



Université d'Ottawa • University of Ottawa



Université d'Ottawa · University of Ottawa

FACULTÉ DE ÉTUDES SUPÉRIEURES
ET POSTDOCTORALES

FACULTY OF GRADUATE AND
POSTDOCTORAL STUDIES

Istvan HERNADVOLYGI

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

Ph.D. (Computer Science)

GRADE - DEGREE

School of Information, Technology and Immunology

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Automatically Generated Lower Bounds for Search

R. Holte

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

S. Boyd

S. Matwin

J. Oommen

W. Zhang

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES
SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE
AND POSTDOCTORAL STUDIES

Automatically Generated Lower Bounds for Search

István T. Hernádvölgyi

School of Information Technology and Engineering
University of Ottawa



May 2004

A Thesis

Submitted to the School of Graduate Studies and Research
of the University of Ottawa
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
in Computer Science¹

¹The Doctoral Program in Computer Science is a joint program with Carleton University, administered by the Ottawa Carleton Institute for Computer Science



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-494-01708-2

Our file Notre référence

ISBN: 0-494-01708-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Dedication

I dedicate this work in memory of my mother, *Zsóka*, my grandma, *Baba* and my dear friend *Zombi*. I wish we could celebrate together.

Acknowledgments

I would like to sincerely thank Dr. Robert C. Holte, my thesis supervisor, for his support, encouragement and friendship. I hope one day I can follow in your footsteps, Rob.

I would also like to thank my many friends at the University and at the Rideau Canoe Club who made my stay at the University of Ottawa really fantastic.

My thesis work was supported in part by a research grant and a doctoral scholarship provided by the Natural Sciences and Engineering Research Council of Canada.

Abstract

Heuristic search algorithms (eg. A^* and IDA^*) with accurate lower bounds can solve impressively large problems optimally. Most lower bounds, such as the well known Manhattan Distance heuristic for the sliding-tile puzzles or the Assignment Problem lower bound for the Asymmetric Traveling Salesman problem, are the products of human ingenuity and insight. An alternative approach to obtain lower bounds is to precalculate shortest distances in an abstraction of the original search space which is derived automatically and store the bounds in *pattern databases* (look-up tables). This latter technique, based on the ideas of Culberson and Schaeffer, gained popularity when Korf for the first time solved random instances of Rubik's Cube using pattern databases. While researchers were pushing for solving larger and larger problems, the fact that there exist a very large number of abstract spaces that can provide lower bounds was overlooked. This thesis fills this gap in research by investigating the search performance of lower bounds derived from abstractions. We also use the results of this analysis to automatically derive high performance pattern databases.

First, we establish a very predictable trade-off between search speed and the number of entries in the pattern database. Second, we derive simple statistics that can predict the search performance of pattern databases without performing actual searches in the original state space. Using these results, we derive high performance pattern databases to search for macro-operators and to solve challenging instances of the well known Sequential Ordering Problem (SOP). Macro-search is a good candidate to showcase automatically derived lower bounds since there are many search spaces and each needs a different lower bound. The SOP is an NP-hard optimization problem. We were able to solve an unsolved instance from the TSPLIB. This required a greedy search in the space of abstractions to find a sufficiently accurate lower bound and several novel enhancements to the basic branch and bound algorithm.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Contributions	4
1.3	Thesis Outline	6
2	Preliminaries	8
2.1	State-space Search	8
2.2	Algorithms	15
2.2.1	A*	16
2.2.2	IDA*	17
2.2.3	DFBB	18
2.3	Lower Bounds From Abstraction	20
2.4	Mathematical Foundation	29
2.5	Domain Abstraction	31
2.6	Implementation Framework	37
3	Lower Bounds from Domain Abstractions	43
3.1	Experiments	44
3.1.1	The 3×3 Puzzle	45

3.1.2	Experimental Setup	46
3.2	Space-time Trade-off	48
3.3	High-performance Pattern Databases	57
3.4	Chapter Summary	68
4	Searching for Macro Operators	71
4.1	Subgoals and Macros	72
4.2	Macro-search	75
4.2.1	Uninformed Search	76
4.2.2	Heuristic Search	77
4.3	Reducing the Number of Subgoals	78
4.4	The Rubik's Cube	80
4.5	Domain Abstractions	82
4.6	Results	86
4.7	Chapter Summary	94
5	Application to the Sequential Ordering Problem	96
5.1	The Sequential Ordering Problem	97
5.2	State Space	98
5.3	Lower Bounds	103
5.3.1	The Abstract Space	103
5.3.2	Calculating the Lower Bounds	112
5.3.3	Choosing Domain Abstractions	114
5.3.4	Generating Predecessors in S'	117
5.4	Search	119
5.4.1	Bidirectional Depth-first Branch and Bound	120

5.4.2	Recursive Depth-first Branch and Bound	122
5.4.3	Level-wise Transposition Tables	123
5.4.4	Local Search and Perfect Hashing	124
5.5	Experimental Evaluation	125
5.5.1	Test Sets	126
5.5.2	Error Functions	128
5.5.3	TSPLIB Problems	131
5.5.4	Space-time Trade-off	133
5.5.5	Cost Convergence	135
5.5.6	Constrainedness vs. Search Speed	137
5.5.7	DFBB, RecDFBB and Transposition Tables	139
5.6	Additional Constraints	140
5.6.1	Disjunctive Precedence Constraints	141
5.6.2	Soft Precedence Constraints	142
5.7	Chapter Summary	143
6	Related Work	147
6.1	Abstractions	147
6.2	Pattern Databases	150
6.3	Performance Analysis	153
6.4	Applications	154
6.4.1	Macro-operators	154
6.4.2	The Sequential Ordering Problem	155
6.5	Using Memory to Speed-up Search	157
7	Limitations	159

7.1	Non-surjective Homomorphisms	159
7.2	Limitations in Approach	163
7.3	Remaining Open Questions	166
8	Conclusions	168
8.1	Results and Reflections	168
8.2	Future Work	172
8.3	Final Word	173
A	Checking for Spurious States	175
B	Perfect Minimal Hash Function	178
	Bibliography	183

List of Figures

2.1	The 2×2 puzzle.	13
2.2	State space corresponding to the 2×2 puzzle.	14
2.3	Brute-force search tree for the 2×2 puzzle.	15
2.4	State space homomorphism.	22
2.5	Transpose of the Abstract Space.	25
2.6	A Non-surjective State Space Homomorphism.	28
2.7	Shortcut Abstract Paths	30
2.8	Fringe and Corner Patterns.	32
2.9	A Domain Abstraction for the 4×4 Puzzle.	33
2.10	Abstract State Space for the 2×2 Puzzle.	34
2.11	A Non-surjective Abstraction for the 2×2 Puzzle.	35
2.12	A Framework for state-space Search.	37
3.1	3×3 puzzle.	45
3.2	Distribution of Optimal Solution Distances for the 3×3 Puzzle .	46
3.3	Size of Pattern Database vs. Nodes Expanded	49
3.4	Pattern Database Size and Range of Nodes Expanded for IDA*. .	53
3.5	Pattern Database Size and Range of Nodes Expanded for A*. . .	53
3.6	Size of Pattern Database (x -axis) vs. CPU ticks (y -axis)	55

3.7	d vs. b at Even Levels in the 3×3 Puzzle	59
3.8	K&R Prediction vs. Actual Number of Nodes Expanded with IDA*.	63
3.9	$ave(\phi)$ vs. Actual Number of Nodes Expanded with IDA*.	65
3.10	$h_0(\phi)$ vs. Actual Number of Nodes Expanded with IDA*.	66
3.11	$K\&R(\phi)$ vs. Actual Number of Nodes Expanded with A*.	68
3.12	$ave(\phi)$ vs. Actual Number of Nodes Expanded with A*.	69
3.13	$h_0(\phi)$ vs. Actual Number of Nodes Expanded with A*.	70
4.1	Macro-table Entries Corresponding to Subgoal 1.	74
4.2	Macro-table Entries Corresponding to Subgoal 2.	74
4.3	State-spaces Corresponding to Subgoals 1 and 2.	76
4.4	The <i>Rubik's Cube</i>	81
4.5	Corner-cubie Orientations.	82
4.6	Subgoal States for the Rubik's Cube.	83
4.7	Histograms of solution lengths obtained for 10,000 random cube states	88
5.1	A Helicopter Routing Problem	99
5.2	Precedence Graph for the Helicopter Routing Problem	100
5.3	State-space Lattice for the Helicopter Routing Problem	103
5.4	State Space Lattice Abstraction	104
5.5	The States of $S'_4 = \phi_4(S_4)$	105
5.6	The States of $S'_5 = \phi_5(S_5)$	106
5.7	Edges Connecting State Clusters	107
5.8	Expanding the Abstract State-space	113
5.9	Vertex Order Changes in the Incumbent	120
5.10	Nodes Expanded vs. Memory	134

5.11	Solution Cost vs. Nodes Expanded for 500×500 -3	136
7.1	Dual Representation for the 2×2 Puzzle.	161
7.2	Abstract Space for the “Dual” Encoding of the 2×2 Puzzle. . . .	162
A.1	Abstract State s'_i	176
B.1	Pebbles in Boxes.	180

List of Tables

3.1	Abstraction Granularities for the 3×3 Puzzle.	47
3.2	Rank Correlation for All Pattern Databases with IDA*.	64
3.3	Rank Correlation for All Pattern Databases with A*.	67
4.1	Macro table $T_{18,1}$: Statistics	89
4.2	Macro table T_6 : Statistics	90
5.1	Generating Lower Bounds for p43.4	129
5.2	Results of Solving 500×500 3 with Different Databases	135
5.3	Nodes Expanded and CPU Times with Different Size Pattern Databases	145
5.4	Nodes Expanded and CPU Times with DFBB, RecDFBB and RecDFBB with Transposition Tables	146
B.1	Picking Pebbles in Lexicographic Order.	180

Chapter 1

Introduction

In this thesis, we are investigating the automatic generation of lower bounds¹ for heuristic search algorithms, such as A* [HNR68], IDA* [Kor85b] and branch and bound [LW66]. With inaccurate lower bounds these heuristic search algorithms are not much better than blind search. With accurate lower bounds, however, they can solve impressively large problems (for example the Rubik's Cube [Kor97]). Traditionally lower bounds were designed specifically for a given problem and were usually the products of human ingenuity. Examples include the Minimum Weight Spanning Tree lower bound for the Traveling Salesman Problem [HK70] or the Manhattan Distance heuristic for the $n \times n$ sliding tile puzzles. Recently, however, lower bounds derived from abstractions of the original problems were shown to be very effective, even outperforming human designed lower

¹Lower bounds used by heuristic search algorithms in Artificial Intelligence are also called *heuristics*. In this thesis, we will prefer to use the term *lower bound* to avoid confusion. In Operations Research, a *heuristic solution* corresponds to a suboptimal solution, which in the case of minimization problems, is also an upper bound.

bounds [CS98, Kor97, KF02]. These lower bounds correspond to solutions in abstract spaces, which are induced by simple abstraction functions. The abstract solution costs (the lower bounds) are calculated by an exhaustive expansion and stored in a lookup table called the *pattern database*.

1.1 Motivation

Our abstractions are simple functions that relabel state representations. We took the “pattern abstraction” idea from Culberson and Schaeffer [CS98] and generalized it. In previous work [CS98, Kor97, Ede01a, KF02] only a handful of pattern databases were involved and these were picked by the authors. We, however, viewed this abstraction technique as an enabling technology capable of generating a very large number of lower bounds (often in the millions) which should be exploited to obtain high performance heuristics. We were particularly interested in the following questions.

- How different are pattern databases with respect to search performance?
- What are the features that make one pattern database perform better than another?
- Can one efficiently and automatically design or obtain a pattern database which performs better?
- Are there applications where automatically generated abstractions are preferable?

