University of Alberta

MULTIMAPPING ABSTRACTION AND STATE-SET SEARCH THEORY

by

Bo Pang

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

Master of Science

Department of Computing Science

Fall 2012
Edmonton, Alberta

Abstract

This thesis consists of two parts. First, we invented an abstraction framework called multimapping which allows multiple admissible heuristic values to be extracted from one abstract space. The key idea of this technique is to design a multimapping function which maps one state in the original space to multiple states in the abstract space. The fundamental differences between this new technique and other existing ones is that this technique is completely domain independent and can be implemented without increasing the size of the abstract space. Based on this multimapping framework, we have implemented multimapping domain abstraction for the HIDA* algorithm. To benchmark its performance, extensive experiments have been run on the Sliding Tile Puzzle, Pancake, Topspin and Blocks World domains. The results show that the new technique outperforms existing domain abstraction with a single or multiple abstract spaces in terms of CPU time and memory usage.

The second part of this thesis is focused on the theory of state-set search. This theory investigates the scenario that a set of states are manipulated as a single state-set by a search algorithm. This scenario occurs frequently in abstraction and planning system and we are the first to formally define and analyze it. Based on this theory, we have found a path-based distance, d_{ww}, which is the maximum admissible distance between two state-sets.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Robert Holte for his tremendous help in this project. The discussion with Rob is always inspiring and makes me feel energetic. I have learnt a lot from Rob about the way of tackling challenging problems, great attention to details and other valuable assets. Thank you, Rob!

I would like to thank Neil Burch for providing his PSVN compiler which makes things way more easier and faster. I also want to thank Mehdi Samadi for the help he provided in experiment section.

At last, I would like to thank my family for their endless support and encouragement during my entire education. Thanks to Wendy for her support during the writing process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter gives a introduction to the thesis. First, we are going to give a brief description of state space search problems. Second, after the introduction of existing methods of solving state space search problems our approach is presented. In the third section, we will talk about our contributions in this thesis. Finally, a short description of each chapter is presented.

1.1 Problem Definition

Imagine that you are traveling from Edmonton to Jasper to enjoy the beautiful Rocky Mountains. You jump into the car, and enter the name Jasper into your GPS. Then this magical device guides you through the city, leading you all the way west to the mountains. How does this tiny device help you through the complicated road network? The general problem of this kind is what we are interested in in this thesis. In this particular example, to solve it, the road network is represented by a mathematical model, a graph G. All intersections and points of interest along the road are modeled as nodes in graph G. Roads between nodes are represented by directed edges. Now the problem of navigating from Edmonton to Jasper becomes finding a path from a node to another node in graph G. In general, each edge in G is associated with a non-negative cost. Any path between two nodes in G is also associated with a cost, which is the sum of costs of all edges along the path. What we are interested in is to find a path between two nodes with the lowest cost possible. This lowest cost path from one node to another is called an optimal solution. In the following discussion, we will refer nodes as states and G as a state space. This problem is called the state space search problem.

We also need to solve the state space search problem within a reasonable memory and time limit. This is challenging because unlike road networks, the state space of many problems is usually so large that it is impossible to fit the whole space into memory. For this reason, the state space is stored in the computer implicitly. That is to represent the space by a set of rules which are used to generate the space. Also searching in such a big space could take a long time to find a solution. To tackle the challenge of how to solve state space search problems within reasonable time and memory limit is the focus of the "heuristic search" research community.

1.2 Approach to Problem

The state of the art technique to solve state space search problem uses a search algorithm guided by a heuristic. A heuristic is an estimate of the distance from a state to a goal state. If the heuristic value is guaranteed to never overestimate the true distance, we call the heuristic admissible. With an admissible heuristic, standard search algorithms are guaranteed to produce optimal solutions. The method of generating heuristics is the issue we will focus on in this thesis.

One popular method to create heuristics is abstraction. The idea of abstraction is to use a function to map every state in the original state space to a new state space which we call the abstract space. This function is specially designed so that the distance between states in original space is never less than the distance between their images in the abstract space. Thus the abstract distance can be used as an admissible heuristic. Meanwhile, the abstract space should be much smaller than the original space so that it can be explored efficiently. The size of the abstract space has an important influence on the quality of heuristic extracted from it. With a larger abstract space, the heuristic value is, in general, closer to the true distance, so a search algorithm with this heuristic is faster. However, the improvement of heuristic quality with bigger abstract space does not come free; a bigger abstract space will cause more memory consumption and more time to explore it. If we want to have better heuristic quality without using a bigger abstract space, we need to come up with a more advanced technique of abstraction.

1.2.1 Multimapping Abstraction

The multimapping abstraction we present in this thesis is a novel framework for designing abstractions. There are three important characteristics of multimapping. First, each state in the original space has several images in the abstract space. In contrast, if using traditional abstraction, every state in the original space only has one image in the abstract space. Second, having multiple images for each state in the abstract space does not necessarily increase the size of the abstract space. This is because the abstraction function is specially designed so that all images of a state are in the same abstract space. Third, the distance from each image of a state to the nearest abstract goal state is still an underestimate of the distance from the state to the goal state in the original space. Since there exist several images of a state, and each of these images provide an admissible heuristic, the calculation of the heuristic value of a state in the original space can take the maximum of the heuristic value returned by each of its images. In this way, the heuristic quality can be improved without increasing the size of the abstract space.

1.2.2 State-Set Search Theory

Designing an abstraction function for multimapping also inspired us to investigate another important issue, which is the second part of this thesis: the state-set search theory. This theory investigates the scenario that a set of states are manipulated by a search algorithm as a single state-set. We give

a formal analysis of this scenario and develop four kinds of distances between state-sets. Among these four kinds of distance, we find that the cost of a "weak path" (defined in Chapter 4) between two state-sets P and Q is the largest possible admissible estimate for the true distance between any state in P and any state in Q. Thus it can be used as a way of generating a strong heuristic.

1.3 Contributions of this Research

The main contributions of multimapping abstractions are as follows:

1. We introduced the multimapping abstraction framework, which allows a state in the original space to have multiple images in a single abstract space. We investigated two specific methods to implement the multimapping abstraction framework. One of them is called multimapping domain abstraction, which we showed experimentally to be a successful method. The other method is called the Partial-State technique. This method still needs to be further developed but it inspired us to develop the state-set search theory.

2. We developed three enhancement techniques for the multimapping abstraction framework. These techniques can be implemented together and our experiments show that multimapping domain abstraction implemented with these three techniques is better in terms of memory use and CPU time than multimapping domain abstraction without these enhancements.

3. Experiments both in small scale and in large scale are done to compare multimapping domain abstraction to two existing methods: (1) domain abstraction with one abstract space, and (2) domain abstraction with multiple abstract spaces. The results show that domain multimapping abstraction in most domains has better performance in both CPU time and memory consumption.

Most of this material has been published in the 2012 Symposium on Combinatorial Search [29].

The main contributions of state-set theory are as follows:

1. We investigated the scenario in which a set of states in the original space is manipulated as a state-set by a search algorithm. A state-set theory has been developed to describe the properties and behavior of state-set search. This theory is independent on how a state-set is represented.

2. We discovered four kinds of distances between state-sets. The most important one among them is called a "weak path". Our analysis shows that the cost of a "weak path" between two state-sets P and Q is the largest possible admissible estimate for the true distance between any state in P and any state in Q. Thus this "weak path" can be utilized as a way of generating a strong heuristic.

Most of this material has been published in the 2011 Symposium on Combinatorial Search [28].

1.4 Outline

The rest of the thesis is going to be presented in the following way.

In Chapter 2, the background knowledge needed to understand the whole thesis is presented. This includes: (1) the representation of a state space search problem, (2) the state spaces we are going to use in the experiment sections, and (3) heuristics and heuristic search algorithms.

Chapter 3 begins by presenting the multimapping abstraction framework and an implementation of the framework called multimapping domain abstraction. Then the enhancement techniques for multimapping abstraction are described. In the experiment section, we designed both small scale and large scale experiments to investigate and verify the performance of multimapping domain abstraction. In the last part of the chapter, a new technique called Partial-State abstraction is presented.

In Chapter 4, the formal definition of the state-set theory is presented at the beginning of the chapter. Later, we described how the theory can be used to explain the behavior of some existing planning systems.

In Chapter 5, we summarize the contributions and limitations of the techniques and theories presented in this thesis.

If there is work related to our theory or technique, it is presented within corresponding chapters. Thus, we do not have a specific chapter that focuses on related work.

Chapter 2

Essential Background

In this chapter, we are going to introduce all the background knowledge readers need to have in order to understand the rest of the thesis. First, we are going to introduce the concept of heuristic search in Section 2.1. Next, we are going to talk about the representation of a state space search problem. The method we use to encode problems is presented in Section 2.2. For the purpose of evaluating our technique, we are going to test our method on several problem domains. Those domains are introduced in Section 2.3. The introduction of heuristics and the traditional way of creating heuristics is presented in Section 2.4. The search algorithms used in our experiments are introduced in Section 2.5.

2.1 State Space Search

The state space search problem is to find a solution path between two nodes in a graph G. The graph G is the mathematical model to represent the search space. Each node in G represents a state in the search space. Every directed edge between two nodes, say an edge from node a to node b, represents an action that can change state a to state b. All edges in G are associated with a non-negative cost. In this thesis, the cost for all edges is always 1. A path from node a to node b is a sequence of edges that connect a and b. The cost of this path is the sum of cost of all edges along the path. Because all edges cost 1 in all of our problems in this thesis, the cost of a path is simply the length of the path.

There are two kinds of solution path: optimal and suboptimal. An optimal solution is a path with the lowest cost. We denote the cost of the shortest path from a to b as $c(a, b)$ and the length of it as $d(a, b)$. In this thesis, $c(a, b) = d(a, b)$. A solution with cost larger than the cost of the optimal solution path is called a suboptimal solution. In this thesis, we only focus on the problem of finding an optimal solution.

Dijkstra's algorithm [8] finds an optimal solution from one node to every other node in the graph. However, Dijkstra's algorithm needs to save every node it has visited in memory. Because our problems have a huge state space, running Dijkstra's algorithm on those problems will be infeasible due to its memory requirements. The state-of-the-art technique to solve state space search problem

uses a search algorithm guided by extra information about the state space. This extra information is called a heuristic, and will be introduced in Section 2.4. With this extra information and the search algorithms described in Section 2.5, solving big state space search problems becomes possible. Searching under the guidance of heuristics is called heuristic search.

2.2 State Space Representation

As we mentioned previously, any state space search problem can be formulated as a search problem in a state space. In order to represent a state space, we need a formalism to describe the state space to the computer. If the state space is small, we can store it explicitly into memory by a technique such as an adjacency matrix [4]. However, large spaces cannot be stored in memory explicitly because they require too much memory. For large spaces, we use an implicit representation. Instead of storing every node in memory, an implicit representation describes the rules based on which the entire state space can be generated. In this thesis, we are going to employ the PSVN language [16, 2] to describe and implement state spaces.

First, we will introduce the definition of state space. The following definition of state space is taken from Zilles *et al.*[35].

Definition 1 (State Space) *A state space is a triple $\mathcal{S} = (D, k, \Pi)$ where D is a finite alphabet, $k \in \mathbb{N}$, and $\Pi \subseteq D^k \times D^k$. Every $s \in D^k$ is called a state and every pair $(s, s') \in \Pi$ is called an edge from state s to state s'.*

Note that the states defined in the above are not necessarily reachable from each other. In contrast, another definition commonly used in the literature is to define a state space to be a set of states reachable from a seed state $s_0 \in D^k$ [15, 16, 19]. However, differences between these two kinds of definition do not affect the content of this thesis.

Definition 2 (PSVN State) *A PSVN state s is a vector of fixed length k. For $i = \{1, \ldots, k\}$, the constants in each position i of s is from a finite set of possible values D_i. We define $D = \cup_{i=1}^{k} D_i$.*

Edges between states are represented by operators in PSVN.

Definition 3 (PSVN Operator) *A PSVN operator ω is a pair of vectors $\langle LHS, RHS \rangle$. Each of these vectors is of length k. The LHS describes the precondition of the operator and the RHS describes the effect. For each position i in LHS and RHS, it is either a constant from D_i or a variable.*

An operator defines two things. First, it defines the states to which the operator can be applied. Second, it defines the result of applying the operator. The following description of how a state is matched to an PSVN operator is taken from Burch *et al.* [2].

"For the operator matching rules, a state s can be applied with operator $\omega = \langle LHS, RHS \rangle$ if s matches LHS according to the following rule. For $i = \{1, 2, \ldots, k\}$, if $LHS[i]$ is a constant, $s[i]$ must equal $LHS[i]$. For $i, j = \{1, 2, \ldots, k\}$, if $LHS[i]$ and $LHS[j]$ are the same variable, $s[i]$ must equal $s[j]$".

For producing the result of an operator, there are two situations:

Deterministic Operators

An operator is deterministic if every variable in the RHS of the operator is also in the LHS. Assume $s' = \langle s'_1, \ldots, s'_k \rangle$ is the result of applying the operator $\langle LHS, RHS \rangle$ to $s = \langle s_1, \ldots, s_k \rangle$. For $i, j = \{1, \ldots, k\}$ if $RHS[j] \in D_j$ then $s'[j] = RHS[j]$. If $RHS[j]$ is a variable and $RHS[j] = LHS[i]$ then $s'[j] = s[i]$.

Non-deterministic Operators

An operator is non-deterministic if one or more variables in the RHS of the operator does not occur in the LHS. We call this kind of variable an unbound variable. The result of applying this kind of operator is a set of states. For positions in RHS that are not unbound variables, the corresponding positions in the resulting states are calculated in the same way as for deterministic operators. If position i in RHS is an unbound variable, the value for position i in the resulting states is drawn from D_i and the set of resulting states must include all possible combinations of values for unbound variables.

2.3 Puzzle Domains

In this section, we are going to introduce four problem domains which we will use in our experiments. For each domain, we used a small version and a large version.

2.3.1 Sliding Tile Puzzle

The sliding tile puzzle [32], as illustrated in Figure 2.1, is a l by l grid containing one blank location (represented by 0) and $l^2 - 1$ tiles numbered from 1 to $l^2 - 1$ (some variants have different names for the tiles but the basic mechanism of the puzzle is the same). The only way to change the arrangement of the tiles is to swap the blank tile with one of the adjacent tiles. The goal of this puzzle is to rearrange the tiles to become a desired permutation. In the following, we use "state" to refer to a permutation of tiles.

The small version of the puzzle we used later in our experiments is 3 by 3. This is called the 8-Puzzle. The large version of the puzzle we used is 4 by 4. This is called the 15-Puzzle. The goal states of 8-Puzzle and 15-Puzzle are presented in Figure 2.1. We define the size of a problem domain to be the number of reachable states in the corresponding search space. In this case, the size of 8-Puzzle is $181,440$ ($\frac{9!}{2}$) and the size of 15-Puzzle is $\frac{16!}{2}$. Next, we will define the asymptotic

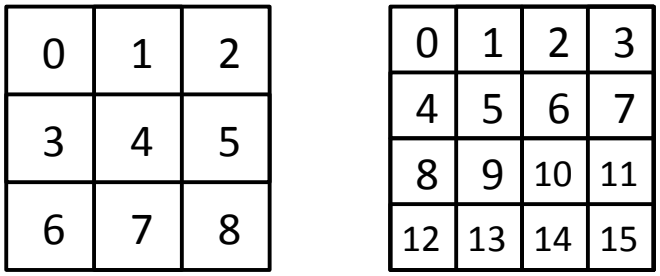


Figure 2.1: Goal states of the 8-Puzzle (left) and the 15-Puzzle (right).

branching factor [11]. The asymptotic branching factor is a measurement of the ratio of the number of children to their parents in a search space. We call it the branching factor for short in the following discussion. For the 8-Puzzle, it is 1.5 at even depths and 2 at odd depths. For 15-Puzzle, it is 2.1304 for both even and odd depths.

Encoding

We use a vector of length l^2 to represent a state for the $l \times l$ sliding tile puzzle. The blank tile is represented by 0. The numbered tiles are represented by the corresponding numbers. The tiles in a state are recorded in top-to-bottom and left-to-right fashion. For example, the 8-Puzzle state in Figure 2.1 is represented by $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8\rangle$. An operator that moves blank to the right when it is in the top left corner is encoded as $\langle 0, A, B, C, D, E, F, G, H\rangle \rightarrow \langle A, 0, B, C, D, E, F, G, H\rangle$.

2.3.2 Pancake Puzzle

The l-Pancake puzzle [9] is a stack of l pancakes numbered from 1 to l. The operators reverse the order of the top i pancakes for $i \in \{2, \ldots, l\}$. For example in the 3-Pancake state presented in the left of Figure 2.2, if we the flip the top 2 pancakes, we will get the state on the right with pancake 2 on the top of pancake 1. The aim of this puzzle is to the rearrange the pancakes to a desired arrangement. In our experiment, we used the 9-Pancake and 14-Pancake puzzles as the small and large versions respectively. The size of the 9-Pancake and 14-Pancake state spaces is $9!$ and $14!$ respectively. The branching factor for the 9-Pancake puzzle is 8 and for the 14-Pancake puzzle is 13.

Encoding

A state of l-Pancake puzzle is a vector of length l. Pancakes are encoded in a top-to-bottom fashion. For example, the left state of the 3-Pancake puzzle in Figure 2.2 is encoded as $\langle 1, 2, 3\rangle$ and the right state is encoded as $\langle 2, 1, 3\rangle$. An operator reversing the top 2 pancakes is encoded as $\langle A, B, C\rangle \rightarrow \langle B, A, C\rangle$

Figure 2.2: An example of the 3-Pancake puzzle.

2.3.3 TopSpin Puzzle

The (20,4)-Topspin puzzle is presented in Figure 2.3. This puzzle consists of 20 tokens in a round track and a turnstile which can reverse 4 tokens at one time. Because tokens are placed in a round track and all of them can be slid along the track, any successive 4 tokens can be slid into the turnstile. The aim of this puzzle is to use the turnstile to rearrange the tokens into a specific order, usually in ascending order from 1 to 20. The small version of this puzzle used in our experiment is (10,4)-Topspin, meaning there are only 10 tokens and a turnstile of capacity 4. The large version we used is (15,4)-Topspin. In the following discussion, the order of tokens in the track is a "state". This state only represents to the relative order rather than the absolute positions of the tokens in the track. Sliding tokens along the track will not change the state of the puzzle. The size of (10,4)-Topspin is $10!$ and the size of (15,4)-Topspin is $\frac{15!}{2}$ [3]. The branching factors for (10,4)-Topspin and (15,4)-Topspin are 10 and 15 respectively.

Figure 2.3: A picture of (20,4)-Topspin Puzzle[1].

Encoding

The encoding of the Topspin puzzle is a little bit tricky. Because tokens have no fixed position in the track, they are encoded in their position relative to a flag token. This flag token can be any token in the track. When we encode the state into a vector, the flag token is fixed at the beginning of the vector and is never moved no matter how we flip the turnstile. For this reason, we can ignore this

[1]This image is taken from `http://www.passionforpuzzles.com/puzzles/top-spin.php`

flag token in the vector of a state. Therefore, a (m, n)-Topspin puzzle state can be encoded with a vector of length $m-1$. For example, in Figure 2.3, if we choose token 1 as the flag token, the vector of the state will be $\langle 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\rangle$. Note that there is no value 1 in this vector. An example of an operator which reverses positions 2 to 5 is

$$LHS : \langle a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s\rangle$$

$$RHS : \langle \mathbf{d}, \mathbf{c}, \mathbf{b}, \mathbf{a}, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s\rangle$$

2.3.4 Blocks World With Table Positions

The (m, n)-Blocks World [35] is a variant of traditional Blocks World puzzle [31]. The (m, n)-Blocks World consists of m blocks numbered from 1 to m, n table positions, and one claw. Blocks can be stacked on each other as long as the stack is at a legal table position. If the claw is empty, it can pick up the top block of any stack. If the claw is holding a block, it can put it down on top of any stack or on an empty table position. The goal of this puzzle is to rearrange all blocks to be at a particular table position with a specific order. The small blocks world puzzle we used is $(8, 3)$-Blocks World and the big one is $(12, 3)$-Blocks World. An example of the $(3, 2)$-Blocks World is presented in Figure 2.4. A state of the blocks world includes the position of each block, including the block held by the claw if any. The size of the (8,3)-Blocks World is $3,265,920$. We have not calculated the exact size of (12,3)-Blocks World but it is certainly larger than $12!$. For branching factor, we do not know the exact value but for both domains it will not exceed 3. Generally, for the Blocks World with n table positions the branching factor will not exceed n.

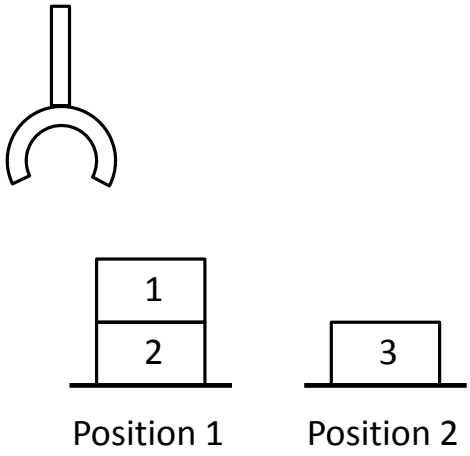


Figure 2.4: An example of the (3,2)-Blocks World.

Encoding

The following method to encode the (m, n)-Blocks World was invented by Zilles and Holte [35]. To encode a state of (m, n)-Blocks World, we used a vector of length $(m + 1) \times n + 1$ as illustrated in the following.

$$\langle \underbrace{i_1}_{\text{name of the block held by the claw}}, \overbrace{i_2, \ldots, i_{m+2}}^{\text{first table position}}, \underbrace{\ldots}_{\text{remaining } n-1 \text{ table positions}} \rangle$$

The first value of the vector is the name of the block held by the claw. It is 0 if no block is held by the claw. The next $m+1$ values represent the first table position and the remaining $(m+1) \times (n-1)$ value represents the remaining $n - 1$ table positions. For the $m + 1$ values that represent a position, the first value represents how many blocks are at this table position. For example i_2 in the above example is the number of blocks located at the first table position. The remaining m values are the names of the blocks at the current position for each level from bottom to top, 0 if empty. For example, the encoding of Figure 2.4 is the following:

$$\langle \underbrace{0}_{\text{is 0 because no block is held by the claw}}, \overbrace{2, 2, 1, 0}^{\text{first table position}}, \underbrace{1, 3, 0, 0}_{\text{second table position}} \rangle$$

Operators that pick up a block on the first table position are:

$$\langle 0, 3, A, B, C, 0, 0, 0, 0 \rangle \rightarrow \langle C, 2, A, B, 0, 0, 0, 0, 0 \rangle$$

$$\langle 0, 2, A, B, 0, 1, C, 0, 0 \rangle \rightarrow \langle B, 1, A, 0, 0, 1, C, 0, 0 \rangle$$

$$\langle 0, 1, A, 0, 0, 2, B, C, 0 \rangle \rightarrow \langle A, 0, 0, 0, 0, 2, B, C, 0 \rangle$$

2.4 Heuristics

A heuristic is an estimate of the distance from the current state s to goal state g. It provides extra information to search algorithms to speed up search. We denote the heuristic value of state s as $h(s)$. The closer $h(s)$ is to the real distance from s to g, the better the heuristic quality is. Heuristics with good quality will largely reduce the number of nodes expanded in the search process [25]. At the beginning of this section, we are going to introduce two important properties of a heuristic: admissibility and consistency. After that, we will introduce how to create a heuristic using abstraction.

2.4.1 Properties of a Heuristic

Admissibility

We say heuristic h is admissible if for all s, $h(s) \leq d(s, g)$ where g is a goal state. That is to say, the heuristic h never overestimates the true distance from any state to the goal state. If $h(s) = d(s, g)$

for all s, the heuristic is called a perfect heuristic. With the guidance of an admissible heuristic, the solution we find using search algorithms like A*[13] and IDA*[22] is guaranteed to be optimal. In this thesis, because we only focus on finding optimal solutions, we will only use admissible heuristics.

Consistency

Another important property of a heuristic is consistency. Heuristic h is consistent if for every state n and every successor of n, p, the following holds:

$$h(n) \leq c(n, p) + h(p)$$

Because $c(n, p) = d(n, p)$ in all domains in this thesis, we can also write the above inequality as

$$h(n) \leq d(n, p) + h(p)$$

This is a desirable property of a heuristic. The reason why consistency is desirable will be discussed in Section 2.5. In this thesis, we only use consistent and admissible heuristics.

2.4.2 Generating Heuristics Using Abstraction

An effective way to generate heuristics is by abstraction. The fundamental idea is to create a smaller space, which retains some information from the original space, and use the true distances in the smaller space as the heuristic. Usually, this smaller space should be small enough so that we can enumerate all the states in the space. An abstraction is a method to generate this smaller space out of the original space. Specifically, we use an abstraction function $\phi(\cdot)$ to map states in the original space to abstract states, and this abstraction function ϕ must have a property called state space homomorphism [16]. The following definition is taken from Hernádvölgyi and Holte [16].

Definition 4 (State Space Homomorphism) *Let S and T be state spaces with operators Ω_S and Ω_T respectively. $\phi : S \to T$ is a state space homomorphism if the following always holds: for every state $u, v \in S$ if $\exists \omega \in \Omega_S$ such that $\omega(s) = t$, then $\exists \omega' \in \Omega_T$ such that $\omega'(\phi(s)) = \phi(t)$.*

If the abstraction function ϕ have the property of state space homomorphism, then for any pair of states p and q in original space, the following always holds:

$$d(\phi(p), \phi(q)) \leq d(p, q)$$

That is to say, the true distance from $\phi(p)$ to the abstract goal state $\phi(g)$ is always an admissible heuristic value for p in the original space [16]. It can be proved that the heuristic generated by abstraction is consistent [16].

s	0	1	2	3	4	5	6	7	8
$\phi(s)$	0	1	1	1	1	2	3	4	5

Table 2.1: An example of a domain abstraction function.

