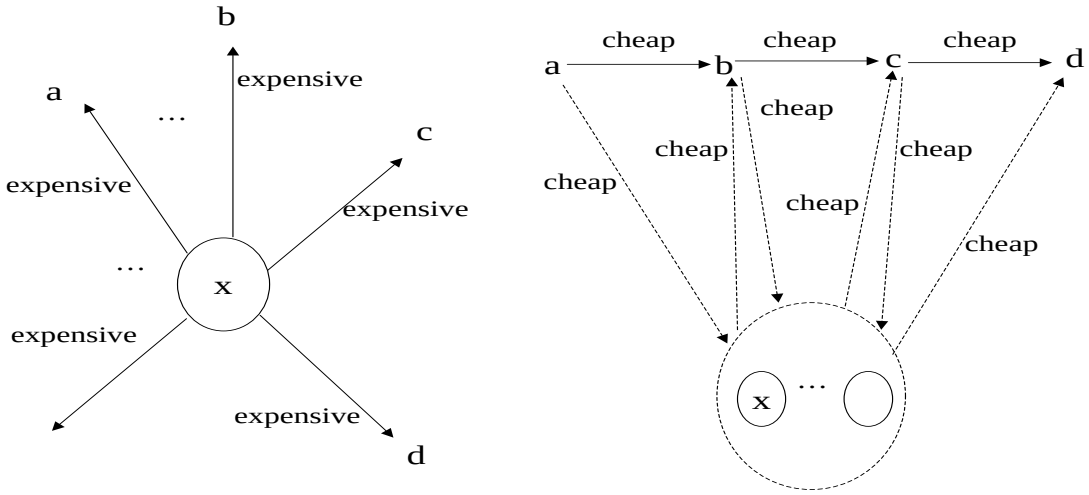


Figure 6.4: Vertex x is connected to each neighbour vertex by expensive outgoing edges. Edges that are not shown are cheap edges. Left: If x is distinguished, at least one expensive edge must be used in the solution path. Right: If x is a don't-care vertex, all expensive edges can be replaced by cheap edges.

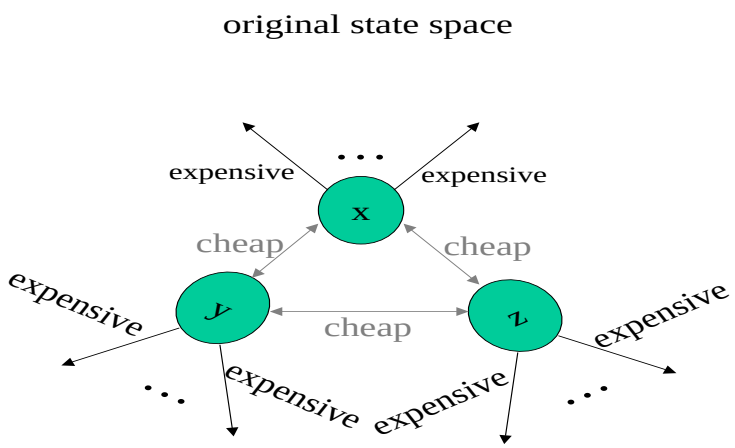


Figure 6.5: Vertices x,y and z are consisting of an expensive-edge cluster.

In practice (e.g. ft53.1 in TSPLIB) there exists a cluster in which some vertices are neighbours connected by cheap edges while they are connected to other vertices by very expensive edges. All other edge costs are not expensive. We refer to this cluster of vertices as the *expensive-edge cluster* since these vertices are connected to other vertices by expensive edges. Formally, given a set of

vertices V, a cluster of vertices U ($U \subset V$) is an expensive-edge cluster provided that $u_i \rightarrow u_j$ is not an expensive edge and $u_i \rightarrow v$ is an expensive edge, $\forall i \neq j, u_i, u_j \in U, v \in V \setminus U$. Note that vertex x shown in Figure 6.4 is just a single-node expensive-edge cluster.

Figure 6.5 is an example of this type. The vertices x, y and z form an expensive-edge cluster because $u_i \rightarrow u_j$ is not an expensive edge $\forall i \neq j, u_i, u_j \in \{x,y,z\}$ and $u_i \longrightarrow v$ is an expensive edge, $\forall u_i \in \{$x,y,z$\}$ and $v \in V \setminus \{$x,y,z$\}$.

In the examples shown in Figures 6.4 and 6.5, expensive edges are edges starting from vertices of the expensive-edge cluster U. In general, an expensive-edge cluster can also refer to the set of vertices whose incoming edges are expensive. When we design abstractions, vertices of the expensive-edge cluster have the top priority to be included in the same set of distinguished vertices, since at least one expensive edge must be used in the abstract solution path.

Given an SOP instance represented by a weighted graph $G = (V, E, C, P)$ and the expensive edge set $expE \subset E$, we identify the expensive-edge cluster U if it satisfies one of the following definitions.

- If $u_i \rightarrow v \in expE$ and $u_i \rightarrow u_j \notin expE, \forall i \neq j, u_i, u_j \in U, v \in V \setminus U$ and $1 \leq |U| < |V|$, then U is an expensive-edge cluster.
- If $v \rightarrow u_i \in expE$ and $u_i \rightarrow u_j \notin expE, \forall i \neq j, u_i, u_j \in U, v \in V \setminus U$ and $1 \leq |U| < |V|$, then U is an expensive-edge cluster.

Cheap Edges

The second key for good abstractions is to eliminate the use of cheap edges as much as possible in the abstract state space. Take Figure 6.6 for example. Vertex x is connected with many of its neighbours by cheap edges. Here we refer to x as a *cheap-edge cluster* as it is a vertex connected with many cheap edges. Given the cheap edge set $cheapE$, we define that a cheap-edge cluster U's nearest neighbours are vertices connected to U by cheap edges, i.e., $NN(U) = \{v : v \notin U, u \in U$ and $(u \rightarrow v \in cheapE$ or $v \rightarrow u \in cheapE)\}$. In Figure 6.6, if vertex x and its nearest neighbours $NN(\{x\})$ are distinguished in the abstract state space, only two cheap edges will be used in the abstract solution path. However, if x is a don't-care vertex and its nearest neighbours are distinguished, more cheap edges can be used because there exist many cheap edges between the distinguished vertices and don't-care vertices. As shown in the right part of Figure 6.6, the path a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d can be replaced by the path a $\rightarrow$ x $\rightarrow$ b $\rightarrow$ x $\rightarrow$ c $\rightarrow$ x $\rightarrow$ d which consists of more than one cheap edge. As a result, the heuristic value may be too low to be a good one. Thus it is important to put x and its nearest neighbours in the same abstract state space. In the example shown in Figure 6.6, vertex x is connected with many outgoing and incoming cheap edges. In general a cheap-edge cluster can also be a vertex with only incoming or outgoing cheap edges.

In practice a cheap-edge cluster may be connected with a group of other cheap-edge clusters by

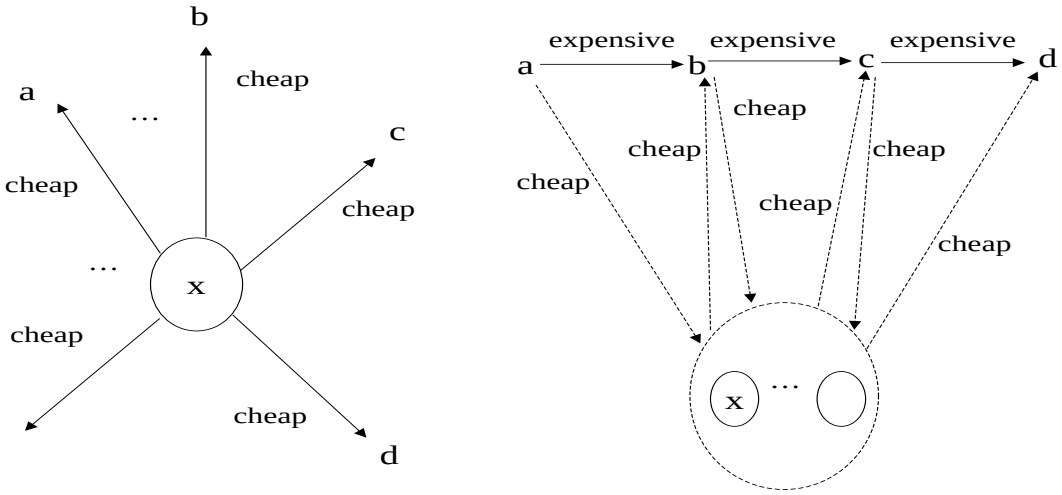


Figure 6.6: x is connected to many of its neighbours by cheap edges. Edges that are not shown are expensive edges. Left: If x and its nearest neighbours are distinguished, only two cheap edges can be used in the solution path. Right: If x is a don't-care vertex but its neighbours are not, many expensive edges can be replaced by cheap edges between distinguished vertices and x.

cheap edges. A cheap-edge cluster consisting of a single vertex (shown in Figure 6.6) is a single-node cheap-edge cluster which can also be called *cheap-edge core* in the remaining of this chapter.

Given a weighted directed graph $G = (V, E, C, P)$ and the cheap edge set $cheapE \subset E$, we classify two basic types of the cheap-edge clusters as follows.

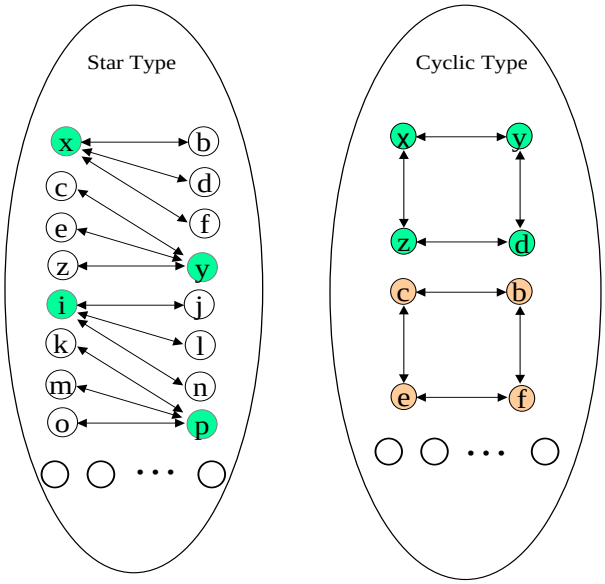


Figure 6.7: Two basic types of the cheap-edge cluster. Only cheap edges are shown in the state space.

- **Star Type**. $U = \{u\}, V = \{v_1, v_2, ..., v_m\}$. If $\forall v_i \in V$, $v_i \rightarrow u \in cheapE$ then U is a cheap-edge cluster of the incoming star type consisting of a single vertex; if $\forall v_i \in V$, $u \rightarrow v_i \in cheapE$ then U is a cheap-edge cluster of the outgoing star type. For example, there are four cheap-edge clusters (i.e., {x},{y},{i},{p}) in the left part of Figure 6.7.

- **Cyclic Type**. $U = \{u_1, u_2, ..., u_m\}$. If $\forall i \in [1..m-1]$, $u_i \rightarrow u_{i+1} \in cheapE$ and $u_m \rightarrow u_1 \in cheapE$, then U is a cheap-edge cluster of the cyclic type. In the cheap-edge cluster of this type, each element can be regarded as a cheap-edge cluster connected with some neighbours by incoming cheap edges or outgoing cheap edges. For example, in the right part of Figure 6.7, there are two cheap-edge clusters of the cyclic type, {x, y, z, d} and {c, b, e, f}.

Precedence Constraints

When designing abstractions, we first consider the edge cost and regard constraints as the secondary consideration due to two main reasons as follows.

- Precedence constraints have played a role for the DFBB (described in Figure 6.2) to detect and prune the states representing a violation. If there are as many precedence constraints as possible, the Hamiltonian path is defined by the constraints and the heuristics don't matter in this situation.

- If there exist only a few constraints, the quality of heuristics cannot be improved provided that there are many cheap edges available to form a cheap abstract solution path. In this situation, edge costs play a more important role for the heuristics.

Therefore constraints are regarded as the secondary consideration. For example, if two vertices have the same number of nearest neighbours it is better to have the vertex involved with more constraints as distinguished, which will preserve more constraints in the abstract state space.

In short, expensive-edge clusters, cheap-edge clusters and the precedence constraints are underlying keys when we design greedy methods to generate abstraction-based heuristics. Since h_{add} and h_{max} have different definitions and computations in the abstract state space, the ways of applying these keys for h_{add} and h_{max} may differ.

6.2.2 Different Considerations For h_{add} and h_{max}

Figure 6.8 illustrates an abstract state space. D is the set of distinguished intermediate vertices, R is the set of don't-care vertices. Since the start s and goal g vertices are always distinguished in each abstraction, we require that $D \cup R = V \setminus \{s, g\} = \mathcal{I}$. According to the definitions in Section 6.1, there are two main differences in the computation of abstractions for h_{add} and h_{max}.

One difference is that for h_{add} each edge from R to D is zero (i.e., $C(u \rightarrow v) = 0, \forall u \in R, v \in D$). Therefore in additive abstraction we are interested in the cheap-edge cluster on incoming cheap

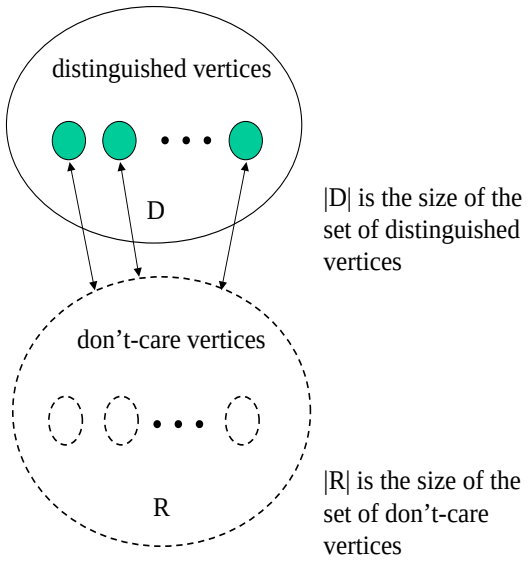


Figure 6.8: An Abstract State Space.

edges. For example, in Figure 6.9 suppose the edges that are not shown are expensive. In the additive abstraction (shown on the right part of Figure 6.9), the cost of each edge from R to D is always zero. If x and its neighbours are distinguished, only one incoming cheap edge will be used in the solution path; while if x is in R but its neighbours are in D, there are many cheap edges between R and D can be used in the solution path. For example, the path a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d can be replaced by the path a $\rightarrow$ x $\rightarrow$ b $\rightarrow$ x $\rightarrow$ c $\rightarrow$ x $\rightarrow$ d which consists of more than one cheap edge. To minimize the use of the cheap edges, vertex x and its corresponding neighbours should be added to the same distinguished set.

The other difference is that for h_{add} we don't count edges between don't-cares (i.e., $C(u \rightarrow v) = 0, \forall u, v \in R$), while for h_{max} the edges between don't-cares also count to construct the abstract solution.

Take the cheap-edge cluster of the star type for example. In additive abstractions we only consider incoming cheap-edge clusters while in standard abstractions we should consider both incoming and outgoing cheap-edge clusters. Furthermore, in order to identify the cheap-edge cluster in standard abstractions, we consider the number of incoming (or outgoing) cheap edges exclusively instead of the total number of cheap edges. It is because at most two edges will be used in the solution path if the vertex is connected with m mixed (i.e., both incoming and outgoing) cheap edges, while up to one cheap edge will be used for the vertex connected with m incoming (or outgoing) cheap edges. As shown in Figure 6.10 there are two cheap-edge clusters $\{x\}$ and $\{y\}$. x has 3 incoming cheap edges and 3 outgoing cheap edges, while y has 6 incoming cheap edges and 0 outgoing cheap edges. If x and its nearest neighbours $NN(\{x\})$ are distinguished, at most **two** cheap edges will be used in the solution path, while if y and $NN(\{x\})$ are distinguished, at most **one** cheap edge will be used

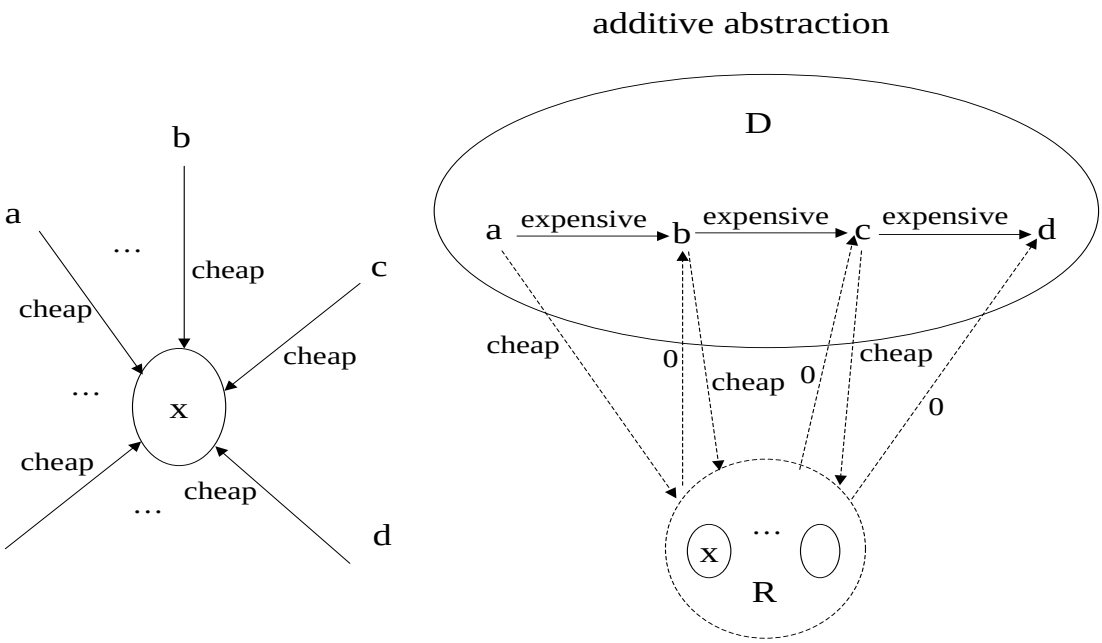


Figure 6.9: An example of the cheap-edge cluster on incoming cheap edges. Other edges not shown in the figure are expensive. Left: If x and its nearest neighbours are distinguished, at most one incoming cheap edge will be used in the solution path. Right: If x is in R, but its neighbours are in D, many expensive edges can be replaced by cheap edges between D and R.

in the solution path. Therefore y has the top priority to be identified first although both x and y are connected with 6 cheap edges.

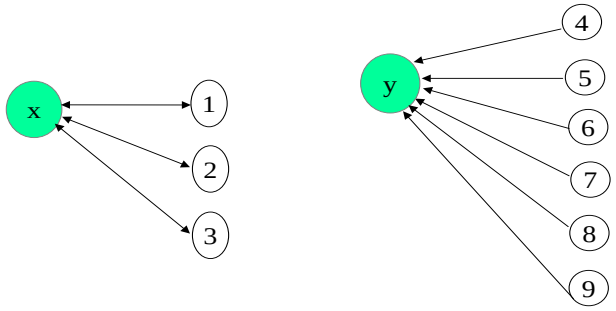


Figure 6.10: Two cheap-edge clusters $\{x\}$ and $\{y\}$. Each cheap-edge cluster is connected with 6 cheap edges.