- How do pattern database heuristics perform in domains where operators have different costs?

This last question arose because in previous work pattern databases were used to solve problems where all operators had unit costs.

Automatically derived precalculated lower bounds to speed up search were considered by Frieditis [Pri90] for STRIPS [FN71] planning and by Christofides, Mingozzi and Toth [CMT81] for solving routing problems. It was, however, the work of Culberson and Schaeffer [CS94] which awoke interest in memory based lower bound heuristics. In particular, Korf [Kor97] solved random instances of the Rubik's Cube for the first time and shortly after Korf and Felner [KF02] solved random instances of the much larger 5×5 puzzle with pattern databases. Instead of solving impressively large problems, we were more interested in studying pattern databases and using them in a more general setting where high performance abstractions are obtained automatically and the fact that there is a vast number of abstractions is considered in the derivation of the abstraction.

We were also interested in finding applications where we can make a clear case for using automatically generated heuristics instead of human derived ones. In macro-search there are several different search spaces where each space would have to be searched with a different heuristic. Therefore macro-search was an ideal application to showcase automatically generated heuristics. The Sequential Ordering Problem (SOP) is a well known scheduling problem, based on the Traveling Salesman Problem, which proved to be very hard. In fact there are small SOPs (with only 40-50 vertices) which are still unsolved today [TSP]. Even powerful techniques, such as Lagrangian and Linear Programming relaxations,

were not good enough to solve some small instances. Hence the SOP was a good candidate to put our automatically generated lower bounds to the test.

1.2 Contributions

Our thesis makes the following research contributions.

- For the first time we performed experiments which involved a large number of pattern database heuristics and from our results we
 - established a space-time trade-off originally conjectured by Korf [Kor97].
 - identified simple statistics which can be used to identify high performance pattern databases without searches in the original state space.

The early results were published in [HH99b, HH00].

- For the first time we used automatically generated lower bounds to search for optimal macro-operators. Previously macro-operators were obtained by uninformed search. We also made two novel modifications to macro-search.
 - Due to the speedup obtained by our technique, we were able to reduce the number of subgoals and thereby obtain much better quality solutions.
 - We solved some of the subgoals on-demand by heuristic search with our automatically generated pattern databases while others were solved by the usual macro-composition.

This work was published in [Her01, Her00].

- We extended abstraction techniques to take costs into account². We also generalized the “pattern abstraction” of Culberson and Schaeffer [CS98] and thus we are able to generate a larger class of abstractions.
- We discovered a frequently occurring phenomenon that was overlooked in previous research. *Non-surjective* abstractions could result in abstract spaces that are more difficult to search than the original spaces. We identified some causes and for our applications we also suggested solutions.
- We chose the Sequential Ordering Problem (SOP), a well known optimization problem, as our testbed. To tackle this very hard problem, we also made the following research contributions.
 - We solved one unsolved instance of a SOP from the TSPLIB [TSP].
 - We devised a novel way of building pattern databases which is applicable to solve a number of optimization problems where the state space exhibits a certain level structure (there is a different abstraction for each level).
 - For the first time, we used greedy search in the space of pattern databases to obtain high performance pattern databases.
 - We invented two novel search algorithms. Bidirectional Depth-first Branch and Bound (BiDFBB) is a general search algorithm applicable to search state spaces, if the inverses of the operators are known. Recursive Depth-first Branch and Bound (RecDFBB) is specifically designed for the SOP but would also be suitable to solve problems that

²While Christofides *et al.* [CMT81] generated memory based lower bounds for the first time for state spaces with operator costs, our abstractions are substantially different.

exhibit a certain recursive property. In particular, it is applicable to solve problems which are derived from the Traveling Salesman Problem (TSP).

This work was published in [Her03a].

1.3 Thesis Outline

- Chapter 2: We introduce our abstraction mechanism and describe how we build pattern databases. We also define our search framework, which was used to implement our search tools.
- Chapter 3: We report our experiments which involve thousands of pattern databases and were designed to gain insight into the space-time trade-off (how much doubling memory to store the pattern database speeds up search) and to evaluate simple statistics that can predict the relative search performance (rank) of pattern databases without performing actual searches in the original state space.
- Chapter 4: In macro-search a solution is confined to pass through a sequence of subgoals. Each subgoal is associated with a set of macro-operators and, as far as search is concerned, each subgoal also represents a search space. Because of the many state spaces, automatic generation of search heuristics is preferable. We chose the Rubik's Cube as our testbed. In this chapter we detail our experiments and discuss our results. We were able to find macro-operators more efficiently and we also improved on solution quality by reducing the number of subgoals.

- Chapter 5: We describe our new way of building pattern databases with greedy search and our novel search algorithms. The testbed is the Sequential Ordering Problem, which is also a well studied problem in Scheduling and in Operations Research. We also present the results of our experiments that show the effectiveness of our techniques.
- Chapter 6: We describe related research.
- Chapter 7: This chapter discusses the limitations of our work. We pay particular attention to non-surjective abstractions.
- Chapter 8: We provide a brief reflection of our contributions and discuss future work.

Chapter 2

Preliminaries

In this chapter we describe the algorithms and closely related work which provide the context of our thesis and therefore are essential to understand and interpret our results. We define state space search, introduce the standard algorithms that will be used in the thesis and discuss the domain abstractions which induce our heuristics. We also describe a general framework that comprises the implementation of our search tools.

2.1 State-space Search

State-space search is a general problem solving paradigm. A problem is modeled by successive state changes that lead to a goal state from a given start state. It is applicable if we can encode state change as a successor generator $\Delta(s)$ over the states s and can identify the goal state g . In terms of Δ , an initial state and a cost function we can define our problem space.

Definition 1 (State Space) *Given a successor generator $\Delta(s)$ over states, a seed state s_0 and a cost function, the state space $S = \langle s_0, \Delta, \text{cost} \rangle$ is a graph corresponding to the transitive closure of Δ from s_0 . The states of S are the states reachable from s_0 and there is an edge uv in S if there is a transition $v \in \Delta(u)$. The edges are weighted by $\text{cost}(uv)$.*

In this thesis, we consider three types of problems which can be solved in a state space by search.

Definition 2 (Path Finding Problem) *Given a start state $s \in S$ and a goal state g , find a path $\langle s_1 = s, s_2, \dots, s_n = g \rangle$ such that $s_{i+1} \in \Delta(s_i)$.*

Definition 3 (Least-cost Path Problem) *Given a start state $s \in S$ and a goal state g , find a path $\langle s_1 = s, s_2, \dots, s_n = g \rangle$ such that $s_{i+1} \in \Delta(s_i)$ and*

$$\sum_{i=1}^{n-1} \text{cost}(s_i, s_{i+1})$$

is minimum.

Definition 4 (Shortest Path Problem) *Given a start state $s \in S$ and a goal state g , find path $\langle s_1 = s, s_2, \dots, s_n = g \rangle$ such that $s_{i+1} \in \Delta(s_i)$ and n is minimum.*

A shortest path problem is a special case of the least-cost path problem, where all costs are 1. In this general formulation it is not required that the goal state g is unique nor that it is included in S . The goal state in the above definitions can be replaced with a goal condition. In our thesis, however, the problems all have

a unique goal state and the lower bounds are derived specifically for the chosen goal state. Furthermore, to be able to solve a search problem in S , it is also required that the start and goal states are included in the closure of Δ from s_0 . For the state spaces we consider in this thesis, we can offer self-evident choices for s_0 . In the $n \times n$ sliding tile puzzle spaces and the Rubik's Cube, we choose s_0 to be the goal state. We can do this without loss of generality since the closure of Δ from any state s from which g can be reached is isomorphic to the closure from g . In the case of the SOP, there is only one unique start state for each SOP problem and therefore it can be used as the seed state.

If the state space graph is small enough to be explicitly stored in computer memory and the costs are non-negative, Dijkstra's single-source all shortest paths algorithm [Dij59] can find the least cost path in $O(|S| \log |S| + |E|)$ time [CLR90], where $|S|$ is the number of states in the state-space and $|E|$ is the number of edges. While this algorithm is asymptotically optimal if nothing more is known about the state space than S and E , in this thesis we are mainly concerned with state spaces which are so large that they cannot be stored explicitly and cannot be enumerated efficiently. Therefore Dijkstra's algorithm or any algorithm that stores a significant portion of the space does not apply. Instead of explicit representations, such as adjacency matrices and adjacency lists, we will deal with a much more compact representation of state spaces. For the problems in this thesis and for many search problems studied in the literature, the successor generator can succinctly be represented by a set of *operators*.

Definition 5 (Operator) *An operator o is a partial function over states which returns the successor $v = o(u)$ if o applies to state u .*

Definition 6 (Implicitly Generated State Space) *Let $O = \{o_1, o_2, \dots, o_n\}$ be a set of operators. Each operator $o_i \in O$ is associated with a cost $c_i \in C$. The state space $S = \langle s_0, O, C \rangle$ is the transitive closure of a seed state s_0 and the operators O . The states of S are the reachable states from s_0 by applying the operators O .*

Since we are only concerned with implicitly generated state spaces, we can give an exact definition of what we mean by search in this thesis.

Definition 7 (State Space Search) *State Space Search is an algorithm*

$$A = \langle s, g, S = \langle s_0, O, C \rangle \rangle$$

which solves a path finding problem, a shortest path problem or a least-cost path problem in S by returning a path $\langle o_1, o_2, \dots, o_k \rangle$ in terms of the operators $o_i \in O$ such that

$$(o_1 \circ o_2 \circ \dots \circ o_k)(s) = g$$

Note that the state path $p = \langle s_1 = s, s_2, \dots, s_{k+1} = g \rangle$ is clear from the operator path since $s_i = (o_1 \circ o_2 \circ \dots \circ o_{i-1})(s)$.

Our algorithms must find a solution if one exists but we do not require that they terminate if a solution does not exist.

Heuristic search algorithms use a *heuristic* to prune branches from the search tree and in some cases to order the nodes to be explored as well. We must differentiate between nodes and states. A node corresponds to a unique state in the state space, however the same state can be generated more than once during

search and these generations correspond to distinct nodes in the search tree. For us, the *heuristic* is always a lower-bound estimate of reaching the goal state from a node in the search tree. In the case of the shortest-path problem this lower-bound is an underestimate of the *number of operators* that must be applied to reach the goal state, while in the case of a least-cost path problem the lower bound is on the *total cost* of the operator sequences that reach the goal. Traditionally such lower bounds are called *admissible heuristics* in Artificial Intelligence.

Definition 8 (Admissible Heuristic) *A heuristic estimate $h(n)$ for a node n in the search tree is called admissible if it is a lower bound on the minimum cost (or length) of reaching the goal state from n .*

In this thesis we prefer the term *lower bound* instead to avoid confusion. The term heuristic in Operations Research often refers to a solution which is suboptimal and hence corresponds to an upper-bound.

A useful measure to compare search algorithms is the count of nodes expanded to reach the goal state from the start state. *Nodes expanded* are the nodes in the search tree whose successors have been generated by the successor generator (in our case, the operators). Each node n during search is associated with a value $f(n)$ which is used by the search algorithm.

Definition 9 (f -value) *The f -value of a node n is defined as*

$$f(n) = c(n) + h(n) \tag{2.1}$$

where $c(n)$ is the cost of reaching n from the start state and $h(n)$ (heuristic value) is an estimate of the cost of reaching the goal state from n .

In our case, $h(n)$ is purely a function of the state the node is associated with. On the other hand, g is a function over nodes, because the same state can be reached by different operator paths during search.

Definition 10 (Brute-force Search Tree) *Given a search algorithm A , a successor generator Δ and a start state s , the brute-force search-tree (BFST) is a tree rooted at s which corresponds to the trace of algorithm A when $h(.) = 0$ for all states and goal states are ignored.*

The brute-force search-tree includes the nodes generated during search and it also contains the branches that would be pruned off due to a better h . The topology of the brute-force search tree depends on the search algorithm as it must take intrinsic pruning by the search algorithm into account. For example, many search algorithms utilize a cache of states encountered during search to eliminate duplicate nodes. These nodes are not part of the BFST either.