Domain Abstraction

Domain abstraction [16] is an abstraction method based on a total function $\phi : D \rightarrow D'$, where $|D'| < |D|$. This ϕ induces a state space abstraction, which we will also call ϕ, as follows: if $s = \langle s_1, \ldots, s_k \rangle$, then $\phi(s) = \langle \phi(s_1), \ldots, \phi(s_k) \rangle$.

It has been proven that domain abstraction has the property that $d(\phi(p), \phi(q)) \leq d(p, q)$ for any pair of p, q in original space [16].

Example 1 (Using domain abstraction on the 8-Puzzle) *An example of a domain abstraction is presented in Table 2.1. Suppose the original state of the 8-Puzzle is* $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8 \rangle$. *After applying this domain abstraction function, the abstract state is* $\langle 0, 1, 1, 1, 1, 2, 3, 4, 5 \rangle$*, as illustrated in Figure 2.5.*

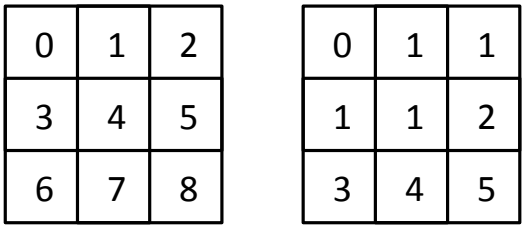


Figure 2.5: An example of an 8-Puzzle state and its abstraction.

Granularity

Granularity [15] is a property of a domain abstraction function. The following defintion is taken from Hernadvolgyi *et al.* [15]:

Definition 5 (Granularity) *The granularity of a domain abstraction* $\phi : D \rightarrow D'$ *is a vector* $\langle n_1, \ldots, n_j \rangle$*, where* $j = |D'|$*. Each* n_i *is the number of constants in* D *being mapped to constant* $d_i \in D'$*. We reorder* n_i *so that* $n_i \geq n_{i+1}$ *for all* i*. Sometimes, when* $|D|$ *is clear from the context, we will omit the* 1*'s in the vector.*

Example 2 *The granularity of domain abstraction in Table 2.1 is* $\langle 4, 1, 1, 1, 1, 1 \rangle$.

We use granularity to classify abstractions into different categories in the experiments in Chapter 3. An important reason to do this is because the granularity of the domain abstractions used in

this thesis determines the size of the abstract space generated by the domain abstraction. However, granularity does not generally determine the size of a abstract space created by domain abstraction [15].

Projection

A projection abstraction [10, 35] ϕ is defined by a subset $\{i_1, \ldots, i_m\} \subset \{1, \ldots, k\}$. Applying ϕ to state $\langle s_1, \ldots, s_k \rangle$ gives state $\langle s_{i_1}, \ldots, s_{i_m} \rangle$.

The basic idea of projection abstraction is to eliminate certain fixed positions in the state vector.

Example 3 *Let ϕ be a projection abstraction which eliminates second position in a state and s be state $\langle 1, 2, 3, 4 \rangle$. Then $\phi(s) = \langle 1, 3, 4 \rangle$.*

Pattern Database

A pattern database[6] is a common technique to represent heuristics created using abstraction. A pattern database is lookup table containing the distances from each abstract state to the abstract goal state. To get a heuristic value for a state s, we look for the corresponding abstract state $\phi(s)$ in the pattern database and return the value.

A pattern database is created by using breadth-first search [4] backward from the abstract goal state. In this way, when the search reaches a state the distance from this state to the abstract goal state is immediately acquired and stored in the PDB. This process continues until all abstract states are visited and stored in the PDB.

2.5 Search Algorithms

In this section, we are going to introduce three heuristic search algorithms, A*, IDA* and HIDA*. Even though we only use HIDA* in our experiments, the other two serve as a foundation to understand HIDA*. In the following discussion, for a state s, we use $h(s)$ to denote the heuristic value of s and $g(s)$ to denote the cost of the path from the start state to s. The f-value of s is defined as $f(s) = g(s) + h(s)$. We say a state is expanded when a search algorithm applies the operators to it and generates all its children.

2.5.1 A*

A*[13] is a kind of best first search algorithm [30]. It uses two lists of states: OPEN and CLOSED . OPEN is a list of states the algorithm has reached but not yet expanded. It is always sorted according to the f-value of the states. CLOSED is a list of states the algorithm has already expanded. A* always expands a state in OPEN with the lowest f-value. For a successor p of the expanded state, if p does not exist in either OPEN or CLOSED, it will be added to OPEN. If p already exists in

OPEN and the newly generated p has a lower f-value, the f-value of p in OPEN will be updated. If p already exists in CLOSED and the newly generated p has a lower f-value, p will be removed from CLOSED and be added to OPEN with the new f-value. The algorithm terminates when the goal state is expanded. If used with an admissible heuristic, A* is guaranteed to return an optimal solution*[13]. If the heuristic is consistent, it is guaranteed that A* will not re-expand any state [13].

The drawback of A* is that it has a high memory requirement. This is because it will store every state it has reached during the search process. If the search space is large and the heuristic is imperfect, A* will usually run out of memory before finding a solution, and thus be an infeasible approach.

2.5.2 IDA*

IDA* [22] is a search algorithm often used in large search spaces. This is because the memory requirement of IDA* is proportional to the length of the solution it finds. Unless the solution length is exponentially long, IDA* will have a low memory requirement for large spaces. The idea of IDA* is to perform depth-first search [4] within a certain cost bound. Any state with its f-value higher than the bound is immediately pruned. If the goal is not found in the current iteration, the bound is increased to the minimum f-value of the pruned states in this iteration. The algorithm terminates immediately if a goal state is found with an f-value equal to the current cost bound. Like A*, provided with an admissible heuristic IDA* will find an optimal solution[22].

Besides having a low memory requirement in most cases, another advantage of IDA* is that it is easy to implement. Because it does not involve implementation of OPEN and CLOSED like A*, the code for IDA* is short and succinct. However, in IDA* most states in the search tree will be expanded numerous times and this can be a big computational cost. The overall overhead of IDA* is dominated by the last iteration [22]. Pseudocode for IDA* is shown in Algorithm 1.

Algorithm 1 IDA*

```
IDA(start, goal)
    bound ← h(start)
    while goal not found:
        bound ← DFS(start,goal,0,bound)

DFS(s, goal, g, bound)
    If s == goal: exit with success
    g ← g + 1
    newbound ← ∞
    Iterate over x ∈ children(s):
        f(x) ← g + h(x)
        if f(x) ≤ bound: f(x) ← DFS(x, goal, g, bound)
        if f(x) < newbound: newbound ← f(x)
    Return newbound
```

2.5.3 HIDA*

HIDA*[18] is a search algorithm that combines IDA* with hierarchical abstractions. Hierarchical abstractions, as illustrated in Figure 2.6, are designed to be a set of abstract spaces labeled from level 1 to level l. For each $i = \{2, \ldots, l\}$, abstract space level i is an abstract space for level $(i-1)$ created by treating abstract space level $(i-1)$ as an original space and applying the abstraction ϕ_{i-1}. Abstract space level 1 is an abstract space for the original space. We call the original space the base level in the following discussion. In the hierarchy of abstractions, the heuristic value for any state s in level i can be obtained by calculating the true distance between $\phi_i(s)$ to the closest abstract goal state in abstract space level $(i+1)$.

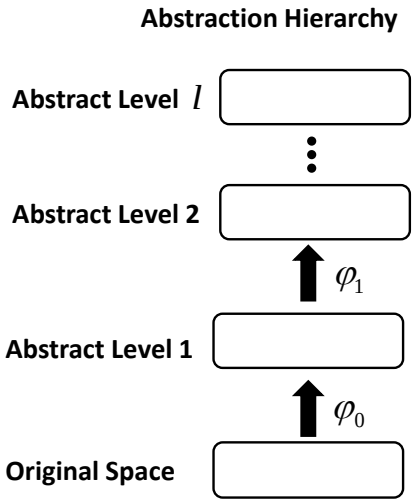


Figure 2.6: Abstraction Hierarchy

The idea of HIDA* is to apply the IDA* algorithm to the original space and all the abstract spaces. If a heuristic value of state s in original space or abstract space level from 1 to $(l-1)$ is needed, IDA* is used to search in the immediately higher level abstract space and return the true distance as a heuristic value. If the heurisitic value of any state in level l is needed, a default heuristic value 0 is used. During the search, the heuristic value for any state s in abstract space is stored in memory to prevent wasting resource to recompute it again in the future. HIDA* also uses two techniques to improve its search performance. The first technique is called P-g caching [21], which is presented in bold lines in Algorithm 2. It improves the heuristic value during the search. Another technique is called optimal path caching. It saves search effort by detecting if the optimal path from the current abstract state to goal abstract state has already been found during a previous search.

An advantage of HIDA* is that it does not require providing a PDB or a heuristic function for guidance. This algorithm will calculate heuristic values on its own and only calculate the heuristic

values needed to solve an instance. For this reason, if the number of problem instances is small, the overhead of generating a PDB might outweigh the overhead of using HIDA*. Thus, HIDA* is ideal for situations in which the number of problem instances is small or the goal state of each instance is different.

Algorithm 2 HIDA*

```
HIDA(start, goal)
    bound ← h(start,goal)
    while goal not found:
        bound ← DFS(start,goal,0,bound)
    For all nodes x on the solution path:
        cache[x].exact ← True
        cache[x].dist ← distance from x to goal
```

```
DFS(s, goal, g, bound)
    If s == goal: exit with success
    g ← g + 1
    newbound ← ∞
    Iterate over x ∈ children(s):
        // P-g caching
        cache[x].dist ← max(cache[x].dist, bound-g, h(x,goal))
        f ← g + cache[x].dist
        // Optimal path caching
        if (f==bound) and (cache[x].exact == True):
            exit with success
        if f ≤ bound: f ← DFS(x, goal, g, bound)
        if f < newbound: newbound ← f
    Return newbound
```

```
h(s,goal)
    If at the top abstraction level, return 0
    if cache[φ(s)].exact == False:
        HIDA(φ(s),φ(goal))
    Return cache[φ(s)].dist
```

2.6 Summary

In this chapter, we introduced the essential information needed to understand our research. We have introduced the basic concept of heuristic search, the PSVN representation of state spaces, four problem domains, the properties of heuristics, and three search algorithms (A*, IDA*, and HIDA*).

Chapter 3

Multimapping Abstraction

In this chapter, we are going to introduce the multimapping abstraction technique. The fundamental idea of this technique is about making abstractions such that a state in the original space could have multiple images in the abstract space. Multimapping abstraction itself is a framework rather than a specific method. The details of this framework will be discussed at the beginning of this chapter. After that we are going to present two specific methods of implementing the multimapping abstraction framework. The first method, based on domain abstraction, will be presented in Section 3.1. Extensive experiments are carried out in different problem domains and our results show that multimapping abstraction with domain abstraction is superior to standard alternatives in terms of both memory usage and CPU time . The second method incorporates multimapping with a Partial-State technique, which is a completely new method. This method will be presented in Section 3.4. For the second method, only limited experiments are done to evaluate this technique. Most of the material in this chapter has been published in the 2012 Symposium on Combinatorial Search [29].

3.1 Multimapping Abstraction Framework

The framework of multimapping abstraction is defined by the following two characteristics. Firstly, the output of abstraction $\phi(\cdot)$ is a set of abstract states rather than a single abstract state. Secondly, the following holds for any pair of states s, t in the original space:

$$\forall s' \in \phi(s) : \min_{t' \in \phi(t)} d(s', t') \leq d(s, t)$$

We define n to be the **Mapping Factor**, which is the number of abstract states an original state will be mapped to. In order to guarantee admissibility and maintain a good quality of heuristic value, $h(s)$ is extracted from the abstract space as follows:

$$h(s) = \max_{s' \in \phi(s)} \min_{g' \in \phi(g)} d(s', g')$$

g in this formula represents the goal state. This idea is illustrated in Figure 3.1. In this figure, let s and g be states in the original space and each of them have three images in the abstract space, namely

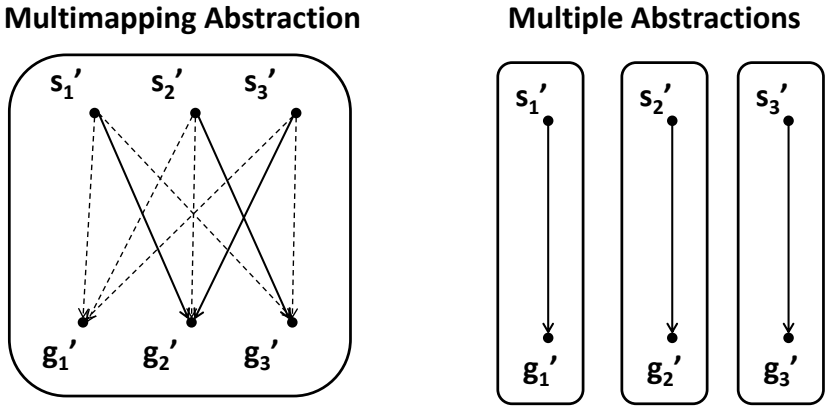


Figure 3.1: Difference between multimapping Abstraction and Multiple Abstractions.

$\phi(s) = \{s'_1, s'_2, s'_3\}$ and $\phi(g) = \{g'_1, g'_2, g'_3\}$. To get the heuristic value for s in this multimapping abstraction, there are 9 distances we can use, $d(s'_i, g'_j)$ for $i, j \in \{1, 2, 3\}$. First, we calculate the minimum distance for each abstract state s' by $\min_{g' \in \phi(g)} d(s', g')$. In this particular case, assume that the three distances we get are $d(s'_1, g'_2)$, $d(s'_2, g'_3)$ and $d(s'_3, g'_2)$ shown as bold lines in Figure 3.1. The final heuristic of s is the maximum distance among these three, and in this case we assume it is $d(s'_3, g'_2)$. This distance is used as the heuristic value for s. We can prove the distance we have chosen is an admissible heuristic providing that the multimapping abstraction $\phi(\cdot)$ fits into the multimapping abstraction framework that is defined at the beginning of this section. Also it is consistent but asymmetric as we will discuss below.

Admissible Heuristic

Theorem 4 *Given a multimapping abstraction defined as above, then*

$$h(s) = \max_{s' \in \phi(s)} \min_{g' \in \phi(g)} d(s', g') \tag{3.1}$$

provides admissible heuristic value for s.

Proof. Based on definition of multimapping, we know that for each $s'_i \in \phi(s)$, $\min_{g' \in \phi(g)} d(s'_i, g')$ provides an admissible heuristic, therefore picking the maximum value returned by each s'_i still provides an admissible heuristic. So $h(s)$ is guaranteed to be an admissible heuristic.

Consistent Heuristic

Theorem 5 *The heuristic defined above is consistent.*

Proof. Let us assume s and t are two states in original space. In order to prove that $h(s) \leq d(s,t)+h(t)$, we first choose two abstract states s' and g'_s such that $d(s', g'_s) = h(s)$. For any original state t, there must exist t' such that $d(s', t') \leq d(s,t)$ according to the definition of multimapping. Let us assume the closest abstracted goal state to t' is g'_t, then we have $d(s', g'_s) \leq d(s', g'_t)$. Because s', t' and g'_t are in the same abstract space, the triangle inequality holds for them, *i.e.,*

$$d(s', g'_t) \leq d(s', t') + d(t', g'_t)$$

We have:

$$\begin{aligned} h(s) &= d(s', g'_s) \\ &\leq d(s', g'_t) \\ &\leq d(s', t') + d(t', g'_t) \\ &\leq d(s,t) + d(t', g'_t) \end{aligned}$$

Because $h(t) = \max_{d' \in \phi(t)} d(d', g'_d) \geq d(t', g'_t)$, we have:

$$h(s) \leq d(s,t) + h(t)$$

Heuristic h is consistent. □

Asymmetric Heuristic

First, we need to define symmetric state space.

Definition 6 (Symmetric Space and Heuristic) *A state space is symmetric if for any pair of states p and q we have $d(p,q) = d(q,p)$. A heuristic h is symmetric if for any pair of states p and q we have $h(p,q) = h(q,p)$.*

We call a state space or a heurisitc asymmetric if this symmetric property does not hold. For heuristic based on multimapping, it is asymmetric even if the abstract space is symmetric. The heuristic calculated from start state to goal state might be different from the heuristic calculated from goal state to start state. We call the calculation from start state to goal state the forward fashion and the other way the backward fashion in the following discussion. The fact that the forward and backward calculation can be different is illustrated in the following example.

Example 6 *s and t are two states in original space. Given a multimapping with a Mapping Factor of 2, each of them is mapped to two states, s'_1, s'_2, t'_1 and t'_2 respectively. First, let us choose s to be the start state and t to be the goal state, we have*

$$h(s,t) = \max\{\min\{d(s'_1, t'_1), d(s'_1, t'_2)\}, \min\{d(s'_2, t'_1), d(s'_2, t'_2)\}\}$$

If we choose t to be the start state and s to be the goal state, we will have

$$h(t,s) = \max\{\min\{d(t'_1, s'_1), d(t'_1, s'_2)\}, \min\{d(t'_2, s'_1), d(t'_2, s'_2)\}\}$$

Suppose $d(s'_1, t'_1) = 1$, $d(s'_1, t'_2) = 2$, $d(s'_2, t'_1) = 3$, $d(s'_2, t'_2) = 4$, then we will have $h(s,t) = 3$ and $h(t,s) = 2$. Therefore, it is possible that $h(s,t)$ does not equal $h(t,s)$ in certain situations.

In the experiment section, we will investigate this issue further by comparing the differences of heuristic values calculated in the forward fashion and the backward fashion. We can also utilize this property to enhance heuristic quality by taking maximum of the heuristic values calculated in the forward and backward fashions. However, this enhancement of the heuristic value will come with increased computational overhead so we did not explore this in our experiment.

Differences Between Multimapping and Multiple Abstractions

The multiple abstractions technique [17, 20] also involves multiple mappings and multiple lookups of heuristic values. In multiple abstractions technique, we first create m independent abstract spaces. Let $h_1(s) \ldots h_m(s)$ be the heuristic values of state s extracted from these independent abstract spaces. The final heuristic value of s is calculated as follows:

$$h(s) = \max_i h_i(s).$$

The advantage of multiple abstractions technique is that the heuristic extracted from m small abstract spaces of size $\frac{z}{m}$ using multiple abstractions technique tends to be better than the heuristic extracted from a single abstract space of size z in general [17].

The fundamental differences between multimapping and multiple abstractions can be illustrated using Figure 3.1, where s and g are states in the original space. s'_i and g'_i are the abstract states corresponding to s and g respectively. For the multiple abstractions technique, the heuristic value is extracted from isolated abstract spaces. We say those abstract spaces are isolated because there is no path from an abstract state in one abstract space to any abstract state in another abstract space. Besides, there is only one goal abstract state in each abstract space for multiple abstractions, therefore it does not need to choose minimum distance to different abstract goal states like the multimapping technique does.

For the multimapping abstraction framework, we aim to have all abstract images of a state in the same abstract space. In order to make multimapping have this property, the abstraction function of multimapping must be carefully designed. If the abstraction function cannot be designed in this way, the multimapping abstraction framework will become multiple abstractions and the features and advantages of multimapping abstraction framework will disappear.

Differences Between Multimapping and Symmetry

Symmetry techniques [34, 12, 24, 7] also involve multiple lookups when a heuristic value for a state s is needed. Assuming $\mathcal{S}$ is a state space, the idea of a symmetry technique is to design a set of

functions $sim_i(\cdot) : \mathcal{S} \rightarrow \mathcal{S}$ such that $d(s, g)$ is guaranteed to be the same as $d(sim_i(s), g)$. In this case, $h(sim_i(s))$ is guaranteed to be an admissible heuristic value for s. The final heuristic value of s can be generated by $\max\{h(s), \max_i\{h(sim_i(s))\}\}$. An example of using symmetry in the $(3, 3)$-Blocks World is given below.

Example 7 *In $(3, 3)$-Blocks World puzzle, lets assume that the goal state is*

$$\langle 0,\ \ 3, 1, 2, 3,\ \ 0, 0, 0, 0,\ \ 0, 0, 0, 0\rangle.$$

For state s

$$s = \langle 0,\ \ 0, 0, 0, 0,\ \ 1, 1, 0, 0,\ \ 2, 2, 3, 0\rangle$$

a symmetry state $sim(s)$ is

$$sim(s) = \langle 0,\ \ 0, 0, 0, 0,\ \ 2, 2, 3, 0,\ \ 1, 1, 0, 0\rangle.$$

If there exists a path π from s to the goal state, there must exist a path π_{sim} with equal length of π from $sim(s)$ to the goal state. We can create this π_{sim} from π by changing every action on the second and third table positions to an action on the third and second table positions respectively. In such cases, $\max\{h(s), h(sim(s))\}$ can be used as a heuristic value for both s and $sim(s)$.

There is a major structural difference between the multimapping and the symmetry technique. In the symmetry technique, the multiple lookups are generated by having multiple original states but only one abstraction. However, in the multimapping technique, multiple lookups are generated by having multiple abstractions but there is only one original state.

Another difference for implementing these two techniques is that symmetry techniques are domain specific while multimapping is not. It is not guaranteed that a symmetry technique can be implemented in any domain. For example, in the Glued-15 Puzzle we used in Section 3.3.4 where the tile 9 is fixed, symmetry technique cannot be applied. While multimapping abstraction is a general method and can be implemented in any domain.

Implementation with Domain Abstraction

Up to now, we have given a framework for multimapping abstraction. To implement this idea, all we need to do is to define an abstraction which takes a single state as input and gives a set of states as output. In this section, we propose a simple extension of domain abstraction to implement this idea. We call this technique multimapping domain abstraction.

The key idea is to design a set of abstraction $\Phi(\cdot) = \{\phi_1(\cdot), \phi_2(\cdot), \cdots, \phi_n(\cdot)\}$, where each $\phi_i(\cdot)$ is a normal domain abstraction. $\Phi(\cdot)$ defined in this way is a multimapping domain abstraction. For an input state s , applying $\Phi(\cdot)$ would be equivalent to applying each abstraction $\phi_i(\cdot)$ to the state s. The result is a set of abstract states, $\Phi(s) = \{\phi_1(s), \phi_2(s), \cdots, \phi_n(s)\}$. Each domain abstraction $\phi_i(\cdot)$ should be carefully designed so that $\{\phi_1(s), \phi_2(s), \cdots, \phi_n(s)\}$ are reachable from each other.

Theorem 8 *Domain multimapping conforms to the definition of multimapping abstraction.*

Proof. Firstly, domain abstraction generates a set of abstract states. Secondly, given s and t, for each $\phi_i(s)$, because $\phi_i()$ is an abstraction we must have $d(\phi_i(s), \phi_i(t)) \leq d(s,t)$ □

Example 9 *We choose 9-Pancake as the domain to illustrate this idea. Suppose we have a start state* $s = \langle 1,4,6,5,7,0,8,3,2 \rangle$*, and we define the multimapping function to be* $\Phi(\cdot) = \{\phi_1(\cdot), \phi_2(\cdot)\}$ *which means any state in original space will be mapped to two abstract states. For this particular example, we design* $\phi_1(\cdot)$ *and* $\phi_2(\cdot)$ *to be functions presented in Table 3.1. Applying* $\Phi(\cdot)$ *to start state s, we have* $\Phi(s) = \{s'_1, s'_2\} = \{\langle a,b,6,b,7,0,8,a,a \rangle, \langle b,a,6,a,7,0,8,a,b \rangle\}$*. These* s'_1 *and* s'_2 *are abstract states in the same abstract space because all permutations of tiles are reachable from each other in the Pancake problem space. The heuristic value for* s *derived from the abstract space, simply takes the maximum of the distance from* s'_1 *,* s'_2 *to the closest goal abstract state.*

s	0	1	2	3	4	5	6	7	8
$\phi_1(s)$	0	a	a	a	b	b	6	7	8
$\phi_2(s)$	0	b	b	a	a	a	6	7	8

Table 3.1: Domain abstractions ϕ_1 and ϕ_2.

3.2 Enhancements

In this section, we are going to introduce three enhancements for multimapping domain abstraction. These three enhancements do not have anything to do with the algorithm but only involve the design of the abstraction.

3.2.1 Choosing A Good Mapping Factor

The mapping factor affects the performance of multimapping abstraction in two ways. Firstly, the heuristic value is calculated by taking a maximum over the states in $\phi(s)$. If there is a larger number of such states, or a higher mapping factor, the final heuristic value will be higher. This is the bright side of a higher mapping factor. Secondly, when computing the distance to goal for each state in $\phi(s)$, a minimum is also taken over all the states in $\phi(g)$. If there is a higher number of states in $\phi(g)$, which means a higher mapping factor, the distance for each state in $\phi(s)$ will be lower. This is a harmful effect caused by a high mapping factor.

As discussed, a higher mapping factor is a double-edged sword, so the question is what Mapping Factor that will give us best heuristic quality?

We have chosen the 8-Puzzle domain to investigate this issue. We design a first level abstraction configuration with granularity of $\langle 3,3,2,1 \rangle$. In all of our experiments, the blank tile is never abstracted and is kept unique, so only tiles 1 to 8 are abstracted.

For the 8-Puzzle domain with a $\langle 3, 3, 2 \rangle$ granularity on tiles 1 to 8, there are 280 different choices of domain abstractions. Multimapping domain abstraction of mapping factor n can be generated by choosing n different domain abstractions and using them together. We generated 280 domain multimapping abstractions for each mapping factor. In our experiment, the goal state is set to be $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ where 0 represents the blank. The start states we used for evaluating the abstractions are 500 randomly generated states from all reachable states in the problem domain. For each abstraction, the median and average number of nodes expanded at the base level in solving the 500 test cases was recorded. The results (truncated to integer values) for each mapping factor are presented in Table 3.2.

Mapping Factor	Median	Average
1	3742	3699
2	1725	1824
3	1549	1545
4	1597	1610
5	1693	1699
⋮	⋮	⋮
24	4108	4100

Table 3.2: Number of Nodes Expanded at the Base Level for Different Mapping Factors.