A cheap-edge cluster U and its nearest neighbours $NN(\{U\})$ should be put into the same set

of distinguished vertices. However, there are situations when the size of distinguished set (i.e., $|D|$) is too small to include all cheap-edge clusters and their nearest neighbours in one abstraction. It may lead to the result that some cheap edges are left between don't-cares. According to the second difference, for additive abstractions, there is less harm to have some cheap edges between don't-cares. But the computation of h_{max} also counts the edges between don't-cares, so here we discuss two schemes (Scheme I and Scheme II) to eliminate the use of cheap edges when computing h_{max}.

- Scheme I is to have as many cheap-edge clusters and the corresponding nearest neighbours as possible in the distinguished set. That is, we minimize the use of cheap edges between distinguished vertices by clustering as many nearest neighbours as possible in the distinguished set;

- Scheme II is to avoid as many cheap edges as possible between don't-care vertices. For this purpose, we can avoid having cheap edges between don't-care vertices by only putting into the distinguished set one end point of each cheap edge. That is, more cheap edges may exist between distinguished vertices and don't-care vertices.

Both schemes have their own benefits and drawbacks for different situations. Figure 6.11 is an example to discuss the benefits and drawbacks of these two schemes for h_{max}. The left part illustrates the original state space; the middle part is the abstract state space constructed using the first scheme (i.e., Scheme I); the right part shows the abstract state space constructed using the second scheme (i.e., Scheme II). $|D|$ is the number of distinguished vertices and $|R|$ is the number of don't-care vertices. Assume that all cheap-edge clusters and their nearest neighbours have been identified and labelled in the original state space. Only cheap edges are shown in the original state spaces and other edges are expensive.

In the original state space (shown on the left part of Figure 6.11) of the example, if the cost of an edge $u \rightarrow v$ is less than or equal to 10, we regard the edge $u \rightarrow v$ as a cheap edge. There is a cycle consisting of four vertices x, y, z and d connected with each other by eight directed cheap edges. Assume that $|D| = 4$. Applying Scheme I, we have distinguished set $D = \{$x,y,z,d$\}$, such that at most three cheap edges will be used to construct the abstract solution path. While applying Scheme II, $D = \{$x,d,c,f$\}$ and at most four cheap edges are used to construct the abstract solution path. Since the total number of edges in either the original solution path or the abstract solution path is always the same, even one more cheap edge being used to replace an expensive one may lead to a very small heuristic value.

For this example, if using Scheme I, $D = \{$x,y,z,d$\}$ and the edge cost is defined as follows.

- $C(u \rightarrow v) = 1, \forall u, v \in D$.

- $C(u \rightarrow v) = 10, \forall u, v \in R$.

- $C(u \rightarrow v) = 100, \forall u \in D, v \in R$.

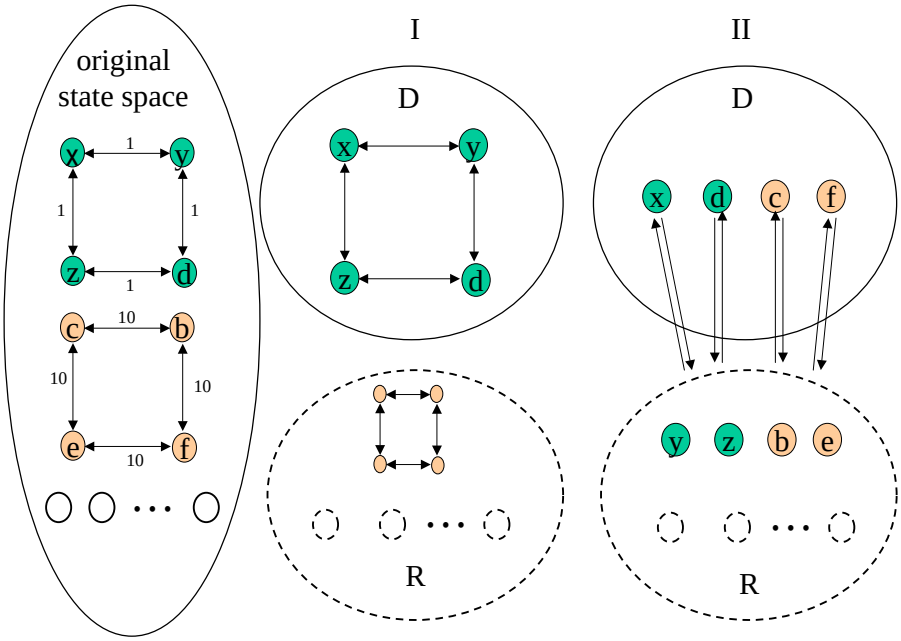


Figure 6.11: An example to compare two schemes.

- $C(u \to v) = 100, \forall u \in R, v \in D.$

And using Scheme II, $D = \{$x,d,c,f$\}$ and the edge cost is defined as follows.

- $C(u \to v) = 100, \forall u, v \in D.$
- $C(u \to v) = 100, \forall u, v \in R.$
- $C(\text{x} \to v) = C(v \to \text{x}) = C(\text{d} \to v) = C(v \to \text{d}) = 1, \forall v \in R.$
- $C(\text{c} \to v) = C(v \to \text{c}) = C(\text{f} \to v) = C(v \to \text{f}) = 10, \forall v \in R.$

Suppose that $|D| = 4, |R| = 5$ in the abstract state space shown in the middle part of Figure 6.11. One abstract solution path may be x $\to$ y $\to$ d $\to$ z $\to u \to u \to u \to u \to u$ where $u \in R$, the heuristic value $h_I = 1 + 1 + 1 + 100 + 10 + 10 + 10 + 10 = 143$; and in the abstract state space shown in the rightmost part of Figure 6.11 one abstract solution path may be $u \to$ x $\to u \to$ d $\to u \to$ c $\to u \to$ f $\to u$ where $u \in R$, $h_{II} = 1 + 1 + 1 + 1 + 10 + 10 + 10 + 10 = 44$. Scheme I beats Scheme II in this situation because using Scheme I minimizes the use of the cheapest edges.

However, the advantage of Scheme I decreases as $|R|$ increases. For example if we change $|R| = 7$ and keep the other conditions, then two more edges between don't-care vertices must be added to the abstraction solution path. Thus $h_I = 143 + 20 = 163$ and $h_{II} = 44 + 200 = 244$. Scheme II outperforms Scheme I in this situation because Scheme II maximizes the minimum edge cost between don't-care vertices.

6.2.3 Summary of Concepts

For both h_{add} and h_{max} our goal is to maximize the use of expensive edges and eliminate the use of cheap edges as much as possible.

As shown in Figures 6.4 and 6.5, an expensive-edge cluster is defined to be a vertex or a group of vertices that are connected to all other vertices by expensive edges, while the vertices in the same expensive-edge cluster are connected to each other by cheap edges. The expensive-edge cluster has the top priority to be distinguished in the abstractions, although it does not always exist in all instances. Once there exists an expensive-edge cluster consisting of $\{u_1, u_2, ..., u_m\}$, we mark $u_1, u_2, ..., u_m$ distinguished before applying any other greedy method for cheap-edge clusters. Note that the size of an expensive-edge cluster should be small enough to be included in the distinguished set. Otherwise we neglect the larger expensive-edge clusters in the abstraction because it will be ineffective when leaving some part of an expensive-edge cluster in the set of don't-care vertices.

A cheap edge cluster is defined to be a vertex or a set of vertices that are connected to some other vertices by cheap edges. Here we do not require a cheap edge cluster to be connected to all other vertices by cheap edges. Figure 6.7 present two basic types of the cheap-edge cluster. Since the cheap-edge cluster U has the close relationship with its nearest neighbours $NN(U)$, we have two obvious choices. Either we put U and $NN(U)$ in the same set of distinguished vertices; or we just have U to be distinguished. The former method eliminates the use of cheap edges in the distinguished set as much as possible, but it may leave some cheap edges between don't-care vertices.

For additive abstractions, the computation of h_{add} does not count the edges between don't-cares. Thus no matter the value of $|R|$ it is important to put U and $NN(U)$ to the same distinguished set in order to eliminate the use of cheap edges as much as possible.

The computation of h_{max} differs mainly in that edges between don't-care vertices also count. Given $|D|+|R|$, the total number of vertices, it is evident that enlarging the value of $|D|$ will enhance the quality of h_{max}. However, the memory requirements increase significantly when the abstraction contains more distinguished vertices. Therefore there exists certain situations in which the value of $\frac{|D|}{|D|+|R|}$ must be relatively small and sometimes we are more interested to study how to define good h_{max} in these situations, i.e., $|D| \leq |R|$. According to the discussion in the previous section, when $|D| \leq |R|$ it is better to apply Scheme II : we just put one endpoint of each cheap edge into the distinguished set to maximize the minimum edge cost between don't-care vertices.

We first consider the edge costs and then regard constraints as the secondary consideration. In Sections 6.3 and 6.4 we will have more details of greedy methods for identifying the expensive-edge clusters and the cheap-edge clusters. In the following sections, we refer to each single element in the cheap-edge cluster as the *cheap-edge core*.

6.3 Greedy Methods for h_{add}

In this section we present algorithms to solve the problem defined as follows:

- **Input:** An SOP instance ($G = \langle V, E, C, P\rangle$ and the cost matrix M), and a sequence of numbers $< n_1, n_2, ..., n_k >$ specifying the size of the distinguished set in each abstraction where k is the total number of abstractions and $\Sigma_{i=1}^{k} n_i \leq |\mathcal{I}|$ where $\mathcal{I} = V \setminus \{s, g\}$[2].
- **Output:** a sequence of distinguished sets $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

Algorithms are described in the pseudocode and each line of comments begins with "//". Some issues such as error handling are ignored in order to convey the essence of the algorithm more concisely. Likewise, some variables such as $< n_1, n_2, ..., n_k >$ and $< D_1, D_2, ..., D_k >$ are global variables throughout all subroutines, and we will not repeat the claim in the subroutines for the purpose of conciseness.

In Section 6.2 we described the expensive-edge cluster and the cheap-edge cluster that should be distinguished to build good abstractions. In this section greedy methods are applied. First we present a method called CHECK_EXPENSIVE to check for expensive-edge clusters. Then we introduce the method (CHEAP_EDGE_ADD) to identify the cheap-edge clusters. CHEAP_EDGE_ADD is a general method consisting of subroutines that are designed for cheap-edge clusters of the star type and the cyclic type, respectively. Finally in Section 6.3.3 we combine these greedy methods to consider both the expensive-edge cluster and the cheap-edge cluster for designing good abstractions. Note that greedy methods do not always yield optimal solutions, thus there exist certain situations which make these methods not yield the best abstractions. More observations and discussions are presented in the sections on experiments and future directions.

6.3.1 The CHECK_EXPENSIVE Procedure

As discussed in Section 6.2.3, the expensive-edge cluster does not always exist in all instances. Once a cluster of vertices is identified to be an expensive-edge cluster, these vertices have the top priority to be distinguished in the abstractions.

An *expensive edge* is defined by an edge $u \longrightarrow v$ with an edge cost no less than a parameter ω.

Given a weighted graph $G(V, E, C, P)$, an expensive-edge cluster can be identified if it satisfies one of the following definitions.

1. If $\exists U$, $1 \leq |U| < |V|$ such that $C(u_i \longrightarrow v) \geq \omega$ and $C(u_i \rightarrow u_j) < \omega, \forall i \neq j, u_i, u_j \in U$ and $v \in V \setminus U$, then U is an expensive-edge cluster.

[2]Here we are assigning the intermediate vertices $\mathcal{I} = V \setminus \{s, g\}$ to each distinguished set. The start s and goal g vertices are always distinguished in each abstraction.

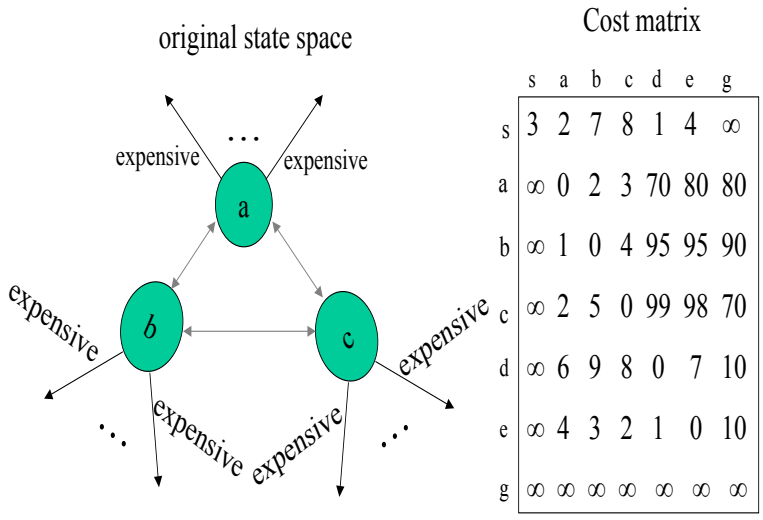


Figure 6.12: An instance with an expensive-edge cluster {a, b, c}.

2. If $\exists U$, $1 \leq |U| < |V|$ such that $C(v \longrightarrow u_i) \geq \omega$ and $C(u_i \rightarrow u_j) < \omega, \forall i \neq j, u_i, u_j \in U$ and $v \in V \setminus U$, then U is an expensive-edge cluster.

We take Figure 6.12 for example. If ω is set to be 50 for this instance, any edge whose cost is larger than or equal to 50 is an expensive edge. Since $\forall u_1$, $u_2 \in$ {a,b,c} and $v \in \{s, g$,d,e$\}$, $C(u \rightarrow v) \geq \omega$ and $C(u_1 \rightarrow u_2) < \omega$, {a,b,c} is an expensive-edge cluster.

CHECK_EXPENSIVE is a subroutine to check for the expensive-edge clusters. Its inputs include some global variables such as M, $< n_1, n_2, ..., n_k >$ and ω. The output is a sequence $< D_1, D_2, ..., D_k >$ indicating the distinguished set for each abstraction.

From the definitions of the expensive-edge cluster and the description in Figure 6.13 the operation of CHECK_EXPENSIVE is fairly straightforward. The **FOR** loop of lines 4-21 uses the definitions of expensive-edge clusters to examine each vertex.

Lines 6-11 detect the expensive-edge cluster consisting of a single vertex. If not all of u's outgoing edges are expensive and not all of u's incoming edges are inexpensive, we examine the existence of an expensive-edge cluster consisting of a set of vertices U in lines 12-20 using the last two definitions.

No matter what the size of U, one expensive edge must be used in the abstract solution path. Suppose an expensive-edge cluster U consisting of m vertices is found. The number of expensive edges which must be used in the abstract solution path per vertex of the expensive-edge cluster is

Algorithm: CHECK_EXPENSIVE

Key Concept: To identify expensive-edge clusters and assign them to distinguished sets.

Input: M (the cost matrix)
$< n_1, n_2, ..., n_k >$ (a sequence of numbers specifying the size of each distinguished set)
ω (the lower bound of the expensive edge cost.)

Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| \leq n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

$D_i = \phi, \forall i$. // Initialization.
$CIndex$=0.// Initialize the cluster index.
$U_0 = \phi$.
FOR each vertex u **DO**
  Define U_{mark}= $\cup_{j=0}^{CIndex} U_j$.//U_{mark} include vertices identified to belong to an expensive-edge cluster.
  IF ($u \notin U_{mark}$) **AND** ($M_{u,v} \geq \omega, \forall v \in V \setminus \{u\}$) //all outgoing edges are expensive.
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = \{u\}$.
  ELSE IF ($u \notin U_{mark}$) **AND** ($M_{v,u} \geq \omega, \forall v \in V \setminus \{u\}$) //all incoming edges are expensive.
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = \{u\}$.
  // Check the first definition for the expensive-edge cluster consisting of more than one vertex.
  ELSE IF ($\exists U$, $u \in U$ **AND** 2$\leq |U| < |V|$) **AND** ($M_{u,u_i} < \omega$,$\forall u_i \in U \setminus \{u\}$) **AND** ($U \cap U_{mark} = \phi$)
      AND ($M_{u_i,v} \geq \omega$ **AND** $M_{u_i,u_j} < \omega$,$\forall u_i$,$u_j \in U$, $v \in V \setminus U$)
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = U$.
  // Check the second definition for the expensive-edge cluster consisting of more than one vertex.
  ELSE IF ($\exists U$, $u \in U$ **AND** 2$\leq |U| < |V|$) **AND** ($M_{u,u_i} < \omega$,$\forall u_i \in U \setminus \{u\}$) **AND** ($U \cap U_{mark} = \phi$)
      AND ($M_{v,u_i} \geq \omega$ **AND** $M_{u_i,u_j} < \omega$,$\forall u_i$,$u_j \in U$, $v \in V \setminus U$)
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = U$.
  END IF
END FOR
IF ($CIndex > 0$)
  SORT all expensive-edge clusters by the size,i.e., $< U_{\tau(1)}, ..., U_{\tau(CIndex)} >$, $\forall i$, $|U_{\tau(i)}| \leq |U_{\tau(i+1)}|$.
  SORT $< n_{\pi(1)}, n_{\pi(2)}, ..., n_{\pi(k)} >$ such that $n_{\pi(i)} \leq n_{\pi(i+1)}$,$\forall i$.
  $t = 1$. // $\pi(t)$ is the index of each distinguished set.
  FOR $i = 1$ to $CIndex$ **DO**
    IF ($t \leq k$ **AND** $|D_{\pi(t)} \cup U_{\tau(i)}| \leq n_{\pi(t)}$)
      $D_{\pi(t)} = D_{\pi(t)} \cup U_{\tau(i)}$.
    ELSE
      REPEAT
        $t = t + 1$.
      UNTIL ($|D_{\pi(t)} \cup U_{\tau(i)}| \leq n_{\pi(t)}$ **OR** $t > k$).
      IF ($t > k$)
        break; // no more distinguished set to accommodate the cluster.
      ELSE
        $D_{\pi(t)} = D_{\pi(t)} \cup U_{\tau(i)}$.
      END IF
    END IF
  END FOR
END IF

Figure 6.13: The method to check for the expensive-edge cluster.

$1/m$.

Therefore, the smaller value of m, the greater value of $1/m$. Given a set of expensive-edge clusters the greedy method begins by adding to the distinguished set the expensive-edge cluster of smallest size because the smallest expensive-edge cluster has the greatest value of $1/m$ and it has the top priority to be included in the distinguished set.