To illustrate the concepts and definitions introduced so far, let us consider the 2×2 tile puzzle as an example. There are three tiles with labels 1, 2 and 3 (Figure

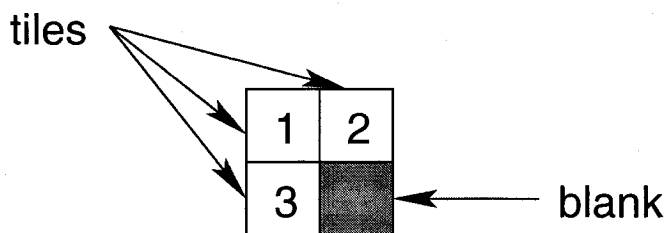


Figure 2.1: The 2×2 puzzle.

2.1). There is one tile missing so a neighboring tile can be slid into the vacant position. First we need a state representation. A natural way to encode a state

state space has 12 states as shown in Figure 2.2. A portion of the brute-force search tree for a search algorithm that does not use duplicate node elimination is shown in Figure 2.3. Note that the brute-force search-tree without duplicate

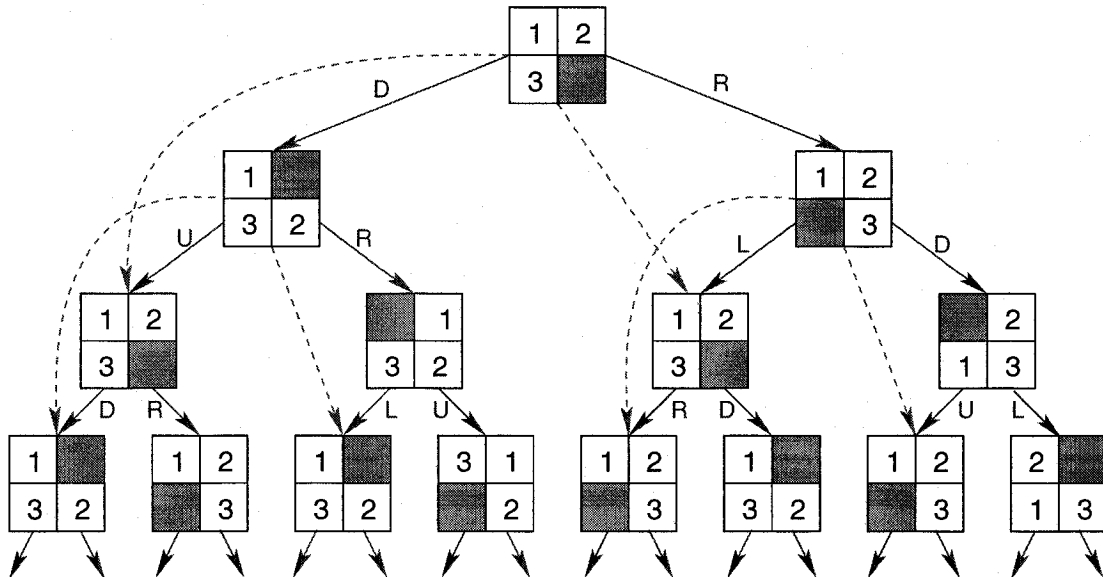


Figure 2.3: Brute-force search tree for the 2×2 puzzle.

elimination is infinite. The dashed arrows in Figure 2.3 identify duplicate nodes.

2.2 Algorithms

The search algorithms used in this thesis are all based on three well known algorithms. These are A^* [HNR68], IDA^* [Kor85b] and DFBB [LW66]. These algorithms all use the f -value of a node (Definition 9, page 12) to prune nodes from the search tree. Moreover, when $h(n)$ does not overestimate the true distance from n to the goal then these algorithms will find an optimal solution if the goal is reachable.

2.2.1 A*

A* uses an open and a closed list to store nodes already reached. The nodes generated but not yet expanded are on the open list and the ones that have already been expanded are on the closed list. The open list is usually implemented as a priority queue where the node's priority is its f -value (lower f -value implies higher priority).

A* terminates when the goal node is expanded. If there are two nodes with the same f -value, the node which has the lower h value (consequently further away from the start state) should be assigned higher priority. Otherwise many nodes with the same f -value of the goal node could be taken off the open list before it is the goal node's turn. Even with this tie breaking rule, it cannot be guaranteed that the goal node is the first to come off from the nodes on the open list with the f -value of the goal. If the state space is finite and the goal state is not reachable, A* exhaustively enumerates the entire space and therefore it can be used to detect if the goal state does not exist. The drawback of using A* is its memory requirements. For large spaces it is often the case that the sizes of the open and closed lists exceed available memory. In the worst case, the time and space complexity of A* is the same as breadth-first search.

Definition 11 (Consistent Heuristic) *A heuristic estimate h is said to be consistent if*

$$h(u) \leq \min \text{cost}(uv) + h(v), \quad \forall u, v \in S \quad (2.2)$$

where $\min \text{cost}(uv)$ is the minimum cost of reaching v from u in the state space or ∞ if v is not reachable from u .

For A^* , *consistent lower bounds* guarantee that a node put on the closed list (expanded) will not be put back on the open list again (re-expanded) [Pea84].

Definition 12 (Monotone Heuristic) *A heuristic estimate h is said to be monotone if*

$$h(u) \leq \text{cost}(uv) + h(v), \quad u, v \in S, \quad v \in \Delta(u) \quad (2.3)$$

where $\text{cost}(uv)$ is the cost of the uv edge.

Pearl [Pea84] showed that *consistency* and *monotonicity* are equivalent properties.

Since A^* with a monotone heuristic does not generate duplicates, its brute-force search-tree is a spanning tree of the state space where each root-to-node path segment is of minimal cost.

2.2.2 IDA*

The algorithm often used in practice to search large spaces is IDA* [Kor85b]. IDA* is an iterative algorithm which performs a series of depth-first searches. On each iteration a cost bound B is given and B is increased from one iteration to the next. If the f -value of a node exceeds this B then it is pruned from the search tree for the current iteration. The smallest f -value exceeding B among the generated nodes is returned and will be used as the cost bound for the next iteration. The first iteration starts with $B = h(s)$, where s is the start state. If the operator costs widely vary, IDA* may need a large number of iterations to bump up B to the cost of the goal node. IDA*, however, is asymptotically optimal when the operator costs are the same [Kor85b]. Unlike A^* , there is

no node ordering or checking for duplicate nodes. Therefore IDA* is very fast with respect to node generation. As soon as the goal state is expanded, IDA* terminates and the optimal solution is found. On the last iteration the number of nodes expanded depends on when the goal state is encountered. Since there is no node ordering, in the worst case it is possible that all nodes with f -value $f(n) \leq f(g)$ are expanded. With unit cost operators, the last iteration dominates all other iterations and therefore complexity analysis of IDA* [KR98, KRE01] is usually only concerned with the number of nodes expanded on the last iteration in the worst case. This is the number of nodes n in the brute-force search-tree with $f(n) \leq f(g)$.

2.2.3 DFBB

As mentioned earlier, IDA* is not appropriate if the operator costs substantially vary. The SOP (Chapter 5) is defined by an $n \times n$ cost matrix and therefore there are $O(n^2)$ different operator costs. This renders IDA* inapplicable for the SOP. A* is also impractical because of the large state space size of the problem instances we would like to solve.

The *branch and bound* (BB) [LW66] search algorithm is often used for optimization problems. Let $Z(u)$ be the value of the objective function (the function to be optimized) for a state u . The state in many optimization problems is a partial set of assignments of the state variables. In a minimization problem the aim is to find a state g for which $Z(g)$ is minimal. Let x denote the best currently known feasible solution. x is also called the *incumbent*. A solution is feasible if it satisfies the problem constraints. The algorithm consists of the following steps from the

start state s .

branch select a node u not yet expanded and generate its successors (the selection criterion – such as best bound, most recent, *etc* – may have significant impact on performance and memory requirements)

bound for each successor v , generate a lower bound $z(v)$.

fathom for each successor v , apply some or all of the following fathoming (eliminating) tests:

1. $z(v) > Z(x)$: v is worse than the best known solution so far.
2. v 's subtree contains no feasible solutions.
3. v is feasible. If $z(v) < Z(x)$, then $x = v$.

The algorithm terminates when there are no more remaining subproblems. For fathoming test 3 to apply, v must be the best feasible solution in the subtree of v . Since v is a partial solution, it cannot be by itself feasible. However, in many branch and bound algorithms (like Balas' Additive Algorithm for 0-1 Integer Programs [Bal65]) $z(v)$ corresponds to a greedy completion and with this completion v could be feasible. To adapt BB for a general state-space search problem, we can only use fathoming test one, which we can restate as $f(u) > Z(x)$, because the lower bound for a node u is its f -value. For the problems we consider in this thesis, fathoming test 2 is not necessary since we only generate feasible nodes. We do not store the completion paths that correspond to $h(u)$ and therefore we cannot evaluate fathoming test 3 efficiently. If BB always selects the last generated node to be expanded next then the memory requirements are only $O(bk)$, where k is the search depth and b is the branching factor. k could

actually be much larger than the minimum depth of a goal node. On the other hand, in many optimization problems all solutions are at the same depth. This is because the problems are often formulated as binary integer programs over n variables. Each branch in the search corresponds to setting a variable to 0 or 1 and therefore no path is deeper than n . In particular, in matching problems, job-shop scheduling and problems corresponding to Hamiltonian paths or circuits, the depth of the solution is known ahead of time. BB with the branch selection criterion of the most recently generated node is also called Depth-first Branch and Bound (DFBB). Unlike A^* and IDA^* , DFBB expands nodes which have higher f -value than that of the goal and as a consequence, the first solution found is usually not optimal. If DFBB is used to solve an Integer Programming problem with n decision variables, then a feasible solution is found in $O(n)$ steps. Potentially many more suboptimal solutions of decreasing cost will be generated before an optimal solution is reached. Therefore DFBB is also an *anytime* algorithm.

Definition 13 (Anytime Search Algorithm) *A search algorithm A is anytime if it can return a solution even if it is stopped before completion. If A is allowed to run longer it may return a better quality solution.*

Zhang in [Zha93] showed that *truncated* DFBB (DFBB stopped before completion) is particularly effective to obtain good quality solutions for the asymmetric TSP.

2.3 Lower Bounds From Abstraction

Our lower bounds correspond to optimal costs in an abstracted version of the original state space. In practice, most such estimates come from abstract models

[Pea84, Pri93, CS98, CMT81, MBR97]. In particular, we are interested in *state-space homomorphisms* which guarantee lower bounds.

Definition 14 (State-space Homomorphism) *Let S and S' be state spaces. $\phi : S \rightarrow S'$ induces a state-space homomorphism, if*

1. ϕ is a total function
2. for every edge $uv \in S$ where $\phi(u) \neq \phi(v)$ there is an edge $\phi(u)\phi(v) \in S'$ (reachability requirement)
3. $\text{cost}(\phi(u)\phi(v)) \leq \text{cost}(uv)$ (cost requirement)

The function ϕ is called the abstraction function or abstraction for short.

In this definition we make it clear that ϕ is applied to the states and that the connectivity of S is preserved in S' . Suppose we designate $g \in S$ as our goal state. The minimum cost of reaching the abstract goal state $g' = \phi(g)$ in S' from a state $s' = \phi(s)$ is a lower bound on the minimum cost of reaching g from s in S . Figure 2.4 depicts a state space homomorphism. S has 10 states while S' has only 4. If the edge cost restriction in Definition 14 is preserved, it is clear that the minimum cost path from s' to g' in S' cannot have a higher cost than the minimum cost path from s to g in S .