From Table 3.2, we can see that both the median and average of number of nodes expanded are showing the same trend. Both numbers begin with a high value then continue to drop as the mapping factor increases. Both reach a minimum when the mapping factor is 3. At this point, the average number of nodes expanded at the base level is only 50% of that when the mapping factor is 1. After that, the numbers rise with each further increase of the mapping factor. When the mapping factor reaches 24, both numbers exceed the value when the mapping factor is 1.

In this experiment, the best heuristic quality occurs when the mapping factor is 3. In the rest of the experiments in this chapter, we will use 3 as the mapping factor. Using 3 as mapping factor for all domains might not be the best choice because 3 is chosen only based on a experiment done in the 8-Puzzle domain. However, for our convenience we used 3 in all our following experiments.

3.2.2 Goal Aggregation

Another enhancement technique we used to improve multimapping is called goal aggregation (GA). The basic idea of this technique is to map goal state g to abstract states g'_i which are near to each other. For example, in the 9-Pancake puzzle, with the abstractions presented in Table 3.3 the goal state $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ will be mapped to two abstract states $\langle 1, 3, 2, 1, 2, 4, 2, 1, 8 \rangle$ and $\langle 1, 2, 3, 1, 2, 4, 2, 1, 8 \rangle$. These two abstract states are only one step away from each other in the abstract space.

s	0	1	2	3	4	5	6	7	8
$\phi_1(s)$	1	3	2	1	2	4	2	1	8
$\phi_2(s)$	1	2	3	1	2	4	2	1	8

Table 3.3: Multimapping domain abstractions used for goal aggregation.

The reason we want to minimize the distance Δ between abstract goal states is that this is an upper bound of the "harm" that can be done by taking the minimum. This idea can be illustrated by the following inequality: for any abstract state s',

$$\max_{g' \in \phi(g)} d(s', g') \leq \Delta + \min_{g' \in \phi(g)} d(s', g')$$

For small Δ, the harm of taking the minimum is limited.

There is another benefit when we implement this technique with HIDA*. Because the goal abstract states are close to each other, the abstract space around them will be fully explored and cached in memory after a few searches. This will speedup the following searches and reduce memory usage.

Goal aggregation is not effective in all domains. For some domains, it is impossible to map the goal state to abstract goal states which are close to each other. For example, in the (8,3)-Blocks World domain, at least 8 steps are required to transform one abstract goal state to another abstract state which we can map the goal state to. This is because all disks must be stacked at the first table position so that we can abstract the goal state to the abstract state. Swapping the top two disks requires 8 steps. If we want an additional abstract goal state, it will require shuffling the top 3 disks, which needs at least 12 steps. In our experiment, we have used abstractions of the (8,3)-Blocks World with granularity of $\langle 3, 3 \rangle$. The average heuristic value in this case is 24.7, so a Δ of 12 is probably too big to be useful.

3.2.3 Remapping

When implementing multimapping domain abstraction with HIDA*, we noticed that there is an issue about heuristic quality when searching with multiple goal states. In this setting, there is only one goal state at the base level. However, this goal state will be mapped to several abstract states in the first abstract level. The problem we are facing now is how to get effective heuristic values to guide search in the first abstract level. The only way we can guarantee an admissible heuristic value is to do the following. For any state s' in abstract level 1 (the abstract space immediately above base level), the heuristic for this abstract state is

$$h(s') = \min_{g'_i \in \phi(g)} d(\phi_1(s'), \phi_1(g'_i))$$

where $\phi_1()$ is a single abstraction mapping states from level 1 to next higher abstract level. We will use a single abstraction at this level and all levels above, the reason will be explained later. The heuristic generated this way for s' is actually very weak. Since we don't know which goal state is

closest to s', we have to consider all g'_i and as long as we want to maintain admissibility, we have to choose the smallest heuristic value. In an initial experiment, this issue degraded performance to a great extent, especially in large domains like the 15-Puzzle. The method we used to solve this is called "remapping". Simply put, we define $\phi_1(\phi_0(g))$ to be a single state. In this way, all g'_i are mapped to a single state at the second abstract level and we no longer need to choose the smallest value to maintain admissibility. In this way, we have

$$h(s') = d(\phi_1(s'), \phi_1(g'_i))$$

Here, different i will give us the same result since $\phi_1(g'_i)$ will be the same state no matter which i we choose. This actually imposes a constraint on the design of the abstraction, so that $\phi_1(\phi_0(g))$ is a single state rather than a set of states. We sacrifice this flexibility of design to get a boost in performance. To implement this technique, all we need to do is to design the abstraction with the property we described above. An example of multimapping domain abstraction with remapping is the following.

Example 10 *In the 8-Puzzle domain, the abstraction presented in Table 3.4 will have the remapping property. We can verify this property by applying $\phi_0()$ to any state s, getting two resulting states, and then applying $\phi_1()$ to the two resulting states, the result will be equivalent to applying abstraction $\phi_1(\phi_0(s))$. In other words, any state will be mapped to 2 different abstract state by $\phi_0()$, then these two states will be mapped to a single state by $\phi_1()$.*

s	0	1	2	3	4	5	6	7	8
$\phi_0^1(s)$	0	1	1	1	2	2	2	3	4
$\phi_0^2(s)$	0	1	2	2	1	1	2	3	4
$\phi_1(s)$	0	1	1	3	4	5	6	7	8
$\phi_1(\phi_0(s))$	0	1	1	1	1	1	1	3	4

Table 3.4: Multimapping domain abstractions used for remapping. ϕ_0 consists of two abstractions ϕ_0^1 and ϕ_0^2 while ϕ_1 is the abstraction at the next level.

The key point of remapping is to keep number of abstract goal states small in the abstract space above the first level. Keeping number of abstract goal states small is also the reason that we use a single abstraction at level 1 and higher. Using a multimapping abstraction to go from one abstract level to the next will make number of goal abstract states grow very quickly.

3.2.4 Combining Remapping and Goal Aggregation

Remapping and goal aggregation both involve restricting the design of an abstraction, and each technique brings different constraints on the design of the abstraction. For some granularities, if we want to generate a remapping abstraction, it is hard to also design it as a goal aggregation abstraction as well. In order to combine both techniques, we have to carefully choose the granularity. The following example shows how sometimes it is difficult to combine the two techniques together.

Example 11 *Suppose in the 9-Pancake Puzzle, we want to design a multimapping abstraction using both goal aggregation and remapping, with a granularity of* $\langle 4 \rangle$ *at the first level. Each level higher will have one more tile abstracted. The first step is to pick a domain abstraction and apply it to the goal state. In this example, we pick* ϕ_0^1 *as in Table 3.5, and the resulting abstract goal state* g' *would be* $\langle 1, 1, 1, 1, 2, 3, 4, 5, 6 \rangle$*. Next, we calculate two adjacent abstract states of* g'*, which are* $\langle 2, 1, 1, 1, 1, 3, 4, 5, 6 \rangle$ *and* $\langle 3, 2, 1, 1, 1, 1, 4, 5, 6 \rangle$*. With these three abstract states, we can design our goal aggregation multimapping abstraction by using the three abstractions that map the goal state to these three abstract states. This give us* ϕ_0^1, ϕ_0^2 *and* ϕ_0^3 *as in Table 3.5. However, if we also want to apply remapping, we need to map* $(1, 2, 3)$ *to the same tile at next level which makes the next level with a granularity at least of* $\langle 6 \rangle$*. This illustrates where goal aggregation and remapping have conflicts. It is impossible for us to make next higher abstract level with granularity of* $\langle 5 \rangle$ *and we need to sacrifice this flexibility of design to combine goal aggregation and remapping.*

s	0	1	2	3	4	5	6	7	8
$\phi_0^1(s)$	1	1	1	1	2	3	4	5	6
$\phi_0^2(s)$	2	1	1	1	1	3	4	5	6
$\phi_0^3(s)$	3	2	1	1	1	1	4	5	6

Table 3.5

3.3 Experiments

The experiments are split into four parts. First, we are going to investigate how much difference it will make if we calculate the heuristic value in forward fashion or in backward fashion. Second, we are going to investigate what is the best enhancement method for multimapping domain abstraction. After we determine this, we will implement this enhancement as the default for multimapping domain abstraction in the remaining experiments. Third, we are going to compare multimapping domain abstraction against a single domain abstraction and against multiple domain abstractions in small problem domains. This experiment with the small domains is designed to be very thorough so that we have a good understanding of the behaviors of each abstraction technique. Finally, we are going to test multimapping domain abstraction, single domain abstraction and multiple domain abstractions in large problem domains to verify the behaviors we found in small domains. The experiment in the large domains is small in scale because the computational cost is huge in large domains thus only limited experiments are feasible.

There are 4 small problem domains used in the Section 3.3.1, Section 3.3.2, Section 3.3.3. They are the 9-Pancake, (8,3)-Blocks World, (10,4)-Topspin and 8-Puzzle.The description of these four domains was presented in Section 2.3. For each small domain, we generated 500 random solvable start states as test cases. The large problem domains we use in the Section 3.3.4 are the 15-Puzzle,

Glued 15-Puzzle, 14-Pancake, (15,4)-Topspin and (12,3)-Blocks World. Except for the 15-Puzzle, we generated 100 random solvable start states as test cases for each large problem domain. For the 15-Puzzle, we used the 100 standard test cases [22].

We use Hierarchical IDA* as our search algorithm to implement with each abstraction. The abbreviations DA-HIDA*, MM-HIDA* and MA-HIDA*, are used to refer to HIDA* implemented with single domain abstraction, multimapping domain abstraction and multiple domain abstractions respectively. The difference between three methods is illustrated in Figure 3.2. For DA-HIDA*, every abstract level only consists of one abstract space and each state is mapped to just one state at the next higher level. For MA-HIDA*, each abstract level consists of more than one abstract space, each state in the original space is mapped to one state in each space at the first abstract level. After that, each abstract state in each abstract space is only mapped to one abstract state in one space at the next higher level. For MM-HIDA*, each state in the original space is mapped to multiple abstract states in the abstract space immediately above original space. Each abstract state is mapped to one abstract state at higher abstract level.

All the experiments are run on a computer with two AMD Opteron 250 (2.4GHz) CPUs and 8GB of memory.

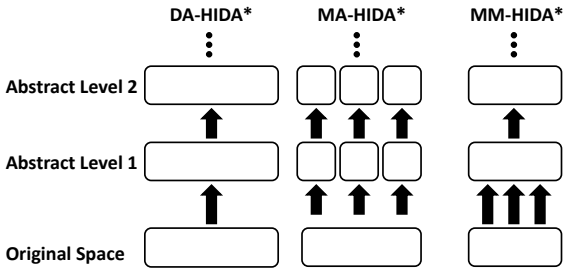


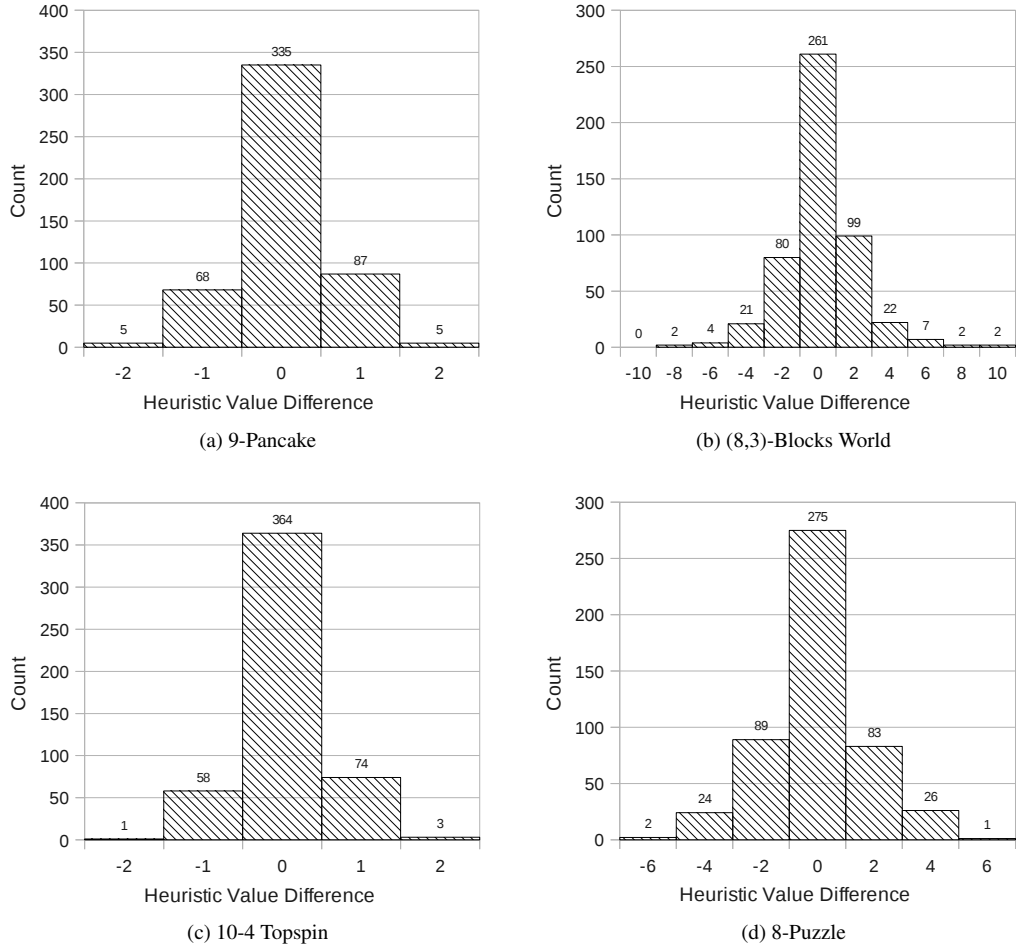
Figure 3.2: Differences between three HIDA* setups.

3.3.1 Difference Between the Forward and Backward Heuristic Calculation

In this section, we are going to investigate how heuristic values will be affected if we calculate them in the forward fashion or backward. As reported in Section 3.1, the heuristic values calculated in the forward fashion and backward fashion could be different, but we have no idea how different they will be. We have set up a small experiment to explore this issue. The results show that the majority of states in the tested problem domains will have the same heuristic value, and neither the forward fashion nor the backward fashion is a better choice than the other.

We have used the four small domains mentioned above, 9-Pancake, (8,3)-Blocks World, (10,4)-Topspin and 8-Puzzle, to investigate this issue. For each domain, we generated 500 random reachable states as test cases and randomly created a multimapping domain abstraction for each problem domain. The first level of all multimapping domain abstractions are designed to be $\langle 3, 3 \rangle$ with a mapping factor of 3. This is the only level that affects the heuristic value of states in the base level.

For each test case, we calculated the forward heuristic and the backward heuristic respectively, as $h_f(s)$ and $h_b(s)$. The histograms of $h_f(s) - h_b(s)$ are shown for each domain in Figure 3.3



(a) 9-Pancake

(b) (8,3)-Blocks World

(c) 10-4 Topspin

(d) 8-Puzzle

Figure 3.3: Heurisitc Value Difference Between the Forward and Backward Calculation.

From Figure 3.3, firstly, we can see that all four figures show the same trend. More than half of the test states in each domain have the same heuristic value no matter whether it is calculated in the forward fashion or in the backward fashion. Secondly, the histogram is generally symmetric around the value 0. This tells us that neither calculation in the forward fashion nor in the backward fashion will generate a superior heuristic to the other. However, it is technically hard to utilize this phenomenon by taking the maximum of $h_f(\cdot)$ and $h_b(\cdot)$. This is because when calculating in the backward fashion, the start states become the goal state and change constantly in the search process. Because of this, we have to calculate $h_b(\cdot)$ from scratch every time and this computational cost is obviously huge compared to the benefits it brings.

3.3.2 The Best Version of Multimapping

In this section, we are going to find out what is the best enhancement technique for multimapping domain abstraction. Experiments are designed first to compare the enhanced multimapping domain abstraction against plain multimapping domain abstraction in order to find out how much benefits of CPU time and memory usage we can get from these enhancement techniques. Second, the enhancement techniques are compared against each other and the results show us that combining goal aggregation and remapping is the best enhancement. This enhancement is implemented as the default technique for multimapping domain abstraction in the subsequent experiment sections.

In this section, we also used 4 domains, 8-Puzzle, 9-Pancake, (10,4)-Topspin and (8,3)-Blocks World, and we generated 500 random start states for each domain, . The multimapping domain abstraction we used is a 4-level abstraction. The abstraction in the first level is a $\langle 3, 3\rangle$ multimapping domain abstraction[1] with a mapping factor of 3. In level 2, the two tiles representing the abstracted tiles from the base level are mapped to the same tile so the granularity of level 2 is $\langle 6\rangle$. In level 3 and 4, an additional tile is added to the abstracted tiles in each of these level so the granularity of levels 3 and 4 are $\langle 7\rangle$ and $\langle 8\rangle$. An example of this multimapping domain abstraction is presented in Table 3.6.

s	0	1	2	3	4	5	6	7	8
$\phi_1^1(s)$		1	3	4	2	2	2	1	1
$\phi_1^2(s)$		3	1	2	1	1	2	4	2
$\phi_1^3(s)$		3	1	1	2	4	2	2	1
$\phi_2(s)$			1						
$\phi_3(s)$			1	1					
$\phi_4(s)$			1	1	1				

Table 3.6: $\phi_1^1(s)$, $\phi_1^2(s)$ and $\phi_1^3(s)$ are three abstractions at the first level, $\phi_2(s)$, $\phi_3(s)$ and $\phi_4(s)$ are abstractions at second, third and forth level respectively.

We designed four versions of multimapping domain abstractions for each domain. (1) Plain, (2) remapping, (3) goal aggregation and (4) remapping with goal aggregation. The number of abstractions of each kind is presented in Table 3.7. For the (8,3)-Blocks World, we did not implement goal aggregation, thus only two kinds of abstraction are used. For plain multimapping (MMP), we generate each by randomly choosing 3 different $\langle 3, 3\rangle$ domain abstractions and using them together. remapping (MMRM) is generated in a similar way, but we require the 6 tiles that will be abstracted at level 1 to be the same for all three domain abstractions, namely $\phi_1^1(s)$, $\phi_1^2(s)$ and $\phi_1^3(s)$, at the first level. For generating goal aggregation multimappings (MMGA), we first pick a random domain abstraction ϕ_1^1, apply it to the goal state to get an abstract state s'_{g1}. Second, we calculate two closest abstract states of s'_{g1}, called s'_{g2} and s'_{g3}, that the goal can be mapped to. Third, we calculate the two domain abstractions ϕ_1^2 and ϕ_1^3 such that $\phi_1^2(g) = s'_{g2}$ and $\phi_1^3(g) = s'_{g3}$. Using ϕ_1^1, ϕ_1^2 and ϕ_1^3 we will

[1]For 8-Puzzle, (10,4)-Topspin and (8,3)-Blocks World only tiles 1 to 8 are abstracted. For 9-Pancake, tiles 0 to 8 are abstracted.

	8-Puzzle	9-Pancake	(10,4)- Topspin	(8,3)-Blocks World
MMP	280	840	280	280
MMRM	280	840	280	280
MMGA	180	772	267	N/A
MMGARM	100	68	13	N/A

Table 3.7: Number of Abstractions for Each Domain and Enhancement Techniques.

get a goal aggregation multimapping. The combined remapping and goal aggregation (MMGARM) are generated by inspecting each MMGA abstractions to see if the 6 tiles that will be abstracted at level 1 are the same for all three abstractions; if so we put it into the MMGARM category and remove them from the MMGA category. For this reason, MMGA and MMGARM always add up to the number of 280 (840 for the 9-Pancake). In the following experiments, we are going to compare techniques in terms of (1) number of nodes expanded at the base level, (2) memory usage[2] and (3) CPU time (in seconds).

MMRM Against MMP

First, we are going to compare remapping and plain multimapping. The results are presented in Figure 3.4, Figure 3.5 and Figure 3.6, starting at page 40. For each data point, the x-coordinate represents the average performance measurement for a particular test case (start state) averaged across all the MMRM abstractions and the y-coordinate represents the analogous average for plain multimapping. Points below the diagonal mean that plain multimapping outperforms the MMRM abstraction on that start state and points above the diagonal mean the opposite.

From these results, we can see that the remapping outperforms the plain multimapping domain abstraction in terms of CPU and memory (Figure 3.6 and Figure 3.5) but loses in terms of the number of nodes expanded at the base level (Figure 3.4). The poorer performance in nodes expanded at the base level is an indicator of a decrease of heuristic value quality because of remapping. This is brought about by the constraints of constructing remapping abstractions. Plain multimapping domain abstraction are constructed by randomly selecting $\langle 3, 3 \rangle$ domain abstractions and using them together. When constructing remapping abstractions we cannot freely choose domain abstractions, thus we could miss some good abstraction combinations.

MMGA Compared to MMP

Next, we compared goal aggregation to plain multimapping. The results are presented in Figure 3.7, Figure 3.8 and Figure 3.9 (page 41), which present the results in the same manner as Figure 3.4, Figure 3.5 and Figure 3.6.

This experiment is only done in three domains because goal aggregation was not implemented in the (8,3)-Blocks World domain for the reason stated in Section 3.2.2.

[2]In this thesis, memory usage is measured in terms of the number of entries cached in memory.

From these results, we can see that goal aggregation is slightly better than plain multimapping in terms of memory usage and CPU time (Figure 3.9 and Figure 3.8). However, goal aggregation does not generate the better heuristic values we expected. The nodes expanded at the base level are almost the same as plain multimapping in the 9-Pancake and (10,4)-Topspin and slightly worse in the 8-Puzzle (Figure 3.7). The decrease of heuristic value quality can be explained by two reasons. First, constructing goal aggregation abstraction also involves constraints on which abstractions can be used together, which may limit the quality of the heuristic. Second, besides the goal state, other states are also mapped to abstract states which are close to each other. For example, in 9-Pancake domain let $\langle 0, 1, 2, 3, 4, 5, 6, 7, 8\rangle$ be goal state and ϕ_1, ϕ_2 in Table 3.8 be two abstractions applied to it. The corresponding two abstract goal states are $\langle 1, 3, 2, 1, 2, 4, 2, 1, 8\rangle$ and $\langle 1, 2, 3, 1, 2, 4, 2, 1, 8\rangle$ which are only one step away. However, if we apply ϕ_1 and ϕ_2 to $\langle 3, 1, 2, 0, 4, 5, 6, 7, 8\rangle$, we will get the same two abstract states which are also one step away. Because start states could also be mapped to adjacent abstract states, the fundamental advantage of multimapping is compromised, resulting in degraded heuristic value quality.

s	0	1	2	3	4	5	6	7	8
$\phi_1(s)$	1	3	2	1	2	4	2	1	8
$\phi_2(s)$	1	2	3	1	2	4	2	1	8

Table 3.8: Domain abstractions ϕ_1 and ϕ_2.

Comparison of MMRM, MMGA and MMGARM

Next, the comparison is made between remapping, goal aggregation and remapping with goal aggregation. For the (8,3)-Blocks World domain, we only have remapping so this comparison is only made in the 8-Puzzle, 9-Pancake and (10,4)-Topspin domains. The results are presented in Figure 3.10, Figure 3.11 and Figure 3.12 (page 42). These figures are presented using box plots. For each problem domain there are three box plots represents the three enhancement techniques. The first one is for remapping (MMRM) and the second and the third are for goal aggregation (MMGA) and remapping with goal aggregation (MMGARM) respectively. A data point in the figures represents the average performance of a particular abstraction over all the test cases (start states). The box contains the data points of exactly half of the abstractions for the corresponding technique. The horizontal line inside the box shows the median performance. The bottom line and the top line represent 75^{th} and 25^{th} percentiles respectively. The vertical line below the box extends to the best performance or to 1.5 times the interquartile range IQR, whichever is larger. If there are performance values beyond $1.5 \times IQR$, they are plotted as individual points. The vertical line above the box is analogous.

According to the results of the experiments, MMGARM beats the other two techniques in terms of both memory usage and CPU time in all three domains (Figure 3.11 and Figure 3.12), which

makes MMGARM the most desirable technique. MMRM technique is the second desirable technique because it beats MMGA in terms of memory usage and CPU time in all three domains. In the following experiment section, we will use MMGARM as the default version of multimapping domain abstraction.

3.3.3 Comparison With Other Methods Using Small State Spaces

In this section, we are going to compare the MMGARM version of MM-HIDA* to DA-HIDA* and MA-HIDA* in an extensive way in small problem domains. We like to use small domains because it is practical to do large amount of experiments in these domains to obtain statistically persuasive results. The domains we used in this experiment are the 8-Puzzle, 9-Pancake, (10,4)-Topspin and (8,3)-Blocks World. We also used the same 500 test cases for each domain as in Section 3.3.2. For the abstraction design, we used the same granularity and 4-level design as described in Section 3.3.2. For MM-HIDA*, we used the remapping goal aggregation technique for the 8-Puzzle, 9-Pancake and (10,4)-Topspin. For (8,3)-Blocks World domain, we only used remapping. The performance measurements we used are the same as we used previously, namely: number of nodes expanded at the base level, CPU time and memory usage. The total number of nodes expanded is usually an important performance measurement used in numerous papers, but in our experiment, we found that this indicator is always consistent with CPU time. Therefore, for the purpose of having experiment results succinct, we decided to only present CPU time. This experiment is split into three parts. First, we will compare MM-HIDA* to DA-HIDA*. Next, MM-HIDA* is compared to MA-HIDA*. Finally, we compare all three techniques based on a different perspective. All results show us that MM-HIDA* is the best technique in terms of CPU time and memory usage.

Comparison With A Single Domain Abstraction

First, we are going to compare MM-HIDA* to DA-HIDA*.

For DA-HIDA*, we generated all 280 abstractions which abstract tiles 1 to 8 on the first level. This set of 280 abstractions are used on the 8-Puzzle, (10,4)-Topspin and (8,3)-Blocks World. For 9-Pancake, we need to abstract all 9 tiles (0 to 8), so the complete set consists of 840 abstractions.

In Figure 3.13 (page 43), the number of nodes expanded at the base level is presented. For every test case, we calculate the average number of nodes expanded across all the abstractions of each type, so each point in the plot is an average across 280 abstractions (840 for the 9-Pancake) and there are 500 data points in each plot. From these results, we can see that multimapping expanded fewer nodes at the base level than domain abstraction in all domains. This supports our claim that multimapping abstraction provides better heuristics than domain abstraction given that the size of abstract space is the same.