Here the expensive-edge clusters are sorted by size in increasing order. In line 23, $\tau(i)$ ($i \in [1, CIndex]$) is the index indicating the new order of all expensive-edge clusters. Then the distinguished sets are also sorted by the required size in non-decreasing order. The reason for this order is that larger expensive-edge cluster can only be accommodated by larger distinguished sets and if we first fill the large distinguished set with many small expensive-edge cluster, then there is no distinguished set to accommodate the large expensive-edge cluster. In line 24, $\pi(i)$ ($i \in [1, k]$) is the index indicating the new order of all distinguished sets.

The second part of CHECK_EXPENSIVE is in lines 22-40. It starts if there exists some expensive-edge clusters (i.e., if $CIndex > 0$). It sorts the expensive-edge clusters by the size $|U_i|$ in non-decreasing order and also sorts the sequence that specifies the size of each distinguished set by the value in non-decreasing order. The **FOR** loop of lines 26-39 obtains expensive-edge clusters in turn from the sequence $< U_{\tau(1)}, ..., U_{\tau(CIndex)} >$ and fills each distinguished set to capacity. Lines 30-32 are executed to find a distinguished set that is big enough to include the current expensive-edge cluster U_i. The algorithm terminates when the current expensive-edge cluster $U_{\tau(i)}$ is too big to be included in the last distinguished set (i.e., $|D_{\pi(k)} \cup U_{\tau(i)}| > n_{\pi(k)}$), or all expensive-edge clusters have been assigned to the distinguished sets (i.e., $i > ClusterIndex$).

In our experiments, we set ω to be the value min+(max-min)$\times 50\%$ where min and max are the minimum value and maximum values in the cost matrix, respectively. So in the following experiments any edge cost no less than (min+(max-min)$\times 50\%$) is regarded as an expensive-edge cost. Based on this definition of ω, in the range of our experiments on larger instances of TSPLIB we can identify one expensive-edge cluster in each of the instances ft53.1-4 and the identification of expensive-edge cluster significantly improves the quality of heuristic values.

6.3.2 The CHEAP_EDGE_ADD Procedure

No matter what type the cheap-edge cluster is, the underlying key is to identify the cheap-edge core and its corresponding nearest neighbours, and add them to the same distinguished set for additive abstractions. Considering both the star type and the cyclic type, we design CHEAP_EDGE_ADD which is more general for practical use.

The inputs of CHEAP_EDGE_ADD include G, α (the parameter used in the subroutine to set δ, the upper bound of the cheap edge cost.), $< n_1, n_2, ..., n_k >$ and $< D_1^0, D_2^0, ..., D_k^0 >$ (a sequence of partly designated distinguished sets). The output is $< D_1, D_2, ..., D_k >$ which specifies the distinguished vertices for each abstraction.

Algorithm: CHEAP_EDGE_ADD
Key Concept: considering cheap-edge clusters of the star type and the cyclic type.
Input: G, α, $< n_1, n_2, ..., n_k >$, and
$< D_1^0, D_2^0, ..., D_k^0 >$ where $D_i^0 \cap D_j^0 = \emptyset, \forall i \neq j$ and $|D_i^0| \leq n_i, \forall i$.
Output: $< D_1, D_2, ..., D_k >$ such that $D_i^0 \subseteq D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.
1 SORT all edges by edge costs, $S =< e_1, ..., e_m >, \forall i, C(e_i) \leq C(e_{i+1})$.
2 $D_i = D_i^0, \forall i \in [1, k]$. // Initialization.
3 **FOR** $i = 1$ **To** k **Do**
4 Obtain the cheap-edge core x from the outputs of **CHECK_INCOMING_STAR**(i,S,α).
5 **IF** ($|D_i| < n_i$)
6 Define $E_x = \{z \rightarrow x : z \in D_i\}.//E_x$ is the set of x's incoming edges.
7 Select the edge $u \rightarrow v$ from E_x where $C(u \rightarrow v) \leq C(e), \forall e \in E_x$.
8 **NEAREST_NEIGHBOUR**(i, u, v)
9 **END IF**
10 **END FOR**

Figure 6.14: The greedy method designed for instances with cheap-edge clusters of both the cyclic type and the star type.

Figure 6.14 gives the pseudocode for CHEAP_EDGE_ADD. After the initialization (lines 1-2), the **FOR** loop of lines 3-9 adds vertices to the distinguished set for each abstraction. First it calls CHECK_INCOMING_STAR to select a cheap-edge core x. In CHECK_INCOMING_STAR x and its corresponding nearest neighbours are assigned to the current distinguished set. If more vertices are still needed for the distinguished set (i.e., if $|D_i| < n_i$), CHEAP_EDGE_ADD calls the procedure NEAREST_NEIGHBOUR to start from the cheap-edge core x and construct a piece of path by the nearest neighbour strategy. All vertices along the path are assigned to the distinguished set until the set is big enough.

In this algorithm α is a parameter to define the upper bound of the cheap edge cost used in the subroutine CHECK_INCOMING_STAR. We discuss the usage of α in the next section.

There is a step of sorting edges by costs in non-decreasing order in this algorithm. In the same manner we define $P_e = \{u : x \rightarrow y = e, u <_p x$ or $x <_p u$ or $y <_p u$ or $u <_p y\}$. If $|P_{e_i}| > |P_{e_j}|$ and C(e_i)=C(e_j), then e_i is in front of e_j in the sequence,i.e., $S =< e_1, ..., e_i, e_j, ..., e_m >$.

CHEAP_EDGE_ADD does not always yield optimal solutions, thus there exist certain situations which make it not yield the best abstractions. More observations and discussion are presented in the sections on experiments.

CHEAP_EDGE_ADD is an algorithm considering cheap-edge clusters of the incoming star type and the cyclic type. We focus on the incoming star type because we don't count the edges starting from a don't-care vertex as we discussed in Section 6.2.2. Alternatively, if we change the definition to charge the primary cost of a move when a distinguished state is reached, it is easy to adapt the CHEAP_EDGE_ADD to focus on the outgoing star type as follows.

- In line 4 CHECK_OUTGOING_STAR is used instead to output the cheap-edge core of the outgoing star type. The CHECK_OUTGOING_STAR procedure will be described later in this

section.

- In line 6 E_x is adapted to be $\{x \rightarrow z : z \in D_i\}$, which is the set of x's outgoing edges.

CHEAP_EDGE_ADD calls two subroutines: CHECK_INCOMING_STAR and NEAREST_NEIGHBOUR. The CHECK_INCOMING_STAR can be replaced by CHECK_OUTGOING_STAR if we change the definition of edge costs in additive abstraction. The remainder of this section introduces the function and structures of these procedures.

The CHECK_INCOMING_STAR Procedure

Algorithm: CHECK_INCOMING_STAR
Key Concept:To add a cheap-edge cluster of the incoming star type to the distinguished set.
Input: $Index$ (the index of the current distinguished set)
S (a sequence of edges $< e_1, ..., e_m >, \forall i, C(e_i) \leq C(e_{i+1})$)
α (a parameter used to define the cheap edge cost.)
Output: x (the cheap-edge core).

```
1  Define marked = ∪_{j≠Index} D_j.
2  Obtain Δ from the output of SET_INCOMING_DELTA(marked,α,S).
3  CheapNo[v] = 0,∀v ∉ marked. // Initialization.
4  FOR i = 1 To m DO
5      Define β_i = {z, y} where y → z = e_i.
6      IF (β_i ∩ marked =∅)
7              CheapNo[z]=CheapNo[z]+1.
8              IF (CheapNo[z]=Δ)
9                x = z; break; // terminate the loop and z is identified to be a cheap-edge core.
10             END IF
11     END IF
12 END FOR
13  D_Index = D_Index ∪ {x}.
14 FOR each vertex u ∈ {u : C(u → x) ≤ C(e_i), u ∉ marked } Do
15     IF (|D_Index| < n_Index)
16             D_Index = D_Index ∪ {u}.
17     ELSE
18             break;
19     END IF
20 END FOR
```

Figure 6.15: The greedy method designed to identify cheap-edge cores of the star type.

For a cheap-edge core of the star type, we set the criteria to identify it. We require that a cheap-edge core is connected with at least Δ cheap edges. Here we apply the procedure called SET_INCOMING_DELTA to set the value of Δ. The SET_INCOMING_DELTA procedure will be described later in this section.

CHECK_INCOMING_STAR is a subroutine to identify a cheap-edge cluster of the star type. It takes several global variables as inputs, such as the graph G representing an instance and $<$

$D_1, D_2, ..., D_k >$ which specifies the distinguished vertices for each abstraction. Besides those variables the inputs also include $Index$ (the index of the current distinguished set), S (a sequence of edges sorted in increasing order of the size), and α (a parameter used to define the upper bound of the cheap edge cost). This algorithm computes the global values $< D_1, D_2, ..., D_k >$ and also outputs a cheap-edge core x.

Figure 6.18 presents the pseudocode of CHECK_INCOMING_STAR. First it obtains the value of Δ from the output of SET_INCOMING_DELTA which is described in Figure 6.16. Δ is the designated number of cheap edges on a cheap-edge core of the star type to be identified. The **FOR** loop of lines 4-12 is to detect a cheap-edge core x which is connected with Δ incoming cheap edges. The value of Δ described in Figure 6.16 guarantees the existence of x. Line 7 is executed provided that two endpoints of a cheap edge are not in other distinguished sets. Line 9 terminates the loop when the number of cheap edges on x first reaches Δ (i.e., $CheapNo[x]$=Δ). Note that $C(e_i)$ is also obtained when terminating the **FOR** loop of lines 4-12 and $C(e_i)$ is used as the upper bound of edge costs to select $x's$ nearest neighbours in line 13. The algorithm first adds x to the current distinguished set (line 13) and the **FOR** loop of lines 14-20 iteratively adds each of x's nearest neighbours u to the current distinguished set. Here the nearest neighbours are connected with x by $x's$ incoming edges with cost less than $C(e_i)$. Line 17 guarantees that this algorithm never fills the distinguished set over its capacity. This algorithm terminates when all of x's corresponding nearest neighbours are added to the current distinguished set, or the current distinguished set is big enough.

In the subroutine CHECK_INCOMING_STAR, Δ is a criteria to select the cheap-edge core. If Δ is set to be only one, the overall algorithm of CHEAP_EDGE_ADD given in Figure 6.14 applies the nearest neighbour strategy to add vertices to the distinguished set, which focuses on a cheap-edge cluster of the cyclic type if there exists some; if Δ is very close to the size of the distinguished set, the algorithm of CHEAP_EDGE_ADD adds a cheap-edge core with as many nearest neighbours as possible to each distinguished set, which focuses on a cheap-edge cluster of the star type.

In our experiments, Δ is specified by the SET_INCOMING_DELTA procedure. The key concept is to count the incoming cheap edges for each vertex and choose the maximum number of incoming cheap edges of a single vertex.

The inputs of the **SET_INCOMING_DELTA** procedure include some global variables such as G (the graph representing the instance). In addition the inputs also include marked (the set of vertices that are in other distinguished sets), α (the parameter used to define the cheap edge cost) and S (a sequence of edges sorted by costs in increasing order). The output is Δ which specifies the number of edges of a cheap-edge core of the star type.

Figure 6.16 gives the pseudocode for SET_INCOMING_DELTA. Line 1 defines the set of edges E' whose endpoints are not distinguished in other abstractions. Lines 2-3 define the minimum and the maximum edge cost, respectively. Line 4 defines σ which is used as the upper bound of the cheap edge cost. After the initialization (lines 5-6), the **WHILE** loop of lines 7-12 counts the number of

Algorithm: SET_INCOMING_DELTA

Key Concept: to specify the number of incoming cheap edges connected with a cheap edge core of the star type.

Input: marked (the set of vertices that are in other distinguished sets.)
α (the parameter used to define the cheap edge cost.)
S (a sequence of edges $< e_1, ..., e_m >, \forall i, C(e_i) \leq C(e_{i+1})$)

Output: Δ

1 $E' = \{y \rightarrow x \in \{e_1, ..., e_m\} | \{x, y\} \cap$ marked=$\emptyset\}$.
2 $min = min_{e \in E'} C(e)$.
3 $max = max_{e \in E'} C(e)$.
4 $\sigma = min + (max - min) \times \alpha$. //$\sigma$ is the upper bound of the cheap edge cost.
5 $degree[v] = 0, \forall v \notin marked$.
6 $i = 1$. // Initialize the edge index i.
7 **WHILE** ($C(e_i) \leq \sigma$) **DO**
8 **IF** ($e_i \in E'$ AND $y \rightarrow x = e_i$)
9 $degree[x]$=$degree[x]$+1.// update the in-degree
10 **END IF**
11 i=i+1;
12 **END WHILE**
13 $\Delta = max_{v \notin marked}\{degree[v]\}$.

Figure 6.16: The method to set Δ, the number of cheap edges connected with the cheap-edge core of the star type.

cheap edges for each unmarked vertices. The last line defines the value of Δ.

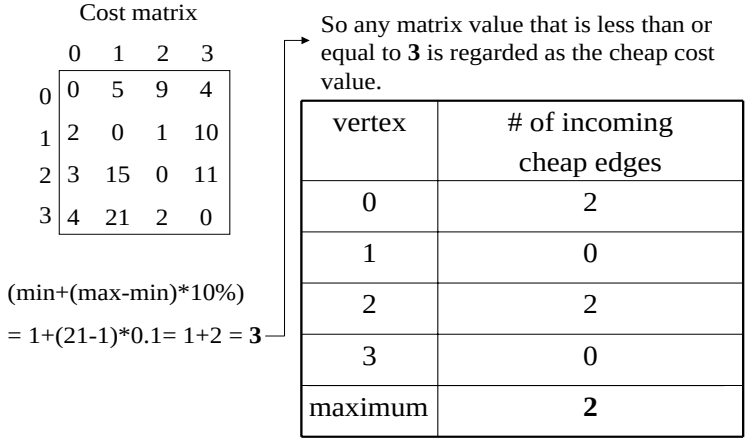


Figure 6.17: An example to calculate Δ.

In the algorithm the parameter α is used to define the cheap edge cost. In our experiments, α is 10%.

Figure 6.17 gives an example for the computation of the cheap edge cost σ and Δ. In this example, $\alpha = 10\%$, min=1 and max=21 as we do not count the value in the diagonal line. So any

matrix value which is less than or equal to 3 (i.e., σ=1+(21-1)$\times$10%=3) is regarded as a cheap cost value. As shown in the table of Figure 6.17, assuming that vertices 0,1,2,3 are not distinguished in any abstraction we calculate the total number of cheap incoming edges for each vertex by looking in the corresponding column. Δ is estimated to be 2 as the maximum number of cheap edges for each column is 2.

The CHECK_OUTGOING_STAR Procedure

Algorithm: CHECK_OUTGOING_STAR
Key Concept:To add a cheap-edge cluster of the outgoing star type to the distinguished set.
Input: $Index$ (the index of the current distinguished set)
S (a sequence of edges $< e_1, ..., e_m >, \forall i, C(e_i) \leq C(e_{i+1})$)
α (a parameter used to define the cheap edge cost.)
Output: x (the cheap-edge core).

```
1  Define marked = ∪_{j≠Index} D_j.
2  Obtain Δ from the output of SET_OUTGOING_DELTA(marked,α,S).
3  CheapNo[v] = 0,∀v ∉ marked. // Initialization.
4  FOR i = 1 To m DO
5      Define β_i = {x, y} where x → y = e_i.
6      IF (β_i ∩ marked =∅)
7          CheapNo[x]=CheapNo[x]+1.
8          IF (CheapNo[x]=Δ)
9            break; // terminate the loop and x is identified to be a cheap-edge core.
10         END IF
11     END IF
12 END FOR
13  D_Index = D_Index ∪ {x}.
14 FOR each vertex u ∈ {u : C(x → u) ≤ C(e_i), u ∉ marked } Do
15     IF (|D_Index| < n_Index)
16         D_Index = D_Index ∪ {u}.
17     ELSE
18         break;
19     END IF
20 END FOR
```

Figure 6.18: The greedy method designed to identify cheap-edge cores of the outgoing star type.

In the description of CHEAP_EDGE_ADD, we mentioned that if we change the definition to charge the primary cost of a move when a distinguished state is reached, we will adapt CHEAP_EDGE_ADD to focus on the cheap-edge cluster of the outgoing star type. One step is to apply the subroutine CHECK_OUTGOING_STAR instead of CHECK_INCOMING_STAR.

Figure 6.18 gives the pseudocode of CHECK_OUTGOING_STAR which has a similarity to the pseudocode of CHECK_INCOMING_STAR. The differences lie in the following three lines.

- In line 2, obtain Δ from the output of **SET_OUTGOING_DELTA**(marked,α,S). The pseudocode of SET_OUTGOING_DELTA is almost the same to that of SET_INCOMING_DELTA

(shown in Figure 6.16) except that in SET_OUTGOING_DELTA, line 9 is updated to be $degree[y]$=$degree[y]$+1 which counts the out-degree of the start point of the edge e_i.

- In line 5, $x \rightarrow y = e_i$, i.e., x is changed to be the start point and y is the end point of the edge e_i. Thus line 7 counts the number of outgoing edges.

- In line 13, $u \in \{u : C(x \rightarrow u) \leq C(e_i),\ u \notin$ marked $\}$, i.e., vertex u is x's neighbour connected by a cheap edge from x to u.

The NEAREST_NEIGHBOUR Procedure

Algorithm: NEAREST_NEIGHBOUR
Key Concept: To add unmarked nearest neighbours to the current distinguished set.
Input: $Index$ (the index of the current distinguished set)
$start$ (the start point of the partial path)
end (the end point of the partial path)
Output: $< D_1, D_2, ..., D_k >$.
1 $PathS = \{start, end\}$. //Initialize a set of vertices on the path.
2 **REPEAT**
//To define a set of the end point's outgoing edges.
3 Define E_1=$\{u \rightarrow v$:$u = end$ and $v \notin ((\cup_{j \neq Index} D_j) \cup PathS)\}$.
//To define set of the $start$ point's incoming edges.
4 Define E_2=$\{u \rightarrow v$:$v = start$ and $u \notin ((\cup_{j \neq Index} D_j) \cup PathS)\}$.
5 Select the shortest edge e from $E_1 \cup E_2$ such that $C(e) \leq C(e')$, $\forall e' \in E_1 \cup E_2$
6 Define $\beta_e = \{x, y\}$ where $x \rightarrow y = e$.
7 **IF** ($x = end$) //The partial path is extended at the end point.
8 $end = y$
9 **ELSE** // $y = start$, The partial path is extended at the $start$ point.
10 $start = x$
11 **END IF**
12 $PathS = PathS \cup \beta_e$.
13 $D_{Index} = D_{Index} \cup PathS$.
14 **UNTIL** ($|D_{Index}| = n_{Index}$)

Figure 6.19: The greedy method using nearest neighbour strategy.

NEAREST_NEIGHBOUR is a subroutine applying the nearest neighbour strategy. Its inputs include some global variables such as the graph G representing an instance and $< D_1, D_2, ..., D_k >$ which specifies the distinguished vertices for each abstraction. Besides those variables the inputs also include $Index$ (the index of the current distinguished set), and two endpoints of a cheap edge (x,y). The outputs include the global variables such as $< D_1, D_2, ..., D_k >$.