Definition 15 (Refinable Path) *A path*

$$p' = \langle u'_1, u'_2, \dots, u'_k \rangle$$

in S' is refinable if there is a path

$$p = \langle \dots, u_1, \dots, u_2, \dots, u_k, \dots \rangle$$

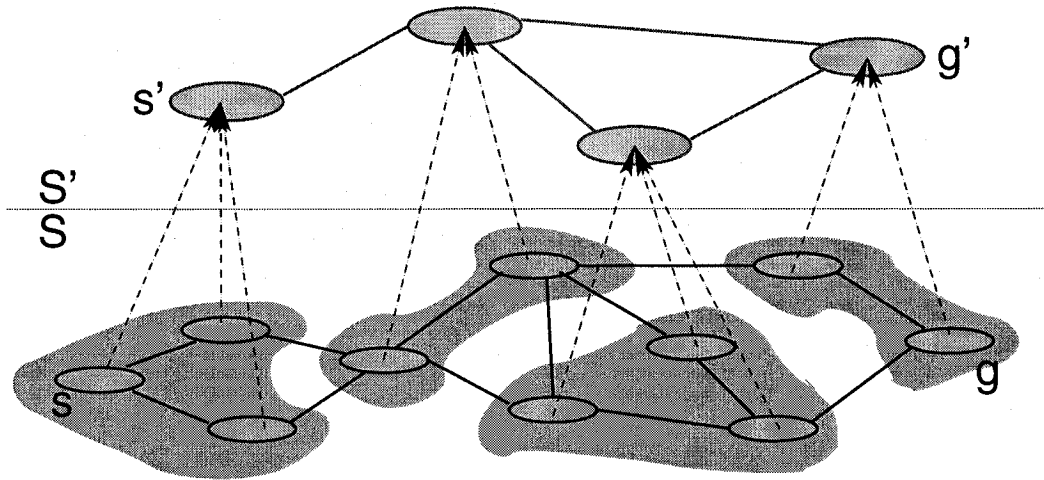


Figure 2.4: State space homomorphism.

in S which includes states u_i for each $u'_i \in p'$ (in the same order) such that $\phi(u_i) = u'_i$.

In the case of a state space homomorphism, every path $p \in S$ has an image in S' but in general it is not true that all paths $p' \in S'$ are refinable in S . We will show examples of unrefinable paths in S' in Section 2.4.

If S' is small enough to be stored in memory, then we can build a hashtable H where the entries are the costs from every abstract state $s' \in S'$ to g' and indices are the abstract states. The lower bound $h(s)$ then becomes

$$h(s) = H[\phi(s)] \quad (2.4)$$

We will refer to H as a pattern database.

Definition 16 (Pattern Database) *A pattern database is a look-up table of lower bounds indexed by abstract states.*

The term “pattern database” comes from [CS94] but the idea was independently developed in Operations Research, Artificial Intelligence and Mathematics. Christofides *et al.* [CMT81] used precalculated dynamic programming tables (lower bounds) of relaxed state spaces for constrained versions of the Traveling Salesman Problem. Culberson and Schaeffer [CS94, CS98] applied tile abstractions on the 15-puzzle and stored the lower bounds obtained from the abstract spaces in *pattern databases*. Frieditis [Pri93] also mentions precalculating *heuristics* for the Rubik’s Cube. This abstraction was later included in the pattern databases used by Korf [Kor97] to actually solve random instances of the Rubik’s Cube optimally for the first time. Herbert Kociemba’s algorithm [Koc], also independently developed, uses 3 look-up tables of lower-bounds to derive very short solutions to the Rubik’s Cube.

While the conceptual picture of a state-space homomorphism as exemplified in Figure 2.4 (page 22) is easy to adopt, we must work with a generative model. We are particularly interested in state spaces that are so large that they cannot be enumerated and therefore the calculation of S' must be performed without calculating S . In most previous work on pattern databases, lower bounds were obtained by *expanding* the abstract space using a generator. The notable exception is the work of Holte *et al.* [HPZM96] who used abstractions to speed-up search in explicitly stored graphs and therefore calculated the abstractions from the actual state space. We will build our generative model of pattern databases from the recursion

$$\gamma_v = \begin{cases} 0, & \text{if } v = s \\ \min_{u \in \Delta(u)} (\gamma_u + \text{cost}(uv)), & \text{otherwise} \end{cases} \quad (2.5)$$

In Equation 2.5, s is the start state and γ_v is the minimal cost of reaching state v from state s . Dijkstra's algorithm provides an efficient solution to the recursion when all costs are non-negative. This is going to be the case for all of our state spaces and therefore we assume in the following discussion that $cost(uv) \geq 0$. In Equation 2.5, γ_v is cost from state s but what we need is the cost to the goal. We rewrite Equation 2.5 as

$$\gamma_u = \begin{cases} 0, & \text{if } u = g \\ \min_{v \in \Delta^{-1}(u)} (\gamma_v + cost(vu)), & \text{otherwise} \end{cases} \quad (2.6)$$

Equation 2.6 is a simple recursive formulation of the least-cost path completion problem. It requires that we have a generator Δ^{-1} which returns the predecessors of a state in S . This does not mean that the edges must be invertible but rather that we can calculate the states that have transitions to the state in question. Note that even though the predecessors are generated, the forward costs are used. The path cost is measured in the original direction of the edges. If Equation 2.6 is solved with Dijkstra's algorithm, then in essence the *transpose* of the state space is expanded.

Definition 17 (Transpose of a Graph) *Let $G = \langle V, E \rangle$ be a graph with vertices V and edges E . The transpose of G is a graph $G^T = \langle V, E^T \rangle$ with the exact same vertices and edge costs but the direction of the edges is reversed. If $G = \langle V, C \rangle$, where C is a cost matrix, then G^T is simply $\langle V, C^T \rangle$.*

Since we do not have an explicit representation, Δ^{-1} generates the predecessors.

If the operators are all invertible and all costs are 1, then $\Delta = \Delta^{-1}$ and Equation 2.6 becomes

$$\gamma_u = \begin{cases} 0, & \text{if } u = g \\ \min_{u \in \Delta(v)} (\gamma_v + 1), & \text{otherwise} \end{cases} \quad (2.7)$$

which can be solved more efficiently by a breadth-first expansion of the space from the goal state. This is the technique used most frequently in previous pattern database work [CS98, Kor97].

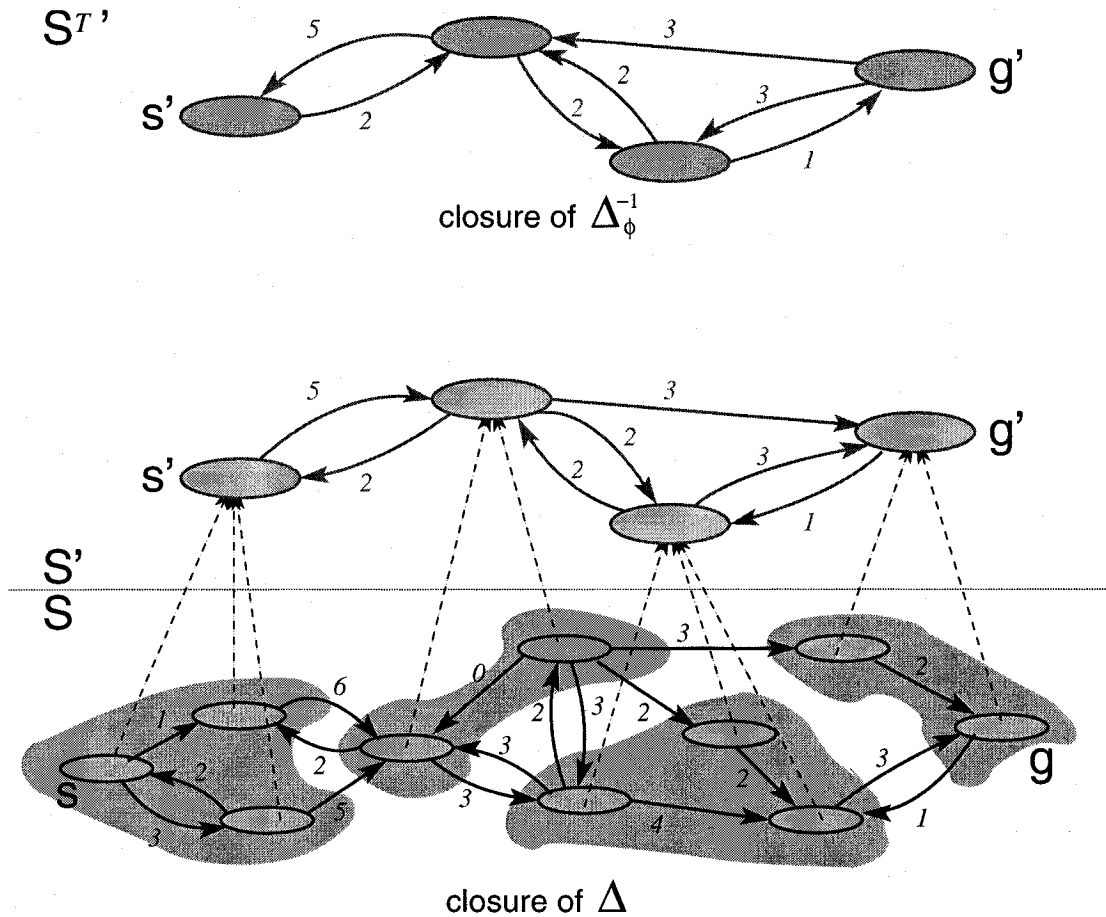


Figure 2.5: Transpose of the Abstract Space.

Figure 2.5 depicts the general scenario where the edges are directed and have different costs. $\phi : S \rightarrow S'$ is a state space homomorphism (the three requirements

of Definition 14 are satisfied). The lower bound $h(u)$ for $u \in S$ is the cost of the cheapest $u'g'$ path in S' . Performing a separate search from each $u' \in S'$ to g' would be a very inefficient technique to derive all lower bounds. On the other hand, with non-negative costs, Dijkstra's single source all shortest path algorithm in $S^{T'}$ (the transpose of S') from source g' is an efficient algorithm to obtain all lower bounds. Since S is the closure of Δ and we assume that S is so large that we cannot expand it to derive the lower bounds, we need a generator Δ_ϕ^{-1} which induces $S^{T'}$. The subscript ϕ on Δ_ϕ^{-1} designates that it returns predecessors in the abstract space and therefore it possibly requires a different implementation than that of Δ^{-1} . Δ_ϕ^{-1} induces $S^{T'}$ when

$$v \in \Delta(u) \iff u' \in \Delta_\phi^{-1}(v') \quad (2.8)$$

and

$$cost(v'u') \leq \min_{\substack{u \in \phi^{-1}(u') \\ v \in \phi^{-1}(v')}} cost(uv) \quad (2.9)$$

In Equation 2.9, $\phi^{-1}(u')$ is the set of preimages of u . That is,

$$u \in \phi^{-1}(u') \iff \phi(u) = u' \quad (2.10)$$

In our experience, it is not always straightforward to implement Δ_ϕ^{-1} such that it satisfies Equation 2.8. However

$$v \in \Delta(u) \implies u' \in \Delta_\phi^{-1}(v') \quad (2.11)$$

is sufficient to preserve connectivity in $S^{T'}$.

Next, we state a theorem which is the theoretical foundation of pattern databases in this thesis.

Theorem 1 (Pattern Database Lower-bound Theorem) *Let Δ_ϕ^{-1} and function cost satisfy the conditions of Equations 2.11 and 2.9 respectively. γ_u , defined as*

$$\gamma_u = \begin{cases} 0, & \text{if } u = g \\ \min_{u' \in \Delta_\phi^{-1}(u)} (\gamma_{u'} + \text{cost}(u'u)), & \text{otherwise} \end{cases} \quad (2.12)$$

is a lower bound on the cost of any path that reaches g from u in S .