The comparison of memory usage between MM-HIDA* and DA-HIDA* is presented in Figure 3.14, the measurement unit in these figures is the number of nodes cached during the HIDA*

search process. MM-HIDA* has lower memory use in all four domains. In the 9-Pancake and (10,4)-Topspin, we notice that the plot converges to a point where DA-HIDA* and MM-HIDA* have the same memory consumption. This is because harder cases will have the abstract space thoroughly explored for these two particular domains, thus DA-HIDA* and MM-HIDA* will use the same amount of memory.

CPU time is presented in Figures 3.15. We see that MM-HIDA* is faster than DA-HIDA* in all four domains. Taking all these results together, we can see that MM-HIDA* beats DA-HIDA* in terms of heuristic quality, memory usage and CPU time.

Comparison With Multiple Domain Abstractions

In this section, we are going to compare MM-HIDA* to MA-HIDA*. The problem domains, test cases and measurements of performance are the same as in the previous section.

The abstractions we used for MM-HIDA* are the same we used in the previous experiments. For MA-HIDA*, the abstractions are also 4-level and are made from MMGARM abstractions by treating the three abstractions at the first level of MMGARM as separate domain abstractions. We generated the same number of abstractions as MMGARM for the 8-Puzzle, (10,4)-Topspin and 9-Pancake. For the (8,3)-Blocks World, the abstractions are generated in the same fashion as for MMRM. The number of nodes expanded at the base level is presented in Figure 3.16 (page 44). From these results, we can see that MA-HIDA* generates fewer nodes at the base level than MM-HIDA*. This result is what we expected because MA-HIDA* does not take the minimum over a set of abstract goal states.

In terms of memory usage, the experiment results are presented in Figure 3.17. These results all show the same trend, MA-HIDA* always consumes more memory than MM-HIDA*. This is also what we expected because the size of the abstract space for MA-HIDA* is 3 times bigger than MM-HIDA* and both abstract spaces tend to be totally explored when the test cases become harder.

For overall CPU time, the results are presented in Figure 3.18. As with memory usage, MM-HIDA* has better CPU time than MA-HIDA* in all domains.

Final Comparison

In this section, we will present our experiment results in another way to see how MM-HIDA* compares to the other two methods. We compute the average number of nodes expanded at the base level, memory usage and CPU time for each abstraction and plot abstractions according to method category. Each point in each figures is an average over 500 test cases. The results are presented using the same type of box plots as in Figure 3.10 to 3.12

The average number of nodes expanded at the base level is presented in Figure 3.19 (page 45). In all four domains, multiple abstractions achieved the least number of nodes expanded at the base level while domain abstraction always expanded more nodes at the base level than the other two methods. Multimapping tends to be in the middle of the two. Since multiple abstractions used an

abstract space 3 times larger than the other two methods, it is no surprise that it has better heuristic quality and expanded fewer number of nodes at the base level.

The memory usage of each method is presented in Figure 3.20. Since multiple abstractions uses abstract spaces 3 times bigger than the other two methods, it is no surprise that it uses more memory than the other two. Even though the size of abstract space of multimapping and domain abstraction are the same, multimapping tends to use less memory than domain abstraction.

CPU time results are presented in Figure 3.21. They resemble the memory usage results. From these results, we can see that multimapping abstractions have the best CPU time in all four domains.

Overall, multimapping domain abstraction has the best CPU time and memory usage among the three methods.

3.3.4 Experiment with Large State Spaces

In this section, we will run a set of experiments in large state spaces in order to verify the conclusions we made using small spaces.

The large domains we have chosen are the 15-Puzzle, the Glued 15-Puzzle (the tile in the second column, second row from bottom cannot be moved from its goal location), the 14-Pancake Puzzle, (15,4)-TopSpin and (12,3)-Blocks World. For the 15-Puzzle, we used the standard 100 start states [22] in our experiment. For all other domains, we generated 100 random start states as test cases in our experiments. Because the size of these spaces is very large, it is not practical to have extensive experiments like we did in the small spaces. Instead, we hand-generated 5 abstractions for each of DA-HIDA*, MM-HIDA* and MA-HIDA*. First, we made 5 abstractions for DA-HIDA* for each domain. For MM-HIDA*, we used mapping factor of 3 and we made the size of the abstract space equal to that of DA-HIDA*. In order to do this, we designed abstractions for MM-HIDA* to be the same granularity as the abstractions for DA-HIDA*. MM-HIDA* is designed to be related to DA-HIDA*. Each abstraction of MM-HIDA* is made with one abstraction from the DA-HIDA* together with two abstractions determined by goal aggregation and remapping. In some situations, we cannot make two abstractions from the abstraction from DA-HIDA* because of the conflicts explained in Section 3.2.4. In this case, we generate MM-HIDA* from a new abstraction. For MA-HIDA*, we used the same abstractions as MM-HIDA* except for the 15-Puzzle where we used coarser-grained abstractions because of memory limitations. Therefore the total memory usage of MA-HIDA* should be substantially larger than MM-HIDA* and DA-HIDA*. The abstractions we used for each domain and abstraction method are presented in Appendix A. For multimapping, we used remapping and goal aggregation for all domains except the (12,3)-Blocks World. For (12,3)-Blocks World, we only used remapping. We will investigate goal aggregation in the (12,3)-Blocks in the next section. Since the number of abstractions is quite small, we cannot make conclusions statistically. These results are only to verify conclusions we made in the previous sections. Results are presented in Tables 3.9 to 3.13. In each table, there are three columns on the right which de-

scribe the performance measurements. Nodes represents the number of nodes expanded at the base level. CPU and Mem represent average CPU time in seconds and average memory consumption as entries cached in memory, respectively. The left wide column describes which kind of abstractions the experiment used. There are 5 abstractions of each kind and they are all numbered. For MM and MA abstractions, if there is an asterisk beside the number that means this MM abstraction is made from corresponding DA abstraction. For MA abstractions if there is an asterisk beside the number that means this MA used the same abstractions as corresponding MM. All results are sorted according to average CPU time.

DA	MM	MA	Nodes	CPU(s)	Mem ($\times 10^7$)
	1*		3,669,519	768.2	2.784
	2*		1,667,888	782.0	2.758
	5*		7,945,182	798.1	2.802
1			20,571,539	806.4	2.758
2			5,545,817	914.6	2.687
5			24,547,168	1,029.4	2.753
	3*		2,910,399	1,040.6	4.430
		1	552,289	1,224.7	3.673
3			54,322,104	1,253.1	4.453
		5	906,147	1,344.0	4.297
		2	789,905	1,360.4	4.295
		4	1,007,468	1,557.2	5.092
		3	926,771	1,616.2	4.435
	4*		1,178,024	1,681.8	5.978
4			16,793,560	1,909.7	5.834

Table 3.9: The 15-Puzzle.

DA	MM	MA	Nodes	CPU(s)	Mem ($\times 10^6$)
	1*		78,519	54.2	2.176
1			1,172,133	58.4	2.143
	4*		3,104,027	60.3	1.529
	2*		3,777,306	65.4	2.163
3			4,247,013	73.8	2.287
	3*		345,574	73.9	2.474
2			9,108,588	74.2	2.087
5			23,194,543	98.5	2.144
4			20,474,809	103.5	1.469
	5*		12,363,685	104.2	2.189
		1*	29,862	112.8	4.196
		4*	1,405,176	114.3	3.269
		2*	1,502,349	115.2	4.307
		5*	4,333,375	131.3	4.430
		3*	108,947	134.9	4.687

Table 3.10: The Glued 15-Puzzle.

From these results, we can see that MM-HIDA* takes the best ranking in all 5 domains. In 15-4 Topspin and 14-Pancake, MM-HIDA* takes the top 5 rankings. In the 15-Puzzle, MM-HIDA* takes the top 3 positions. In Glued 15-Puzzle, MM-HIDA* takes 4 out of 6 top positions. In (12,3)-

DA	MM	MA	Nodes	CPU(s)	Mem ($\times 10^6$)
	5*		587,931	284.6	7.646
	3*		454,853	293.3	7.976
	4*		480,962	304.8	7.934
	2*		217,028	312.2	7.887
	1*		206,675	322.1	7.758
2			1,369,956	397.9	8.892
1			1,176,908	401.2	8.778
4			2,861,843	507.0	11.745
		5*	118,043	525.1	13.870
3			1,818,312	531.6	11.780
		4*	114,719	535.1	14.197
		2*	79,196	540.1	13.837
		1*	46,751	565.3	13.849
		3*	102,077	593.6	14.350
5			1,253,427	635.1	11.963

Table 3.11: The 14-Pancake Puzzle.

DA	MM	MA	Nodes	CPU(s)	Mem ($\times 10^6$)
	2*		11,870	143.5	5.390
	1*		12,312	147.6	5.433
	4*		12,471	150.7	5.387
	3*		12,849	153.8	5.328
	5*		13,498	160.8	5.456
4			47,907	183.3	6.649
2			52,970	186.8	6.676
3			47,905	197.2	6.604
1			43,576	199.5	6.720
5			47,959	217.5	6.651
		2*	3,506	279.7	9.838
		4*	3,541	286.0	9.783
		1*	3,338	300.5	9.874
		3*	3,458	301.5	9.783
		5*	3,759	309.8	10.018

Table 3.12: The (15,4)-TopSpin Puzzle.

Blocks World domain, MM-HIDA* does not have as big an advantage over other abstractions as it does in the other domains.

In terms of memory usage, it is almost the same trend as CPU time. This is because, in general, the longer HIDA* runs, the more memory it consumes, so lower CPU time means the algorithm is likely to use less memory. From the results presented, we can see that multimapping also has a big advantage in memory usage.

In conclusion, the results in the large domain verified the performance advantage revealed in the small domains. It shows that multimapping domain abstraction takes less time and memory than single domain abstraction and multiple domain abstractions.

DA	MM	MA	Nodes	CPU(s)	Mem ($\times 10^6$)
2			43,896	77.4	2.062
3			589,399	106.4	2.464
	2*		6,202,221	123.1	2.117
		2*	2,109	126.3	3.506
	4*		1,065,182	131.8	3.514
	1*		3,138,661	138.1	3.033
1			6,310,237	143.7	2.930
5			6,271,341	147.6	3.531
	3*		7,647,763	166.4	2.538
		3*	22,149	187.4	4.441
4			18,038,164	194.9	3.510
		1*	5,214	204.1	5.295
	5*		13,099,419	218.9	3.648
		4*	46,776	222.8	6.567
		5*	48,895	229.3	6.690

Table 3.13: The (12,3)-Blocks World.

Apply GA Abstraction on Blocks World Domain

We mentioned that we do not apply GA abstraction on Blocks World domain because the distances between abstract goal states are too far away compared to the average heuristic value, as discussed in Section 3.2.2. However, because the (12,3)-Blocks World is much larger, implementing this technique could be beneficial. In this experiment, we generated 5 MMGARM abstractions[3] for the (12,3)-Blocks World. These 5 abstractions are generated from scratch and not from any domain abstraction. We compared MMGARM with DA, MM (remapping version) and MA and the experiment results are presented in Table 3.14. The results show us that MMGARM performs better than the other three methods in terms of CPU time. But the advantage of MMGARM is less obvious than it is in the other domains.

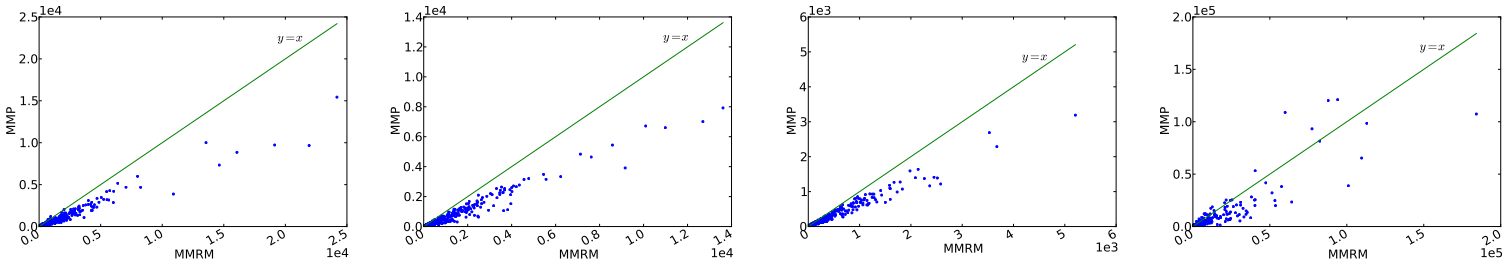
3.3.5 Conclusions

From the experiments in this chapter, we have understood some properties of multimapping domain abstraction. Firstly, in terms of heuristic quality, this technique could provide better heuristic values then a single domain abstraction. This is achieved with the same size of abstract space. Secondly, for HIDA* equipped with multimapping domain abstraction, the experiment results show that MM-HIDA* outperformed both DA-HIDA* and MA-HIDA* in terms of CPU time and memory usage.

[3]The MMGARM abstractions for (12,3)-Blocks World are described in Appendix A.

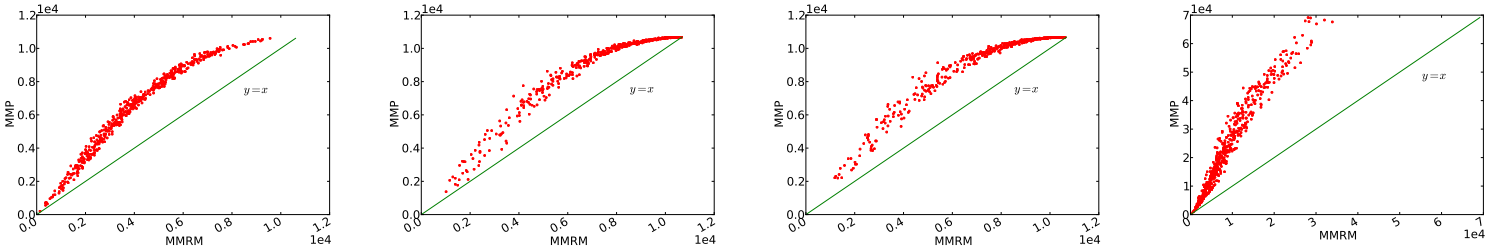
DA	MM	MA	MMGARM	Nodes	CPU(s)	Mem ($\times 10^6$)
			1	67,970	60.3	1.308
2				43,896	77.4	2.062
			5	521,272	89.3	1.945
3				589,399	106.4	2.464
			3	506,178	117.1	2.566
	2*			6,202,221	123.1	2.117
		2*		2,109	126.3	3.506
			2	316,069	128.1	3.987
	4*			1,065,182	131.8	3.514
			4	232,302	132.0	3.550
	1*			3,138,661	138.1	3.033
1				6,310,237	143.7	2.930
5				6,271,341	147.6	3.531
	3*			7,647,763	166.4	2.538
		3*		22,149	187.4	4.441
4				18,038,164	194.9	3.510
		1*		5,214	204.1	5.295
	5*			13,099,419	218.9	3.648
		4*		46,776	222.8	6.567
		5*		48,895	229.3	6.690

Table 3.14: The (12,3)-Blocks World with the MMGARM technique.



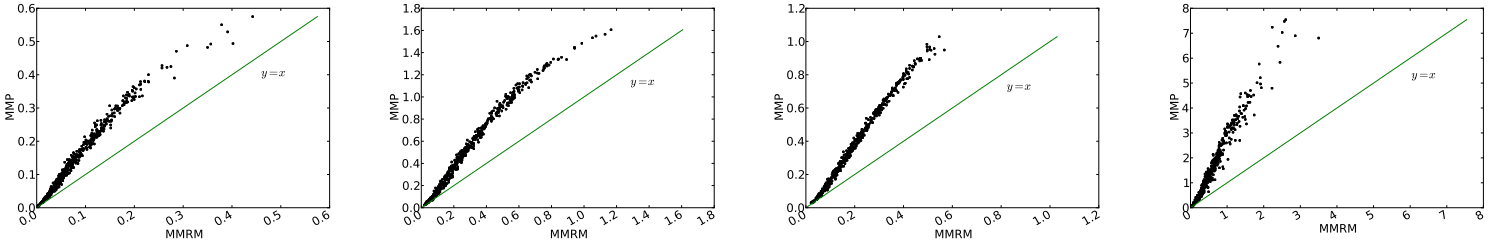
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.4: Nodes expanded at the base level: multimapping domain abstraction (remapping) Vs. multimapping domain abstraction (plain).



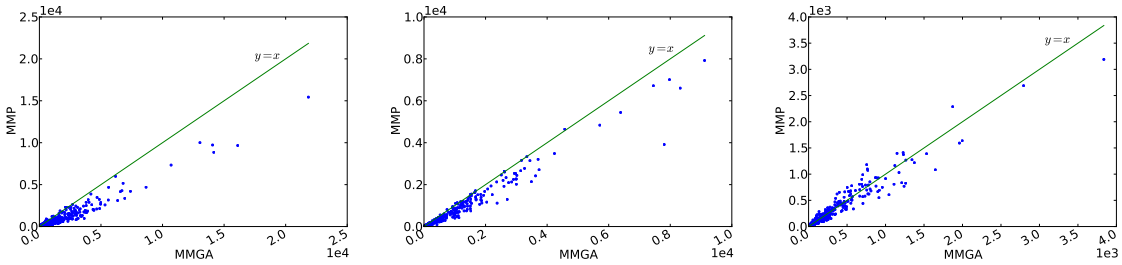
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.5: Memory usage: multimapping domain abstraction (remapping) Vs. multimapping domain abstraction (plain).



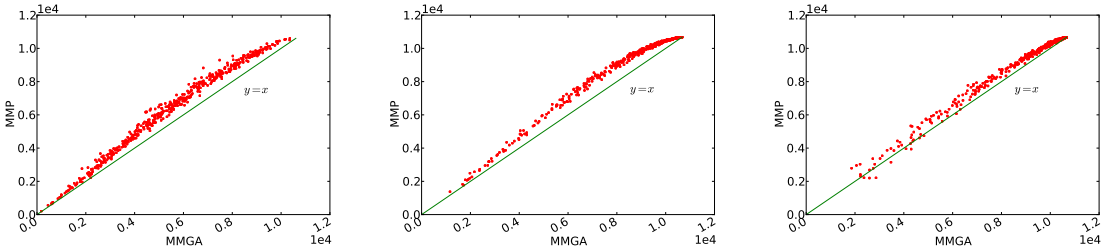
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.6: CPU time: multimapping domain abstraction (remapping) Vs. multimapping domain abstraction (plain).



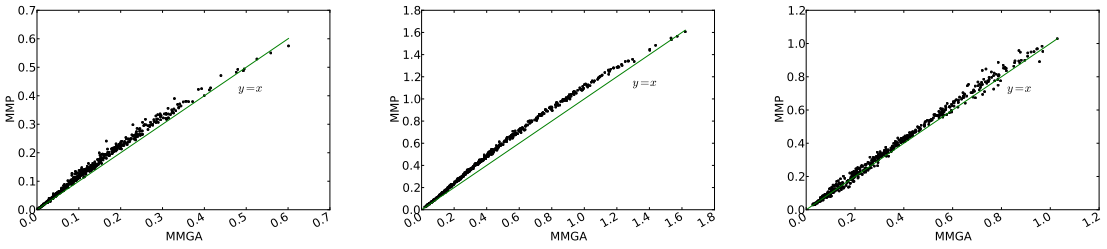
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.7: Nodes expanded at the base level: multimapping abstraction (goal aggregation) vs. multimapping abstraction (plain).



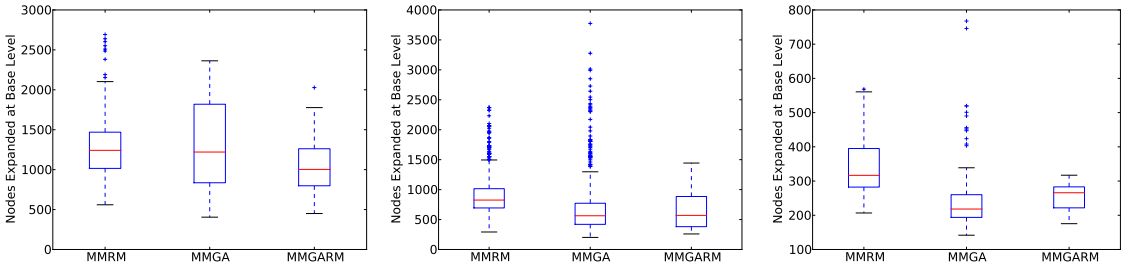
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.8: Memory usage: multimapping abstraction (goal aggregation) vs. multimapping abstraction (plain).



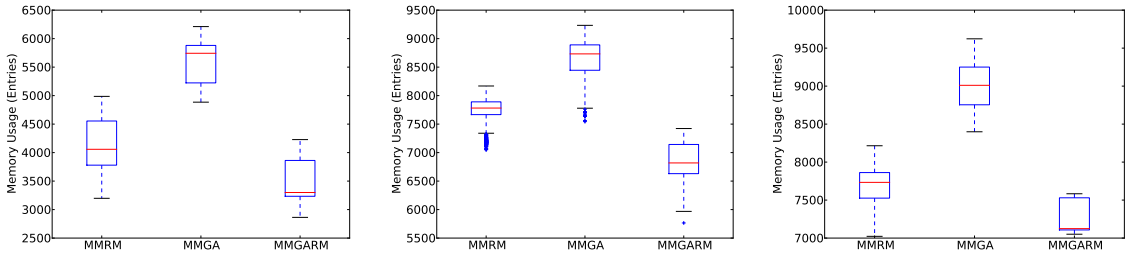
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.9: CPU time: multimapping abstraction (goal aggregation) vs. multimapping abstraction (plain).



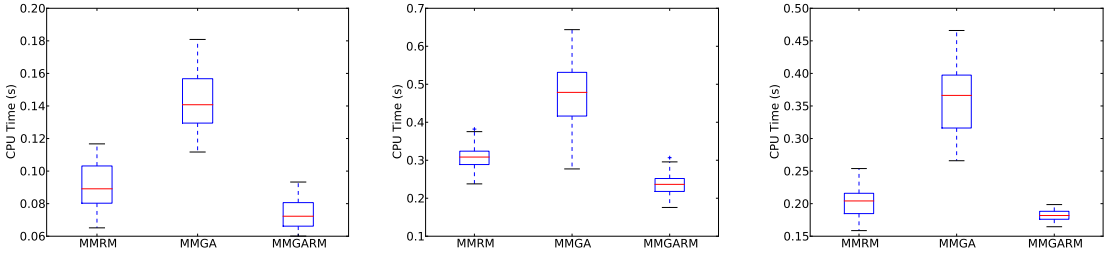
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.10: Nodes expanded at base level: abstraction comparison between MMRM, MMGA and MMGARM.



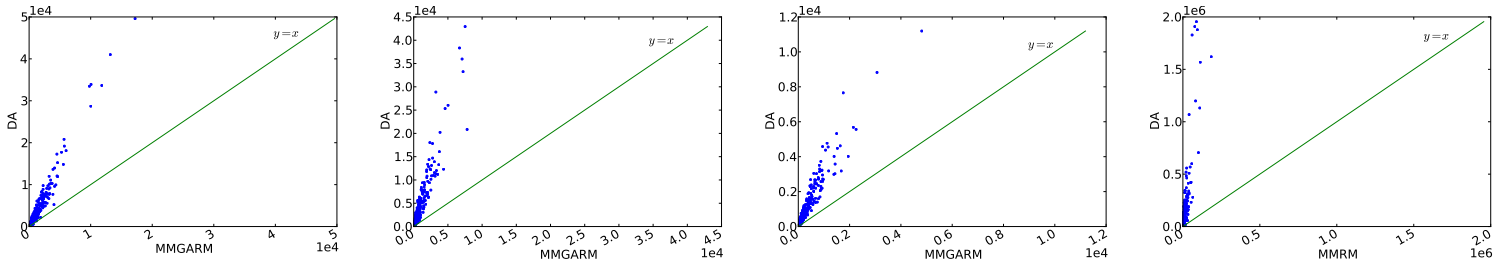
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.11: Memory usage: abstraction comparison between MMRM, MMGA and MMGARM.



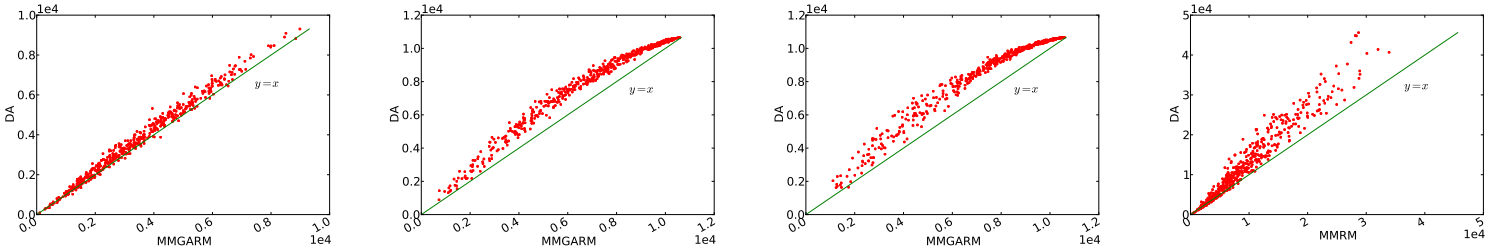
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin

Figure 3.12: CPU time: abstraction comparison between MMRM, MMGA and MMGARM.