Figure 6.19 gives the pseudocode of NEAREST_NEIGHBOUR. When it is called, this algorithm starts from the edge $start \rightarrow end$, then iteratively applies the nearest neighbour strategy to construct a piece of a path. When we apply the nearest neighbour strategy to choose the unmarked nearest neighbour we choose in the range of vertices that are not in other distinguished sets and not in the current path $PathS$ (i.e.,$v \notin ((\cup_{j \neq Index} D_j) \cup PathS)$). In line 5 a shortest edge is selected

from $E_1 \cup E_2$. Recall that NEAREST_NEIGHBOUR is called by CHEAP_EDGE_ADD only when $|D_i| < n_i$, therefore $E_1 \cup E_2 \neq \emptyset$. All vertices along the path are added to the distinguished set D_{Index} until the set is big enough, i.e., it iteratively adds a nearest vertex that is not distinguished in other distinguished sets and does not belong to the current path $PathS$. We store the vertices along the path in $PathS$ as shown in line 12. Line 13 adds all vertices along the path to the current distinguished set.

This method can detect some cheap-edge cores of the cyclic type and mark these vertices distinguished until the current distinguished set is big enough. For example, in the original state space shown on the left part of Figure 6.11, there exists a cycle consisting of x, y, z and d connected by edges that cost one. Assume that other edge costs are larger than one. Given distinguished vertices x and y, this method will also mark vertices z and d distinguished provided that they are not marked in other distinguished sets. However, if z or d is distinguished in another abstraction, this method will not identify this cheap-edge cluster. Thus there also exist certain situations which make this method not yield the best abstractions.

Note that NEAREST_NEIGHBOUR has a step to select the shortest edge. If there is more than one candidate, that is, there is more than one vertex that has the same distance to one of the endpoints of the current path, we choose the vertex that has more constraints related to the distinguished set of this abstraction. Formally, we define $P_v = \{u : u <_p v \text{ or } v <_p u, u \in D_{Index}\}$ where D_{Index} is the current distinguished set. Suppose U is the set of candidates which have the same distance to one endpoint of the current path. If $|P_x| > |P_y|, \forall y \in U$, then vertex x is the choice.

As we have discussed in a Section 6.2, we often focus on cheap edges. It is because the cheap edges play very important roles when designing good abstractions. If there is more than one candidate for a choice, constraints are the secondary consideration.

6.3.3 Combining Greedy Methods for h_{add}

Algorithm: HYBRID ADD
Key Concept: To combine greedy methods for expensive-edge cluster and cheap-edge core.
Input: G,M,ω,α,$< n_1, n_2, ..., n_k >$.
Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.
1 CHECK_EXPENSIVE (M,$< n_1, n_2, ..., n_k >$,ω).
2 CHEAP_EDGE_ADD (G, α, $< n_1, n_2, ..., n_k >$, $< D_1, D_2, ..., D_k >$).

Figure 6.20: The hybrid method

Our methods for defining abstractions combine the greedy methods for expensive-edge cluster and cheap-edge core. Figure 6.20 gives the pseudocode of HYBRID ADD. First we call the procedure CHECK_EXPENSIVE to detect the expensive-edge clusters, then we call the greedy algorithm to add the cheap-edge clusters and their corresponding neighbours to the distinguished sets.

6.4 Greedy Methods for h_{max}

In this section we design abstractions for h_{max}. We present algorithms to solve the problem defined as follows:

- **Input:** An SOP instance ($G = \langle V, E, C, P \rangle$ and the cost matrix M), and a sequence of numbers $< n_1, n_2, ..., n_k >$ specifying the size of the distinguished set in each abstraction, where k is the total number of abstractions and $\Sigma_{i=1}^{k} n_i \leq |\mathcal{I}|$ where $\mathcal{I} = V \setminus \{s, g\}$[3].
- **Output:** a sequence of distinguished sets $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

As for the output, we require that $D_i \cap D_j = \emptyset, \forall i \neq j$ although distinguished sets of abstractions for h_{max} can be overlapped. Our reason for this is that during search DFBB need information (i.e., heuristics) of some states, and the information may be more helpful if some designated vertices are distinguished in the abstraction. For example, in the leftmost part of Figure 6.21 edges of the search tree are labelled by the vertex to be visited next, and states are labelled with numbers. For state 1, there are three vertices (d, e and x) left before DFBB reaches a goal state, and information is needed to determine the best order of these last three vertices in the solution path. Note that we have the same heuristic values for these three states (state 2, state 3 and state 4) in the first abstraction (shown in the middle of Figure 6.21) because vertices d, e and x are all don't-care vertices in the first abstraction. Therefore we need an abstraction in which vertices d, e and x are distinguished. The second abstraction (shown on the right part of Figure 6.21) is a good abstraction for this situation.

In reality it is impossible to include all combinations of vertices in abstractions. For each abstraction we consider the underlying keys to a good abstraction, i.e., we identify the expensive-edge clusters and the cheap-edge clusters in each abstraction.

One main difference between algorithms for h_{add} and those for h_{max} lies in that h_{max} is calculated by counting all edge costs along the abstract path, while h_{add} is calculated by only counting primary costs of edges along the path. In this section, first we describe the method called CHECK_EXPENSIVE_MAX to check for the expensive-edge clusters for standard abstractions. Then we present the greedy method CHEAP_EDGE_MAX designed for the cheap-edge clusters. Finally in Section 6.4.3 we combine these methods to consider both the expensive-edge clusters and the cheap-edge clusters for designing good abstractions.

6.4.1 The CHECK_EXPENSIVE_MAX Procedure

Figure 6.4.1 gives the pseudocode for the CHECK_EXPENSIVE_MAX procedure. It is similar to the CHECK_EXPENSIVE procedure in Section 6.3.1. The only difference lies in line 24 where we sort the sequence to specify the size of each distinguished set by the non-increasing order. The

[3]Here we are assigning the intermediate vertices $\mathcal{I} = V \setminus \{s, g\}$ to each distinguished set. The start s and goal g vertices are always distinguished in each abstraction.

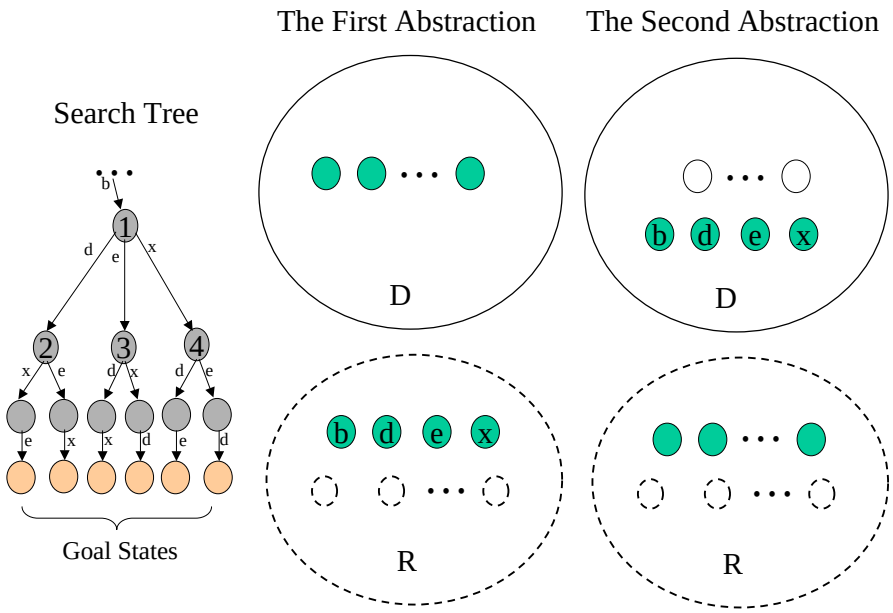


Figure 6.21: An example to explain how to choose complementary abstractions.

reason for this is that for additive abstractions, each abstraction contributes to the value of h_{add}, while for standard abstractions, only one abstraction will contribute to the value of h_{max}. Here we first add the expensive-edge clusters to the larger distinguished set as the abstraction of larger size often determines the value of h_{max}.

6.4.2 The CHEAP_EDGE_MAX Procedure

Our experiments are focused on large instances for which it is impossible for the abstraction to contain many distinguished vertices compared to the total number of vertices. According to the discussions in Section 6.2.3, we apply Scheme II in this situation.

CHEAP_EDGE_MAX is designed based on Scheme II which was described in Section 6.2. It aims to maximize the minimum edge cost between don't-care vertices in order to avoid using cheap edges many times (if $|R|$ is much larger than $|D|$, i.e., the size of the don't-care set is much larger than the size of the distinguished set).

Figure 6.23 gives the pseudocode for the CHEAP_EDGE MAX procedure. The inputs include $G, < n_1, n_2, ..., n_k >$ and $< D_1^0, D_2^0, ..., D_k^0 >$ (a sequence of initial distinguished sets). The output is $< D_1, D_2, ..., D_k >$ which specifies the distinguished vertices for each abstraction.

For each distinguished set that is not big enough (line 5), this algorithm iteratively adds edges with the same edge costs to a subgraph $G'(V', E')$ and for each subgraph it tries to find a minimum vertex cover for E' such that at least one vertex of the vertex cover connects with each edge of E'. The problem of finding a minimum vertex cover was one of Karp's 21 NP-complete problems

Algorithm: CHECK_EXPENSIVE_MAX

Key Concept: To identify expensive-edge clusters and assign them to distinguished sets.

Input: M (the cost matrix)
$< n_1, n_2, ..., n_k >$ (a sequence of numbers specifying the size of each distinguished set)
ω (the lower bound of the expensive edge cost.)

Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| \leq n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

$D_i = \emptyset, \forall i$. // Initialization.
$CIndex$=0.// Initialize the cluster index.
$U_0 = \emptyset$.
FOR each vertex u **DO**
  Define U_{mark}= $\cup_{j=0}^{CIndex} U_j$.//U_{mark} include vertices identified to belong to an expensive-edge cluster.
  IF ($u \notin U_{mark}$) **AND** ($M_{u,v} \geq \omega, \forall v \in V \setminus \{u\}$) //all outgoing edges are expensive.
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = \{u\}$.
  ELSE IF ($u \notin U_{mark}$) **AND** ($M_{v,u} \geq \omega, \forall v \in V \setminus \{u\}$) //all incoming edges are expensive.
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = \{u\}$.
  // Check the first definition for the expensive-edge cluster consisting of more than one vertex.
  ELSE IF ($\exists U$, $u \in U$ **AND** 2$\leq |U| < |V|$) **AND** ($M_{u,u_i} < \omega$,$\forall u_i \in U \setminus \{u\}$) **AND** ($U \cap U_{mark} = \emptyset$)
      AND ($M_{u_i,v} \geq \omega$ **AND** $M_{u_i,u_j} < \omega$,$\forall u_i$,$u_j \in U$, $v \in V \setminus U$)
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = U$.
  // Check the second definition for the expensive-edge cluster consisting of more than one vertex.
  ELSE IF ($\exists U$, $u \in U$ **AND** 2$\leq |U| < |V|$) **AND** ($M_{u,u_i} < \omega$,$\forall u_i \in U \setminus \{u\}$) **AND** ($U \cap U_{mark} = \emptyset$)
      AND ($M_{v,u_i} \geq \omega$ **AND** $M_{u_i,u_j} < \omega$,$\forall u_i$,$u_j \in U$, $v \in V \setminus U$)
    $CIndex$ = $CIndex$ + 1.
    $U_{CIndex} = U$.
  END IF
END FOR
IF ($CIndex > 0$)
  SORT all expensive-edge clusters by the size,i.e., $< U_{\tau(1)}, ..., U_{\tau(CIndex)} >$, $\forall i$, $|U_{\tau(i)}| \leq |U_{\tau(i+1)}|$.
  SORT $< n_{\pi(1)}, n_{\pi(2)}, ..., n_{\pi(k)} >$ such that $n_{\pi(i)} \geq n_{\pi(i+1)}$,$\forall i$.
  $t = 1$. // $\pi(t)$ is the index of each distinguished set.
  FOR $i = 1$ to $CIndex$ **DO**
    IF ($t \leq k$ **AND** $|D_{\pi(t)} \cup U_{\tau(i)}| \leq n_{\pi(t)}$)
      $D_{\pi(t)} = D_{\pi(t)} \cup U_{\tau(i)}$.
    ELSE
      REPEAT
        $t = t + 1$.
      UNTIL ($|D_{\pi(t)} \cup U_{\tau(i)}| \leq n_{\pi(t)}$ **OR** $t > k$).
      IF ($t > k$)
        break; // no more distinguished set to accommodate the cluster.
      ELSE
        $D_{\pi(t)} = D_{\pi(t)} \cup U_{\tau(i)}$.
      END IF
    END IF
  END FOR
END IF

Figure 6.22: The method to check for the expensive-edge cluster.

Algorithm: CHEAP_EDGE_MAX

Key Concept: Applying Scheme II for the cheap-edge cores.

Input: $G, < n_1, n_2, ..., n_k >$, and $< D_1^0, D_2^0, ..., D_k^0 >$ where $D_i^0 \cap D_j^0 = \emptyset, \forall i \neq j$.

Output: $< D_1, D_2, ..., D_k >$ such that $D_i^0 \subseteq D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

E': a set of edges in G, $E' \subseteq E$.

V': the set of endpoints of E', $V' = \{u, v : u \rightarrow v \in E'\}$.

G': a subgraph consisting of E' and V'.

$Degree[v]$: the degree (in-degree plus out-degree) of vertex v in G'. It is updated whenever G' changes.

```
1  SORT all edges by edge costs, S =< e_1, ..., e_m >, ∀i, C(e_i) ≤ C(e_{i+1}).
2  D_i = D_i^0, ∀i. // Initialization.
3  FOR Index=1 To k DO
4    i = 1; // i is the index of the edge.
5    WHILE (|D_Index| < n_Index) DO
6       Define marked = ∪_{j=1}^{k} D_j.
7       E' = ∅, V' = ∅
8       Cost = C(e_i).// set the current edge cost.
9       WHILE (C(e_i) = Cost) OR (E' = ∅) DO
10            Define β_i = {u, v} where e_i = u → v.
11            IF (β_i ∩ marked =∅) // both endpoints of e_i are not distinguished in any abstraction.
12              E' = E' ∪ {e_i}. // add e_i to G'
13              V' = V' ∪ β_i. // add endpoints of e_i to G'
14            END IF
15            i = i + 1.
16            IF (C(e_i) ≠ Cost) AND (E' = ∅)
17              Cost = C(e_i). // change the current edge cost.
18            END IF
19      END WHILE
20      REPEAT
21        Select v' from V' where Degree[v'] ≥ Degree[u], ∀u ∈ V'.
22        D_Index = D_Index ∪ {v'};
23        V' = V' \ {v'}. // delete v' from G'
24        E' = E' \ {u → v', v' → u}, ∀u ∈ V' . // delete all edges on v' from G'
25      UNTIL ((E' = ∅) OR (|D_Index| = n_Index)).
26    END WHILE
27 END FOR
```

Figure 6.23: The CHEAP_EDGE MAX procedure.

[59] and is therefore a classical NP-complete problem in computational complexity theory. Here we apply the greedy strategy to this problem by choosing the vertex with the largest degree (the sum of in-degree and out-degree). Lines 6-7 initialize the set to mark the distinguished vertices and the subgraph $G'(V', E')$. Line 8 sets a standard edge cost $Cost$ to form a group of edges which have the same edge cost ($Cost$).

The **WHILE** loop of lines 9-19 adds edges together with two endpoints on these edges to the subgraph G'. Line 15 obtains the index of the next edge. Lines 16-18 changes the current standard edge cost if there is no edge in E'. This happens when all edges with costs $Cost$ have been marked distinguished in another abstraction.

The **REPEAT** loop of lines 20-25 iteratively selects the vertex v' with the largest degree in G',

adds v' to the distinguished set, deletes v' as well as edges on v' from $G'(V', E')$, and updates the degree of the remaining vertices. Note that if two vertices u, v have the same degree (i.e., $Degree[u] = Degree[v]$), the vertex involved with more precedence constraints will be chosen. Formally we define P_x =$\{y : y \in D_{Index}$ and $(x <_p y$ or $y <_p x)\}$. If $Degree[u] = Degree[v]$ and $|P_u| > |P_v|$, then u is the choice. The loop is executed until $|D_{Index}| = n_{Index}$ or all edges in E' are deleted, i.e., the distinguished set forms a vertex cover for E' that is constructed in the **WHILE** loop of lines 9-19.

There is a step of sorting edges by costs in non-decreasing order in this algorithm (line 1). As in previous sections we define $P_e = \{u : x \rightarrow y = e, u <_p x$ or $x <_p u$ or $y <_p u$ or $u <_p y\}$. If $|P_{e_i}| > |P_{e_j}|$ and C(e_i)=C(e_j), then e_i is in front of e_j in the sequence, i.e., $S =< e_1, ..., e_i, e_j, ..., e_m >$.

6.4.3 Combining Greedy Methods for h_{max}

Algorithm: HYBRID MAX
Key Concept: To combine greedy methods for expensive-edge cluster and cheap-edge core.
Input: G, M, ω, $< n_1, n_2, ..., n_k >$.
Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.
1 CHECK_EXPENSIVE_MAX (M, $< n_1, n_2, ..., n_k >$, ω).
2 CHEAP_EDGE_MAX (G, $< n_1, n_2, ..., n_k >$, $< D_1, D_2, ..., D_k >$).

Figure 6.24: The hybrid method

Our methods for defining abstractions combine the greedy methods for the expensive-edge clusters and the cheap-edge clusters. Figure 6.24 illustrates HYBRID MAX. First we call the procedure CHECK_EXPENSIVE_MAX to detect the expensive-edge clusters, then we call the greedy algorithm to add cheap-edge clusters to the distinguished sets.

6.5 The Greedy Method for Precedence Constraints

A distinct feature of the SOP instance is the existence of the precedence constraints and it is attractive to design abstractions regarding these constrains as the first consideration. For the purpose of comparison we define a greedy method called GREEDY_CONSTRAINT to group vertices by just considering the precedence constraints instead of the edge costs.

Assume that the transitive closure of the precedence relation has been computed for G. The inputs of the GREEDY_CONSTRAINT procedure include G and $< n_1, n_2, ..., n_k >$. The output is $< D_1, D_2, ..., D_k >$ which specifies the distinguished vertices for each abstraction.