Before proving Theorem 1 (in Section 2.4), we would like to draw attention to the consequences of relaxing Equation 2.8 by Equation 2.11. We require only that the set of abstract states returned by $\Delta_\phi^{-1}(v')$ include the images of all states u in S which have a transition to v . This requirement holds even if $\Delta_\phi^{-1}(v')$ returns abstract states which have no preimages in the original space. Such states will be referred to as *spurious states*. For a state space abstraction to be useful, it should be smaller than the original space. An abstraction that produces many spurious states defeats its purpose.

Definition 18 (Non-surjective Abstraction) *An abstraction $\phi : S \rightarrow S'$ is non-surjective, if there exists a state $u' \in S'$ such that there is no $u \in S$ with $\phi(u) = u'$.*

A non-surjective state space homomorphism is depicted in Figure 2.6. The states with dashed boundaries do not have preimages in the original space. Nonetheless, it is clear, that adding extra states and edges still results in lower bounds as the required edges are included in the minimization. Unfortunately, it is also possible that paths passing through spurious states have cheaper costs and this can be the cause of less accurate lower bounds. We would like to stress that non-surjective

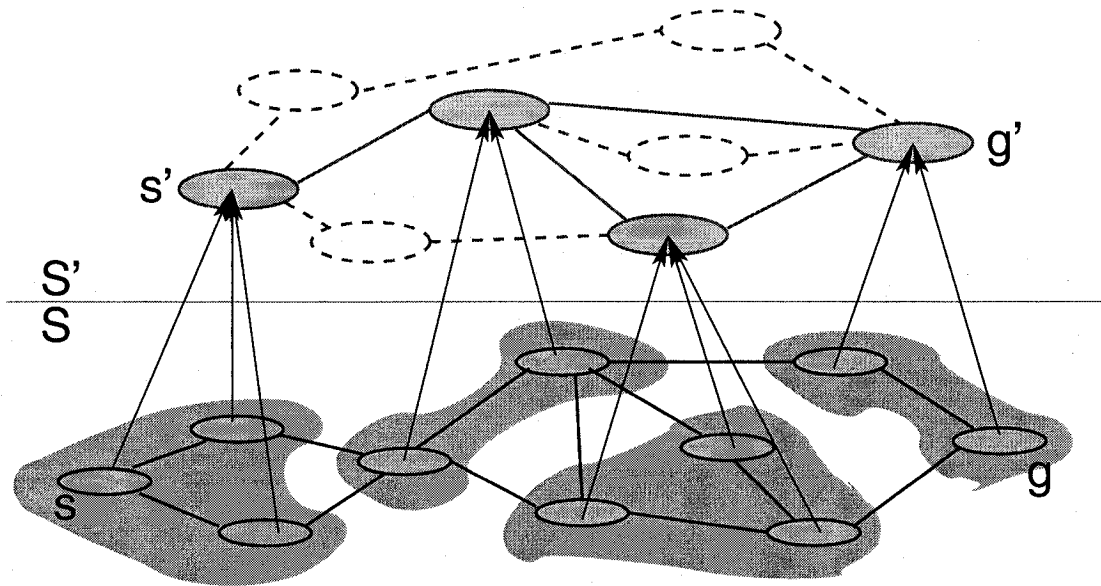


Figure 2.6: A Non-surjective State Space Homomorphism.

state space homomorphisms arise because we use a generative model to calculate S' . If the abstractions were directly applied to the states of S (like the STAR abstraction in [HPZM96]) the problem would not occur.

In our experience, non-surjective abstractions arise very often and we will show in Section 7.1 that there are pathological encodings of search spaces where practically all abstractions are non-surjective. A naïve solution is to require Δ_ϕ^{-1} to only return those abstract states which have preimages in S . It is often, however, very hard to design a predecessor generator which does not yield spurious states and if the number of such states is relatively small compared to the size S' , the abstraction could still be useful. We believe that this phenomenon was overlooked in previous research and therefore we dedicate Section 7.1 to provide an in-depth description and examples. Also, for the Sequential Ordering Problem (Chapter 5), a straightforward implementation of the predecessor generator would produce spurious states.

2.4 Mathematical Foundation

First we state and then prove that state space homomorphisms are useful to derive lower bounds.

Theorem 2 (State Space Homomorphism Theorem) *Let $\phi : S \rightarrow S'$ be a state space homomorphism (Definition 14, page 21) and let $s, g \in S$ be any two states. The cost of the least cost path between $\phi(s)$ and $\phi(g)$ in S' is a lower bound on the cost of reaching g from s in S .*

Proof 2.1 (Theorem 2) If there is no path between s and g in S then their distance is ∞ and any value is a lower bound. Suppose g is reachable from s via the path

$$p = \langle s_1 = s, s_2, \dots, s_n = g \rangle$$

Then according to the reachability requirement in Definition 14 (page 21), the path

$$p' = \langle s'_1 = \phi(s), s'_2 = \phi(s_2), \dots, s'_n = \phi(g) \rangle$$

exists and, by the cost requirement, its cost cannot exceed the cost of p . $\square$

While p' must exist when p exists, it is instructive to consider the cases when there is a cheaper path than p' in S' between s' and g' . The cases are depicted in Figure 2.7. Suppose $s_i, s_j \in p$ where $i < j$ and $s_k \notin p$. In the first case, $\phi(s_i) = \phi(s_j)$ indicating that p' has a loop and this is why p' is cheaper. In the second case, $\phi(s_k) = \phi(s_i)$ and there is an abstract path from s'_k to g' which is also cheaper than the cost of the path from s'_i to g' that includes s'_j (the completion of s'_i corresponding to p'). In the third case, s'_k is a spurious state (one that has no

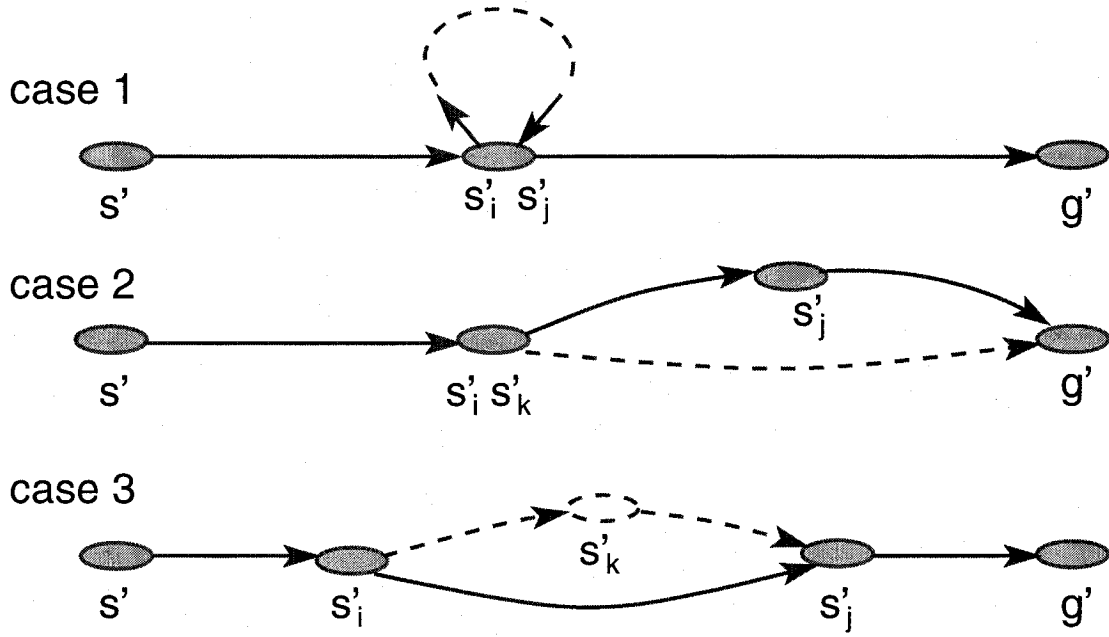


Figure 2.7: Shortcut Abstract Paths

preimage) and it creates a shortcut between s'_i and s'_j . Note that this path has no refinement in the original space. If s and g are not connected in S , they can become connected in the abstract space in cases 2 and 3. The scenarios depicted in Figure 2.7 also apply when $s'_i = s'$ or $s'_j = g'$.

Next we prove Theorem 1 (page 27) which states that every total function ϕ over the states of S can be used to derive lower bounds.

Proof 2.2 (Theorem 1, Page 27) First we show that the closure of Δ_ϕ^{-1} from g' is a homomorphism of S^T (the transpose of S) by showing that the 2 conditions of Definition 14 (page 21) are satisfied.

1. $\Delta_\phi^{-1}(v')$ by definition must generate a u' for every u with $v \in \Delta(u)$. Therefore if there exists an edge $uv \in S$ then there is an edge $v'u'$ in $S^{T'}$ (Equation 2.11).

2. The cost of $v'u' \in S^{T'}$ by definition is at most the cost of $uv \in S$ (Equation 2.9).

Hence, according to Theorem 2, $\gamma_{s'}$ is a lower bound on $\text{cost}(g', s')$. Since the minimum cost path from s' to g' in S' has the same cost as the cheapest path from g' to s' in S'^T , $\gamma_{s'}$ is also a lower bound on the cost of reaching g from s . $\square$

As we discussed earlier, for many state spaces (like the $n \times n$ puzzles and the Rubik's Cube) it is the case that $\Delta = \Delta^{-1}$ and the actual implementation of Δ does not have to be modified at all to generate predecessors in the abstract space. We view this situation as a special case which is not true for many state spaces, including the SOP (Chapter 5).

Theorem 3 (Monotonicity Theorem) γ_u defined in Theorem 1 (Page 27) is a monotone heuristic (Definition 12).

Proof 2.3 (Theorem 3) Holte *et al.* in [HPZM95] proved that cheapest costs to g' in S' when S' is a homomorphism are monotone heuristics. We proved above that $\gamma_{s'}$ is the cost of the cheapest path from s' to g' . $\square$

As a consequence A^* does not expand duplicate nodes with our lower bounds.

2.5 Domain Abstraction

In this section we describe our particular abstraction technique. It is an extension of the *pattern* abstraction introduced by Culberson and Schaeffer in [CS94]. The

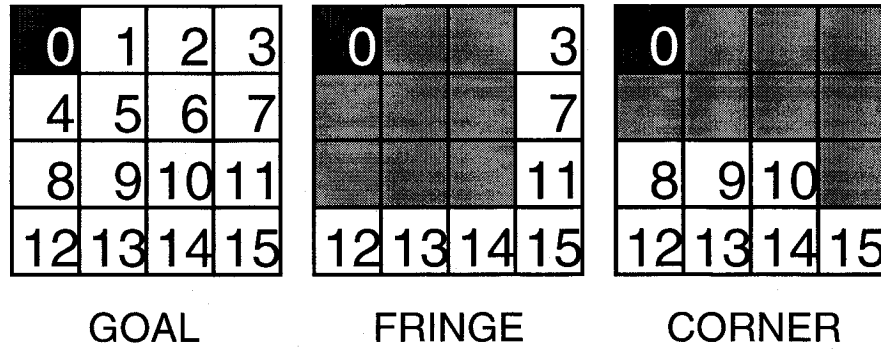


Figure 2.8: Fringe and Corner Patterns.

fringe and *corner* patterns (from [CS94]) for the 4×4 puzzle are depicted in Figure 2.8. They correspond to physically scrubbing off the numbers from some of the tiles. The resulting puzzle represents a much smaller state space and according to Theorem 1 (page 27), the length of the shortest path to solve a problem in the abstract space is a lower bound on the shortest number of moves for a corresponding problem in the original puzzle. If the state representation is a vector of tile labels, then the fringe and corner patterns can be defined as relabeling functions of the state representations.