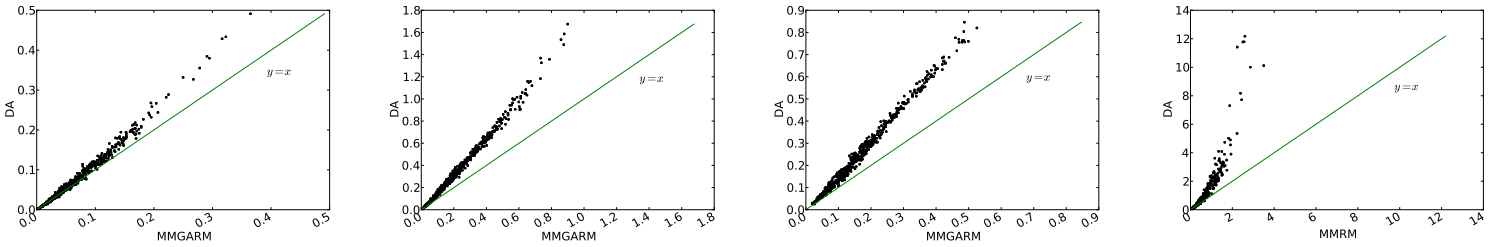
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.13: Nodes expanded at the base level: multimapping abstraction (GARM) vs. domain abstraction.



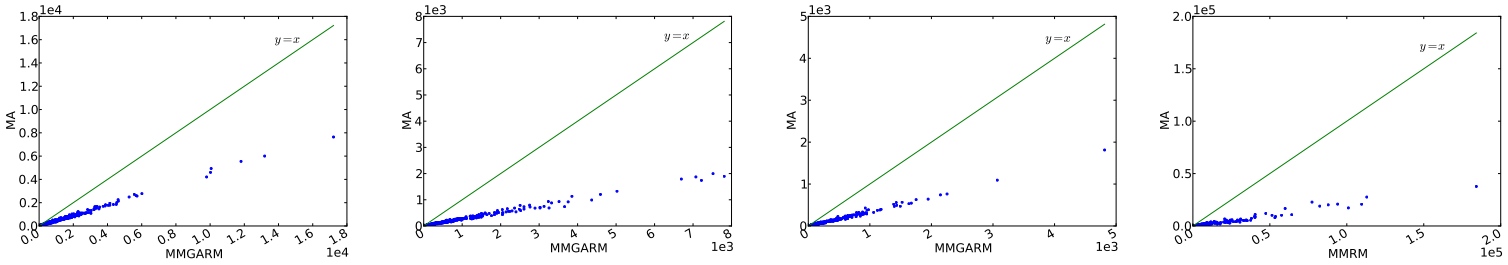
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.14: Memory usage: multimapping abstraction (GARM) vs. domain abstraction.



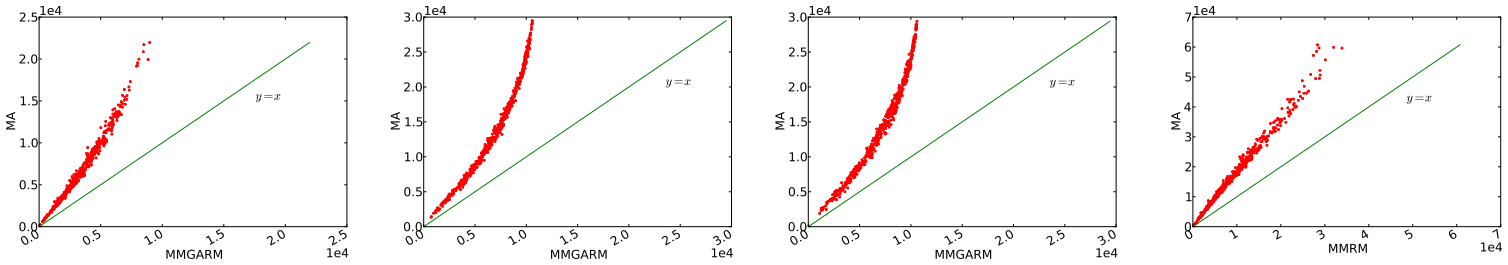
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.15: CPU time: multimapping abstraction (GARM) vs. domain abstraction.



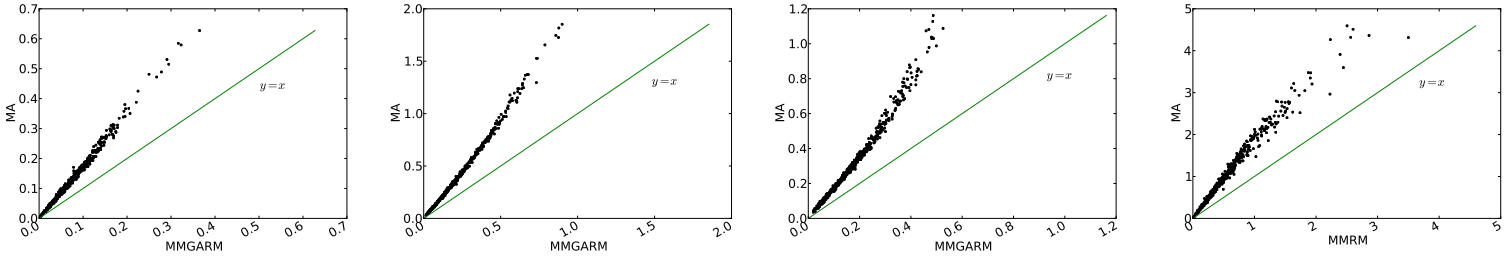
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.16: Nodes Expanded at the base level: Multimapping Abstraction (GARM) vs. Multimapping Abstraction (MA).



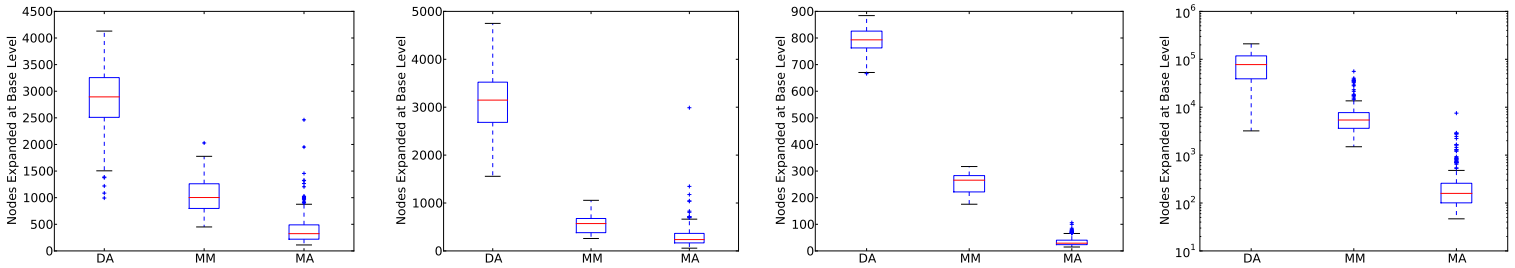
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.17: Memory Usage: Multimapping Abstraction (GARM) vs. Multimapping Abstraction (MA).



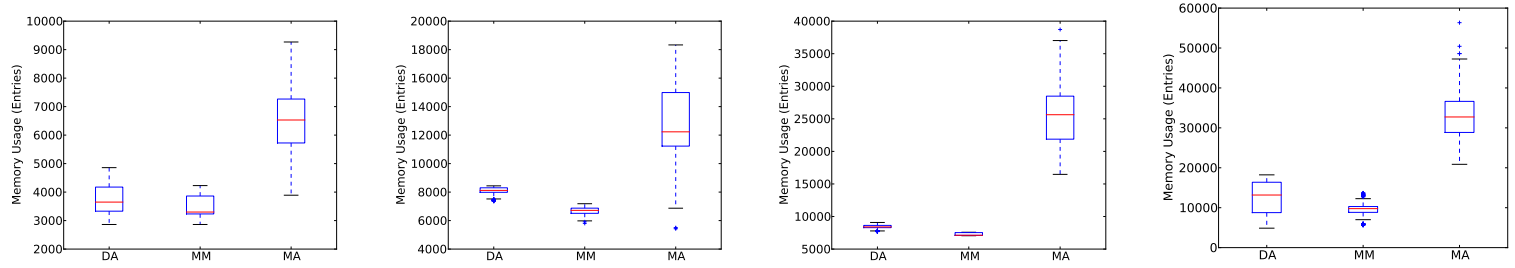
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.18: CPU Time: Multimapping Abstraction (GARM) vs. Multimapping Abstraction (MA).



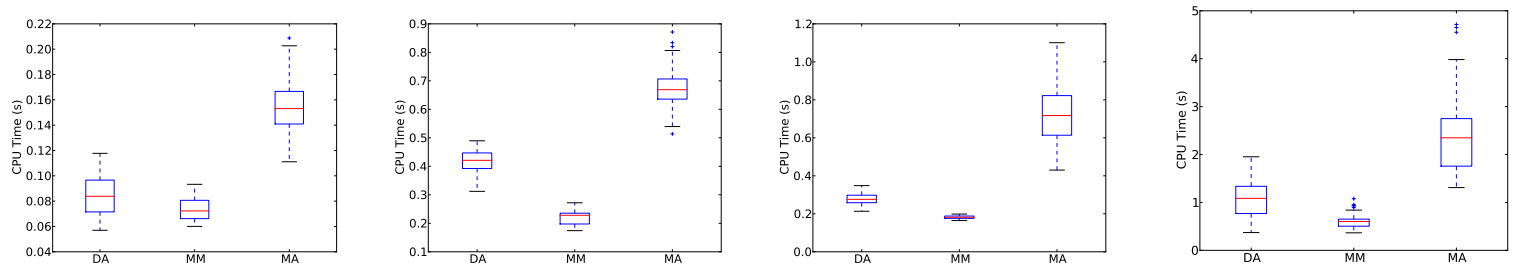
(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.19: Nodes Expanded at the base level: comparison between DA, MM and MA.



(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.20: Memory Usage: comparison between DA, MM and MA.



(a) 8-Puzzle (b) 9-Pancake (c) (10,4)-Topspin (d) (8,3)-Blocks World

Figure 3.21: CPU Time: comparison between DA, MM and MA.

3.4 Implementation with Partial-State Abstraction

In this section, we are going to introduce a new technique, called Partial-State abstraction, to represent abstract states. The reason it is called Partial-State abstraction is that only a part of the state is specified. The unspecified part of the state are variables which make the Partial-State represent a set of states rather than a specific one. For Partial-State abstraction, we developed corresponding operators and rules for operator matching and goal testing. The system altogether is a state representation and abstraction technique. However, this method is still a premature method with significant drawbacks. We have only performed limited experiments on this technique. This method inspired us and led us to develop a theory about manipulating states as a single state-set and multimapping in general, which will be presented in Chapter 4.

3.4.1 Definition

Partial-State

We continue to use the PSVN representation of a state. A state is a vector $\langle D_1, D_2, \ldots, D_k \rangle$ where $k \in \mathbb{N}$, each D_i is a finite set called a "domain". A Partial-State is a vector $\langle V_1, V_2, \ldots, V_k \rangle$ where each V_i is drawn from $D_i \cup X$. X here is a set of variable symbols distinct from the values in all the D_i. A Partial-State will represent a set of states in the following way. A state s is contained in a Partial-State s^p iff $\forall i$, $s_i^p \in D_i \Longrightarrow s_i^p = s_i$, and we denote this as $s \in s^p$.

Example 12 *We define the following states and Partial-States:*

$$\begin{aligned} s_1 &= \langle 1, 2, 3, 4 \rangle \\ s_2 &= \langle 1, 3, 2, 4 \rangle \\ s^p &= \langle 1, 2, x_1, x_2 \rangle \end{aligned}$$

In this example, we have $s_1 \in s^p$ and $s_2 \notin s^p$.

Multiple occurrences of variable x in a partial state carries a special meaning. In the above example, x_1 and x_2 do not exert any constraints on s^p. If we define s^p as $\langle 1, 2, x_1, x_1 \rangle$ then the third and fourth state values are required to be the same. This is a feature of Partial-State that enhances its representation power.

Because the Partial-State has the power to represent a set of states, it can be considered as an abstract state and this is the foundation of our new technique. The way we abstract an ordinary state is to turn values in the state into variables from X. In this way, the Partial-State we make will contain the original state plus a set of other states. We describe the abstraction function with a vector $\langle m_1, m_2, \ldots, m_{|D|} \rangle$ where $D = \cup_{i=1}^{k} D_i$ and m_i specifies how many occurrences of value d_i are to be replaced by a variable symbol. Assuming the number of occurrences of d_i is n_i in the

original state, an abstraction function is legal only if $0 \leq m_i \leq n_i$. If $0 < m_i < n_i$, we are facing a situation of choosing m_i out of the n_i occurrences and that is where multimapping comes from for this technique.

Example 13 *Let $D = \{1, 2, 3\}$, state s be $\langle 1, 1, 1, 2, 3, 3\rangle$ and the partial-state abstraction be $\langle 2, 0, 1\rangle$. There are three ways to choose two of the three 1's and two ways to choose one of the two 3's, so this abstraction maps $\langle 1, 1, 1, 2, 3, 3\rangle$ to six different abstract states ($\langle 1, x_1, x_2, 2, 3, x_3\rangle$, $\langle x_1, 1, x_2, 2, 3, x_3\rangle$, $\langle x_1, x_2, 1, 2, 3, x_3\rangle$, $\langle 1, x_1, x_2, 2, x_3, 3\rangle$, $\langle x_1, 1, x_2, 2, x_3, 3\rangle$, $\langle x_1, x_2, 1, 2, x_3, 3\rangle$). By construction, all of them contain the state $\langle 1, 1, 1, 2, 3, 3\rangle$.*

An ordinary state in PSVN notation can be viewed as a special Partial-State which only covers one state. In the following discussion, anything applicable to a Partial-State also is applicable to an ordinary state.

Abstraction using Partial-State

In order to create a multimapping abstraction with the Partial-State technique, it is required that there are duplicate constants in a state vector. But this is not a common case for most problem domains. Our solution is to use domain abstraction on states at first to create duplicated constants and then apply the Partial-State technique. We also use granularity to describe abstraction using the Partial-State. The definition of the granularity of the Partial-State is given below:

Definition 7 (Partial-State Granularity) *The granularity of the Partial-State is defined as $\langle n\rangle$ where n indicates how many constants in the original state are mapped to variables.*

An example of abstraction using domain abstraction and Partial-State is presented as follows.

Example 14 *Given a state $\langle 1, 2, 3, 4\rangle$ with $D = \{1, 2, 3, 4\}$, we want to generate abstract states of granularity of $\langle 2\rangle$ with multimapping and the Partial-State technique. First, we apply a domain abstraction which maps $(1, 2) \rightarrow 1$ and $(3, 4) \rightarrow 3$ to the state. The resulting state is $\langle 1, 1, 3, 3\rangle$. Next, we apply Partial-State abstraction $\langle 1, 0, 1, 0\rangle$ which will pick a tile 1 and a tile 3 and turn them into variables in X. The resulting states are four Partial-States of granularity of $\langle 2\rangle$ $\langle x_1, 1, x_2, 3\rangle$, $\langle x_1, 1, 3, x_2\rangle$, $\langle 1, x_1, x_2, 3\rangle$, $\langle 1, x_1, 3, x_2\rangle$.*

Representation Power Between Domain Abstraction, Projection and Partial-State Technique

In this section, we are going to discuss the representation power of three different abstraction techniques: domain abstraction, projection and the Partial-State technique. In order to discuss this issue, we will first formally define representation power. Then comparisons are made between the Partial-State technique and the other two techniques.

Definition 8 (Representation Power) *First, we define P_A to be an abstract state w.r.t abstraction A in problem space $\mathcal{S}$. For abstraction A and B. We say A and B have equal representation power*

iff for any P_A we can find a P_B to cover the same set of states in original space, and for any P_B we can find a P_A to cover the same set of states in original space. We say A has better representation power than B iff for any P_B we can find a P_A to cover exactly the same set of states covered by P_B, but for same P_A, there does not exist a P_B such that P_B will exactly cover the same set of states covered by P_A. We say the representation power of A and B is incomparable iff neither for all P_A we can find a P_B to cover exactly the same set of states covered by P_A nor for all P_B we can find a P_A to cover exactly the same set of states covered by P_B.

Based on this definition, we claim that Partial-State has better representation power than projection. The reason for this claim is that for every abstract state $P_{projection}$ generated by projection, we can substitute the positions that are projected out with distinct variables in the Partial-State, resulting a Partial-State P_{PS} which covers exactly the same set of states of $P_{projection}$. However, for an abstracted state generated by Partial-State technique, if the same variable occurs twice or more in the Partial-State, projection cannot generate a corresponding abstract state. For example in state $\langle 1, 2, 3, 4\rangle$ with $D = \{1, 2, 3, 4\}$, a Partial-State $\langle 1, x_1, x_1, 4\rangle$ will never have a corresponding abstract state generated by projection. This is because projection has no way to enforce that the second and third position have the same value.

Domain abstraction and Partial-State abstraction are incomparable. This can be shown using two examples. Given a state $\langle 1, 2, 3, 4\rangle$, we apply a domain abstraction $(2, 3) \rightarrow 2$ on it and get $\langle 1, 2, 2, 4\rangle$ as result. This abstract state can represent 4 states: $\langle 1, 2, 2, 4\rangle$, $\langle 1, 3, 3, 4\rangle$, $\langle 1, 2, 3, 4\rangle$ and $\langle 1, 3, 2, 4\rangle$. However, it is impossible to design a Partial-State to represent these 4 states. On the other hand, a Partial-State $\langle 1, x_1, x_1, 4\rangle$ can represent 4 states: $\langle 1, 1, 1, 4\rangle$, $\langle 1, 2, 2, 4\rangle$, $\langle 1, 3, 3, 4\rangle$ and $\langle 1, 4, 4, 4\rangle$. These 4 states also cannot be represented using a single abstract state generated by domain abstraction.

Operators

We can use PSVN operators with the Partial-State technique. A PSVN operator can be seen as two Partial-States. An operator is defined as a pair $\langle LHS, RHS\rangle$, where LHS and RHS look like the Partial-States defined above. LHS defines the precondition of the operator and we can apply the operator to any state contained in LHS . RHS has two meanings. Firstly, it defines a set of states which are the possible results of the operator. Secondly, RHS and LHS together define how an operator acts on a state.

Applicability of Operators

Given a Partial-State s^p and an operator $\omega = \langle LHS, RHS\rangle$, ω can be applied to s^p iff for all i, $LHS[i] \in D_i$ and $s^p[i] \in D_i \implies s^p[i] = LHS[i]$. The meaning of this applicability rule is that when considered as Partial-State, $LHS \cap s^p \neq \emptyset$. In other words, the set of states represented by s^p and LHS have some common states. We leave the discussion of the reason behind this applicability

rule to the next chapter.

The result of the operator is produced using rules presented in Section 2.2.

Example 15 *Given a Partial-State* $P = \langle 1, x_1 \rangle$ *and an operator* $\langle 1, A \rangle \rightarrow \langle A, 1 \rangle$*, left part of the operator matches* P *because* 1 *in the first position matches the corresponding* 1 *in* P *and* A *is bound to* x_1*. The resulting state is* $\langle x_1, 1 \rangle$*.*

Goal Testing

A big difference between the Partial-State technique and the multimapping domain abstraction is that in the Partial-State technique we do not abstract the goal state. This actually overcomes the problem that performance will degrade once the mapping factor increase beyond a certain point. We consider Partial-State s^p have reached goal state g when $g \in s^p$.

Duplicate Testing

We will face the problem of detecting duplicated Partial-States when we implement the Partial-State technique with search algorithms. A typical example would be detecting if a newly generated Partial-State already exists in the CLOSED list of the A* algorithm. Because a Partial-State can represent a set of states, a more fundamental question is that if the states contained in the newly generated Partial-State are already covered by Partial-States in the CLOSED list. A straightforward way to answer this question is to explicitly enumerate all the states contained in the newly generated Partial-State and test each to see if it is contained in a Partial-State in the CLOSED list. To implement this detection will require us to explicitly list all states contained in a Partial-State and this will introduce huge computational cost. We still have not found an efficient way to do this test. Therefore the method we used in our experiment is to simply detect if exactly the same Partial-State already exists in the cached states.

3.4.2 Experiments

In this section, limited experiments are done to compare the Partial-State technique with domain abstraction. The experiment results show that Partial-State technique does not show much advantage over domain abstraction.

Experiment Setting

The algorithm we used in this experiment is HIDA*. We use DA-HIDA* and PS-HIDA* to refer to HIDA* implemented with domain abstraction and the Partial-State technique respectively. We only used two small domains, the 8-Puzzle and 9-Pancake, in this experiment. For each domain, we generated 500 solvable start states as test cases. To measure the performance of each technique, we calculated the average number of nodes expanded at the base level, CPU time (in seconds) and the number of memory entries cached, as we did in Section 3.3.2.

The number of levels in the hierarchy of abstractions and the granularity of each level are designed to be the same for both techniques. The granularity of the first level is $\langle 4 \rangle$ and for each level higher an additional tile will be abstracted, so the granularity for the second, third and fourth level is $\langle 5 \rangle$, $\langle 6 \rangle$ and $\langle 7 \rangle$. In both domains, only tiles 1 to 8 will be abstracted. At the first level of domain abstraction 4 tiles out of these 8 tiles are chosen and mapped to the same tile. In this way we can generate a total of 70 domain abstractions. For the Partial-State technique, we first apply a domain abstraction of granularity $\langle 2, 2, 2 \rangle$ on the state to create 3 pairs of duplicated tiles, then pick one out of each pair and one additional tile, 4 tiles in total, to make into variables. In this way, the granularity of the first level of a Partial-State abstraction still has a granularity of $\langle 4 \rangle$, and the mapping factor is 8. This is because there are 8 different ways to choose one occurence out of each of the 3 pairs of duplicate tiles. For each $\langle 2, 2, 2 \rangle$ domain abstraction, we can generate 2 Partial-State abstractions[4], thus the total number of Partial-State abstractions are 840. The same domain abstractions and Partial-State abstractions are used in both the 8-Puzzle and 9-Pancake problem domains.

Example 16 *An example of abstraction used for the Partial-State technique is presented at Table 3.15.*

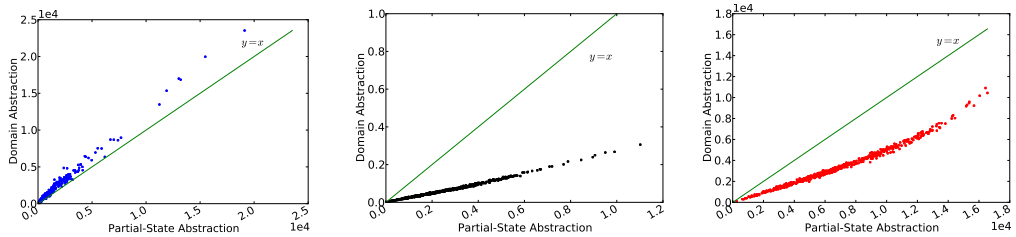
s	1	2	3	4	5	6	7	8
Step 1	1	1	2	2	3	3	7	8
Step 2	1	x_1	x_2	2	3	x_3	x_4	8

Table 3.15: An example of abstraction used for the Partial-State technique. Step 1 is to apply a domain abstraction to s to create 3 pairs of duplicated tiles. Step 2 is to pick one out of each pair and one additional tile, 4 tiles in total, to make them into variables. The final abstracted Partial-State is $\langle 1, x_1, x_2, 2, 3, x_3, x_4, 8 \rangle$

Results and Discussions

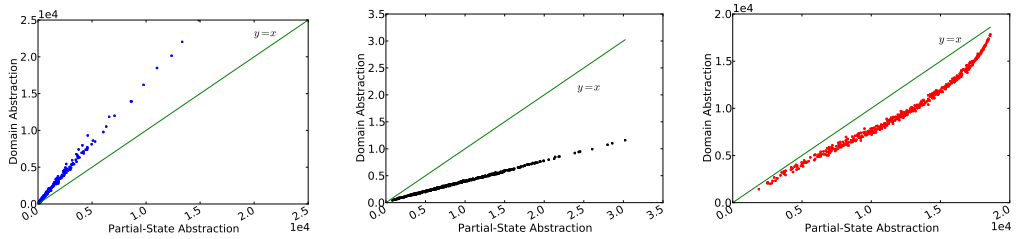
The results for the 8-Puzzle and 9-Pancake are presented in Figure 3.22 and Figure 3.23 respectively. Another perspective in boxplots is presented in Figure 3.25 and Figure 3.24. The trends shown in both domains are the same. For the number of nodes expanded at the base level, Partial-State abstraction performs better than domain abstraction. Because the granularities of the first level of all abstractions are $\langle 4 \rangle$, the size of abstract space of the first level is the same for both Partial-State abstraction and domain abstraction. This is to say the improved heuristic quality of Partial-State abstraction is gained without using a larger abstract space. However, both the CPU time and memory usage for Partial-State abstraction is inferior to that of domain abstraction. This shows us that in order to get a better heuristic value, the Partial-State technique has done more work in the abstract levels than domain abstraction. The workload in the abstract levels is large enough that the overall CPU time and memory usage have been compromised.

[4] 3 tiles are chosen from 3 pairs of duplicated tiles, and then we always need to choose one additional tile from the remaining 2 tiles.



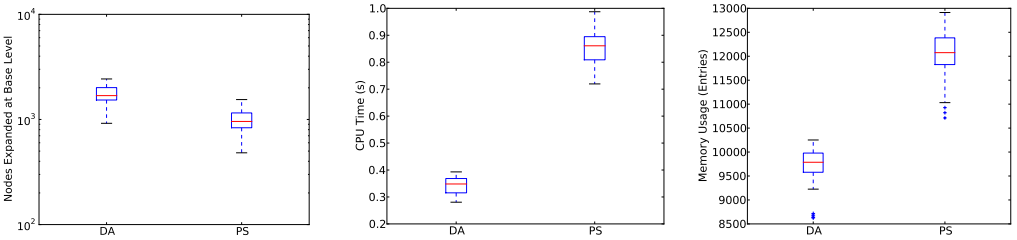
(a) Nodes Expanded at Base Level (b) CPU Time (c) Memory usage

Figure 3.22: 8-Puzzle: Partial-State Abstraction vs. Domain Abstraction.