Figure 6.25 gives the pseudocode for GREEDY_CONSTRAINT. The key concept is to group vertices with corresponding precedence constraints. The **FOR** loop of lines 2-20 fills each distinguished set to capacity. The **WHILE** loop of lines 3-19 iteratively defines P_v to be the set of vertices

Algorithm: GREEDY_CONSTRAINT
Key Concept: To group vertices with corresponding precedence constraints.
Input: $G, < n_1, n_2, ..., n_k >$
Output: $< D_1, D_2, ..., D_k >$ where $D_i \subseteq V$, $|D_i| \leq n_i$, and $D_i \cap D_j = \emptyset, \forall i \neq j$.

```
1  D_i = ∅, ∀i. // Initialization.
2  FOR i = 1 To k Do// i is the index of the distinguished set.
3    WHILE (|D_i| < n_i)
4      Define marked = ∪_{j≠i} D_j.
5      Define P_v = {u : (u <_p v or v <_p u) and u ∉marked}.
6      Select x where x ∉ (D_i∪ marked) and ∀y ∉ (D_i∪ marked), |P_x| ≥ |P_y|.
7      IF (|P_x| = 0)// the situation when no more constraints are to be considered.
8        RETURN //terminate the overall method and return < D_1, D_2, ..., D_k >.
9      END IF
10     D_i = D_i ∪ {x}.
11     Sort all vertices of P_x, < u_1, u_2, ..., u_m > such that ∀i,u_i ∈ P_x and |P_{u_i}| ≥ |P_{u_{i+1}}|.
12     FOR u = u_1 TO u_m Do // add each vertex of P_x to D_i.
13       IF (|D_i| < n_i)
14         D_i = D_i ∪ {u}
15       ELSE
16         break // to terminate the FOR loop if |D_i| = n_i.
17       END IF
18     END FOR
19   END WHILE
20 END FOR
```

Figure 6.25: The greedy method designed to group vertices with corresponding precedence constraints.

having constraints related to v, selects a vertex x related with more precedence constraints, adds x to D_i, and adds each vertex of P_x to D_i, until $|D_i| = n_i$. Line 4 defines the set of vertices that are not distinguished in other abstractions. Line 5 defines P_v to be the set of vertices having constraints related to v. Line 6 selects a vertex x that is not distinguished in any abstractions and is related to the largest number of precedence constraints. Lines 8 is executed to terminate the overall algorithm if no more constraints are to be considered. Line 10 adds vertex x to D_i. Lines 11-18 adds each vertex of P_x to D_i, until $|D_i| = n_i$.

Algorithm: GreedyP ADD
Key Concept: To combine greedy methods by first considering the constraints and then considering edge costs.
Input: G, α, $< n_1, n_2, ..., n_k >$.
Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \phi, \forall i \neq j$.

```
1 GREEDY_CONSTRAINT(G, < n_1, n_2, ..., n_k >).
2 CHEAP_EDGE_ADD (G, α, < n_1, n_2, ..., n_k >, < D_1, D_2, ..., D_k >).
```

Figure 6.26: The hybrid method for h_{add}

As shown in Figures 6.26 and 6.27 the GREEDY_CONSTRAINT method is used to create abstractions for both h_{add} and h_{max} with the combination of methods considering edge costs.

Algorithm: GreedyP MAX
Key Concept: To combine greedy methods for expensive-edge cluster and cheap-edge core.
Input: $G, < n_1, n_2, ..., n_k >$.
Output: $< D_1, D_2, ..., D_k >$ such that $D_i \subseteq V$, $|D_i| = n_i$, and $D_i \cap D_j = \phi, \forall i \neq j$.
1 GREEDY_CONSTRAINT($G, < n_1, n_2, ..., n_k >$).
2 CHEAP_EDGE_MAX II ($G, < n_1, n_2, ..., n_k >, < D_1, D_2, ..., D_k >$).

Figure 6.27: The hybrid method for h_{max}

6.6 Experimental Results

First, we define the following heuristics used in the experiments.

- greedy h_{add}: h_{add} generated from abstractions defined by the HYBRID ADD method.
- greedy h_{max}: h_{max} generated from abstractions defined by the HYBRID MAX method.
- random h_{add}: additive heuristics generated by abstractions defined by choosing distinguished vertices randomly.
- random h_{max}: h_{max} generated from abstractions defined by choosing distinguished vertices randomly.
- greedyP h_{add}: h_{add} generated from abstractions defined by the GreedyP ADD method.
- greedyP h_{max}: h_{max} generated from abstractions defined by the GreedyP MAX method.
- neighbourhood heuristics $h_{neighbor}$: given an instance of the SOP defined by a weighted directed graph $G = \langle V, E, C, P \rangle$ with the start s and goal g vertices designated, and given an intermediate state $t = (\ell, \mathcal{U})$ and a goal state $goal = (g, \emptyset)$, recall that in Section 6.1.6 neighbourhood heuristics for an SOP state t are defined as follows.

  $h_{out}(t, goal) = \sum_{v \in \mathcal{U} \cup \{\ell\}} Out(v)$ where $Out(v)$ is the minimum edge cost of v's outgoing edges whose other endpoint is $u, u \in \mathcal{U} \cup \{g\}$.

  $h_{in}(t, goal) = \sum_{v \in \mathcal{U} \cup \{g\}} In(v)$ where $In(v)$ is the minimum edge cost of v's incoming edges whose other endpoint is $u, u \in \mathcal{U} \cup \{\ell\}$.

  In our experiments the neighbourhood heuristic value $h_{neighbour}$ is the maximum value of the above two heuristics, i.e., $h_{neighbour}(t, goal) = max\{h_{out}(t, goal), h_{in}(t, goal)\}$.

The aim of our experiments is to answer the following key questions.

- How do greedy h_{add} and h_{max} compare to random h_{add} and h_{max}?
- How do algorithms for creating greedy h_{add} and h_{max} that focus on edge costs compare to those that focus on constraints for the SOP?

- How do greedy h_{add} and h_{max} compare to neighbourhood heuristics for the SOP?
- How does greedy h_{add} compare to greedy h_{max} for the SOP?
- How well do the ideas learned for creating good additive abstractions for non-unit-cost problems while studying the SOP transfer to other problems with a different structure?

6.6.1 Experiments with Selected SOP Instances of TSPLIB

The DFBB algorithm for the SOP described in Section 6.1.5 is used in the experiments. Algorithms are coded in C on a machine with an AMD Athlon(tm) 64 Processor 3700+ with a 2.4 GHz clock rate and 2GB main memory.

The experiments of this section are running on benchmark problems contained in the Traveling Salesman Problem Library (TSPLIB) [87, 89]. Twenty larger SOP problems are chosen from TSPLIB and DFBB is run on them with a 30-minute time limit. We select these problems according to three criteria. First the selected problems are of different sizes and structures. Second the problems have more than 40 vertices. Third, problems of the same size have different number of random precedence constraints. According to [2], problems selected from TSPLIB can be classified as follows.

- Problems ft53.x, ft70.x, ry48p.x and kro124p.x are generated from ATSP instances of TSPLIB by adding k random precedence constraints to the $n \times n$ cost matrix, where $k = \frac{n}{4}, \frac{n}{2}, n, 2 \times n$ corresponding to the problem name extension (.1, .2, .3, .4).
- Problems prob100 and prob42 are randomly generated problems.
- Problems rbg378a and rbg358a are real-life problems derived from a stacker crane application[1].

Note that the above numbers ($\frac{n}{4}, \frac{n}{2}, n, 2 \times n$) are just the number of random constraints added to the instances with a check to guarantee that there are no cycles for the constraints. Since the precedence relation is transitive (i.e., $x <_p y$, $y <_p z \Rightarrow x <_p z$), the number of precedence constraints may increase when the transitive closure of the relation is computed for each instance.

Table 6.2 presents more detailed properties of each SOP instance. The **Name** column shows the name of the instance. The **n** column gives the size of the cost matrix in the instance. The **Best Known Solution** column provides the optimal solution cost (or the best known solution range if the optimal cost is not known) of each instance listed on TSPLIB [2]. The **No.of Constraints** column is the total number of constraints among the intermediate vertices after the transitive closure has been computed. The constraints related to the start and the goal vertices are not counted in this column, because to designate the start and goal vertices there are always $n - 1$ constraints for the start and goal vertices, respectively. The $P\%$ column provides the actual percentage of constraints after the transitive closure has been computed. $P\% = (m \times 100\%)/\binom{|\mathcal{I}|}{2}$ where m is the total number of

Name	n	Best Known Solution	No.of Constraints	$P\%$	Edge Range
ry48p.1	49	[15220,15805]	12	1.1	[54,2782]
ry48p.2	49	[15524,16666]	26	2.4	[54,2782]
ry48p.3	49	[18156,19894]	132	12.2	[54,2782]
ry48p.4	49	[29967,31446]	596	55.1	[54,2782]
ft53.1	54	[7438,7570]	12	0.9	[21,1834]
ft53.2	54	[7630,8335]	30	2.3	[21,1834]
ft53.3	54	[9473,10935]	217	16.4	[21,1834]
ft53.4	54	14425	759	57.2	[21,1834]
ft70.1	71	39313	17	0.7	[331,2588]
ft70.2	71	[39739,40422]	48	2.1	[331,2588]
ft70.3	71	[41305,42535]	215	9.2	[331,2588]
ft70.4	71	[52269,53562]	1,325	56.5	[331,2588]
kro124p.1	101	[37722,40816]	33	0.7	[81,4545]
kro124p.2	101	[38534,41677]	68	1.4	[81,4545]
kro124p.3	101	[40967,50876]	256	5.5	[81,4545]
kro124p.4	101	[64858,76103]	2,305	47.5	[81,4545]
prob100	100	[1024,1385]	41	0.9	[1,500]
prob42	42	243	19	2.4	[1,100]
rbg378a	380	[2761,2883]	63,585	89.2	[0,33]
rbg358a	360	[2518,2599]	56,536	88.4	[0,33]

Table 6.2: Properties of selected SOP instances in TSPLIB.

constraints among the intermediate vertices after the transitive closure has been computed (i.e., the values shown in the **No.of Constraints** column), and $\binom{|\mathcal{I}|}{2}$ is the total number of unordered pairs of intermediate vertices. The **Edge Range** column shows the range of the edge cost, i.e., the range of values in the cost matrix.

Table 6.3 shows the scheme of abstractions and the size of memory that we used for each SOP instance. The **Name** and **n** columns are as in Table 6.2. The **Abs** column shows the set of abstractions used to generate heuristics. For example (10*2)(9*3) means that there are 5 abstractions: 2 abstractions with 10 distinguished vertices in each abstraction, and 3 abstractions with 9 distinguished vertices in each abstraction. The **Memory** column indicates the total size of each set of PDBs used to store heuristics.

Note that the number of distinguished vertices in each of our abstractions ranges from 8 to 10. In our experiments that is enough to show the benefits of using our greedy methods when designing abstractions. In addition each corresponding PDB can be computed in reasonable time and accessed easily for the heuristic search algorithms.

In the following Tables 6.4–6.10, we compare heuristics according to the suboptimal solution costs found by DFBB within the 30-minute time limit. The **Problem** column shows the name of the TSPLIB instance. The **n** column gives the size of the cost matrix in the instance. The third and fourth columns of Tables 6.4–6.10 are the suboptimal solution costs found by DFBB with the 30-minute time limit using the heuristics as shown on the headings of these columns. For example, in Table 6.4, the third and fourth columns are the suboptimal solution costs found by DFBB using greedy h_{add}

Problem	n	Abs	Memory
ry48p.1	49	(10*2)(9*3)	792,832
ry48p.2	49	(10*2)(9*3)	792,832
ry48p.3	49	(10*2)(9*3)	792,832
ry48p.4	49	(10*2)(9*3)	792,832
ft53.1	54	(9*4)(8*2)	608,256
ft53.2	54	(9*4)(8*2)	608,256
ft53.3	54	(9*4)(8*2)	608,256
ft53.4	54	(9*4)(8*2)	608,256
ft70.1	71	(10*6)(9*1)	2,376,960
ft70.2	71	(10*6)(9*1)	2,376,960
ft70.3	71	(10*6)(9*1)	2,376,960
ft70.4	71	(10*6)(9*1)	2,376,960
kro124p.1	101	(9*11)	2,813,184
kro124p.2	101	(9*11)	2,813,184
kro124p.3	101	(9*11)	2,813,184
kro124p.4	101	(9*11)	2,813,184
prob100	100	(9*10)(8*1)	2,645,504
prob42	42	(10*4)	757,760
rbg378a	380	(10*36)(9*2)	83,662,848
rbg358a	360	(10*34)(9*2)	74,840,064

Table 6.3: The scheme of abstractions and the size of memory that we used.

and random h_{add}, respectively. The $\delta\%$ column compares the suboptimal solution costs obtained by using two different heuristics. It is measured as $\frac{U_2-U_1}{U_{best}} \times 100\%$ where U_1 is the solution cost presented in the third column, U_2 is the solution cost presented in the fourth column, and U_{best} is the best upper bound of the solution presented in Table 6.2. A value of $\delta\%$ greater than zero indicates that U_1 is closer to the best known solution than U_2, and vice versa. The larger absolute value of $\delta\%$, the greater difference between U_1 and U_2 relative to U_{best}. The **Ratio** column compares greedy h_{add} (or greedy h_{max}) to the best known upper bound of the solution presented in Table 6.2. It is measured as $\frac{U_1}{U_{best}}$ where U_1 is the solution cost presented in the third column, and U_{best} is the best upper bound of the solution presented in TSPLIB. A ratio close to one indicates that U_1 is close to U_{best}.

greedy h_{add} vs. random h_{add}

Table 6.4 compares greedy h_{add} and random h_{add}. The **random** h_{add} column shows the average value of suboptimal solution costs obtained by 10 different h_{add} heuristics, each based on choosing the distinguished vertices randomly. For most instances, values of the $\delta\%$ column are greater than 10%. Note that for instance prob100 the value of $\delta\%$ is greater than 25%.

The results indicate that for instances of different sizes and structures, greedy h_{add} outperforms random h_{add} in terms of the quality of suboptimal solution found within a 30-minute time limit.

Problem	n	greedy h_{add}	random h_{add}	$\delta\%$	Ratio
ry48p.1	49	17,129	18,879	11%	1.08
ry48p.2	49	17,721	19,846	13%	1.06
ry48p.3	49	21,793	23,811	10%	1.10
ry48p.4	49	33,307	36,654	11%	1.06
ft53.1	54	8,472	9,538	14%	1.12
ft53.2	54	10,561	12,017	17%	1.27
ft53.3	54	12,376	14,242	17%	1.13
ft53.4	54	16,246	17,295	7%	1.13
ft70.1	71	42,512	44,676	6%	1.08
ft70.2	71	44,163	46,621	6%	1.09
ft70.3	71	46,894	49,492	6%	1.10
ft70.4	71	57,976	60,523	5%	1.08
kro124p.1	101	48,065	50,663	6%	1.20
kro124p.2	101	50,129	54,728	11%	1.20
kro124p.3	101	63,808	70,149	11%	1.07
kro124p.4	101	90,022	97,202	9%	1.18
prob100	100	1,766	2,163	29%	1.28
prob42	42	270	303	14%	1.11
rbg378a	380	3,998	4,185	7%	1.41
rbg358a	360	3,954	4,299	13%	1.52

Table 6.4: greedy h_{add} vs. random h_{add}

Problem	n	greedy h_{max}	random h_{max}	$\delta\%$	Ratio
ry48p.1	49	18,069	19,813	11%	1.14
ry48p.2	49	18,911	20,352	9%	1.13
ry48p.3	49	22,623	23,403	4%	1.14
ry48p.4	49	34,848	36,272	5%	1.11
ft53.1	54	8,916	9,759	11%	1.18
ft53.2	54	9,527	11,843	28%	1.14
ft53.3	54	13,986	14,253	2%	1.28
ft53.4	54	16,594	17,259	5%	1.15
ft70.1	71	44,349	45,487	3%	1.13
ft70.2	71	45,260	47,433	5%	1.12
ft70.3	71	49,260	50,591	3%	1.16
ft70.4	71	60,695	61,767	2%	1.13
kro124p.1	101	49,204	50,559	3%	1.22
kro124p.2	101	53,406	57,349	9%	1.28
kro124p.3	101	66,631	70,344	6%	1.11
kro124p.4	101	97,155	98,560	5%	1.25
prob100	100	1,900	2,284	28%	1.37
prob42	42	271	306	14%	1.12
rbg378a	380	4,083	4,299	8%	1.44
rbg358a	360	4,125	4,456	13%	1.59

Table 6.5: greedy h_{max} vs. random h_{max}

greedy h_{max} vs. random h_{max}

Table 6.5 compares greedy h_{max} and random h_{max}. The **random** h_{max} column shows the average value of suboptimal solution costs obtained by 10 different h_{max} heuristics, each based on choosing

the distinguished vertices randomly.

For most instances, the values on the $\delta\%$ column are greater than or close to 5%. Note that for instances ft53.2 and prob100 the ratios are greater than 25%.

The results indicate that for instances of different sizes and structures, greedy h_{max} outperforms random h_{max} in terms of the quality of suboptimal solution found within the time limit.

greedy h_{add} vs. greedyP h_{add}

Problem	n	greedy h_{add}	greedyP h_{add}	$\delta\%$	Ratio
ry48p.1	49	17,129	18,013	6%	1.08
ry48p.2	49	17,721	19,794	12%	1.06
ry48p.3	49	21,793	23,171	7%	1.10
ry48p.4	49	33,307	36,669	11%	1.06
ft53.1	54	8,472	9,487	13%	1.12
ft53.2	54	10,561	11,693	14%	1.27
ft53.3	54	12,376	14,533	20%	1.13
ft53.4	54	16,246	16,694	3%	1.13
ft70.1	71	42,512	43,333	2%	1.08
ft70.2	71	44,163	45,937	4%	1.09
ft70.3	71	46,894	50,089	8%	1.10
ft70.4	71	57,976	59,767	3%	1.08
kro124p.1	101	48,065	49,653	4%	1.20
kro124p.2	101	50,129	53,921	9%	1.20
kro124p.3	101	63,808	64,176	1%	1.07
kro124p.4	101	90,022	93,290	4%	1.18
prob100	100	1,766	2,001	17%	1.28
prob42	42	270	281	5%	1.11
rbg378a	380	3,998	4,080	3%	1.41
rbg358a	360	3,954	4,156	8%	1.52

Table 6.6: greedy h_{add} vs. greedyP h_{add}

Table 6.6 compares greedy h_{add} and greedyP h_{add}. The results show that greedy h_{add} produces better solutions than greedyP h_{add} in the given time limit. The greatest value of $\delta\%$ is 20% for the instance ft53.3, which indicates that greedy h_{add} substantially outperforms greedyP h_{add} for this type of instance. The advantage of greedy h_{add} on ft70.x, kro124p.x and rbgxxxa are modest. One reason is due to the increasing number of constraints which strengthen the importance of constraints and weaken the importance of edge costs.

greedy h_{max} vs. greedyP h_{max}

Table 6.7 compares greedy h_{max} and greedyP h_{max}. The greatest value of $\delta\%$ is 27% for the instance ft53.2, while for instances ft70.4 and kro124p.2 $\delta\%$ is less than 0. The advantage of using greedy h_{max} is not consistent for different instances.