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\text{fringe}(t)$	0	-	-	3	-	-	-	7	-	-	-	11	12	13	14	15
$\text{corner}(t)$	0	-	-	-	-	-	-	-	8	9	10	-	12	13	14	15

The covered tiles (represented by the label $-$) while indistinguishable, still play an important role in the abstraction as they interact with the pattern tiles. We generalized this idea.

Definition 19 (Domain Abstraction) *Let D and D' be finite domains with $|D'| \leq |D|$. Any total function $\phi : D \rightarrow D'$ is a domain abstraction.*

Figure 2.9 depicts an example of a domain abstraction. It can be written as

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\phi(t)$	0	*	*	o	+	*	*	*	+	+	+	o	o	+	o	o

Without loss of generality, we can assume that both D and D' in Definition 19 are sets of consecutive integers starting at 0. This way we can define a canonical form of domain abstractions where the following conditions must hold.

1. $D = \{0, 1, 2, \dots, |D| - 1\}$ and $D' = \{0, 1, 2, \dots, |D'| - 1\}$
2. If $x < y$ and $\forall_{w < x} \phi(w) \neq \phi(y)$ then $\phi(x) < \phi(y)$

Our domain abstraction in canonical form is

t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\phi(t)$	0	1	1	2	3	1	1	1	3	3	3	2	2	3	2	2

The entries where the new label first appears is shown in bold. Consider the

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

GOAL

0	*	*	o
+	*	*	*
+	+	+	o
o	+	o	o

DOMAIN
ABSTRACTION

Figure 2.9: A Domain Abstraction for the 4×4 Puzzle.

domain abstraction

$$\phi(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{otherwise} \end{cases}$$

Applied to the 2×2 puzzle (defined in Section 2.1), ϕ induces the state space depicted in Figure 2.10 (the original state space is shown in Figure 2.1, page 13).

If we take $g = \langle 1, 2, 3, 0 \rangle$ as the original goal state, then $\phi(g) = \langle 1, 1, 1, 0 \rangle$ and we

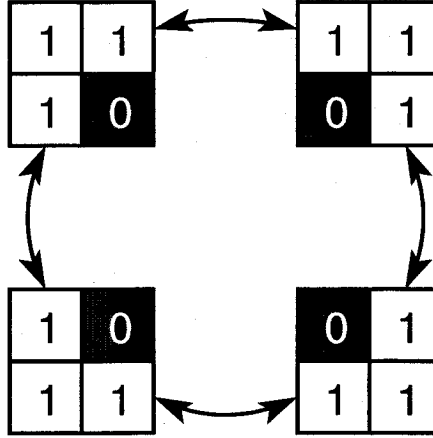


Figure 2.10: Abstract State Space for the 2×2 Puzzle.

can build the following pattern database.

	s'	h
$H =$	$\langle 1, 1, 1, 0 \rangle$	0
	$\langle 1, 1, 0, 1 \rangle$	1
	$\langle 1, 0, 1, 1 \rangle$	1
	$\langle 0, 1, 1, 1 \rangle$	2

From this,

$$h(\langle 1, 2, 0, 3 \rangle) = H[\phi(\langle 1, 2, 0, 3 \rangle)] = H[\langle 1, 1, 0, 1 \rangle] = 1$$

This is actually the true distance to reach g from state $\langle 1, 2, 0, 3 \rangle$. On the other hand, $h(\langle 2, 3, 0, 1 \rangle)$ is also 1, but this state is 5 moves from the goal state. The states $\langle 3, 1, 2, 0 \rangle$ and $\langle 2, 3, 1, 0 \rangle$ are both 4 moves from the goal but their lower bound estimates are 0.

Often a label in the domain has special significance. The label 0 in our encoding of the 2×2 puzzle represents the blank tile. The position of the blank alone decides the applicable operator. Some abstractions that do not treat the blank as special result in non-surjective abstractions (Definition 18, page 27). Consider the domain abstraction

$$\psi(x) = \begin{cases} 0, & \text{if } x = 0 \text{ or } x = 3 \\ x, & \text{otherwise} \end{cases}$$

ψ induces the abstract state space depicted in Figure 2.11. This abstract space

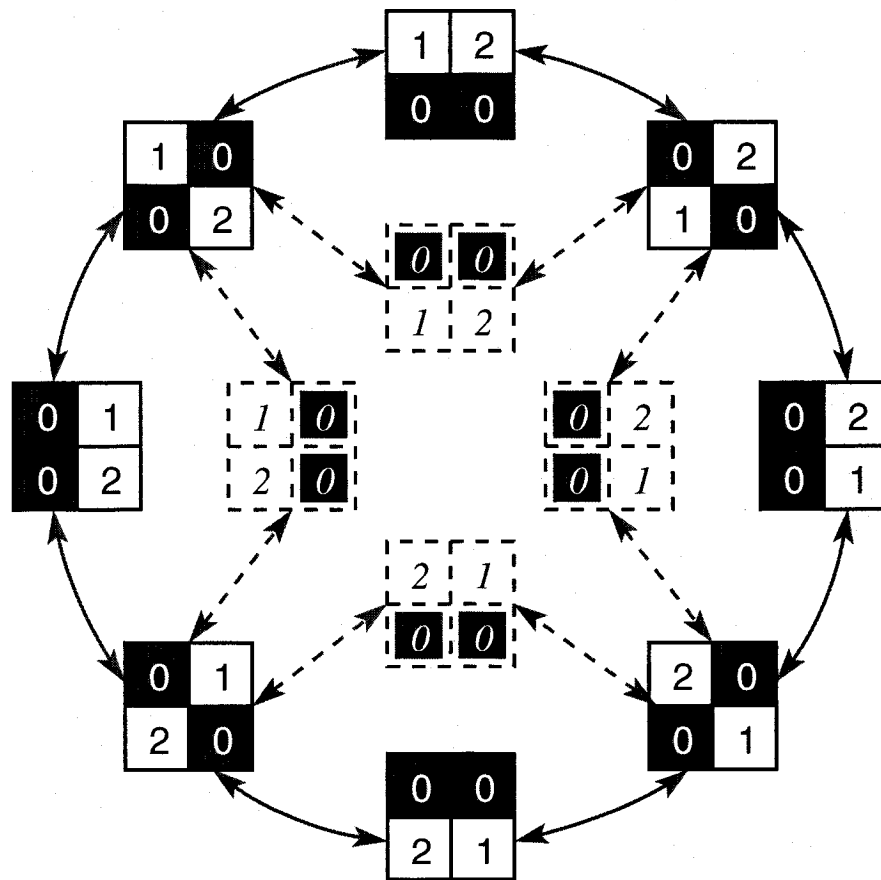


Figure 2.11: A Non-surjective Abstraction for the 2×2 Puzzle.

is obtained by applying the operators starting from the abstract goal state. The

abstract states enclosed in dashed boundaries do not have preimages in the original state space (Figure 2.1, page 13). If we were to apply ψ directly to the state space, then the states and edges (identified with the dashed notation), would simply not occur. These states appear in S' because we built it using a generator. The $n \times n$ puzzle spaces are state isomorphic to the elements of the Alternating Group A_n which are generated by even transpositions. Therefore half of the $n!$ possible permutations are not reachable. Since we defined state spaces in terms of a seed state and operators (Definition 6, page 11), the unreachable half is not part of the original state space. However, when two blank tiles are introduced as the result of the abstraction, the two halves become connected. For the $n \times n$ puzzles, the only restriction in this representation to guarantee surjectivity is to keep the blank's identity unique.

For the $n \times n$ puzzles and for the state spaces we study in this thesis, the size of the abstract space depends on the *granularity* of the domain abstraction.

Definition 20 (Granularity) *Let D and D' be finite domains and let $\phi : D \rightarrow D'$ be a domain abstraction. ϕ has granularity $\Gamma_\phi = \langle n_1, n_2, \dots, n_k \rangle$, if*

$$\sum_{i=1}^m n_i = |D|$$

where n_i 's are the number of labels of D which map to the same label in D' . The 1's in the granularity vector represent labels which remain unique in the abstract space. For brevity, when $|D|$ is clear from the context, we omit these 1's. In the canonical form of the granularity, $n_1 > n_2 > \dots > n_k$.

For example, the granularity of the “Fringe” abstraction depicted in Figure 2.8 is $\langle 8, 1, 1, 1, 1, 1, 1, 1, 1 \rangle$ or $\langle 8 \rangle$. The granularity of the domain abstraction shown in

Figure 2.9 is $\langle 5, 5, 5, 1 \rangle$ or $\langle 5, 5, 5 \rangle$. For the state spaces and domain abstractions used in this thesis, the granularity alone determines the size of the abstract space and therefore the number of entries in the corresponding pattern database.

2.6 Implementation Framework

In this section we define a general framework for state space search. Our frame-

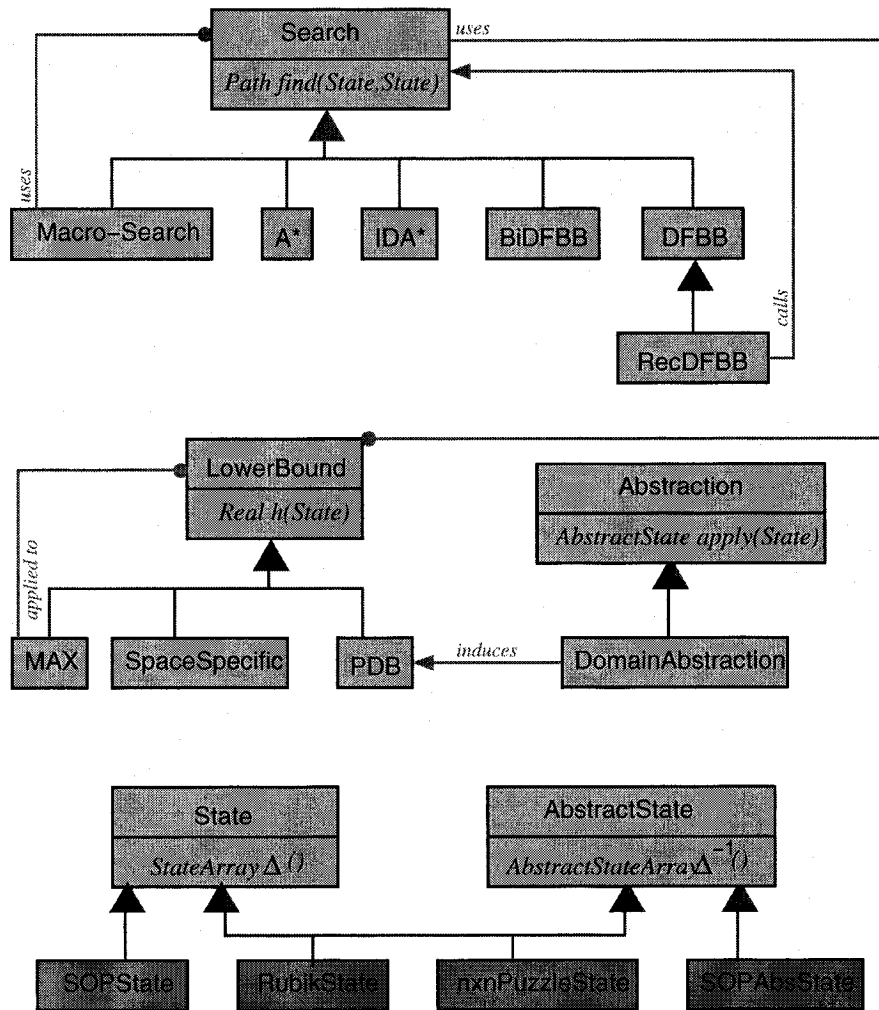


Figure 2.12: A Framework for state-space Search.

work in UML notation [BRJ98] is depicted in Figure 2.12. The search algorithms we consider are A^* , IDA^* , DFBB and macro-search. Except for macro-search, which will be detailed in Chapter 4, these algorithms directly use the lower bounds to prune the search tree. Macro-search has a case base of precalculated macro-operators to derive a solution and does not perform an actual search to solve the problem. The macro-operators, however, are obtained by some search technique (which is indicated by the *uses* association). Our macro-search is also a hybrid algorithm. It solves some of the subgoals by regular heuristic search and the rest are solved by retrieving the appropriate macro-operators. Recursive DFBB (RecDFBB) and Bidirectional DFBB (BiDFBB) are our inventions. RecDFBB starts off as a regular DFBB but it also takes advantage of the fact that a partially completed SOP can be viewed as a SOP by itself and therefore it can be solved independently. BiDFBB switches direction each time a new better cost solution is discovered. We found that this often results in a faster convergence to the optimal solution. BiDFBB and RecDFBB will be discussed in detail in Chapter 5.