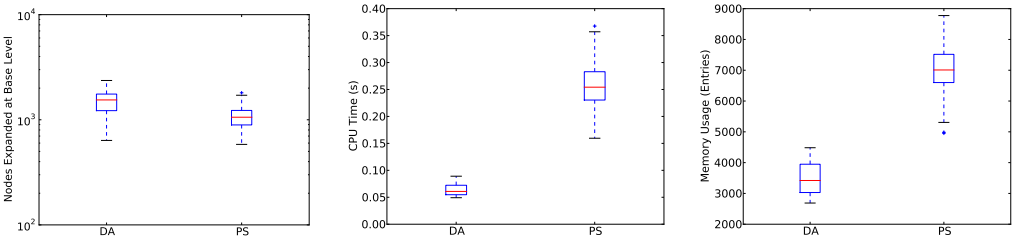
(a) Nodes Expanded at Base Level (b) CPU Time (c) Memory usage

Figure 3.23: 9-Pancake: Partial-State Abstraction vs. Domain Abstraction.



(a) Nodes Expanded at Base Level (b) CPU Time (c) Memory usage

Figure 3.24: 9-Pancake: Abstractions Comparison.



(a) Nodes Expanded at Base Level (b) CPU Time (c) Memory usage

Figure 3.25: 8-Puzzle: Abstractions Comparison.

At the beginning, we mentioned that this is a premature technique and it is still far away from being implemented practically. This technique suffers from the following problems:

- When we apply an operator on a Partial-State, it is possible that the resulting Partial-State will contain fewer states. We call this **the shrinking problem**. For example, suppose we have Partial-State $s^p = \langle 1, 1, x_1, x_2 \rangle$ and operator $\langle 1, x_3, 1, 1 \rangle \Longrightarrow \langle 2, x_3, 2, 2 \rangle$. The operator will turn s^p to $\langle 2, 1, 2, 2 \rangle$ which is a fully grounded state. This is the major shortcoming preventing the use of this technique in most domains. We will discuss this "shrinking" problem in more detail in Chapter 4.

- When we abstract a ground state into a Partial-State, this Partial-State might contain some states which are unreachable in the original space. For example, in 2×2 sliding tile puzzle, we want to turn a state $\langle 0, 1, 2, 3 \rangle$ into a Partial-State of granularity of $\langle 2 \rangle$. A possible result is $\langle 0, x_1, x_2, 3 \rangle$. However, this Partial-State will contain some unreachable states such as $\langle 0, 3, 3, 3 \rangle$. This is because the information that only one tile of 1 and one tile of 2 are turned into variables is not encoded in the Partial-State. Currently, our technique cannot encode this information into Partial-States, thus in some domains, some Partial-States might mismatch some operators. This problem that abstract states contain some illegal states is not specific to the Partial-State technique. It also occurs in domain abstraction and projection. The related problem is well studied by Zilles and Holte [36, 35].

For these reasons, the Partial-State technique is far from practical at present, but it overcomes a problem which arises with multimapping domain abstraction. With Partial-State technique, the goal state is not abstracted when solving the problem. Therefore with an increase of the Mapping Factor, the performance shouldn't degrade like it does with multimapping domain abstraction.

3.5 Conclusions

In this chapter, we introduced the multimapping framework and two multimapping techniques. The first one, based on domain abstraction, was thoroughly explained and extensive experiments were run to compare the performance between this new technique and existing single and multiple domain abstraction technique. The second method, Partial-State abstraction, is not fully developed, only the basic idea is explained. These techniques inspired us to develop a general theory of abstraction in which each abstracted state is a set of states. This will be presented in the next chapter.

Chapter 4

State-Set Search

This chapter will introduce a state-set search theory. This theory is inspired by the Partial-State abstraction technique, but the situation this theory can be applied to commonly occurs in heuristic search and planning research. For example, in the domain abstraction we described in Chapter 2, each abstract state actually represents a set of states in the original space. This theory studies this kind of state space, a space where each state represents a set of states that from some underlying state space. We call this space the state-set space. To distinguish states in the state-set space and the underlying space in following discussion, we use "state-set" to refer to the states in the state-set space and "state" to refer to the states in the underlying space. The main contribution of this theory is that it shows us theoretically what is the best possible admissible abstract distance (d_{ww}) in the state-set space. This theory can also help us explain certain behaviors of some existing planning and abstraction systems. This chapter is split into two parts. First, we will give a formal analysis of state-set search theory. Second, we will discuss applications of the theory: explaining some behaviors of existing systems, such as Grounded TWEAK [1] and the h^m method [14]. Most of the material in this chapter has been published in the 2011 Symposium on Combinatorial Search [28].

4.1 Formal Analysis

In this section, we will first formally define the state-set space and and two matching mechanisms for state-sets. Based on these two matching mechanisms, we introduced four kinds of distances between any pair of state-sets. First, the very basic notions of state-set space are defined as follows:

Definition 9 (State-set) *Let S be a non-empty set of states. A state-set (with respect to S) is a non-empty subset of S. We equate state $s \in S$ with the state-set $\{s\}$.*

Notationally, we allow a function $f : A \to B$ to be applied to a subset $A' \subseteq A$ instead of just an element of A, with $f(A') = \{f(a) \mid a \in A'\}$. Under this convention $f(\emptyset) = \emptyset$.

Definition 10 (State Multimap) *Let S be a non-empty set of states. A state multimap ω on S is a function from S to 2^S, the powerset of S. If P is a state-set, $\omega(P) = \bigcup_{s \in P} \omega(s)$.*

Definition 11 (State Space, Operator) *A state space is a pair $\mathcal{S} = (S, \Omega)$ where S is a non-empty set of states, and Ω is a set of state multimaps on S called operators. For each operator $\omega \in \Omega$ the set of states to which ω can be applied is $PRE_\omega = \{s \in S \mid \omega(s) \neq \emptyset\}$ and the set of states that can possibly be produced by ω is $POST_\omega = \omega(S)$. Without loss of generality we assume for all $\omega \in \Omega$ that PRE_ω is non-empty and therefore it and $POST_\omega$ are both state-sets.*

By defining an operator to be a multimap, we permit one operator to generate more than one successor of a state. This is to be interpreted like the non-deterministic operators in PSVN—the operator produces all the successors.

These definitions do not allow costs to be associated with operators. We leave this extension for future work.

Definition 12 (State Distance) *Let $\mathcal{S} = (S, \Omega)$ be a state space. A finite sequence of operators $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ is applicable to state $s \in S$ iff $\pi(s) = \omega_z(\omega_{z-1} \ldots \omega_2(\omega_1(s)) \ldots) \neq \emptyset$. π is a path from state $s \in S$ to state $t \in S$ iff $t \in \pi(s)$. The distance $d(s, t)$ between two states is the length of the shortest path from s to t (∞ if no such path exists).*

Definition 13 (State-set Space) *Let $\mathcal{S} = (S, \Omega)$ be a state space. The state-set space induced by $\mathcal{S}$ is $\mathcal{SS} = (2^S, \Omega)$.*

4.1.1 State-set Matching, Paths, and Distances

With the definition of state-set space, now we will introduce two matching mechanisms and four types of paths in state-set space. The two matching mechanisms,weak match and strong match, are defined as follows.

Definition 14 (Strong Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P and Q state-sets w.r.t. S. We say that P strongly matches Q iff $P \subseteq Q$,*

Definition 15 (Weak Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P and Q state-sets w.r.t. S. We say that P weakly matches Q iff $P \cap Q \neq \emptyset$.*

We will say simply that P matches Q where strongly/weakly is determined by the context or where either definition can be used.

Definition 16 (Operator Strong Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. Operator ω is strongly applicable to P iff P strongly matches PRE_ω.*

Definition 17 (Operator Weak Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. Operator ω is weakly applicable to P iff P weakly matches PRE_ω.*

Definitions 16 and 17 are shown graphically in Figure 4.1. State-sets are depicted with circles. State-sets representing operator preconditions are shaded. The upper part of Figure 4.1 shows that

operator ω's precondition contains state-set P; ω is therefore strongly applicable to P. In the lower part of Figure 4.1 operator ω's precondition intersects P but does not contain it; ω is therefore only weakly applicable to P. In both parts of the figure $Q = \omega(P) = \omega(P \cap PRE_\omega)$.

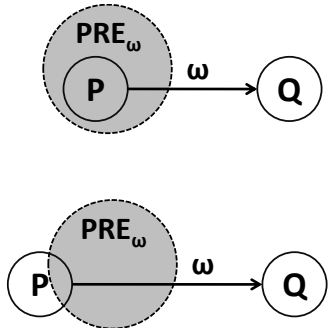


Figure 4.1: (upper) Operator ω is strongly (and weakly) applicable to P. (lower) Operator ω is weakly applicable to P.

Definition 18 (Operator Sequence Strong Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. A finite sequence of operators $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ is strongly applicable to P iff $z = 1$ and ω_1 is strongly applicable to P, or $z > 1$, ω_1 is strongly applicable to P, and the sequence $(\omega_2, \ldots, \omega_z)$ is strongly applicable to $\omega_1(P)$.*

Definition 19 (Operator Sequence Weak Match) *Let $\mathcal{S} = (S, \Omega)$ be a state space and P a state-set w.r.t. S. A finite sequence of operators $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ is weakly applicable to P iff $z = 1$ and ω_1 is weakly applicable to P, or $z > 1$, ω_1 is weakly applicable to P, and the sequence $(\omega_2, \ldots, \omega_z)$ is weakly applicable to $\omega_1(P)$.*

These definitions immediately imply the following.

Corollary 17 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is strongly or weakly applicable to both P and Q. Then $\pi(P) \supseteq \pi(Q)$.*

Corollary 18 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is weakly applicable to Q. Then π is weakly applicable to P.*

Corollary 19 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S such that $P \supseteq Q$, and $\pi \in \Omega^+$ a finite sequence of operators that is strongly applicable to P. Then π is strongly applicable to Q.*

Definition 20 (State-set Distance) *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S, and $\pi \in \Omega^+$ a finite sequence of operators that is applicable to P. π is a path from P to Q iff $\pi(P)$ matches Q. The distance from P to Q, denoted $d(P, Q)$, is the length of the shortest path from P to Q (∞ if no such path exists). We say that Q is reachable from P if there exists a path from P to Q.*

There are actually four different definitions of path, distance, and reachable here, depending on whether strong or weak matching is used to test if π is applicable to P and whether strong or weak matching is used to test if $\pi(P)$ matches Q. The two definitions we will focus on are:

- When strong matching is used throughout Definition 20, a path from P to Q is applicable to all states in P and guaranteed to map each state in P to a state in Q. We call such paths "strong paths" and denote the corresponding distance and operator sequence as $d_{ss}(P, Q)$ and $\pi_{ss}(P, Q)$ respectively.
- When weak matching is used throughout Definition 20, a path from P to Q maps at least one state in P to a state in Q. We call such paths "weak paths" and denote the corresponding distance and operator sequence as $d_{ww}(P, Q)$ and $\pi_{ww}(P, Q)$ respectively.

Figure 4.3 depicts these two definitions, with the upper part showing a strong path from P to Q and the lower part showing a weak path from P to Q. Because every strong path is also a weak path, $d_{ww}(P, Q) \leq d_{ss}(P, Q)$. It can easily happen that $d_{ss}(P, Q)$ is infinite but $d_{ww}(P, Q)$ is finite. For example, this would happen with the sliding tile puzzle if P contained states in which the blank was in different locations. This is shown in the following example.

Example 20 *In the 2×2 sliding tile puzzle, let us assume that* $P = \{\langle 0, 1, 2, 3\rangle, \langle 1, 0, 2, 3\rangle\}$, $Q = \{\langle 3, 2, 1, 0\rangle\}$, *as illustrated in Table 4.2. For any operator* ω, *it is only possible that* ω *is applicable to one of the states in* P, *i.e., weakly applicable to* P. *This is because the blank (tile 0) is at different positions in those two states in* P *while an operator can only be applied to a state with the blank at a specific position.* $d_{ss}(P, Q)$ *is infinite in this example because there is no operator which is applicable to both states in* P.

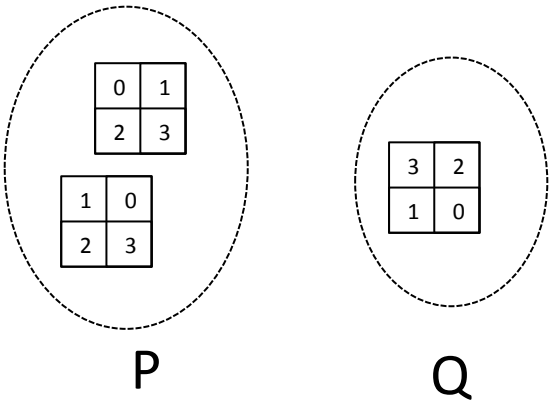


Figure 4.2: State-set P and state-set Q.

Theorem 21 *The fact that there exists a weak path* π *from* P *to* Q *is equivalent to there existing a pair of states* s *and* t *such that* $s \in P$, $t \in Q$ *and* $\pi(s) = t$.

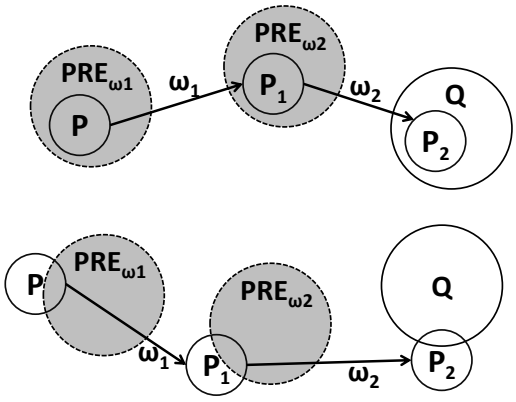


Figure 4.3: Strong (upper) and weak (lower) paths from P to Q.

Proof. First we will prove that the existence of a weak path $\pi = \langle \omega_1, \cdots, \omega_k \rangle$ from P to Q is a sufficient condition for there existing a pair of states s and t such that $s \in P$, $t \in Q$ and $\pi(s) = t$. We choose t to be any state in $\pi(P) \cap Q$. Because $\pi(P) \cap Q \neq \emptyset$, such a t must exist.

We prove the rest using mathematical induction. The base case is that there must exist a $s_k \in \langle \omega_1, \cdots, \omega_{k-1} \rangle(P)$ such that $\omega_k(s_k) = t$. This statement holds because according to the rule of weak matching, $PRE_{\omega_k} \cap \langle \omega_1, \cdots, \omega_{k-1} \rangle(P) \neq \emptyset$. This s_k must exist. The inductive hypothesis for this proof is that there exists s_i such that $\langle \omega_i, \omega_{i+1}, \cdots, \omega_k \rangle(s_i) = t$. We need to prove that there exists $s_{i-1} \in \langle \omega_1, \cdots, \omega_{i-2} \rangle(P)$ such that $\omega_{i-1}(s_{i-1}) = s_i$. This is obvious because $PRE_{\omega_{i-1}} \cap \langle \omega_1, \cdots, \omega_{i-1} \rangle(P) \neq \emptyset$, this s_{i-1} must exist. Because the base case and inductive step both hold, there must exist s_1 such that $\langle \omega_1, \omega_2, \cdots, \omega_k \rangle(s_1) = t$ and $s_1 \in P$. This s_1 is the s we need to find.

Second, we will prove that the existence of a weak path $\pi = \langle \omega_1, \cdots, \omega_k \rangle$ from P to Q is a necessary condition for there existing a pair of states s and t such that $s \in P$, $t \in Q$ and $\pi(s) = t$. The base case of this proof is to let $s_1 = \omega_1(s)$, we have $s_1 \in \omega_1(P)$. This is obvious because $s \in PRE_{\omega_1} \cap P$. The inductive hypothesis for the proof is that there exists $s_i \in \langle \omega_1, \cdots, \omega_{i-1} \rangle(P)$. We need to prove for $s_{i+1} = \omega_i(s_i)$, $s_{i+1} \in \langle \omega_1, \cdots, \omega_i \rangle(P)$. Because $s_i \in PRE_{\omega_i}$, we have $\langle \omega_1, \cdots, \omega_i \rangle(P) \cap PRE_{\omega_i} \neq \emptyset$. Therefore $s_{i+1} \in \langle \omega_1, \cdots, \omega_i \rangle(P)$. Because both base case and inductive step hold, $t \in \langle \omega_1, \cdots, \omega_k \rangle(P)$ thus $\pi(P) \cap Q \neq \emptyset$. π is a weak path from P to Q.

□

The two definitions of path and distance that we will not focus on in this chapter are:

- If, in Definition 20, strong matching is used to test applicability and weak matching to test if $\pi(P)$ matches Q, a path from P to Q is guaranteed to be applicable to all states in P and is guaranteed to map at least one state in P to Q, but it is not guaranteed to map all states in P to Q.

- If, in Definition 20, weak matching is used to test applicability and strong matching to test if $\pi(P)$ matches Q, a path from P to Q is guaranteed to be applicable to at least one state in P,

but is not guaranteed to be applicable to all states in P, and it is guaranteed to map any state in P to which it is applicable to a state in Q.

We are not interested in these two kinds of paths because the distances provided by them can neither serve as an admissible heuristic nor carry practical meaning.

4.2 Properties of Weak and Strong Paths and Distances

In this section, we will discuss some important properties of weak and strong paths.

Theorem 22 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \supseteq Q$.*

- *Let π be a weak path from P to Q. Then π is a weak path from P to R and $d_{ww}(P, R) \leq d_{ww}(P, Q)$.*
- *Let π be a strong path from P to Q. Then π is a strong path from P to R and $d_{ss}(P, R) \leq d_{ss}(P, Q)$.*

Proof.

- Because π is a weak path from P to Q, we have $\pi(P) \cap Q \neq \emptyset$. Since $R \supseteq Q$, we have $\pi(P) \cap R \neq \emptyset$. Therefore, π is also a weak path from P to R and its length is an upper bound for the length of the shortest weak path from P to R.
- Because π is a strong path from P to Q, we have $\pi(P) \subseteq Q$. Since $R \supseteq Q$, we have $\pi(P) \subseteq R$. Therefore, π is also a strong path from P to R and its length is an upper bound for the length of the shortest strong path from P to R.

□

Theorem 23 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \supseteq P$, and π a weak path from P to Q. Then π is a weak path from R to Q and $d_{ww}(R, Q) \leq d_{ww}(P, Q)$.*

Proof. Because π is a weak path from P to Q, there must exist a state $s \in P$ such that $\pi(s) \cap Q \neq \emptyset$. Because $R \supseteq P$, $s \in R$. Therefore π is also a weak path from R to Q and the length of it is an upper bound for the shortest weak path from R to Q. □

The preceding two theorems show that if we have two state-sets, P and Q, and "abstract" them to supersets $P' \supseteq P$ and $Q' \supseteq Q$, then the "abstract" distance $d_{ww}(P', Q')$ is a lower bound on $d_{ww}(P, Q)$. This is not true for d_{ss} since Theorem 23 does not hold for strong paths. Indeed, as the next theorem shows, the opposite holds. It immediately follows that d_{ss} cannot, in general, be used in abstraction systems to define admissible heuristics.

Theorem 24 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P, Q, and R state-sets w.r.t. S such that $R \subseteq P$, and π a strong path from P to Q. Then π is a strong path from R to Q and $d_{ss}(R, Q) \leq d_{ss}(P, Q)$.*

Proof. Because π is a strong path from P to Q, for any state $s \in P$ we have $\pi(s) \subseteq Q$. Because $P \supseteq R$, we have $\pi(R) \subseteq Q$. Therefore π is a strong path from R to Q and its length is an upper bound for the length of the shortest strong path from R to Q. □

Theorem 25 $d_{ww}(P,Q) = \min_{p \in P, q \in Q} d(p,q)$.

Proof.

First, we will discuss the special situation that $d_{ww}(P,Q) = \infty$. Then, we will prove this theorem by contradiction. This is split into two steps, we will prove that it is impossible to have $d_{ww}(P,Q) > \min_{p \in P, q \in Q} d(p,q)$. Next, we will prove that it is impossible to have $d_{ww}(P,Q) < \min_{p \in P, q \in Q} d(p,q)$ either. This leaves $d_{ww}(P,Q) = \min_{p \in P, q \in Q} d(p,q)$ as the only possibility.

For the situation that $d_{ww}(P,Q) = \infty$, using Theorem 21 we know that $\min_{p \in P, q \in Q} d(p,q) = \infty$. Now, assume $d_{ww}(P,Q)$ is finite.

- Assume $d_{ww}(P,Q) < \min_{p \in P, q \in Q} d(p,q)$. Because $\pi_{ww}(P,Q)$ maps at least one state in P to a state in Q, there must exist a pair of states $\langle p', q' \rangle$ such that $p' \in P, q' \in Q$ and $d(p',q') < \min_{p \in P, q \in Q} d(p,q)$. This leads to a contradiction.
- Assume $d_{ww}(P,Q) > \min_{p \in P, q \in Q} d(p,q)$. Let $p \in P$ and $q \in Q$ be states with minimum $d(p,q)$. Because $p \in P$, the shortest path from p to q, π, is also applicable to P according to Corollary 18. According to Corollary 17, $\pi(P) \supseteq \pi(p) = q$, which means π is a weak path from P to Q, therefore we found a path that is shorter than $d_{ww}(P,Q)$. Because $d_{ww}(P,Q)$ is already the shortest path from P to Q by definition 20, this leads to a contradiction.

□

This theorem shows that d_{ww} has the property that is needed to guarantee that abstracting states (or state-sets) by mapping them to supersets and using the d_{ww} between supersets as an estimate of the distances between the original states (or state-sets) will be an admissible heuristic. The other distance measures defined in Definition 20 are not guaranteed to have this property, so a general-purpose abstraction system cannot use them if admissibility is required. Moreover, $d_{ww}(P,Q)$ is the largest distance from P to Q that is guaranteed to be an admissible estimate of the distance between a subset of P and a subset of Q.

The following shows that d_{ss} obeys the triangle inequality.

Lemma 26 *Let $\mathcal{S} = (S, \Omega)$ be a state space, and P, Q and R state-sets w.r.t. S, Then $d_{ss}(P,Q) \leq d_{ss}(P,R) + d_{ss}(R,Q)$.*

Proof. The corresponding operator sequence of the strong paths $\pi_{ss}(P,R)$ and $\pi_{ss}(R,Q)$ can be combined together and the resulting operator sequence which generate a strong path from P to Q. We can do this because $\pi_{ss}(P,R)(P) \subseteq R$ and $R \subseteq PRE_{\omega'}$ where ω' is the first operator in the operator sequence $\pi_{ss}(R,Q)$. So $d_{ss}(P,R) + d_{ss}(R,Q)$ is an upper bound for $d_{ss}(P,Q)$ □

The same is not true of d_{ww}, and therefore it is not a true distance metric. Figure 4.4 shows an example in which d_{ww}violates the triangle inequality. $d_{ww}(P,Q) = 1$ (using operator ω) but $d_{ww}(P,R) = 0$ because $P \cap R \neq \emptyset$ and $d_{ww}(R,Q) = 0$ as well. Hence $d_{ww}(P,Q) > d_{ww}(P,R) + d_{ww}(R,Q)$. Since the standard proof of the consistency of heuristics defined by distances in an abstract space relies on the abstract distances obeying the triangle inequality, the fact that d_{ww} does not obey the triangle inequality raises doubts about the consistency of heuristics based on computing d_{ww} in an abstract space.

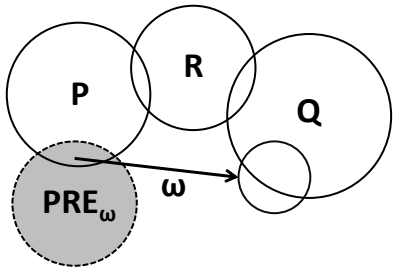


Figure 4.4: Example of d_{ww}violating the triangle inequality.

Theorem 27 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S, we have:*
(1) Let $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ be a shortest weak path from P to Q, and R_i the i^{th} intermediate state-set along the path (i.e., $R_i = (\omega_1, \omega_2, \ldots, \omega_i)(P)$ for $i \in \{1, \ldots, z-1\}$). Then $d_{ww}(R_i, Q) = z - i$, and $d_{ww}(P, R_i,) \leq i$.
(2) Let $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ be a shortest strong path from P to Q, and R_i the i^{th} intermediate state-set along the path (i.e., $R_i = (\omega_1, \omega_2, \ldots, \omega_i)(P)$ for $i \in \{1, \ldots, z-1\}$). Then $d_{ss}(R_i, Q) = z - i$, $d_{ss}(P, R_i,) = i$.

Proof. First, we need to approve that $d_{ww}(R_i, Q) = d_{ss}(R_i, Q) = z - i$. For the $d_{ww}(R_i, Q) = z - i$ part, because π exists, we have $d_{ww}(R_i, Q) \leq z - i$. We prove the following by contradiction. If $d_{ww}(R_i, Q) < z - i$, then we can connect $\pi_{ww}(R_i, Q)$ with the path $(\omega_1, \omega_2, \ldots, \omega_z)(P)$ and have a $d_{ww}(R_i, Q) < z$. Because z is already the shortest length of weak path from P to Q, $d_{ww}(R_i, Q)$ cannot less than $z - i$. Therefore, $d_{ww}(R_i, Q) = z - i$. $d_{ss}(R_i, Q) = z - i$ can be proved in the same way.

Second, we need to prove that $d_{ss}(P, R_i,) = i$, and $d_{ww}(P, R_i,) \leq i$. For $d_{ss}(P, R_i,) = i$, we first can deduce that $d_{ss}(P, R_i) \leq i$ because there is already a strong path from P to R_i. We prove the following by contradiction. If $d_{ss}(P, R_i) < i$, we can concatenate this $\pi_{ss}(P, R_i)$ with the rest of $\pi_{ss}(R_i, Q)$ and get a strong path shorter than z. This contradicts the fact that z is already the shortest strong path from P to Q. Therefore, $d_{ss}(P, R_i) = i$. This does not hold for $\pi_{ww}(P, R_i)$

because $\pi_{ww}(P, R_i)$ and $\pi_{ww}(R_i, Q)$ cannot, in general, be concatenated to be a weak path from P to Q. This is illustrated in the following discussion. □

The surprising part of Theorem 27 is the last part, that $d_{ww}(P, R_i)$ can be less than i. Figure 4.5 illustrates how this can happen. The shortest weak path from P to Q passes through R_1 and then R_2 but there is weak path directly from P to R_2 so $d_{ww}(P, R_2) = 1$ even though R_2 is distance 2 from P on the shortest path to Q. As the figure shows, this happens because the path directly from P to R_2 intersects with R_2 in a state-set (X) from which Q cannot be reached.