Problem	n	greedy h_{max}	greedyP h_{max}	$\delta(\%)$	Ratio
ry48p.1	49	18,069	18,947	6%	1.14
ry48p.2	49	18,911	18,922	+0%	1.13
ry48p.3	49	22,623	23,998	7%	1.14
ry48p.4	49	34,848	35,174	1%	1.11
ft53.1	54	8,916	9,694	10%	1.18
ft53.2	54	9,527	11,745	27%	1.14
ft53.3	54	13,986	14,386	4%	1.28
ft53.4	54	16,594	17,050	3%	1.15
ft70.1	71	44,349	44,647	1%	1.13
ft70.2	71	45,260	48,452	8%	1.12
ft70.3	71	49,260	51,522	5%	1.16
ft70.4	71	60,695	60,673	-0%	1.13
kro124p.1	101	49,204	50,755	4%	1.22
kro124p.2	101	53,406	53,187	-1%	1.28
kro124p.3	101	66,631	72,241	9%	1.11
kro124p.4	101	97,155	97,155	3%	1.25
prob100	100	1,900	2,204	22%	1.37
prob42	42	271	283	5%	1.12
rbg378a	380	4,083	4,155	3%	1.44
rbg358a	360	4,125	4,258	5%	1.59

Table 6.7: greedy h_{max} vs. greedyP h_{max}

Problem	n	greedy h_{add}	$h_{neighbor}$	$\delta\%$	Ratio
ry48p.1	49	17,129	18,306	7%	1.08
ry48p.2	49	17,721	19,258	9%	1.06
ry48p.3	49	21,793	23,006	6%	1.10
ry48p.4	49	33,307	36,940	12%	1.06
ft53.1	54	8,472	9,010	7%	1.12
ft53.2	54	10,561	10,695	2%	1.27
ft53.3	54	12,376	13,941	14%	1.13
ft53.4	54	16,246	16,513	2%	1.13
ft70.1	71	42,512	42,497	0%	1.08
ft70.2	71	44,163	45,901	4%	1.09
ft70.3	71	46,894	50,050	7%	1.10
ft70.4	71	57,976	60,274	4%	1.08
kro124p.1	101	48,065	49,161	3%	1.20
kro124p.2	101	50,129	55,438	13%	1.20
kro124p.3	101	63,808	65,379	3%	1.07
kro124p.4	101	90,022	88,228	-2%	1.18
prob100	100	1,766	1,929	12%	1.28
prob42	42	270	283	5%	1.11
rbg378a	380	3,998	4,122	4%	1.41
rbg358a	360	3,954	3,737	-8%	1.52

Table 6.8: greedy h_{add} vs. $h_{neighbor}$

greedy h_{add} vs. $h_{neighbor}$

Table 6.8 compares greedy h_{add} and $h_{neighbor}$. In terms of the value of $\delta\%$, greedy h_{add} retains its advantage for instances with a low percentage of constraints (i.e., instances with $P\%$ less than 2%

as shown in Table 6.2). One exception is that for ft70.1, the suboptimal solution values found by $h_{neighbor}$ and greedy h_{add} are very close. Note that $h_{neighbor}$ outperforms greedy h_{add} for instances kro124p.4 and rbg358a (the ratios of these instances are less than zero). We notice that in Table 6.2 $P\% = 47.5\%$ for kro124p.4 and $P\% = 88.4\%$ for rbg358a.

The results indicate that the advantage of greedy h_{add} increases as the percentage of constraints decreases. We observe that there exist certain situations (e.g. high percentage of constraints) which make our greedy method not yield the best heuristics.

greedy h_{max} vs. $h_{neighbor}$

Problem	n	greedy h_{max}	$h_{neighbor}$	$\delta\%$	Ratio
ry48p.1	49	18,069	18,306	2%	1.14
ry48p.2	49	18,911	19,258	2%	1.13
ry48p.3	49	22,623	23,006	2%	1.14
ry48p.4	49	34,848	36,940	7%	1.11
ft53.1	54	8,916	9,010	1%	1.18
ft53.2	54	9,527	10,695	14%	1.14
ft53.3	54	13,986	13,941	-0%	1.28
ft53.4	54	16,594	16,513	-0%	1.15
ft70.1	71	44,349	42,497	-5%	1.13
ft70.2	71	45,260	45,901	2%	1.12
ft70.3	71	49,260	50,050	2%	1.16
ft70.4	71	60,695	60,274	-1%	1.13
kro124p.1	101	49,204	49,161	-0%	1.22
kro124p.2	101	53,406	55,438	5%	1.28
kro124p.3	101	66,631	65,379	-2%	1.11
kro124p.4	101	97,155	88,228	-11%	1.25
prob100	100	1,900	1,929	2%	1.37
prob42	42	271	283	5%	1.12
rbg378a	380	4,083	4,122	1%	1.44
rbg358a	360	4,125	3,737	-15%	1.59

Table 6.9: greedy h_{max} vs. $h_{neighbor}$

Table 6.9 compares greedy h_{max} and $h_{neighbourhood}$. We observe that for instances ft53.3, ft53.4 and kro124p.1, the values of $\delta\%$ are zero indicating that the solution costs obtained by greedy h_{max} and $h_{neighbor}$ are very close. For instances ft70.1, ft70.4, kro124p.3, kro124p.4 and rbg358a, $h_{neighbor}$ beats greedy h_{max} as the values of $\delta\%$ are less than zero. But for other problems such as instances ft53.1, ft53.2 and prob100, greedy h_{max} outperforms $h_{neighbor}$.

The results indicate that comparing to the performance of $h_{neighbor}$, the advantage of greedy h_{max} is not consistent.

greedy h_{add} vs. h_{max}

Table 6.10 compares greedy h_{add} and greedy h_{max}. In terms of the value of $\delta\%$, greedy h_{add} outperforms greedy h_{max} for all instances except for instance ft53.2. We should mention that the

Problem	n	greedy h_{add}	greedy h_{max}	$\delta\%$	Ratio
ry48p.1	49	17,129	18,069	6%	1.08
ry48p.2	49	17,721	18,911	7%	1.06
ry48p.3	49	21,793	22,623	4%	1.10
ry48p.4	49	33,307	34,848	5%	1.06
ft53.1	54	8,472	8,916	6%	1.12
ft53.2	54	10,561	9,527	-12%	1.27
ft53.3	54	12,376	13,986	15%	1.13
ft53.4	54	16,246	16,594	2%	1.13
ft70.1	71	42,512	44,349	5%	1.08
ft70.2	71	44,163	45,260	3%	1.09
ft70.3	71	46,894	49,260	6%	1.10
ft70.4	71	57,976	60,695	5%	1.08
kro124p.1	101	48,065	49,204	3%	1.20
kro124p.2	101	50,129	53,406	8%	1.20
kro124p.3	101	63,808	66,631	6%	1.07
kro124p.4	101	90,022	97,155	9%	1.18
prob100	100	1,766	1,900	10%	1.28
prob42	42	270	271	+0%	1.11
rbg378a	380	3,998	4,083	3%	1.41
rbg358a	360	3,954	4,125	7%	1.52

Table 6.10: greedy h_{add} vs. greedy h_{max}

value of $\delta\%$ is -12% for instance ft53.2.

The results indicate that greedy h_{add} is more competitive than greedy h_{max} for most instances of different sizes and structures.

Anytime Behaviours

We compare the anytime behaviours of different heuristics. The following figures demonstrate the anytime behaviours of $h_{neighbour}$, greedy h_{add}/h_{max}, greedyP h_{add}/h_{max} and random h_{add}/h_{max} on large TSPLIB instances. The x axis depicts the time t and the y axis is ratio of the distance from $U(t)$ (the suboptimal solution found at time t) to the best-known solution (or upper bound) U_{best} listed on TSPLIB. The ratio is computed as $\frac{U(t)-U_{best}}{U_{best}} \times 100\%$. The smaller value of the ratio indicates that $U(t)$ is closer to the best-known solution (or upper bound).

We explain the legend of each figure as follows.

- NN: the anytime behaviour of $h_{neighbor}$.
- greedy add: the anytime behaviour of greedy h_{add}.
- greedyP add: the anytime behaviour of greedyP h_{add}.
- random add X: the anytime behaviour of Xth random h_{add}.
- greedy max: the anytime behaviour of greedy h_{max}.

- greedyP max: the anytime behaviour of greedyP h_{max}.
- random max X: the anytime behaviour of Xth random h_{max}.

We select instance prob100 as a sample instance to show the anytime behaviours of different heuristics.

In Figure 6.28, the upper part compares three heuristics $h_{neighbor}$, greedy h_{add}, greedyP h_{add}. The bottom part compares greedy h_{add} with 10 random h_{add}. Greedy h_{add} outperforms other heuristics within 30 seconds and retains its advantage until the end of time limit.

In Figure 6.29, the upper part compares three heuristics $h_{neighbor}$, greedy h_{max}, greedyP h_{max}. The bottom part compares greedy h_{max} with 10 random h_{max}. Greedy h_{max} also outperforms other heuristics within 30 seconds and retains its advantage until the end of time limit (1800 seconds).

Our observations on the anytime behavious of all selected instances are summarizes as follows.

- Greedy h_{add} beats $h_{neighbor}$ for all instances except for instances ft70.1, kro124p.1 and rbg358a. The benefits of using greedy h_{add} begin to show after about 10 seconds for these instances except for instance prob42 and keeps the advantage to the end of the time limit.
- Within the time limit greedy h_{add} beats greedyP h_{add} for all 20 instances.
- Greedy h_{add} beats all random h_{add} on all instances except for instances ft70.3 and prob42. Greedy h_{add} shows its advantage after 10 seconds and keeps the advantage to the end of the time limit. For some instances (e.g., ry48p.2, kro124p.4, prob100), the advantage of using greedy h_{add} is even more evident. For instance ft70.3, greedy h_{add} outperforms nine random h_{add} but there exists one random h_{add} on which DFBB obtains a better suboptimal solution than that obtained by using greedy h_{add}. We note that for instance prob42, greedy h_{add} does not beat all random h_{add} after 10 seconds. Instead, greedy h_{add} starts to show its advantage after more than 400 seconds and later it keeps the advantage until the end of the time limit.
- Within the time limit greedy h_{max} beats $h_{neighbor}$ for instances ry48p.1, ry48p.3, ry48p.4, ft53.1, ft53.2, ft53.3, ft70.3, kro124p.2, prob100, prob42, and rbg378a. But for other instances, $h_{neighbor}$ is comparable to greedy h_{max} or even outperforms greedy h_{max}.
- After 200 seconds greedy h_{max} beats greedyP h_{max} and keep its advantage until the end of the time limit for all instances except for instance ft70.4. For ft70.4, greedyP h_{max} outperforms greedy h_{max} after more than 1000 seconds. Then the suboptimal solution vlaues obtained by greedy h_{max} and greedyP h_{max} are very close until the end of the time limit.
- Greedy h_{max} outperforms all 10 random h_{max} heuristics for instances ry48p.2, ry48p.3, ry48p.4, ft53.1, ft53.2, ft53.4, ft70.2, kro124p4. Note that for instance ft53.2, the advantage of greedy h_{max} is most evident. For instances prob100 (the bottom part of Figure 6.29) and ft70.1 , the suboptimal solution costs obtained by using greedy h_{max} are very close to the

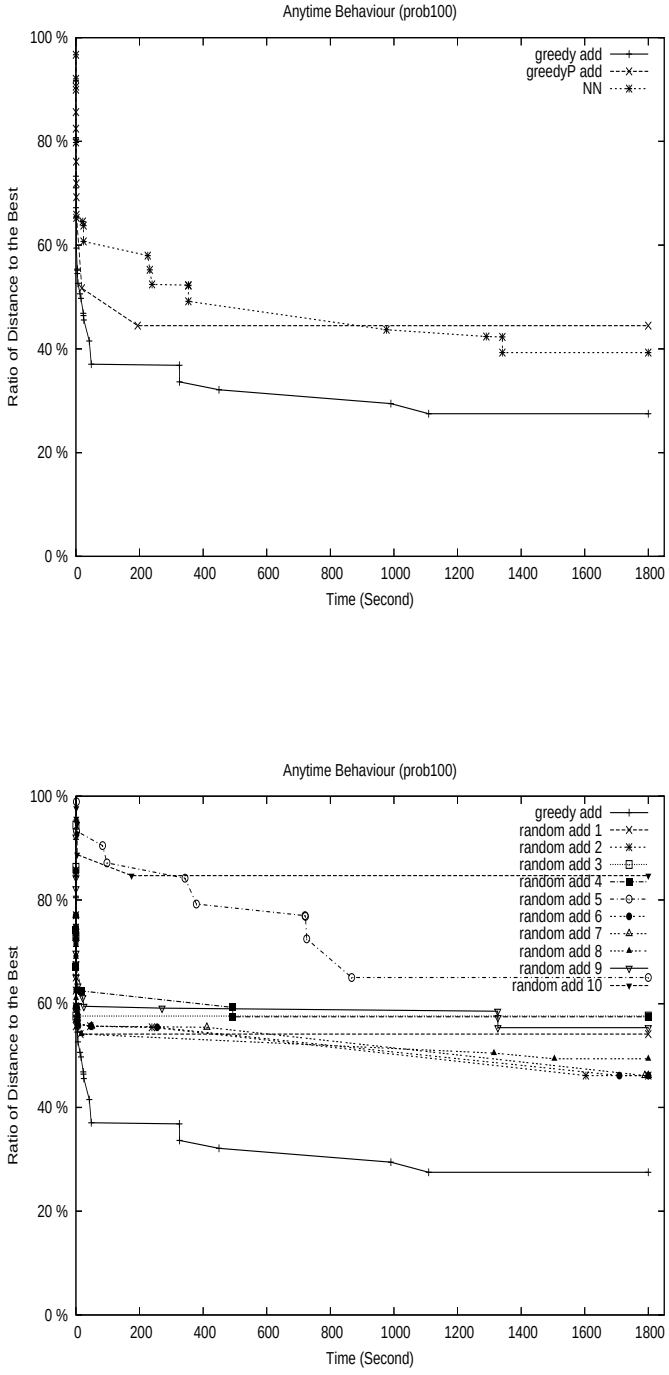


Figure 6.28: The anytime behaviours for prob100, including the anytime behaviours of $h_{neighbor}$, greedy h_{add}, greedyP h_{add} and 10 random h_{add}.

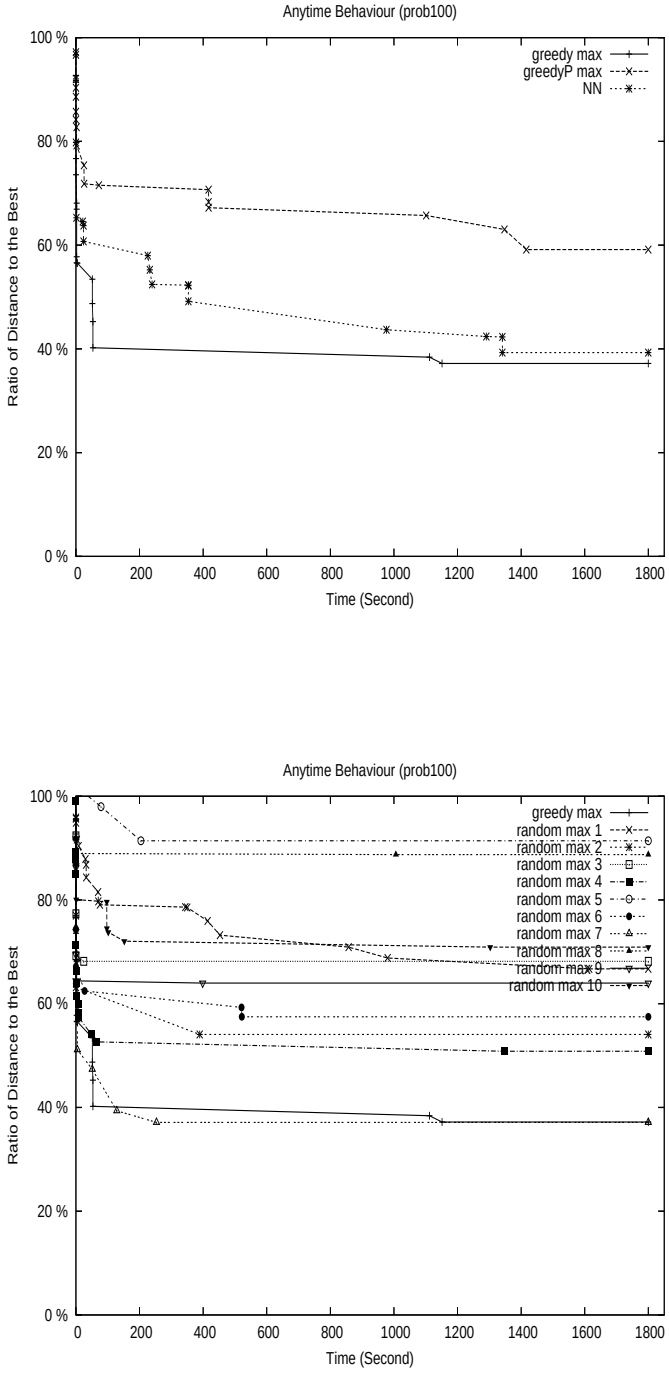


Figure 6.29: The anytime behaviours for prob100, including the anytime behaviours of $h_{neighbor}$, greedy h_{max}, greedyP h_{max} and 10 random h_{max}.

best suboptimal solution costs obtained by using random h_{max}. However, for other instances, using greedy h_{max} did not obtain the best suboptimal solution costs obtained by using random h_{max}.

Evaluating Different Heuristics

Problem	$h_{neighbor}$	greedy h_{add}	greedyP h_{add}	random h_{add}	greedy h_{max}	greedyP h_{max}	random h_{max}
ry48p.1	4	1	2	5	3	6	7
ry48p.2	4	1	5	6	2	3	7
ry48p.3	3	1	4	5	2	7	6
ry48p.4	7	1	5	4	2	3	6
ft53.1	3	1	4	5	2	6	7
ft53.2	3	2	4	7	1	5	6
ft53.3	2	1	7	4	3	6	5
ft53.4	2	1	4	7	3	5	6
ft70.1	1	2	3	6	4	5	7
ft70.2	3	1	4	5	2	7	6
ft70.3	4	1	5	2	3	7	6
ft70.4	3	1	2	4	5	6	7
kro124p.1	2	1	4	6	3	7	5
kro124p.2	6	1	4	5	2	3	7
kro124p.3	3	1	2	5	4	7	6
kro124p.4	1	2	3	6	4	5	7
prob100	3	1	4	5	2	6	7
prob42	4	1	3	6	2	5	7
rbg378a	4	1	2	7	3	6	5
rbg358a	1	2	4	7	3	6	5
avg	3.15	1.20	3.75	5.35	2.75	5.55	6.25

Table 6.11: The rank of the heuristics for each SOP instance.

Table 6.11 ranks different heuristics for each SOP instance. The **Name** column gives the name of the instance. The heading of each remaining column provides the name of each heuristic. For each instance, these columns give the rank of each heuristic based on the suboptimal solution found after 30 minutes. A rank of 1 means that heuristic found a better solution than all the other heuristics. The last row presents the average rank for each heuristic on these TSPLIB instances. Note that the best average rank is 1.20 for greedy h_{add}, which indicates the benefits of using greedy h_{add} on these TSPLIB instances.