In Chapter 3 we use the 3×3 puzzle as a testbed to perform experiments with a large number of pattern databases. These experiments use both A^* and IDA^* . The macro operators, discussed in Chapter 4, will be obtained by IDA^* . SOPs (Chapter 5) will be solved with DFBB, BiDFBB and RecDFBB.

The lower bound $h(s)$ is just a “plug-in” function that these search algorithms use. The `LowerBound` class hierarchy in Figure 2.12 is therefore independent of the `Search` class hierarchy. The lower bounds we consider in this thesis are pattern databases (PDB) and human derived state space specific ones (such as the *Manhattan Distance*). These human derived lower bounds are also very useful

for benchmarking pattern databases. In the case of macro-operators, we also use the maximum of more than one pattern databases (**MAX**), since the maximum of a number of lower bounds is also a lower bound, and never less accurate than the individual estimates. As we stated in Theorem 1 (page 27), any total function can be used to derive a lower bound. In this thesis, we used only domain abstractions but other abstractions, such as projections, could be directly be plugged into our framework by creating a subclass of class **LowerBound**.

The search algorithms can be written purely in terms of a generic state (class **State** in Figure 2.12) which implements the successor generator Δ . The pattern databases can also be built from an abstract state class (**AbstractState**) which implements the predecessor generator Δ^{-1} . For the Rubik's Cube and the sliding-tile puzzle state spaces, the abstract states can use the same state description. For the SOP state space we found that a much more compact abstract state representation is more appropriate.

Our framework as shown in Figure 2.12 provides a high level map of the actual implementation that was needed to solve the problems in this thesis, but at the same time it also shows how one can extend or adopt our design. For example, a new search algorithm can easily be added by deriving it from the abstract class **Search** and this will automatically be picked up in the runtime context of the experiments. If a new state space is to be added, one only has to provide a successor generator and a predecessor generator for the abstract states. In simple cases, like the Rubik's Cube and the sliding tile puzzles, the two are actually the same. Similarly, a new type of abstraction can be used to build pattern databases or other type of lower bounds and this only has to effect this part of the design; the search algorithms do not need to be changed.

We have an implementation of this framework, but we found that in practice to perform large scale experiments (Chapters 3 and 4) or to attack unsolved problems (Chapter 5) one needs a fine tuned implementation capable of competing with the state of the art and a generic framework is just too much computational overhead. Now, we have a library of code that corresponds to the design depicted in Figure 2.12, where we replaced inheritance with type generic code (*template code*) in many modules to avoid chasing pointers and to reduce the memory overhead associated with dynamically allocated memory.

We also have a less efficient interpreted system where problems can be readily specified by production rules and search problems solved with automatically generated lower bounds. In PSVN [HH99a] (Production System, Vector Notation), states are represented by fixed length vectors of labels. An operator has a left-hand side (*LHS*) and a right-hand side (*RHS*). Each position in the *LHS* and *RHS* vectors may be a constant (a label from the domain), a variable or an underscore (_). A variable in an operator's *RHS* must also appear in its *LHS*. An operator is applicable to s if its *LHS* can be unified with s . The act of unification binds each variable in *LHS* to the label in the corresponding position in s . Underscores in *LHS* act as “don't cares”. The *RHS* describes the state that results from applying the operator to s . The *RHS* constants and variables (now bound) specify particular labels and an underscore in a *RHS* position indicates that the resulting state has the same label value as s at that position. For example,

$$o_1 = \langle A, A, 1, -, B, C \rangle \rightarrow \langle 2, -, -, -, C, B \rangle \quad (2.13)$$

is an operator that can be applied to any state whose first two positions have the same value and whose label at the third position is 1. The effect of the operator is to set the value at the first index to 2 and to swap the labels at the last two

positions. For example,

$$o_1(\langle 4, 4, 1, 6, 2, 0 \rangle) = \langle 2, 4, 1, 6, 0, 2 \rangle$$

PSVN operators can be used to describe any finite state space whose states are fixed length vectors of labels chosen from a finite domain. In the extreme case, this could mean that each edge in the graph corresponds to a unique operator. However, for Permutation Groups (Rubik's Cube) or state spaces where the states are permutations of each other ($n \times n$ Puzzles), PSVN can provide compact space definitions. The action of a domain abstraction ϕ on a PSVN operator is to relabel the constants (integers) in the *LHS* and *RHS*. For Theorem 1 to apply, we have to be able to define Δ_ϕ^{-1} such that it preserves connectivity and show that we can assign a cost to each edge in S' which is not larger than the cost of any edge connecting the preimage states. The inverse of o_1 (Equation 2.13) is

$$o_1^{-1} = \langle 2, A, -, -, C, B \rangle \rightarrow \langle A, A, 1, -, B, C \rangle$$

Note, that the introduction of A at the second position in the *LHS* is clear from the context. However, not all PSVN operators are uniquely invertible. For example, consider

$$o_2 = \langle A, A, -, - \rangle \rightarrow \langle 2, 3, -, - \rangle$$

It is not possible to deduce from the resulting state the identity of the original bindings of A . However, we only require that the states generated by Δ_ϕ^{-1} include the inverses. If the original domain is $D = \{0, 1, 2, 3\}$, then

$$o_2^{-1} = \langle 2, 3, -, - \rangle \rightarrow \left\{ \begin{array}{l} \langle 0, 0, -, - \rangle \\ \langle 1, 1, -, - \rangle \\ \langle 2, 2, -, - \rangle \\ \langle 3, 3, -, - \rangle \end{array} \right\}$$

In essence, all possibilities are generated. ϕ is now applied to the inverses and the connectivity requirement of Theorem 1 is established. If all operator costs are 1, then so are the costs in the abstract space and the pattern database is built using a breadth first expansion from the abstract goal with the operator set $\phi(O^{-1})$. Otherwise we use Dijkstra's algorithm to calculate γ_u which naturally takes the minimum if there are more than one edges between two abstract states.

We actually built a totally automatic PSVN system (interpreter, abstraction and search) which could automatically solve specified problems and it was also used to obtain the results published in [HH99b, HH00].

The experiments were run on various Intel based Linux machines in our department (300-600 MHz CPU, 64-256 MB RAM). The SOP experiments were run on a Linux server with 2.4 GHz Intel Pentium IV processor and 1 GB RAM. The entire implementation for all experiments (including some which did not get included in this thesis) comprise over 70,000 lines of C++, Java and Scheme code.

Chapter 3

Lower Bounds from Domain Abstractions

In this chapter we investigate the performance of pattern databases experimentally. In previous work [CS98, Kor97, Ede01a] experiments involved only a few selected pattern databases. Although Korf has conjectured a predictable relationship between the size of the pattern database and its search performance [Kor97], at the time there was not enough data to support his conjecture. For the first time, we performed experiments that involved thousands of pattern databases to investigate the space-time trade-off [HH99b] and to evaluate simple statistics that can be used to predict which pattern databases are likely to perform better. In this chapter, we experimentally establish the space-time trade-off relationship and we also show that simple statistics are effective at ranking pattern databases with respect to search performance.

3.1 Experiments

Since we can generate a large number of domain abstractions for a given state space, we are naturally interested in knowing how they compare in practice. Korf in [Kor97] made two intuitive conjectures. The first predicts the number of nodes expanded of IDA* to be approximately $O(|S|/|S'|)$, where S and S' are the original and abstract spaces. The second is, in his words, “we can characterize the effectiveness of an admissible heuristic by its expected value over the problem space”. Since these conjectures were based on insight and were not derived analytically, we designed experiments which could provide experimental confirmation. In addition, due to the fact that we can generate many different pattern databases, our experiments can provide answers to the following questions.

- How much does doubling memory for a pattern database speed up search?
- How much do pattern databases of the same size differ in performance?
- Can simple statistics rank pattern databases with respect to search performance?

To have conclusive answers, we have to evaluate search performance on a large number of start states with a large number of pattern databases. This immediately rules out using large state spaces. While pattern databases were used to solve random instances of the 5×5 puzzle [KF02] and the Rubik’s Cube [Kor97], these spaces are far too large to perform experiments on this scale. Even the 4×4 puzzle is too large for this purpose. Today, a single instance can be solved under one second with large additive pattern databases¹. To establish the space-time

¹private communication with A. Felner

relationship, we would also have to solve instances with small pattern databases and these searches would take substantially longer. Therefore we chose the 3×3 puzzle as our primary testbed².

3.1.1 The 3×3 Puzzle

1	2	3
4	5	6
7	8	

Figure 3.1: 3×3 puzzle.

The 3×3 sliding tile puzzle space has 181,440 states. Only half of the possible permutations of the tiles are reachable by legal moves from any given state. Figure 3.2 shows the distribution of optimal distances of all states to the goal, when the blank is in the corner position (shown in Figure 3.1). The average distance to the goal is 21.97 moves. The diameter of the state space (longest shortest path between any two states) is 31. In our encoding, a state is a vector of the tile labels indexed by the tile positions, analogous to our state representation of the 2×2 puzzle in Chapter 2. Some domain abstractions which introduce multiple blanks result in non-surjective homomorphisms (as depicted in Figure 18 on page 27 for

²On a smaller scale, we have performed experiments with other state spaces as well, which also support the conclusions we draw in this chapter. Two other small state spaces were used to study the space-time trade-off [HH99b] and two other modest size state spaces were used to test if the pattern databases we ranked best also perform better in practice [HH04]

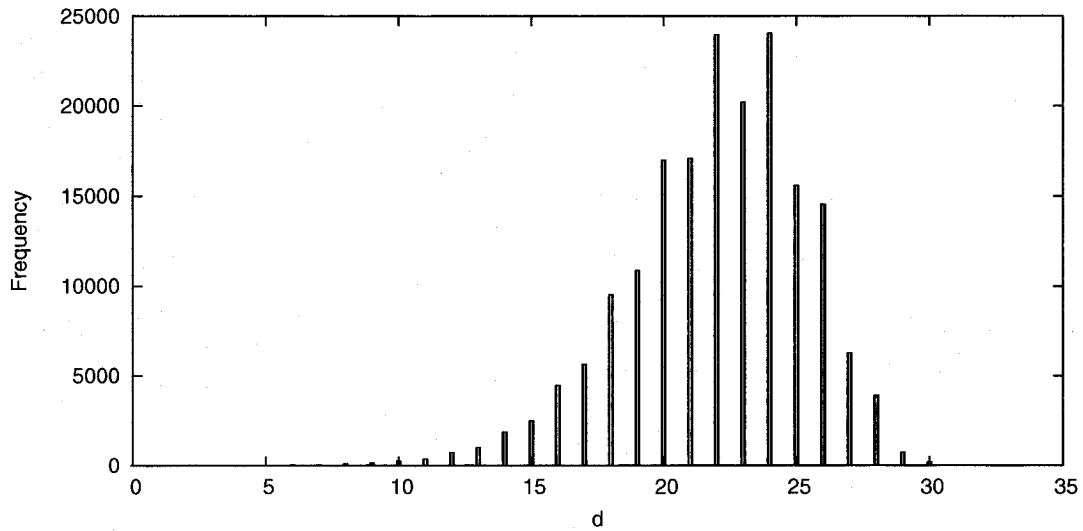


Figure 3.2: Distribution of Optimal Solution Distances for the 3×3 Puzzle

the 2×2 puzzle). Therefore we will only consider domain abstractions which preserve the uniqueness of the blank. There are altogether 4,140 non-isomorphic abstract spaces resulting from blank preserving abstractions. With our encoding, domain abstractions with the same granularity give rise to the same size pattern databases. Table 3.1 lists the different abstraction granularities (Γ), the size of the resulting pattern databases ($|\phi(S)|$) and the number of such abstractions in canonical form (k). The domain abstractions which also correspond to the pattern abstractions invented by Culberson and Schaeffer [CS94] are in bold font.