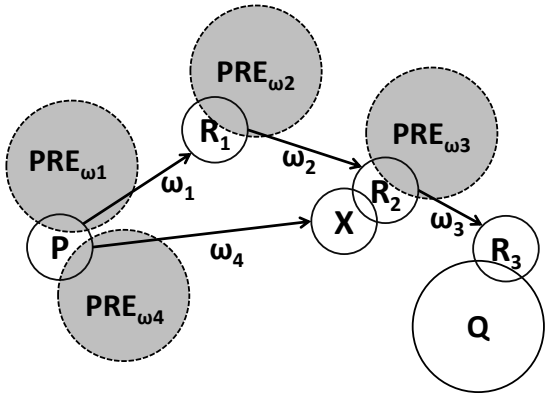


Figure 4.5: The shortest weak path from P to R_2 on the way to Q is not necessarily the shortest weak path from P to R_2.

Because $d_{ww}(R_i, Q) = d_{ss}(R_i, Q) = z - i$, systems like HIDA* [18] and Hierarchical A* [21], which compute abstract distances by searching in the forward direction (from start to goal), will correctly calcuate d_{ww} or d_{ss} for methods that abstract states (or state-sets) by mapping them to supersets.

Definition 21 (Simple Space) *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\mathcal{SS} = (2^S, \Omega)$ the state-set space induced by S. State-set $P \in \mathcal{SS}$ is "simple" if for all state-sets Q and R reachable from P, either $Q = R$ or $Q \cap R = \emptyset$. $\mathcal{SS}$ is simple if all state-sets $P \in \mathcal{SS}$ are simple.*

Theorem 28 *Let $\mathcal{S} = (S, \Omega)$ be a state space, P and Q state-sets w.r.t. S, P a simple state-set, $\pi = (\omega_1, \omega_2, \ldots, \omega_z) \in \Omega^+$ a weak path from P to Q, and R_i the i^{th} intermediate state-set along the path (i.e., $R_i = (\omega_1, \omega_2, \ldots, \omega_i)(P)$ for $i \in \{1, \ldots, z-1\}$). Then $d_{ww}(P, R_i,) = i$.*

Proof. Because P is a simple state-set, if there exists an operator sequence π_1 which can generate a weak path from P to R_i, then $\pi_1(P) = R_i$. We denote the length of this weak path as $d_{\pi_1}(P, R_i)$. We prove the following using contradiction. If $d_{\pi_1}(P, R_i)$ is shorter than $d_{ww}(P, R_i) = i$, then we can concatenate π_1 with operator sequence $(\omega_{i+1}, \cdots, \omega_z)$ and get a new weak path from P to Q which is shorter than $d_{ww}(P, Q)$. Since $d_{ww}(P, Q)$ is already the shortest weak path from P to Q, the length of weak path $d_{\pi_1}(P, R_i)$ cannot be less than i. Therefore i is the shortest possible length of a weak path from P to R_i. Thus $d_{ww}(P, R_i) = i$. □

4.2.1 Inverting Operators and Paths

Definition 22 (Inverse Operator) *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\omega \in \Omega$ an operator. The inverse of ω, denoted ω^{-1}, is defined to be the state multimapping, $\omega^{-1} : POST_\omega \rightarrow PRE_\omega$, such that, for any state $r \in POST_\omega$, $\omega^{-1}(r)$ is defined to be $\{s \in PRE_\omega \mid r \in \omega(s)\}$. If $r \notin POST_\omega$ then $\omega^{-1}(r)$ is defined to be $\emptyset$.*

The definition immediately implies the following.

Lemma 29 *Let $\mathcal{S} = (S, \Omega)$ be a state space and $\omega \in \Omega$ an operator. Then:*

1. *ω^{-1} is an operator in the sense of Definition 11.*
2. *$(\omega^{-1})^{-1} = \omega$, the inverse of ω^{-1}, is ω.*
3. *For any state $s \in PRE_\omega$ and any state $t \in \omega(s)$, $s \in \omega^{-1}(t)$.*
4. *$PRE_{\omega^{-1}} = POST_\omega$.*
5. *$POST_{\omega^{-1}} = PRE_\omega$.*
6. *For any state-set P to which ω is strongly/weakly applicable, ω^{-1} is strongly applicable to $\omega(P)$ and $(P \cap PRE_\omega) \subseteq \omega^{-1}(\omega(P))$.*
7. *For any state-set P to which ω^{-1} is strongly/weakly applicable, ω is strongly applicable to $\omega^{-1}(P)$ and $(P \cap POST_\omega) \subseteq \omega(\omega^{-1}(P))$.*

Proof.

- Lemma $1, 3, 4, 5$ follow immediately from Definition 11 and Definition 22.
- For Lemma 2, first, $(\omega^{-1})^{-1} : PRE_\omega \rightarrow POST_\omega$. Second, we have:

$$\begin{aligned}(\omega^{-1})^{-1}(s) &= \{r \in POST_\omega | s \in \omega^{-1}(r)\} \\ &= \{r \in POST_\omega | s \in PRE_\omega, r \in \omega(s)\} \\ &= \{r \in POST_\omega | r \in \omega(s)\} \\ &= \omega(s)\end{aligned}$$

□

- For Lemma 6, ω^{-1} is strongly applicable to $\omega(P)$ because $\omega(P) \subseteq POST_\omega$ and $PRE_{\omega^{-1}} = POST_\omega$. $(P \cap PRE_\omega) \subseteq \omega^{-1}(\omega(P))$ follows immediately from Definition 22. Lemma 7 follows from Lemma 6 and Lemma 2.

Definition 23 (True Inverse) *Let $\mathcal{S} = (S, \Omega)$ be a state space and ω an operator. ω is said to be a true inverse if $\omega(\omega^{-1}(P)) = P$ for every state-set $P \subseteq POST_\omega$, and ω^{-1} is said to be a true inverse if $\omega^{-1}(\omega(P)) = P$ for every state-set $P \subseteq PRE_\omega$.*

The last two parts of Lemma 29 show that ω^{-1} in not guaranteed to be a true inverse of ω and that neither is ω guaranteed to be a true inverse of ω^{-1}. To be true inverses they should exactly reverse each other's actions, which would require $(P \cap PRE_\omega) = \omega^{-1}(\omega(P))$ and $(P \cap POST_\omega) = \omega(\omega^{-1}(P))$. The following example shows that ω is not guaranteed to be a true inverse.

Example 30 *Assuming in a state space where $D_1 = D_2 = \{1, 2, 3\}$, State-set P only consists of one state $\langle 2, 2\rangle$. Applying operator $\omega : \langle A, A\rangle \rightarrow \langle 1, 1\rangle$, $\omega(P) = \{\langle 1, 1\rangle\}$. $\omega^{-1}(\omega(P)) = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle\}$ according to Definition 22. In this case $(P \cap PRE_\omega) \subset \omega^{-1}(\omega(P))$ and ω^{-1} is not a true inverse. We can also create an example which shows ω is not a true inverse.*

Note that it is possible for one of ω and ω^{-1} to be a true inverse and the other one not.

Example 31 *In the previous example setting, let $\omega^{-1} : \langle 1, 1\rangle \rightarrow \langle A, A\rangle$. For state-set $Q = \{\langle 1, 1\rangle\}$ we have $(Q \cap POST_\omega) = \omega(\omega^{-1}(Q))$. ω is a true inverse but ω^{-1} is not.*

Theorem 32 (Inverses of Weak Paths) *Let $\mathcal{S} = (S, \Omega)$ be a state space, P a state-set, $\pi = \langle \omega_1, \ldots, \omega_k\rangle$ an operator sequence that is weakly applicable to P, and Q any state-set such that $Q \cap \pi(P) \neq \emptyset$. Then $\pi^{-1} = \langle \omega_k^{-1}, \ldots, \omega_1^{-1}\rangle$ is a weak path from Q to P.*

Proof. If $\pi = \langle \omega_1, \ldots, \omega_k\rangle$ is an operator sequence that is weakly applicable to P and $Q \cap \pi(P) \neq \emptyset$, there must exist a state sequence $L = \langle s_0, \ldots, s_k\rangle$ such that $s_0 \in P$, $s_k \in Q$ and $s_{i+1} = \omega_i(s_i)$ for $i \in \{0, \ldots, k-1\}$ according to Theorem 21. If we apply the inverse operator sequence $\pi^{-1} = \langle \omega_k^{-1}, \ldots, \omega_1^{-1}\rangle$ to s_k, we will have $L^{-1} = \langle s_k, \ldots, s_1\rangle$ according to Lemma 29.6 , which is the reverse state sequence of L. Again, because $s_k \in Q$ and $s_1 \in P$, π^{-1} is an weak path from Q to P. □

The equivalent theorem for strong paths does not hold.

4.2.2 Challenge of Building a Pattern Database With State-sets

Pattern Databases (PDBs) [5, 6, 23] are a powerful tool for heuristic search and planning. A PDB is a lookup table containing the distance-to-goal for each abstract state. When a heuristic value is needed for a state s, we search for the abstract state of s, $\phi(s)$, in the PDB and return the corresponding value. A PDB is generated by searching backward from the abstract goal state and saving the distance from every abstract state it reaches from the abstract goal state. Since a PDB is a good tool for heuristic search and planning, we would like to see how they fit in the state-set search theory.

Even though state-set theory provides theoretical analysis of searching with state-sets, which can be considered as abstract state in the PDB terminology, this theory cannot help much in practical aspects of building a PDB. This is because the theory assumes there is always a compact way to represent a set of states, but representing an arbitrary set of states compactly is, in general, infeasible [26]. Assuming g is a goal state, any state-set that is one step away from g can be represented as

$$\omega^{-1}(g) = \{s | \omega(s) = g\}$$

for any operator ω. However, in general it is impossible to represent $\{s|\omega(s) = g\}$ compactly thus a PDB for state-set will not, in general, be practical.

4.3 Planning as State-Set Search

Bäckström [1] describes two planning formalisms that explicitly support reasoning about state-sets (called partial states by Bäckström), Grounded TWEAK and SAS+. Although these are planning formalisms, and not planning systems, they formally define the semantics of operator applicability and goal testing. The following theorems show that in both formalisms strong matching is used for both operator applicability and goal testing.

Theorem 33 *SAS+, as described by Backstrom [1], uses strong matching for testing operator applicability and goal satisfaction.*

Proof Sketch. States in SAS+ are exactly as we define states in Definition 2: vectors of a fixed length, with each position in the vector having its value drawn from a finite set of possible values. In addition, SAS+ has a special symbol (`u`) that allows a state to specify that the value in one or more of its positions is unknown. An SAS+ state with one or more `u` values therefore represents the set of states which agree on the known values and have all possible combinations of values for the unknown positions. This feature resembles the Partial-State technique discussed in Section 3.4.

Operator preconditions, in our sense, are divided into preconditions and prevail conditions in SAS+, but here we will call them all preconditions. An SAS+ state s satisfies the preconditions p of an operator if the values specified by p are known in s. From a state-set point of view, this means that s must be a subset of p (having known values that agree with those specified in p, and possibly more known values, means that s represents a subset of the states that p represents). Hence, SAS+ does strong matching to determine if an operator applies to a state. The goal is defined exactly like a precondition, and matching a state to the goal is done exactly as matching a state to a precondition is done, hence SAS+ also does strong matching to determine if a state matches the goal. □

Theorem 34 *Grounded TWEAK, as described by Backstrom [1], uses strong matching for testing operator applicability and goal satisfaction.*

Proof Sketch. In Grounded TWEAK a state is a set of literals, positive or negative facts that are known to hold in the state. Atoms that have no corresponding literal in a state may be either true or false. Hence, every "state" s in Grounded TWEAK represents the set of states, in our perspective, for which the literals in s are true and the atoms having no corresponding literal in s have all possible combinations of truth value assignments. Operator preconditions are also sets of literals, and precondition p is satisfied by state s if s contains all the literals in p. From a state-set point of view, this means that s must be a subset of p (having all the literals in p, and possibly more, means that s represents a subset of the states that p represents). In other words, Grounded TWEAK does strong

matching to determine if an operator applies to a state. The goal is also a set of literals, and state s satisfies the goal if s contains all the literals in the goal. Again, this is strong matching. □

In the following analysis, it is revealed that the h^m heuristics [14] used in regression planning are actually trying to find a suboptimal strong path from the start state to the set of goal states. Regression planning is a kind of planning in which the search for a plan is done in a backward fashion. In the following discussion, we will use propositional the STRIPS planning [27] formalism to describe states and operators. In this formalism, a state is represented by a set of atoms initially true. An operator consists of three set of atoms: $pre(a)$, $add(a)$ and $del(a)$. Each operator is associated with a cost $cost(a)$. The operators $+$ and $-$ are union and set subtract operations on atoms of a state. In the h^m method, each operator a can be applied on state s in a backward fashion iff $s \cap del(a) = \emptyset$. The result applying a to s in the backward fashion is $s' = (s - add(a)) + pre(a)$. The h^m heuristic is defined as follows, $R(p)$ is the search space of a regression planning problem P:

$$h^m(s) = \begin{cases} 0 & if s \subseteq s_0 \\ \min_{s':(s,a,s') \in R(P)} h^m(s') + cost(a) & if |s| \leq m \\ \max_{s' \subseteq s, |s'| \leq m} h^m(s') & otherwise \end{cases}$$

Note that all lower case letters above refer to sets of atoms rather than state-sets, thus all operators are set operators. In the following proof, we use capitalized letter such as P, Q to represent state-sets, and use the notation $Lit(P)$ to represent the set of atoms that describe P. In this case, $P \subseteq Q \Leftrightarrow Lit(P) \supseteq Lit(Q)$.

Theorem 35 *The length of the shortest path from P to Q found by regression planning, as defined by Haslum and Geffner [14], is* $d_{ss}(P, Q)$.

Proof. This proof is illustrated by Figure 4.6. Let ω' be the backward operator defined in the regression planning method for operator ω, let $O' = \langle \omega'_k, \ldots, \omega'_1 \rangle$ be the path found by regression planning from Q to P, with the states along the path being $\langle Q, Q_k, \ldots, Q_1 \rangle$. Thus, $O'(Q) = Q_1 \supseteq P$. It looks different from the definition of h^m above because here P is a state-set, if P has more atoms than Q_1, it is actually a smaller state-set. From the definition of ω', we can see that $\omega(\omega'(S)) \subseteq S$. This is because $\omega(\omega'(S)) = (Lit(S) - add(a)) + pre(a) + add(a) - del(a)$ could contain more atoms than $Lit(s)$ so that it is a smaller state-set. Next, we are going to prove that $O = \langle \omega_1, \ldots, \omega_k \rangle$ is strongly applicable to P. First, we will prove that O is strongly applicable to Q_1 and $O(Q_1) \subseteq Q$ so that O is also strongly applicable to P and $O(P) \subseteq Q$. The following is proved by mathematical induction. Base case: Because $Q_1 = \omega'_1(Q_2)$, applying ω_1 on both sides, we get $\omega_1(Q_1) = \omega_1(\omega'_1(Q_2)) \subseteq Q_2$. We denote $\omega_1(Q_1)$ by Q'_2, so $Q'_2 \subseteq Q_2$. Inductive hypothesis: assume in a certain position of the forward path, we have $Q'_i \subseteq Q_i$. We want to prove that $\omega_i(Q'_i) \subseteq Q_{i+1}$. Applying ω_i to Q_i, we have $\omega_i(Q_i) = \omega_i(\omega_i^{-1}(Q_{i+1})) \subseteq Q_{i+1}$, so we have $\omega_i(Q'_i) \subseteq Q_{i+1}$. Because the base case and inductive step both hold, $O = \langle \omega_1, \ldots, \omega_k \rangle$ is applicable on Q_1 and $O(Q_1) \subseteq Q$. Since $P \subseteq Q_1$, we have $O(P) \subseteq Q$. This is a strong path because a state

S must contain all the atoms in an operator's precondition, which makes state-set $PRE_\omega \supseteq S$. This conforms to the definition of a strong path.

□

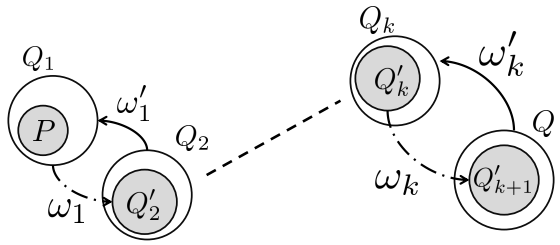


Figure 4.6: Backward and corresponding forward strong path calculated by regression planning. The dashed line in the middle represents omitted state-sets Q_3 to Q_{k-1} and Q'_3 to Q'_{k-1}.

Corollary 36 *$h^m(P,Q)$, as defined by Haslum and Geffner [14], is a lower bound on $d_{ss}(P,Q)$.*

Proof. This follows immediately from Theorem 35 and Theorem 22. If m is small enough such that $h^m(s) = \max_{s' \subseteq s, |s'| \leq m} h^m(s')$ is used, this is equivalent to replacing the strong path from S (S is the state-set equivalent of s) to Q with a strong path from a superset of S to Q.

□

If the same idea was used in forward planning, *i.e.,* replacing a state R reached while searching forward from P to Q with a superset $R' \supseteq R$, the distance calculated would be an upper bound on $d_{ss}(P,Q)$, not a lower bound (see Theorem 24).

4.4 The State-Set View of Abstraction

In order to formally analyze existing abstraction systems, we will first review the definition of them. The following formalization is a blend of ideas from Zilles and Holte [35] and Yang *et al.* [33] and is equivalent to the definition we introduced in Chapter 2.

Definition 24 *A vector state space is a triple $\mathcal{S} = (k, \{D_1, D_2, \ldots, D_k\}, \Omega)$ where $k \in \mathbb{N}$, each D_i is a finite set called a "domain", and Ω is a set of operators. The set of states in $\mathcal{S}$ is $S = D_1 \times D_2 \times \ldots \times D_k$.*

We consider two types of abstraction of vector state spaces.

Domain abstraction. A domain abstraction ψ of vector state space $\mathcal{S} = (k, \{D_1, D_2, \ldots, D_k\}, \Omega)$ is defined by a set $\{\psi_1, \psi_2, \ldots, \psi_k\}$ of mappings $\psi_i : D_i \to E_i$ where $E_i \subseteq D_i$ and, for at least one i, $E_i \neq D_i$. State $\langle \sigma_1, \ldots, \sigma_k \rangle \in D_1 \times D_2 \times \ldots \times D_k$ is mapped by ψ to abstract state $\langle \psi_1(\sigma_1), \ldots, \psi_k(\sigma_k) \rangle$.

Projection. A projection abstraction ψ of vector state space $\mathcal{S} = (k, \{D_1, D_2, \ldots, D_k\}, \Omega)$ is defined by a subset $\{i_1, \ldots, i_m\} \subset \{1, \ldots, k\}$. State $\langle \sigma_1, \ldots, \sigma_k \rangle \in D_1 \times D_2 \times \ldots \times D_k$ is mapped by ψ to abstract state $\langle \sigma_{i_1}, \ldots, \sigma_{i_m} \rangle$.

In both types of abstraction, each abstract state represents a set of states, *i.e.,* is a state-set over $D_1 \times D_2 \times \ldots \times D_k$. The abstract state spaces created by both types of abstraction are simple as defined in Definition 21. For projection this follows from the fact that if two abstract states are not the same, they must differ in the value of at least one variable. No state can have two different values for the same variable and therefore two different state-sets created by projection cannot have any state in common.

The abstract state spaces created by domain abstraction are also simple, for a similar reason. Two different abstract states, α_1 and α_2, created by the same domain abstraction ψ must differ in at least one position. Let i be a position in which they differ ($\alpha_1[i] \neq \alpha_2[i]$). Because ψ_i maps each value in the original domain D_i to one value in the abstract domain E_i, $\alpha_1[i] \neq \alpha_2[i]$ implies that there cannot exist a state s such that $\psi_i(s[i]) = \alpha_1[i]$ and $\psi_i(s[i]) = \alpha_2[i]$. Therefore there is no s mapped to both α_1 and α_2, so $\alpha_1 \cap \alpha_2 = \emptyset$.

If ψ is any abstraction mapping of state space $\mathcal{S}$, and ω is an operator that can be applied to state s, then it is required by a property of abstraction, the state space homomorphism, that ω be applicable to $\psi(s)$, the abstract state corresponding to s. This immediately implies that abstraction systems use weak matching to define if an operator is applicable, since there may be another state, t, such that $\psi(t) = \psi(s)$ but ω is not applicable to t.

A path from P to Q in an abstract space is a sequence of operators π such that $\pi(P) = Q$. This requirement for exact matching is more demanding than strong matching. However, for the types of abstraction we are considering (projection and domain abstraction) exact matching, strong matching, and weak matching are all equivalent when used to test if $\pi(P)$ matches Q.

Having concluded that projection and domain abstraction systems use weak matching to test operator applicability and the equivalent of weak matching to test if $\pi(P)$ matches Q, one is tempted to conclude that these systems compute d_{ww}. This is not true: $d_{abs}(P, Q)$, the distance from P to Q computed by a projection or domain abstraction system, can be strictly smaller than d_{ww}, as the following example illustrates.

Example 37 *A state is a 3-tuple of binary variables and there are only two operators,* $\omega_1 : \langle 1, 1, 1 \rangle \rightarrow \langle 0, 0, 1 \rangle$ *and* $\omega_2 : \langle 1, 0, 1 \rangle \rightarrow \langle 1, 0, 0 \rangle$. *If the first state variable is projected out, states* $\langle 0, 0, 1 \rangle$ *and* $\langle 1, 0, 1 \rangle$ *are mapped to the same abstract state,* $\langle 0, 1 \rangle$*, creating a path of length 2 from abstract state* $\langle 1, 1 \rangle$ *to* $\langle 0, 0 \rangle$*:* $\omega_1(\langle 1, 1 \rangle) = \langle 0, 1 \rangle$ *and* $\omega_2(\langle 0, 1 \rangle) = \langle 0, 0 \rangle$. *Thus* $d_{abs}(\langle 1, 1 \rangle, \langle 0, 0 \rangle) = 2$. *However,* $d_{ww}(\langle 1, 1 \rangle, \langle 0, 0 \rangle) = \infty$. *The difference in distances is because of a subtle difference between how* $\omega_1(\langle 1, 1 \rangle)$ *is defined in the projected space and how it is defined from a state-set point of view. In the latter,* $\omega_1(\langle 1, 1 \rangle)$ *is not* $\langle 0, 1 \rangle$*, it is* $\langle 0, 0, 1 \rangle$. *This is because* $\langle 1, 1 \rangle$ *denotes the state-set*

$\{\langle 0,1,1\rangle, \langle 1,1,1\rangle\}$ *and* ω_1 *maps that state-set to the state-set* $\{\langle 0,0,1\rangle\}$. *No operator is applicable to* $\{\langle 0,0,1\rangle\}$, *hence* $d_{ww}(\langle 1,1\rangle, \langle 0,0\rangle) = \infty$.

An analogous example can be given to show that the distances computed by domain abstraction systems can be strictly less than d_{ww}.

Example 38 *The domain* D *of the state space in this example is* $\{1,2,3,4\}$ *and the length* k *of a state vector is* 2. *Three states we picked in this space are:* $\langle 1,3\rangle$, $\langle 1,2\rangle$, *and* $\langle 3,3\rangle$. *There are only two operators in this state space:* $\omega_1 : \langle 1,2\rangle \rightarrow \langle 2,1\rangle$ *and* $\omega_2 : \langle 3,1\rangle \rightarrow \langle 3,3\rangle$. *The domain abstraction* ϕ *is defined as* $\phi : (2,3) \rightarrow 4$. *In this case, the three states are mapped to two abstract states* $\langle 1,4\rangle$ *and* $\langle 4,4\rangle$. *Operators are mapped to* $\phi(\omega_1) : \langle 1,4\rangle \rightarrow \langle 4,1\rangle$ *and* $\phi(\omega_2) : \langle 4,1\rangle \rightarrow \langle 4,4\rangle$. *Therefore,* $d_\phi(\langle 1,4\rangle, \langle 4,4\rangle) = 2$ *but* $d_{ww}(\langle 1,4\rangle, \langle 4,4\rangle) = \infty$.

We record these observations in the following theorem.

Theorem 39 *Let* $d_{abs}(P,Q)$ *be the distance from* P *to* Q *computed by a projection or domain abstraction system. Then* $d_{abs}(P,Q) \leq d_{ww}(P,Q)$ *and there exist projections and domain abstractions such that* $d_{abs}(P,Q) < d_{ww}(P,Q)$ *for some* P *and* Q.

Proof. $d_{abs}(P,Q) > d_{ww}(P,Q)$ is impossible because d_{abs} is admissible and Theorem 25 established that d_{ww} is the largest distance that is guaranteed to be admissible. Examples 37 and 38 prove the second part of the theorem. □

The reasoning underlying Examples 37 and 38 apply broadly: any state-set space that imposes constraints on state-set reachability beyond those implied by the operator preconditions themselves runs the risk of having to approximate the state-set produced by applying an operator with a superset in order to enforce the extra constraints. Doing this can create paths that would not otherwise exist, which can reduce distances and make state-sets reachable that would not be reachable otherwise ("spurious states" [36]).

On the other hand, abstraction systems that exactly compute d_{ww} run a different risk: if operators are able to reduce the cardinality of a state-set, as happens in Example 37, the number of reachable state sets might become very large. In the worst case, a sequence of operator applications might lead to a state in the original state space making the reachable portion of the abstract space a superset of the original space. This can be viewed as the "problem" that h^m solves in the context of regression planning: when the cardinality of a state-set gets too small, its cardinality is increased artificially by replacing it by a superset (actually, several supersets—see the next section). Here we see that the same "problem" may occur in any abstraction system that attempts to compute d_{ww}.