Summary of the Results on Selected TSPLIB instances

In conclusion, Tables 6.4–6.11 report the comparisons between greedy heuristics and some simple alternative heuristics. According to the observations of these results, we answer the first four questions mentioned at the beginning of this section.

- Both greedy h_{add} and h_{max} beat random h_{add} and h_{max}, respectively (see Tables 6.4 and

6.5).

- The algorithms for creating greedy h_{add} and h_{max} that focus on edge costs outperform those that focus on constraints (see Tables 6.6 and 6.7).
- Compared to neighbourhood heuristics, the benefits of using greedy h_{add} are evident, while the benefits of using greedy h_{max} are not prominent (See Tables 6.8 and 6.9).
- The results empirically show the benefits of using greedy h_{add}. Compared to greedy h_{max}, greedy h_{add} performs more consistently well for instances of different sizes and structures. That is, within the same time limit DFBB using greedy h_{add} can find better solutions.

In the next section we report experimental results on the weighted Pancake puzzle to test whether the ideas learned for creating good additive abstractions for SOP problems will transfer to other problems with a different structure.

6.6.2 Experiments with Weighted Pancake Problems

In this section the key question is as follows.

- How well do the ideas learned for creating good additive abstractions for the SOP transfer to other problems with non-unit edge costs?

We first introduce the weighted Pancake problem as another problem with non-unit edge costs. Recall that a state of the n-Pancake problem is a permutation of n tiles (1, ..., n) and has n-1 successors, with the $(l-1)^{th}$ successor formed by reversing the order of the first l positions of the permutation ($2 \leq l \leq n$). For the n-weighted pancake problem, we randomly create an $n \times n$ matrix M. Given a state s, if tile i is in position 1, and tile j is in position l, then the cost of applying the operator that reverses the first l positions of state s is $M_{i,j}$. That is, the cost of an operation (i.e., a swap) is the cost indicated in M by the tile i that moves out of position 1 and the tile j that moves into position 1.

Suppose that each entry in the cost matrix ranges from 1 to m ($m \geq 1$). Then the average cost of each edge is $\frac{m+1}{2}$. It is clear that the average solution cost increases as the value of m increases, which leads to the result that the search algorithm needs more time to find the optimal solution if IDA* is applied (we will explain the reason why we apply IDA* instead of DFBB shortly). In our experiments we set $m = 10$ and thus the average edge cost is $\frac{10+1}{2} = 5.5$.

Given a set of weighted Pancake instances I, we define some values as follows.

- $\sum_{i \in I} Cost(i)$. $Cost(i)$ is the optimal solution cost for instance i;
- $\sum_{i \in I} maxDepth(i)$. $maxDepth(i)$ is the maximum depth of the optimal solution of instance i;

- $\sum_{i \in I} minDepth(i)$: $minDepth(i)$ is the minimum depth for the optimal solution of instance i;

- $\sum_{i \in I} unitP(i)$. $unitP(i)$ is the optimal solution over the corresponding unit-cost instances of i;

- $\sum_{I} unitP\_Cost(i)$. $unitP\_Cost(i)$ is the solution cost of i's corresponding unit-cost instance. Each solution path is the path first found by IDA^* for the corresponding unit-cost instance. We can use the value of $unitP\_Cost(i)$ as the upper bound for DFBB.

Our experiments are executed on I which includes 100 randomly generated instances of the 12-Pancake problem. For this set we observed that

- $\sum_{i \in I} Cost(i) = 2791$;

- $\sum_{i \in I} maxDepth(i) = 1655$;

- $\sum_{i \in I} minDepth(i) = 1561$;

- $\sum_{i \in I} unitP(i) = 1060$;

- $\sum_{I} unitP\_Cost(i) = 5013$.

It is evident that the average edge cost on the solution path first found by IDA^* for the corresponding unit-cost instance $\frac{\sum_I unitP\_Cost(i)}{\sum_I unitP(i)} = \frac{5013}{1060} \approx 5$ which is the value being close to the average edge cost. However, the average edge cost of the optimal solution path ranges from $\frac{2791}{1561}$ to $\frac{2791}{1655}$, i.e., from 1.68 to 1.79. Note that the average edge cost on these paths is significantly less than 5.5 (i.e., the average edge cost). It is because the edges on the optimal solution paths are only a part of the overall edges in the state space as edges with large costs are always replaced by cheap edges to construct the optimal solution path.

We apply IDA* instead of DFBB to solve weighted Pancake problems because the structure of the state space for the weighted Pancake problem is different from that for the SOP. Recall that in the SOP search space, each step must add a new vertex to the current path and therefore the total steps of a solution equals the total number of the vertices except for the start vertex. In contrast, in the search state space for the weighted Pancake puzzle the search depth of the optimal solution can be much larger than the total number of the tiles, which leads to the result that DFBB may spend a large amount of time without finding any suboptimal solution.

In Figures 6.30-6.32 we compare different search depths to show why we apply IDA* instead of DFBB to the weighted Pancake problems. In these figures, the dashed line is the line of $x = y$ and each diamond dot shows information about one of the 100 12-Pancake problems. All figures are based on results over the same set of 100 weighted Pancake problems. Some dots are overlapped due to the same coordinates so that there may appear to be fewer than 100 dots in the figures.

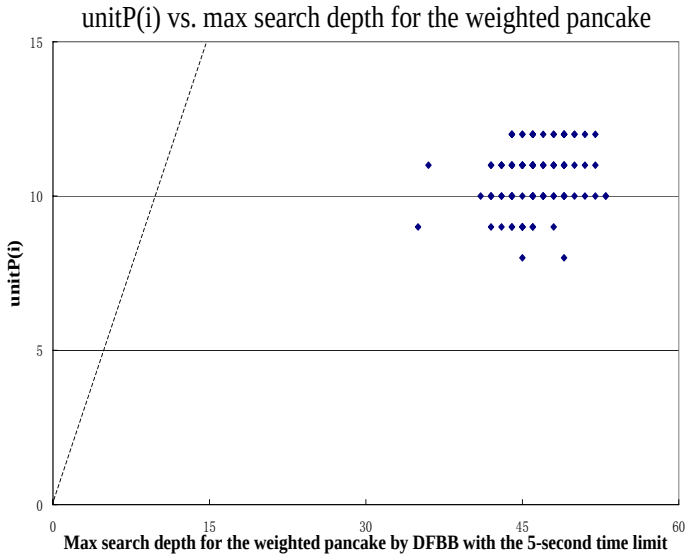


Figure 6.30: Comparing $unitP(i)$ and the maximum search depth of DFBB on the weighted Pancake. The time limit for the DFBB is five seconds for each instance.

Figure 6.30 compares $unitP(i)$ and the maximum search depth of DFBB on the weighted Pancake. The time limit for the DFBB is five seconds for each instance.

A set of random (6-6) additive abstraction-based heuristics is applied when searching with DFBB. There are 3,456 entry values in the PDBs to store (6-6) additive abstraction-based heuristics. The initial upper bound of each weighted Pancake problem is pre-defined by computing the cost of the first-found solution path for the corresponding unit-cost problem. The y axis is unitP(i). The x axis shows the maximum search depth of DFBB with the 5-second time limit in the state space of the weighted Pancake.

For example, there exists a point in Figure 6.30 with coordinates (50,10) which means that the solution depth is 10 for this 12-Pancake problem with unit cost, while the maximum search depth is 50 by DFBB within 5-second time limit for the same problem with non-uniform edge costs. As shown in Figure 6.30, the solution depth of the unit-cost Pancake is always less than 13, while for the same instance with non-uniform costs, the maximum depth of DFBB with the time limit ranges from 35 to 53 in the search space.

Figure 6.31 compares $unitP(i)$ and $maxDepth(i)$ (or $minDepth(i)$). The top part compares $unitP(i)$ and $maxDepth(i)$. The bottom part compares $unitP(i)$ and $minDepth(i)$. IDA^* with additive random (6-6) abstraction-based heuristics is applied to obtain $maxDepth(i)$ and $minDepth(i)$ for each instance i.

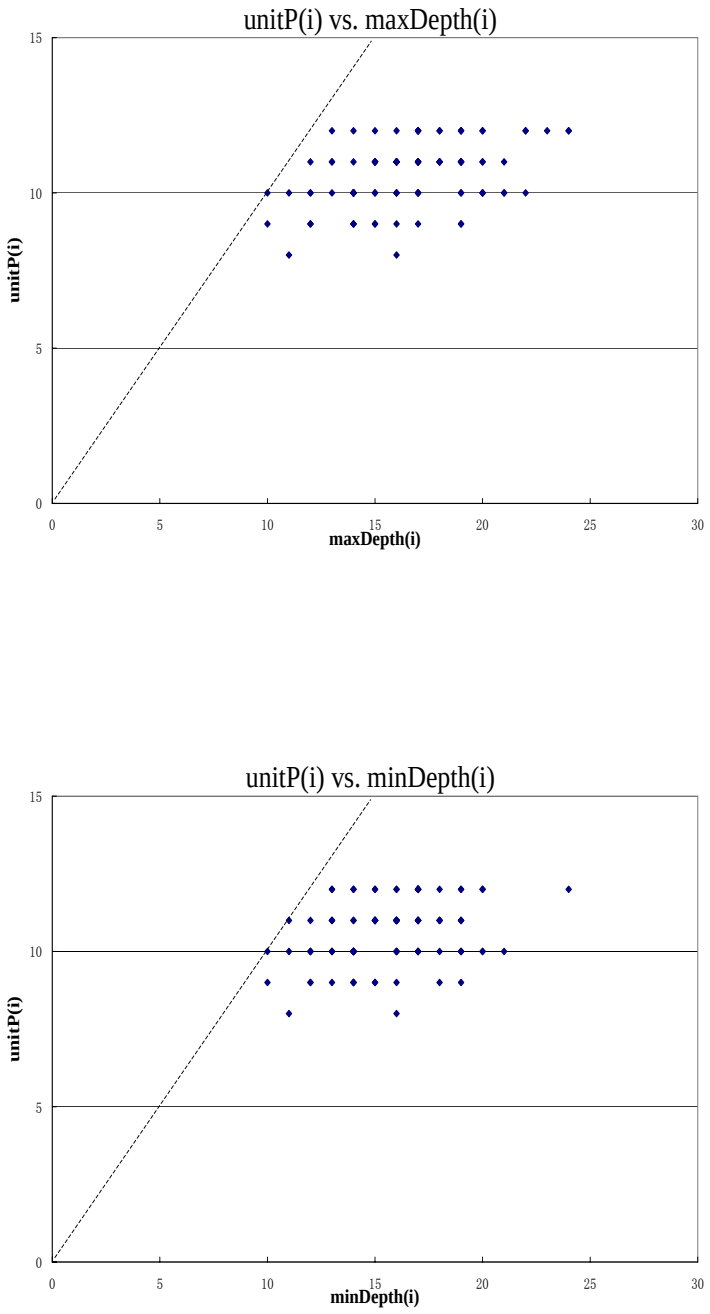


Figure 6.31: Comparing $unitP(i)$ and $maxDepth(i)$ (or $minDepth(i)$). Top: comparing $unitP(i)$ and $maxDepth(i)$. Bottom: comparing $unitP(i)$ and $minDepth(i)$.

The y axis is $unitP(i)$. In the top part of Figure 6.31 the x axis shows $maxDepth(i)$. In the bottom part of Figure 6.31 the x axis shows $minDepth(i)$.

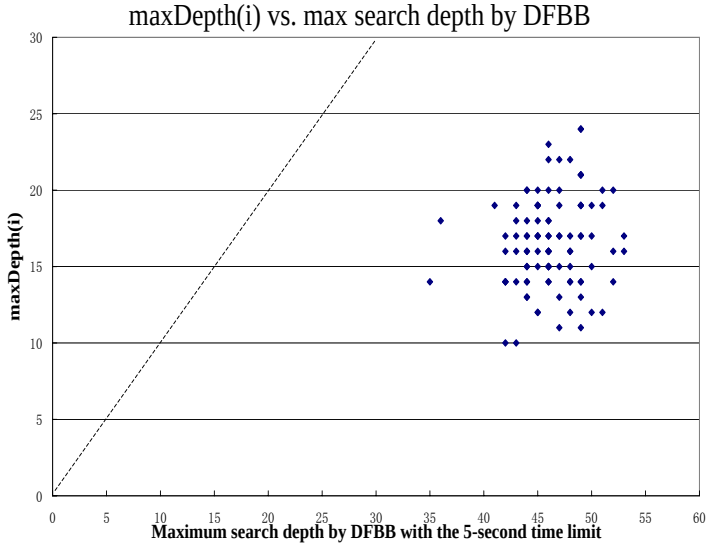
For example, there is a point in the top part of Figure 6.31 with coordinates (12,10) which means that the solution depth is 10 for this Pancake problem with the unit cost, while the maximum search depth is 12 to find the optimal solution to the same problem with non-uniform edge costs. As shown in Figure 6.31, the search depth of the optimal solution to the weighted Pancake ranges from 10 to 24, while the solution depth of the unit-cost Pancake ranges from 8 to 12.

Figure 6.32 compares the different search depths over 100 weighted 12-Pancake problems. For each problem, we obtain the maximum and minimum search depths for IDA^* to find an optimal solution and the maximum search depth for DFBB within a 5-second time limit. A set of random (6-6) additive abstraction-based heuristics is applied for both search algorithms (IDA^* and DFBB). The initial upper bound used by DFBB for each weighted Pancake problem is pre-defined by computing the cost of the first-found solution path for the corresponding unit-cost problem. The x axis shows the maximum search depth using DFBB with the 5-second time limit in the state space of the weighted Pancake. In the top part of Figure 6.32 the y axis is $maxDepth(i)$; in the bottom part the y axis is $minDepth(i)$.

For example, there is a point in the top part of Figure 6.32 with coordinates (50, 15) which means that the maximum solution depth is 15, while the maximum search depth by DFBB with 5-second time limit is 50 which is more than 3 times deeper than the maximum solution depth. As shown in Figure 6.32 DFBB with the time limit always goes deeper than the depth needed to obtain an optimal solution. In the following figure, we compare the initial upper bounds to the actual solution cost to indicate an important reason why DFBB with the time limit always goes deeper than the depth needed to obtain an optimal solution.

Figure 6.33 compares the optimal solution costs ($Cost(i)$) to $unitP\_Cost(i)$ which is used by DFBB as the initial upper bounds. The dashed line in this figure is the line of $y = \frac{1}{2}x$ and the diamond dots show the distribution of the optimal solution cost and the initial upper bound for each problem. The x axis presents the initial upper bounds used by DFBB and the y axis shows the optimal solution costs. For example, there is a point in Figure 6.33 with coordinates (66, 30) which means that the initial upper bound used by DFBB is 66, while the actual solution cost is 30 which is less than the half value of the corresponding initial upper bound for DFBB.

As shown in Figure 6.33 the solution costs of the weighted 12-Pancake problems range from 20 to 35, while the corresponding initial upper bounds range from 24 to 75. Within 100 random instances, about 60% of instances have the initial upper bounds which are equal to or close to twice of the corresponding solution costs. DFBB can go deeper than the depth needed to obtain an optimal solution if the initial upper bound is much larger than the optimal solution cost as it is in these problems. This is an important reason to explain that in our experiments for most instances DFBB with the time limit goes deeper than the depth needed to obtain an optimal solution. Therefore



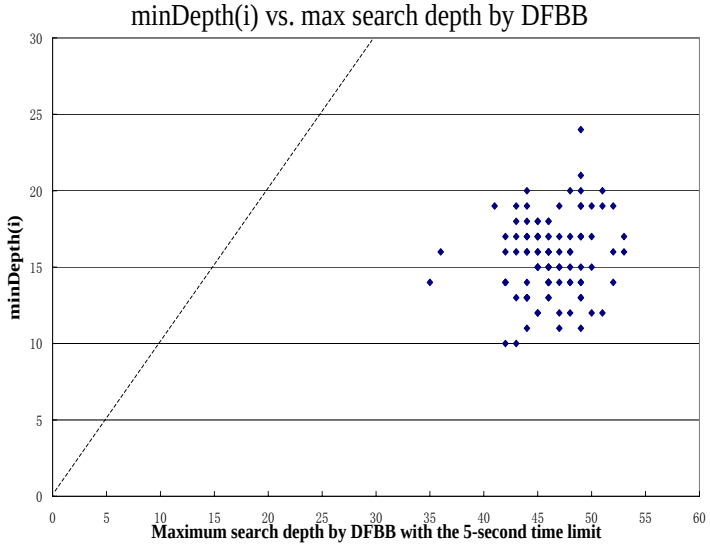


Figure 6.32: Comparing $maxDepth(i)$ (or $minDepth(i)$) and the maximum depth of DFBB with a 5-second time limit over 100 weighted 12-Pancake problems. Top: comparing $maxDepth(i)$ and the maximum search depth of DFBB. Bottom: comparing $minDepth(i)$ and the maximum search depth of DFBB.

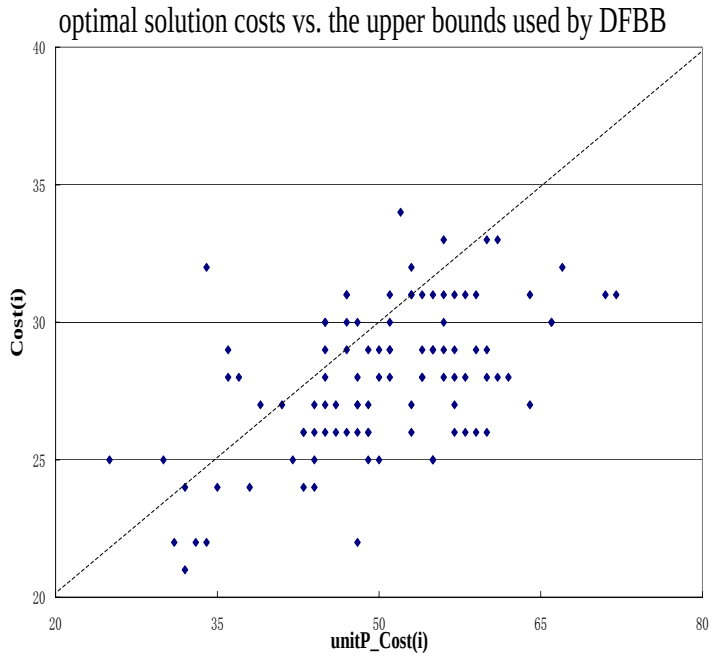


Figure 6.33: Comparing the optimal solution costs to the initial upper bounds used by DFBB. The scale on the y-axis is half the scale on the x-axis, so the diagonal line represents $y = \frac{1}{2}x$.