4.5 Multimapping Revisited

The analogy with h^m leads to a final point that may give an advantage to abstraction systems that exactly compute d_{ww} over existing abstraction systems. When h^m reaches a state-set P whose

cardinality is too small, it does not replace it with just one superset of a sufficiently large cardinality. It enumerates all supersets of P having a sufficiently large cardinality and uses the maximum of their distances to Q as a lower bound estimate for P's distance to Q. An abstraction system could do exactly the same: instead of computing one superset (abstraction) of the given state-set P, several could be computed and the maximum taken over all the distances thus computed. When the abstract state spaces are simple, this idea is exactly equal to using multiple abstractions [17]. However, in non-simple abstract spaces, a given state (or state-set) could have multiple abstract images that are reachable from one another. This is exactly the idea of multimapping studied in Chapter 3. As we saw in Chapter 3, multimapping can result in memory savings over having multiple non-overlapping abstract spaces.

There is an additional reason to consider looking up several supersets of a given state (or state-set). If a consistent heuristic is desired (*e.g.,* if A* search is being done) doing just one lookup is more likely to result in inconsistency than if several lookups are done. Figure 4.7 illustrates this. Suppose a and b are two states (small letter inside circles A and $B_1 \cap B_2$) and ω_1(a)=b. h(a) is computed as $d_{ww}(A, G) = 3$ (the operator sequence is $\omega_1, \omega_2, \omega_3$), where A is a state-set containing a and G is the goal state-set. If h(b) is computed as $d_{ww}(B_2, G) = 1$, an inconsistency in the heuristic values will occur. If it were computed as $d_{ww}(B_1, G)$ or $max(d_{ww}(B_1, G), d_{ww}(B_2, G))$ the heuristic values would be consistent.

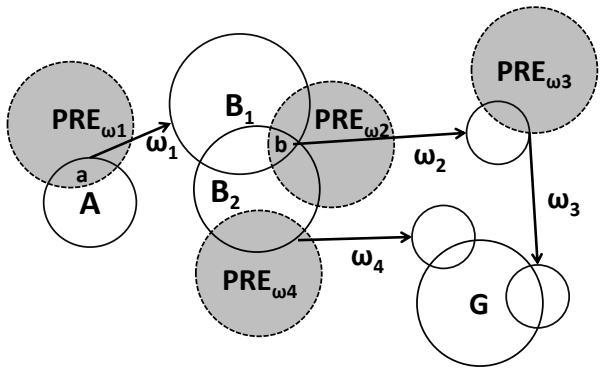


Figure 4.7: Potential for $d_{ww}$to produce an inconsistent heuristic.

4.6 Conclusions

This chapter presents the state-set search theory which investigates the scenario that a set of states are manipulated as a single state-set by a search algorithm. Using this theory, we could explain behaviour of some existing abstraction techniques and planning systems. Based on this theory, we have found that d_{ww}, a path using weak matching and weak goal testing, is the maximum admissible distance between two state-sets. This can be used to generate strong heuristics.

Chapter 5

Conclusions

In this thesis, we presented a new multimapping abstraction framework which allows a state in the original space to be mapped to multiple abstract states within a single abstract space. This technique is a framework rather than a specific method, thus there should be numerous ways to implement it. We have also developed three enhancement techniques for the framework, namely, (1) choosing an appropriate Mapping Factor, (2) remapping and (3) goal aggregation. Choosing an appropriate Mapping Factor is to choose the number of abstract states a state in the original space will be mapped to. Bad choice of Mapping Factor will degrade the quality of heuristic generated by the multimapping domain abstraction. Remapping targets the problem that the number of abstract goal states increases exponentially if multimapping is applied at every level of an abstraction hierarchy. It solves this problem by mapping abstract states which are images of the same state to one abstract state at the next higher level. Goal aggregation prevents abstract goal states from being spread out in the abstract space by deliberately mapping abstract goal states close together in the abstract space. Supported by experiment results, the implementation with all three technique together performed best among implementations with only a subset of the three techniques. To implement this framework, we have introduced two techniques: multimapping domain abstraction and Partial-State abstraction. Multimapping domain abstraction was thoroughly investigated. It is a simple extension of existing domain abstraction by using several domain abstraction functions together. We designed extensive experiments to test the performance of it in terms of CPU time and memory consumption. The results show that multimapping domain abstraction outperforms traditional domain abstraction with a single abstract space and with multiple abstract spaces. Partial-State abstraction is a new way of doing abstraction. The basic idea of Partial-State abstraction is to introduce variables in the PSVN vector, allowing it to represent a set of states. For Partial-State abstraction, we only ran limited experiments to test the technique. The results show that Partial-State cannot completely beat traditional domain abstraction. Thus this technique still needs to be further improved.

Another contribution we made in the thesis is state-set search theory. This theory investigated the scenario that a set of states in the original space is manipulated as a state-set by a search algorithm. In this theory, we formally defined and analyzed the scenario and showed that it gives a new perspective

on abstraction and planning. The most important finding of the theory is that there are naturally four ways to define the distance between state-sets. Among these four kinds of distance, we find that the cost of the shortest weak path between two state-sets P and Q, namely $d_{ww}(P, Q)$, is the largest possible admissible estimate for the true distance between any pair of states in P and Q respectively.

5.1 Limitations

In the multimapping abstraction framework, for each original state, the multiple abstract states for it are required to be contained in the same abstract space. However, this requires careful design of the multimapping abstraction function and knowledge of the domain this technique is working on. If we cannot design an abstraction function meeting this requirement, the implementation would just act like multiple independent abstractions. In this case, the features and advantages of multimapping abstraction will disappear.

Combining enhancement techniques can also be problematic. As discussed in Section 3.2.4, combining goal aggregation and remapping will be impossible under constraints of certain granularity. The performance of multimapping greatly depends on those enhancement techniques and losing any of them will cause noticeable increase in CPU time and memory usage.

Another limitation concerns the mapping factor. The choosing of a mapping factor is quite complicated because both high and low mapping factor will causes heuristic values to decrease. We do not have a theory of choosing a good mapping factor so it must be chosen based on experimental results. The best mapping factor could depend on the specific domain. The practice of using 3 as mapping factor for all domains is not guaranteed to be the best approach to implement multimapping abstraction but only for convenience for our experiments.

The limitation for state-set theory is that even though the d_{ww} is guaranteed to be the best possible admissible abstract distance measurement, its implementation could be problematic because of the shrinking problem described in Section 4.4 and Section 3.4.2. The state-sets along a weak path could contain fewer and fewer states and finally contain only one state. In this case, the search is done in the original space and state-sets can no longer be used as abstract states.

5.2 Final Words

Heuristic search is an important method to solve many real world problems. Generating better heuristics is a powerful way to speed up heuristic search. In this thesis, we first developed a new framework for an abstraction system. With strong evidence from extensive experiments, an implementation of our system proved to outperform two existing popular abstraction systems in terms of both CPU time and memory consumption. We also developed the state-set search theory which helps us to better understand abstraction techniques and planning systems.

Bibliography

[1] Christer Bäckström. Expressive equivalence of planning formalisms. *Artificial Intelligence*, 76:17–34, 1995.

[2] Neil Burch and Robert C. Holte. Automatic move pruning in general single-player games. In Daniel Borrajo, Maxim Likhachev, and Carlos Linares López, editors, *SOCS*. AAAI Press, 2011.

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

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[5] Joseph C. Culberson and Jonathan Schaeffer. Efficiently searching the 15-puzzle. Technical Report 94-08, Department of Computing Science, University of Alberta, 1994.

[6] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Proceedings of the Canadian Conference on Artificial Intelligence*, volume 1081 of *LNAI*, pages 402–416. Springer, 1996.

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

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[9] Harry Dweighter. Problem E2569. *American Mathematical Monthly*, 82:1010, 1975.

[10] Stefan Edelkamp. Planning with pattern databases. In *PROCEEDINGS OF THE 6TH EUROPEAN CONFERENCE ON PLANNING (ECP-01)*, pages 13–24, 2001.

[11] Stefan Edelkamp and Richard E. Korf. The branching factor of regular search spaces. In *AAAI/IAAI*, pages 299–304, 1998.

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

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[14] Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*, pages 140–149, 2000.

[15] István Hernádvölgyi and R. C. Holte. Steps towards the automatic creation of search heuristics. Technical Report TR04-02, Department of Computing Science, University of Alberta, 2004.

[16] István Hernádvölgyi and Robert Holte. PSVN: A vector representation for production systems. Technical Report TR-99-04, Department of Computer Science, University of Ottawa, 1999.

[17] Robert C. Holte, Ariel Felner, Jack Newton, Ram Meshulam, and David Furcy. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, 170(16-17):1123–1136, 2006.

[18] Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, volume 3607 of *LNAI*, pages 121–133. Springer, 2005.

[19] Robert C. Holte and Istvn T. Herndvolgyi. A space-time tradeoff for memory-based heuristics. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99*, pages 704–709. AAAI Press, 1999.

[20] Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *ICAPS*, pages 122–131, 2004.

[21] Robert C. Holte, M. B. Perez, Robert M. Zimmer, and Alan J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI 1996)*, pages 530–535, 1996.

[22] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[23] Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence (AAAI 1997)*, pages 700–705, 1997.

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

[25] Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-a*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[26] Bhaskara Marthi, Stuart J. Russell, and Jason Wolfe. Angelic semantics for high-level actions. In *ICAPS*, pages 232–239, 2007.

[27] Nils J. Nilsson and Richard E. Fikes. Strips: A new approach to the application of theorem proving to problem solving. *Artifical Intelligence*, 2(3-4):189–208, 1971. this paper about strips is the foundation of any further planning research.

[28] Bo Pang and Robert C. Holte. State-set search. In *Symposium on Combinatorial Search*, 2011.

[29] Bo Pang and Robert C. Holte. Multimapping abstractions and hierarchical heuristic search. In *Symposium on Combinatorial Search*, 2012.

[30] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI 1973)*, pages 12–17, 1973.

[31] John K. Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[32] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.

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

[34] Uzi Zahavi, Ariel Felner, Robert C. Holte, and Jonathan Schaeffer. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4-5):514–540, 2008.

[35] Sandra Zilles and Robert C. Holte. Downward path preserving state space abstractions (extended abstract). In Vadim Bulitko and J. Christopher Beck, editors, *Symposium on Abstraction, Reformulation, and Approximation*, 2009.

[36] Sandra Zilles and Robert C. Holte. The computational complexity of avoiding spurious states in state space abstraction. *Artificial Intelligence*, 174(14):1072–1092, 2010.

Appendix A

Abstractions for Large Domains

A.1 15-Puzzle

ϕ_2 to ϕ_9 are all the same for all abstractions for 15-Puzzle. We only describe ϕ_1 in the following subsections.

ϕ_2 to ϕ_9

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\phi_2(s)$			1													
$\phi_3(s)$			1	1												
$\phi_4(s)$			1	1	1											
$\phi_5(s)$			1	1	1	1										
$\phi_6(s)$			1	1	1	1	1									
$\phi_7(s)$			1	1	1	1	1	1								
$\phi_8(s)$			1	1	1	1	1	1	1							
$\phi_9(s)$			1	1	1	1	1	1	1	1						

A.1.1 Domain Abstraction

Domain Abstraction #1

0	1	1	3
2	1	1	4
2	1	1	5
6	7	8	9

Domain Abstraction #2

0	2	2	3
1	1	1	4
1	1	1	5
6	7	8	9

Domain Abstraction #3

0	1	1	1
2	1	1	1
2	3	4	5
6	7	8	9

Domain Abstraction #4

0	2	2	3
1	1	4	5
1	1	6	7
1	1	8	9

Domain Abstraction #5

0	1	1	3
2	2	1	4
1	1	1	5
6	7	8	9

A.1.2 Multimapping Abstraction

Multimapping Abstraction #1

0	1	1	3
2	1	1	4
2	1	1	5
6	7	8	9

0	1	1	3
1	2	1	4
2	1	1	5
6	7	8	9

0	2	1	3
1	1	1	4
2	1	1	5
6	7	8	9

Multimapping Abstraction #2

0	2	2	3
1	1	1	4
1	1	1	5
6	7	8	9

0	1	2	3
2	1	1	4
1	1	1	5
6	7	8	9

0	1	2	3
1	2	1	4
1	1	1	5
6	7	8	9

Multimapping Abstraction #3

0	1	1	1
2	1	1	1
2	3	4	5
6	7	8	9

0	1	1	1
1	2	1	1
2	3	4	5
6	7	8	9

0	2	1	1
1	1	1	1
2	3	4	5
6	7	8	9

Multimapping Abstraction #4

0	2	2	3
1	1	4	5
1	1	6	7
1	1	8	9

0	1	2	3
2	1	4	5
1	1	6	7
1	1	8	9

0	1	2	3
1	2	4	5
1	1	6	7
1	1	8	9

Multimapping Abstraction #5

0	1	1	3
2	2	1	4
1	1	1	5
6	7	8	9

0	2	1	3
1	2	1	4
1	1	1	5
6	7	8	9

0	2	1	3
2	1	1	4
1	1	1	5
6	7	8	9

A.1.3 Multiple Abstraction

From ϕ_2 to ϕ_7:

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\phi_2(s)$			1													
$\phi_3(s)$			1	1												
$\phi_4(s)$			1	1	1											
$\phi_5(s)$			1	1	1	1										
$\phi_6(s)$			1	1	1	1	1									
$\phi_7(s)$			1	1	1	1	1	1								

Multiple Abstraction #1

0	1	1	1
1	1	1	1
1	2	3	4
5	6	7	8

0	2	3	4
5	6	7	8
1	1	1	1
1	1	1	1

0	1	1	2
1	1	1	3
1	1	1	4
5	6	7	8

Multiple Abstraction #2

0	1	1	2
1	1	1	3
1	1	1	4
5	6	7	8

0	1	2	1
3	4	5	1
6	7	8	1
1	1	1	1

0	1	1	1
1	1	1	1
1	2	3	4
5	6	7	8

Multiple Abstraction #3

0	1	1	1
1	1	1	2
3	1	1	4
5	6	7	8

0	1	2	3
4	5	6	1
1	7	8	1
1	1	1	1

0	1	2	3
1	1	1	4
1	1	1	5
1	6	7	8

Multiple Abstraction #4

0	1	2	3
1	1	1	4
1	1	1	5
1	6	7	8

0	1	1	1
2	3	4	1
5	6	7	1
8	1	1	1

0	1	1	2
1	1	3	4
1	1	5	6
1	1	7	8

Multiple Abstraction #5

0	1	1	2
1	1	3	4
1	1	5	6
1	1	7	8

0	1	2	1
3	4	1	1
5	6	1	1
7	8	1	1

0	1	1	1
1	1	1	2
3	1	1	4
5	6	7	8

A.2 Glued 15-Puzzle

ϕ_2 to ϕ_9 are all the same for all abstractions for 15-Puzzle. We only describe ϕ_1 in the following subsections.

ϕ_2 to ϕ_9

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\phi_2(s)$			1													
$\phi_3(s)$			1	1												
$\phi_4(s)$			1	1	1											
$\phi_5(s)$			1	1	1	1										
$\phi_6(s)$			1	1	1	1	1									
$\phi_7(s)$			1	1	1	1	1	1								
$\phi_8(s)$			1	1	1	1	1	1	1							
$\phi_9(s)$			1	1	1	1	1	1	1	1						

A.2.1 Domain Abstraction

Domain Abstraction #1

0	1	1	3
2	1	1	4
2	1	1	5
6	7	8	9

Domain Abstraction #2

0	2	2	3
1	1	1	4
1	1	1	5
6	7	8	9

Domain Abstraction #3

0	1	1	1
2	1	1	1
2	3	4	5
6	7	8	9

Domain Abstraction #4

0	2	2	1
1	1	1	1
3	4	1	5
6	7	8	9

Domain Abstraction #5

0	1	1	3
2	2	1	4
1	1	1	5
6	7	8	9

A.2.2 Multimapping Abstraction

Multimapping Abstraction #1

0	1	1	3
2	1	1	4
2	1	1	5
6	7	8	9

0	1	1	3
1	2	1	4
2	1	1	5
6	7	8	9

0	2	1	3
1	1	1	4
2	1	1	5
6	7	8	9

Multimapping Abstraction #2

0	2	2	3
1	1	1	4
1	1	1	5
6	7	8	9

0	1	2	3
2	1	1	4
1	1	1	5
6	7	8	9

0	1	2	3
1	2	1	4
1	1	1	5
6	7	8	9

Multimapping Abstraction #3

0	1	1	1
2	1	1	1
2	3	4	5
6	7	8	9

0	1	1	1
1	2	1	1
2	3	4	5
6	7	8	9

0	2	1	1
1	1	1	1
2	3	4	5
6	7	8	9

Multimapping Abstraction #4

0	2	2	1
1	1	1	1
3	4	1	5
6	7	8	9

0	1	2	1
2	1	1	1
3	4	1	5
6	7	8	9

0	1	2	1
1	2	1	1
3	4	1	5
6	7	8	9

Multimapping Abstraction #5

0	1	1	3
2	2	1	4
1	1	1	5
6	7	8	9

0	2	1	3
1	2	1	4
1	1	1	5
6	7	8	9

0	2	1	3
2	1	1	4
1	1	1	5
6	7	8	9

A.2.3 Multiple Abstraction

Level 1 abstraction ϕ_1 is the same as for multimapping abstraction.

A.3 14-Pancake

ϕ_2 to ϕ_8 are all the same for all abstractions for 15-Puzzle. We only describe ϕ_1 in the following subsections.

ϕ_2 to ϕ_8

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\phi_2(s)$			1												
$\phi_3(s)$			1	1											
$\phi_4(s)$			1	1	1										
$\phi_5(s)$			1	1	1	1									
$\phi_6(s)$			1	1	1	1	1								
$\phi_7(s)$			1	1	1	1	1	1							
$\phi_8(s)$			1	1	1	1	1	1	1						

A.3.1 Domain Abstraction

Domain Abstraction #1

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1(s)$	1	1	1	1	1	1	2	2	3	4	5	6	7	8

Domain Abstraction #2

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1(s)$	1	1	2	2	2	2	2	2	3	4	5	6	7	8

Domain Abstraction #3

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1(s)$	3	1	1	1	1	1	1	2	2	4	5	6	7	8

Domain Abstraction #4

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1(s)$	3	1	1	2	2	2	2	2	2	4	5	6	7	8

Domain Abstraction #5

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1(s)$	3	4	1	1	1	1	1	1	2	2	5	6	7	8

A.3.2 Multimapping

Multimapping Abstraction #1

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	1	1	1	1	2	2	3	4	5	6	7	8
$\phi_1^2(s)$	2	1	1	1	1	1	1	2	3	4	5	6	7	8
$\phi_1^3(s)$	2	2	1	1	1	1	1	1	3	4	5	6	7	8

Multimapping Abstraction #2

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	2	2	2	2	2	2	3	4	5	6	7	8
$\phi_1^2(s)$	1	2	2	1	1	1	1	1	3	4	5	6	7	8
$\phi_1^3(s)$	2	2	1	1	2	2	2	2	3	4	5	6	7	8

Multimapping Abstraction #3

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	1	1	2	2	1	1	3	4	5	6	7	8
$\phi_1^2(s)$	2	1	1	1	1	2	1	1	3	4	5	6	7	8
$\phi_1^3(s)$	2	2	1	1	1	1	1	1	3	4	5	6	7	8

Multimapping Abstraction #4

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	1	2	2	1	1	1	3	4	5	6	7	8
$\phi_1^2(s)$	2	1	1	1	2	1	1	1	3	4	5	6	7	8
$\phi_1^3(s)$	1	2	2	1	1	1	1	1	3	4	5	6	7	8

Multimapping Abstraction #5

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	2	2	1	1	1	1	3	4	5	6	7	8
$\phi_1^2(s)$	2	1	1	2	1	1	1	1	3	4	5	6	7	8
$\phi_1^3(s)$	1	2	2	1	1	1	1	1	3	4	5	6	7	8

A.3.3 Multiple Abstraction

Level 1 abstraction ϕ_1 is the same as for multimapping abstraction.

A.4 (15,4)-Topspin

ϕ_2 to ϕ_8

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\phi_2(s)$			1												
$\phi_3(s)$			1	1											
$\phi_4(s)$			1	1	1										
$\phi_5(s)$			1	1	1	1									
$\phi_6(s)$			1	1	1	1	1								
$\phi_7(s)$			1	1	1	1	1	1							
$\phi_8(s)$			1	1	1	1	1	1	1						

A.4.1 Domain Abstraction

We used the same domain abstractions as used in 14-Pancake

A.4.2 Multimapping Abstraction

Multimapping Abstraction #1

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	1	1	1	1	2	2	3	4	5	6	7	8
$\phi_1^2(s)$	1	1	1	1	2	2	1	1	3	4	5	6	7	8
$\phi_1^3(s)$	1	1	1	3	2	1	1	2	3	4	5	6	7	8

Multimapping Abstraction #2

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	1	1	2	2	2	2	2	2	3	4	5	6	7	8
$\phi_1^2(s)$	2	2	1	1	2	2	2	2	3	4	5	6	7	8
$\phi_1^3(s)$	1	2	2	2	1	2	2	2	3	4	5	6	7	8

Multimapping Abstraction #3

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	3	1	1	1	1	1	1	2	2	4	5	6	7	8
$\phi_1^2(s)$	3	1	1	1	1	2	2	1	1	4	5	6	7	8
$\phi_1^3(s)$	3	1	1	1	2	1	1	1	2	4	5	6	7	8

Multimapping Abstraction #4

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	3	1	1	2	2	2	2	2	2	4	5	6	7	8
$\phi_1^2(s)$	3	2	2	1	1	2	2	2	2	4	5	6	7	8
$\phi_1^3(s)$	3	1	2	2	2	1	2	2	2	4	5	6	7	8

Multimapping Abstraction #5

s	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\phi_1^1(s)$	3	4	1	1	1	1	1	1	2	2	5	6	7	8
$\phi_1^2(s)$	3	4	1	1	1	1	2	2	1	1	5	6	7	8
$\phi_1^3(s)$	3	4	1	1	1	2	1	1	1	2	5	6	7	8

A.4.3 Multiple Abstraction

Level 1 abstraction ϕ_1 is the same as for multimapping abstraction.

A.5 (12,3)-Blocks World

ϕ_2 to ϕ_8

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_2(s)$		1										
$\phi_3(s)$		1	1									
$\phi_4(s)$		1	1	1								
$\phi_5(s)$		1	1	1	1							
$\phi_6(s)$		1	1	1	1	1						
$\phi_7(s)$		1	1	1	1	1	1					
$\phi_8(s)$		1	1	1	1	1	1	1				

A.5.1 Domain Abstraction

Domain Abstraction #1

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1(s)$	1	1	1	1	1	2	3	4	5	6	7	8

Domain Abstraction #2

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1(s)$	2	3	4	5	6	7	8	1	1	1	1	1

Domain Abstraction #3

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1(s)$	2	3	4	5	1	1	1	1	1	6	7	8

Domain Abstraction #4

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1(s)$	1	2	1	3	1	4	1	5	1	6	7	8

Domain Abstraction #5

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1(s)$	2	1	3	1	4	1	5	1	6	1	7	8

A.5.2 Multimapping Abstraction

Multimapping Abstraction #1

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	1	1	1	1	1	2	3	4	5	6	7	8
$\phi_1^2(s)$	2	1	1	1	1	1	3	4	5	5	6	8
$\phi_1^3(s)$	1	2	1	1	1	1	3	4	5	6	7	8

Multimapping Abstraction #2

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	2	3	4	5	6	7	8	1	1	1	1	1
$\phi_1^2(s)$	1	3	4	5	6	7	8	2	1	1	1	1
$\phi_1^3(s)$	1	3	4	5	6	7	8	1	2	1	1	1

Multimapping Abstraction #3

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	2	3	4	5	1	1	1	1	1	6	7	8
$\phi_1^2(s)$	1	3	4	5	2	1	1	1	1	6	7	8
$\phi_1^3(s)$	1	3	4	5	1	1	2	1	1	6	7	8

Multimapping Abstraction #4

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	1	2	1	3	1	4	1	5	1	6	7	8
$\phi_1^2(s)$	2	1	1	3	1	4	1	5	1	6	7	8
$\phi_1^3(s)$	1	1	2	3	1	4	1	5	1	6	7	8

Multimapping Abstraction #5

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	2	1	3	1	4	1	5	1	6	1	7	8
$\phi_1^2(s)$	1	2	3	1	4	1	5	1	6	1	7	8
$\phi_1^3(s)$	1	1	3	1	4	1	5	2	6	1	7	8

A.5.3 Multiple Abstraction

Level 1 abstraction ϕ_1 is the same as for multimapping abstraction.

A.5.4 Multimapping Abstraction (GARM)

Multimapping Abstraction #1

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	3	4	5	6	7	8	1	1	1	1	1	2
$\phi_1^2(s)$	3	4	5	6	7	8	1	1	1	1	2	1
$\phi_1^3(s)$	3	4	5	6	7	8	1	1	1	2	1	1

Multimapping Abstraction #2

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	1	1	1	3	4	5	6	7	8	1	1	2
$\phi_1^2(s)$	1	1	1	3	4	5	6	7	8	1	2	1
$\phi_1^3(s)$	1	1	1	3	4	5	6	7	8	2	1	1

Multimapping Abstraction #3

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	1	2	3	1	1	1	7	8	9	1	1	2
$\phi_1^2(s)$	1	2	3	1	1	1	7	8	9	1	2	1
$\phi_1^3(s)$	1	2	3	1	1	1	7	8	9	2	1	1

Multimapping Abstraction #4

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	3	1	1	1	4	5	6	7	8	1	1	2
$\phi_1^2(s)$	3	1	1	1	4	5	6	7	8	1	2	1
$\phi_1^3(s)$	3	1	1	1	4	5	6	7	8	2	1	1

Multimapping Abstraction #5

s	1	2	3	4	5	6	7	8	9	10	11	12
$\phi_1^1(s)$	3	4	5	6	7	1	1	1	8	1	1	2
$\phi_1^2(s)$	3	4	5	6	7	1	1	1	8	1	2	1
$\phi_1^3(s)$	3	4	5	6	7	1	1	1	8	2	1	1