DFBB is not a good choice for the weighted Pancake puzzle. In the following experiments IDA* is used instead.

n	Abs	greedy h_{add}	random h_{add}	Ratio	greedy outperforms random
12	(6*2)	345,148,053	1,037,619,898	3.01	81%

Table 6.12: greedy h_{add} vs. random h_{add}

Table 6.12 compares the performance of IDA^* using greedy h_{add} and random h_{add}. The **n** column gives the size of the pancake instance. The **Abs** column shows the set of abstractions used to generate heuristics. Here (6*2) means that there are 2 abstractions with 6 distinguished vertices in each abstraction. The **greedy** h_{add} column is the average number of nodes generated in solving 100 randomly generated start states by using h_{add} generated by HYBRID ADD. The **random** h_{add} column is the average number of nodes generated in solving 100 randomly generated start states by using 100 sets of random h_{add}. The **Ratio** column compares the nodes generated by using two different heuristics. It is measured as $\frac{N_2}{N_1}$ where N_1 is number of nodes presented in the third column,

N_2 is the number of nodes presented in the fourth column. The **greedy outperforms random** column shows the percentage of the 100 randomly generated heuristics that are outperformed by greedy h_{add}.

The results empirically show the benefits when using greedy h_{add} (i.e., h_{add} generated by HYBRID ADD), because the use of greedy h_{add} results in a reduction in nodes generated comparing to using random h_{add}.

n	Abs	greedy h_{max}	random h_{max}	Ratio	greedy outperforms random
12	(6*2)	69,199,466	205,552,030	2.97	77%

Table 6.13: greedy h_{max} vs. random h_{max}

Table 6.13 is the same as Table 6.12 but for h_{max}. As shown in Table 6.13, greedy h_{max} generated by HYBRID MAX outperforms random h_{max} in terms of the number of nodes generated. Greedy h_{max} beats over 70% of the random h_{max} heuristics.

Comparing h_{add} and h_{max} shown in Tables 6.12 and 6.13, we note that IDA^* using greedy h_{max} generates 4 times fewer nodes than IDA^* using greedy h_{add}. In addition random h_{max} also outperforms greedy h_{add}.

These experimental results show that our greedy methods can be applied to some non-unit-cost problems with a different search structure. The resulting heuristics based on greedy abstractions outperforms the random heuristics in terms of the nodes generated. The comparisons of h_{add} and h_{max} show that h_{max} is superior to h_{add} when the size of the weighted Pancake problem is relatively small and there is sufficient memory to store heuristics based on abstractions of relatively large size.

6.7 Chapter Summary

We formalized a novel way of generating additive and non-additive heuristics for the SOP state space. We studied methods of defining additive and non-additive abstractions for the problems with non-unit edge costs.

We explored the key concepts to generate good abstractions for h_{add} and h_{max}, respectively. Some greedy methods of choosing good abstractions were designed and discussed. The trade-off of greedy methods used in the method was discussed.

We ran experiments over selected TSPLIB instances using both h_{add} and h_{max} generated from random abstractions and greedy abstractions. The results indicate the benefits using heuristics based on greedy abstractions for these instances. h_{add} and h_{max} are compared and discussed.

We introduced a new version of the pancake problem called the weighted pancake puzzle. We experimented over random instances of the weighted pancake puzzle. The experimental results show that the ideas learned for creating good abstractions for non-unit-cost problems while studying SOP can be transferred to another problem with a different search structure.

For most TSPLIB instances, greedy h_{add} is superior to greedy h_{max} in terms of the suboptimal solution found within a given time limit. For the weighted pancake problem, the comparisons between h_{add} and h_{max} demonstrate that additive abstractions are not always superior to the standard, maximum-based method for combining multiple abstractions.

Chapter 7

Conclusions and Future Directions

In this thesis, we have presented a formal, general definition of additive abstractions that can be applied to any state space and proved that heuristics based on additive abstractions are consistent as well as admissible.

Our definition formalizes the intuitive idea that abstractions will be additive provided that the cost of each operator is divided among the abstract spaces, and we have presented two specific, practical methods for defining abstract costs, cost-splitting and location-based costs. These methods were applied to two standard state spaces of combinatorial puzzles that did not have additive abstractions according to previous definitions: TopSpin and the Pancake puzzle. Additive abstractions using location-based costs evidently reduce search time for the 17-Pancake puzzle. We also report negative results demonstrating that additive abstractions are not always superior to the standard, maximum-based method for combining multiple abstractions.

A distinctive feature of our definition is that each edge in an abstract space has two costs instead of just one. This feature has enabled us to develop a way of testing if the heuristic value returned by additive abstractions is provably too low (i.e.,infeasible). This test produced no speedup when applied to the Pancake puzzle, but roughly halved the search time for the sliding tile puzzle and in most of our experiments with TopSpin.

Using the new definition, we also explored the applications of additive abstraction-based heuristics in two state spaces with non-uniform edge costs: the Sequential Ordering Problem (SOP) and the weighted Pancake puzzle. We investigated the design of good abstractions for instances with special properties. Experiments showed that compared to some alternative heuristics, well chosen abstraction-based heuristics can enhance the quality of suboptimal solutions for large SOP instances and reduce search time for the weighted Pancake problems.

7.1 Future Directions

We have investigated the general definitions of additive abstractions and addressed several key issues for the design of good abstractions. There are numerous possibilities that can be investigated as

future work:

1. The new theory and methods reported in this thesis are very general—they apply to a wide variety of application domains. However most of the successful abstractions are not chosen automatically for all domains with different properties and structures. We should investigate specific guidelines for automatically applying these new ideas to an arbitrary domain.

2. Methods for defining costs. Our study shows that additive abstractions are not always the best abstraction method. In our experiments the solution cost calculated by an individual additive abstraction can sometimes be very low. In the extreme case, which actually arises in practice, all problems can have abstract solutions that cost 0. This imposes a challenge for defining the primary cost of an abstract state transition to improve the quality of additive abstractions in this situation.

3. Regarding the design of abstractions for the SOP, we define the concepts of the cheap edges and expensive edges. Further research is needed to offer guidelines for these rules considering different structure properties.

4. Greedy Abstractions. Some experiments show that greedy abstractions do not always have the best performance compared to those of random abstractions. We need to investigate the trade-off of different abstraction schemes and improve our greedy abstractions for different domains.

5. Up to now we can only detect the infeasibility of h_{add}. The study of the infeasibility of h_{max} will be the next step of our research. In addition it would be of interest to investigate how to best integrate structure properties into the presented scheme to identify infeasibility more efficiently, and to analyse what impact this would have on the quality of heuristics and the performance of heuristic search.

6. Generally, we can safely add one to an infeasible heuristic value for problems with unit edge cost. But this improvement seems weak when most of the edges cost more than one. It is necessary to explore the method to enhance the increment without losing the admissibility of the infeasible heuristic values. One way is to introduce the second best cost based on the abstraction for the improvement. But there is a space penalty because we need to store more primary costs in memory and it is not clear if it is the best way to use this extra memory.

7. The applications of abstraction-based heuristics are not restricted to the areas of combinatorial problems and the SOP. More powerful abstraction-based heuristics for other pathfinding problems should be addressed.

8. The extension to the heuristic search planner. Different heuristics target different bolts of the planning complexity. It is challenging and interesting to design greedy abstractions to

compose the individual strengths of numerous heuristics which will enable us to solve a larger range of planning tasks and solve each task more efficiently.

Bibliography

[1] Norbert Ascheuer. *Hamiltonian path problems in the on-line optimization and scheduling of flexible manufacturing systems.* PhD thesis, Technical University of Berlin, 1995.

[2] Norbert Ascheuer, Michael Jünger, and Gerhard Reinelt. A branch and cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17(1):61–84, 2000.

[3] Egon Balas and Paolo Toth. Branch and bound methods. In *The Traveling Salesman Problem: a guided tour of combinatorial optimization*, pages 361–401, 1985.

[4] Ranan B. Banerji. *Artificial Intelligence: A Theoretical Approach*. North Holland, 1980.

[5] Richard Bellman. Combinatorial processes and dynamic programming. In *Proceedings of Symposia in Applied Mathematics 10, American Mathematical Society*, pages 217–249, 1960.

[6] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9:61–63, 1962.

[7] Giorgio Carpaneto and Paolo Toth. Some new branching and bounding criteria for the asymmetric travelling salesman problem. *Management Science*, 26(7):736–743, 1980.

[8] Partha Pratim Chakrabarti, Sujoy Ghose, Anurag Acharya, and S. C. De Sarkar. Heuristic search in restricted memory (research note). *Artificial Intelligence*, 41:197–221, 1989.

[9] Ting Chen and Steven Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1–3):269–295, 1996.

[10] Nicos Christofides. The shortest Hamiltonian chain of a graph. *SIAM J.Appl. Math*, 19:689–696, 1970.

[11] Jill Cirasella, David S. Johnson, Lyle A. McGeoch, and Weixiong Zhang. The asymmetric traveling salesman problem: algorithms, instance generators, and tests. In *Proceedings of the third workshop on Algorithm Engineering and Experiments, Springer Lecture Notes in Computer Science 2153*, pages 32–59, 2001.

[12] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Additive-disjunctive heuristics for optimal planning. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 08)*, pages 44–51, 2008.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second Edition)*. The MIT Press, 2001.

[14] Joseph C. Culberson and Jonathan Schaeffer. Efficiently searching the 15-puzzle. Technical Report TR94-08, Department of Computing Science, University of Alberta, 1994.

[15] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, pages 402–416. Springer, 1996.

[16] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[17] Dennis de Champeaux and Lenie Sint. An improved bidirectional heuristic search algorithm. *Journal of the ACM*, 32(3):505–536, 1977.

[18] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[19] John F. Dillenburg and Peter C. Nelson. Perimeter search. *Artificial Intelligence*, 65:165–178, 1994.

[20] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. To max or not to max: Online learning for speeding up optimal planning. In *Proceedings of the Twenty-fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1071–1076, 2010.

[21] W. L. Eastman. *Linear Programming with pattern constraints*. PhD thesis, Harvard University, 1958.

[22] Stefan Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, pages 13–24, 2001.

[23] Stefan Edelkamp. Symbolic pattern databases in heurisitc search planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 274–283, Toulouse, France, 2002.

[24] Esra Erdem and Elisabeth Tillier. Genome rearrangement and planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1139–1144, Pittsburgh, Pennsylvania, 2005.

[25] Ariel Felner and Amir Adler. Solving the 24 puzzle with instance dependent pattern databases. *Proc. SARA-2005, Lecture Notes in Artificial Intelligence*, 3607:248–260, 2005.

[26] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[27] Ariel Felner, Uzi Zahavi, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, pages 1076–1081, Boston, Massachusetts, 2006.

[28] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert Holte. Dual lookups in pattern databases. In *Proceedings of IJCAI-05*, pages 103–108, 2005.

[29] Alan Frieze and Joseph Yukich. Probabilistic analysis of the TSP. In *The Traveling Salesman Problem and its variations*, pages 257–308, 2002.

[30] John Gaschnig. A problem similarity approach to devising heuristics: First results. In *Proceedings of IJCAI-79*, pages 301–307, 1979.

[31] Subrata Ghosh, Ambuj Mahanti, and Dana S. Nau. ITS: An efficient limited-memory heuristic tree search algorithm. In *Proceedings of the Thirteenth Conference on Artificial Intelligence (AAAI-94)*, pages 1353–1358, Seattle, Washington, 1994.

[32] Fred Glover and Lee Tangedahl. Dynamic strategies for branch and bound. *The International Journal of Management Science*, 4(5):571–576, 1976.

[33] B. L. Golden and W. R. Stewart. Empirical analysis of heuristics. In *The Traveling Salesman Problem: a guided tour of combinatorial optimization*, pages 207–249, 1985.

[34] Giovanni Guida and Marco Somalvico. A method for computing heuristics in problem solving. *Information Sciences*, 19:251–259, 1979.

[35] Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and its variations*. Kluwer Academic Publishers, Boston, United States, 2002.

[36] Gregory Gutin, Anders Yeo, and Alexey Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discrete Applied Mathematics*, 117:81–86, 2002.

[37] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4, pages 100–107, 1968.

[38] Patrik Haslum, Blai Bonet, and Héctor Geffner. New admissible heurisitics for domain-independent planning. In *Proceedings of The Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1163–1168, 2005.

[39] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of The Twenty-Second National Conference on Artificial Intelligence (AAAI-07)*, pages 1007–1012, 2007.

[40] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[41] Michael Held and Richard M. Karp. The Traveling Salesman Problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[42] Michael Held and Richard M. Karp. The Traveling Salesman Problem and minimum spanning trees: Part II. *Math. Prog.*, 1:6–25, 1971.

[43] Pavol Hell and Jaroslav Nesetril. Graphs and homomorphisms. *The Oxford Lecture Series in Mathematics and its Applications*, 28, July 2004.

[44] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*, pages 162–169, 2009.

[45] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 176–183, 2007.

[46] Malte Helmert and Gabriele Röger. Relative-order abstractions for the pancake problem. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010),*, pages 745–750, 2010.

[47] Istvan Hernádvölgyi and Robert C. Holte. Experiments with automatically created memory-based heuristics. *Proc. SARA-2000, Lecture Notes in Artificial Intelligence*, 1864:281–290, 2000.

[48] Istvan T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Proceedings of Operations Research 2003*, page 355C362, 2003.

[49] Istvan T. Hernádvölgyi. *Automatically Generated Lower Bounds for Search*. PhD thesis, University of Ottawa, 2004.

[50] Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, 170:1123–1136, November 2006.

[51] Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical search revisited. In *Proceedings of SARA 2005 - Symposium on Abstraction, Reformulation and Approximation, LNAI 3607*, pages 121–133, Springer, 2005.

[52] Robert C. Holte and Istvan T. Hernádvölgyi. Steps towards the automatic creation of search heuristics. Technical Report TR04-02, Computing Science Department, University of Alberta, Edmonton, Canada T6G 2E8, 2004.

[53] Robert C. Holte, Jack Newton, Ariel Felner, R. Meshulam, and David Furcy. Multiple pattern databases. In *Proceedings of ICAPS-04*, pages 122–131, 2004.

[54] Robert C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535, 1996.

[55] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of Vehicle Navigation and Information Systems Conference*, pages 291–296, 1994.

[56] D. S. Johnson and C. H. Papadimitriou. Performance guarantees for heuristics. In *The Traveling Salesman Problem: a guided tour of combinatorial optimization*, pages 145–180, 1985.

[57] David S. Johnson, Gregory Gutin, Lyle A. McGeoch, Anders Yeo, Weixiong Zhang, and Alexei Zverovich. Experimental analysis of heuristics for the ATSP. In *The Traveling Salesman Problem and its variations*, pages 445–488, 2002.

[58] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[59] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, New York, United States, 1972.

[60] Richard M. Karp and J. M. Steele. Probabilistic analysis of heuristics. In *The Traveling Salesman Problem*, pages 181–205. John Wiley and Sons, Chichester, 1985.

[61] Michael Katz and Carmel Domshlak. New islands of tractability of cost-optimal planning. *Journal of Artificial Intelligence Research*, 32:203–288, 2008.

[62] Michael Katz and Carmel Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *Proceedings of 18th International Conference on Automated Planning and Scheduling (ICAPS-08)*, page 174C181, 2008.

[63] Michael Katz and Carmel Domshlak. Structural patterns heuristics via fork decomposition. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-08)*, pages 182–189, 2008.

[64] D. Kibler. Natural generation of admissible heuristics. Technical Report TR-188, University of California at Irvine, July 1982.

[65] Richard E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 1034–1036, 1985.

[66] Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, Providence, Rhode Island, July 1997.

[67] Richard E. Korf. Divide-and-conquer bidirectional search: first results. In *Proceedings of IJCAI-99*, pages 1184–1189, 1999.

[68] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.

[69] Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1202–1207, 1996.

[70] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 910–916, Austin, Texas, 2000.

[71] Richard E. Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier search. *Journal of the ACM*, 52:715–748, 2005.

[72] Jean-Marc Labat and Jean-Charles Pomerol. Are branch and bound and A* algorithms identical? *Artificial Intelligence*, 9:131–143, 2003.

[73] Bradford Larsen, Ethan Burns, Wheeler Ruml, and Robert C. Holte. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of the Twenty-fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 114–120, 2010.

[74] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[75] Lionel Lobjois and Michel Lemaitre. Branch and bound algorithm selection by performance prediction. In *Proceedings of AAAI'98*, pages 353–358, 1998.

[76] G. Manzini. BIDA*: an improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[77] Radu Marinescu and Rina Dechter. Dynamic orderings for AND/OR branch-and-bound search in graphical models. In *Proceedings of European Conference on Artificial Intelligence (ECAI'06)*, pages 138–142, 2006.

[78] Jack Mostow and Armand E. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. *IJCAI*, pages 701–707, 1989.

[79] Y. Narahari. Optimal solution for TSP using branch and bound. *Available at http://lcm.csa.iisc.ernet.in/dsa/node187.html*, 2009.

[80] Dana S. Nau, Vipin Kumar, and Laveen Kanal. General branch and bound, and its relation to A* and AO*. *Artificial Intelligence*, 23:29–58, 1984.

[81] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Pretice Hall, Englewood Cliffs, NJ, 1982.

[82] Judea Pearl. *Heuristics*. Addison Wesley, Reading Mass, 1984.

[83] Ira Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[84] Armand E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[85] W. Pulleyblank and M. Timlin. Precedence constrained routing and helicopter scheduling: Heuristic design. *INTERFACES*, 22(3):100–111, 1992.

[86] Alexander Reinefeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:701–710, 1994.

[87] Gerhard Reinelt. TSPLIB-a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[88] Gerhard Reinelt. *The Traveling Salesman: computational solutions for TSP applications*. Springer-Verlag, Berlin Heidelberg, 1994.

[89] Gerhard Reinelt. TSPLIB, 2010. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[90] Stuart Russell. Efficient memory-bounded search methods. In *Proceedings of ECAI-92*, pages 1–5, 1992.

[91] Anup K. Sen and Amitava Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of IJCAI-89*, pages 297–302, 1989.

[92] Marco Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Science*, 34:48–59, 1984.

[93] Fan Yang. Exploring infeasibility for abstraction-based heuristics. In *AAAI Workshop on Search in Artificial Intelligence and Robotics (WS-08-10)*, pages 134–139, Chicago, USA, 2008.

[94] Fan Yang, Joseph C. Culberson, and Robert Holte. A general additive search abstraction. Technical Report TR07-06, Department of Computing Science, University of Alberta, April 2007.

[95] Fan Yang, Joseph C. Culberson, and Robert Holte. Using infeasibility to improve abstraction-based heuristics. In *SARA 2007 - Symposium on Abstraction, Reformulation and Approximation, LNAI 4612*, pages 413–414, Springer, 2007.

[96] Fan Yang, Joseph C. Culberson, and Robert Holte. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662, 2008.

[97] Weixiong Zhang. Depth-first branch-and-bound versus local search: A case study. In *Proceedings of AAAI-00*, pages 16–22, 2000.

[98] Weixiong Zhang and Richard E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.

[99] Rong Zhou and Eric A. Hansen. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1259–1268, 2003.

[100] Rong Zhou and Eric A. Hansen. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 427–434, 2003.

[101] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. In *Fourteenth International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 92–100, 2004.

[102] Sandra Zilles and Robert Holte. The computational complexity of avoiding spurious states in state space abstraction. *Artificial Intelligence*, 174:1072–1092, 2010.