3.1.2 Experimental Setup

We have selected two sets of start states. T_1 is a randomly chosen pool of 1,000 distinct states which are exactly 22 moves away from the goal. The second set, T_2 , contains 1,000 randomly selected distinct states, where the overall average distance is also 22 moves. The 2,000 states in these two sets are solved with the

Γ	$ \phi(S) $	k	Γ	$ \phi(S) $	k
$\langle 8 \rangle$	9	1	$\langle 4, 2 \rangle$	7,560	420
$\langle 7 \rangle$	72	8	$\langle 3, 3 \rangle$	10,080	280
$\langle 6, 2 \rangle$	252	28	$\langle 3, 2, 2 \rangle$	15,120	840
$\langle 5, 3 \rangle$	504	56	$\langle 4 \rangle$	15,120	70
$\langle 6 \rangle$	504	28	$\langle 2, 2, 2, 2 \rangle$	22,680	105
$\langle 4, 4 \rangle$	630	35	$\langle 3, 2 \rangle$	30,240	560
$\langle 5, 2 \rangle$	1,512	168	$\langle 2, 2, 2 \rangle$	45,360	420
$\langle 4, 3 \rangle$	2,520	280	$\langle 3 \rangle$	60,480	56
$\langle 5 \rangle$	3,024	56	$\langle 2, 2 \rangle$	90,720	210
$\langle 4, 2, 2 \rangle$	3,780	210	$\langle 2 \rangle$	181,440	28
$\langle 3, 3, 2 \rangle$	5,040	280	$\langle \rangle$	181,440	1

Table 3.1: Abstraction Granularities for the 3×3 Puzzle.

4,140 different pattern databases with both the A* and IDA* algorithms. This altogether constitutes 16,560,000 searches. For each of these searches we record the total number of nodes expanded and the heuristic estimate h_0 for the start state. For IDA* searches we also count the total number of nodes expanded on the last cost level. Since it is expected that deeper searches expand more nodes, to determine the space-time trade-off we use the states in T_1 where the depth of the search is fixed.

On the other hand, we use T_2 when we evaluate how well our statistics rank search performance, as in a realistic application, the pattern databases are expected to perform well on random instances selected from the entire space.

We also have solved our test problems with the Manhattan Distance (MD) heuristic, which is a well known hand-crafted lower bound for the $n \times n$ puzzles. The value of the MD heuristic is the sum of block distances of the tiles (excluding the blank) from their goal positions.

3.2 Space-time Trade-off

Figure 3.3 shows the size of the pattern database (x axis) vs. the number of nodes expanded (y axis) during a search with that pattern database. Both axes are on the logarithmic scale. Each datapoint corresponds to solving the instances of T_1 (where each state is with $d = 22$ from the goal) with all pattern databases of the same size. For example, there are 56+28 pattern databases with size $m = 504$ (these are the ones with granularity $\langle 5, 3 \rangle$ and $\langle 6 \rangle$, Table 3.1 page 47), therefore there are two datapoints where $m = 504$, one represents 56,000 searches and

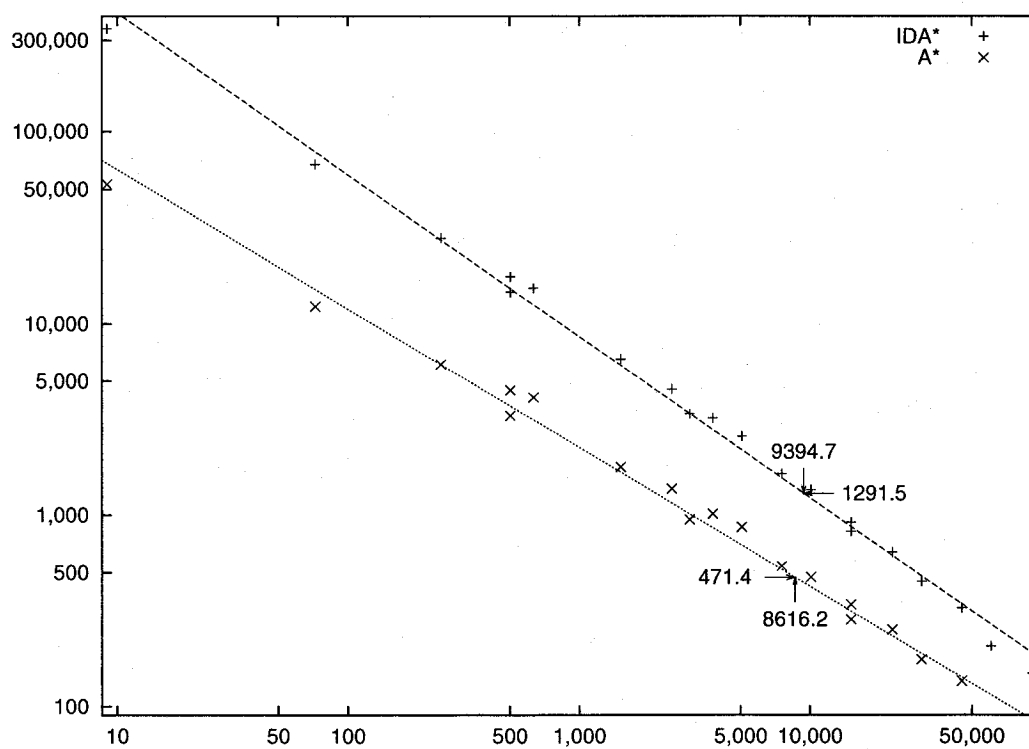


Figure 3.3: Size of Pattern Database vs. Nodes Expanded

the other 28,000. The plots in Figure 3.3 also identify the performance of the Manhattan Distance heuristic with our chosen pool of start states. On average, 1,291.5 nodes are expanded with IDA* and 471.4 nodes with A* with the MD heuristic. The least squares regression lines are also shown. Interpolating from the regression lines, IDA* and A* need a 10,000 entry pattern database (which is only 5.5% of the size of the original space) generated by our domain abstractions to beat the Manhattan Distance. It is very clear that the trade-off for both A* and IDA* is very predictable. The correlation between the datapoints and the regression lines is 0.994913 for IDA* and 0.990542 for A*. However, the slopes of the lines are not -1, which would indicate an exact one-to-one trade-off. The equation of the regression line for IDA* is

$$\log t_{\text{IDA}^*} = -0.84368 \log m + 14.8815 \quad (3.1)$$

and for A* it is

$$\log t_{\text{A}^*} = -0.725522 \log m + 12.7291 \quad (3.2)$$

where t is the number of nodes expanded during search with a pattern database whose size is m . If the slope of the regression line is in $(-1, 0]$, then doubling memory to store the pattern database actually results in more than half the nodes expanded. In these experiments this is the case for both IDA* and A*. While, expectedly, A* expands fewer nodes than IDA*, the trade-off is better for IDA*. For the 3×3 Puzzle, we can refine Korf's conjecture and state that with very high correlation the trade-off is

$$tm^\alpha = c \quad (3.3)$$

c is a constant and α is the slope of the regression line. α also seems to be search algorithm dependent. We also have observed over 0.9 least squares correlations

with two other state spaces [HH99b] with similar values of α .

Establishing this relationship is very important. It is clear that using more memory will speed-up search. The only counter example we have for this is the data point for A^* (Figure 3.3) where the pattern database with size 3,024 outperforms the one with 3,780 entries. The former expands 943.45 nodes on average while the latter 1011.87. The difference is relatively small both in pattern database size and in nodes expanded. It must be noted, however, that for both A^* and IDA^* the databases with 3,024 entries perform proportionally better than the pattern databases which are slightly larger. The pattern databases with size 3,024 have granularity $\langle 5 \rangle$ while the ones with size 3,780 have granularity $\langle 4, 2, 2 \rangle$ which means that there are 6 distinct tile labels in the $\langle 5 \rangle$ pattern databases and only 4 in the $\langle 4, 2, 2 \rangle$'s. This would indicate that when the abstractions give rise to approximately the same size pattern databases, the more label rich abstractions should be preferred. This is further corroborated by the abstractions with granularities $\langle 5, 3 \rangle$, $\langle 6 \rangle$, $\langle 3, 2, 2 \rangle$ and $\langle 4 \rangle$. The $\langle 5, 3 \rangle$ and $\langle 6 \rangle$ abstraction yield pattern databases with 504 entries. The $\langle 6 \rangle$ abstractions have 4 distinct tile labels and the $\langle 5, 3 \rangle$ abstractions have 3. The average number of nodes expanded with the granularity $\langle 6 \rangle$ pattern databases is 21% fewer with IDA^* and 35% fewer with A^* than with the granularity $\langle 5, 3 \rangle$ pattern databases. Similarly, the $\langle 4 \rangle$ and $\langle 3, 2, 2 \rangle$ granularity pattern databases are of the same size, but the $\langle 4 \rangle$ abstractions have 6 distinct labels and the $\langle 3, 2, 2 \rangle$'s 5. The $\langle 4 \rangle$ pattern databases expand 12% fewer nodes with IDA^* and 20% fewer nodes with A^* than the pattern databases with $\langle 3, 3, 2 \rangle$ granularity. This suggests that the original pattern abstractions, which are also included in our richer set of domain abstractions, are particularly effective. Nonetheless, there are many more domain abstractions than pattern

abstractions and it is clear from our experiments that larger pattern databases result in faster searches regardless of the form of the domain abstraction. In the case of the SOP (Chapter 5) we will rely on a rich set of domain abstractions since we will perform greedy searches in the space of abstractions.

The results of Figure 3.3 report the averages over all pattern databases of a given size. We are also interested in the range of nodes expanded among pattern databases of the same size. Figures 3.4 and 3.5 show the size of the pattern database (x -axis) and the number of nodes expanded (y -axis) for IDA* and A* respectively. Both axes are on the logarithmic scale. Each data point represents the searches with a single pattern database and the states in T_1 . The data points for the $\langle 5, 3 \rangle$ and $\langle 6 \rangle$ as well as for the $\langle 3, 3, 2 \rangle$ and $\langle 4 \rangle$ pattern databases are shown side by side so they are visually distinguishable, even though they actually overlap since the pattern databases are of the exact same size ($\langle 5, 3 \rangle$ and $\langle 4 \rangle$ on the left).

The horizontal lines (labeled MD) correspond to the performance of the Manhattan Distance. In the case of A* there are a few pattern databases with size 3,024 (1.67% of the size of the original space) which are actually better than MD. It is also clear from both figures that the best pattern database of a certain size is often better than an average one which is 2-3 times larger. For example, for A*, the best performing pattern database with size 3,024 is better than the worst with size 15,120. Therefore, it is clear, that the size of the pattern database alone is insufficient to predict its relative performance with respect to other pattern databases. The ranges of nodes expanded with the same size pattern databases varies, but on average there is a 2.5 fold difference between the best and the worst with IDA* and a 3 fold difference with A*. It is also evident from the figures that

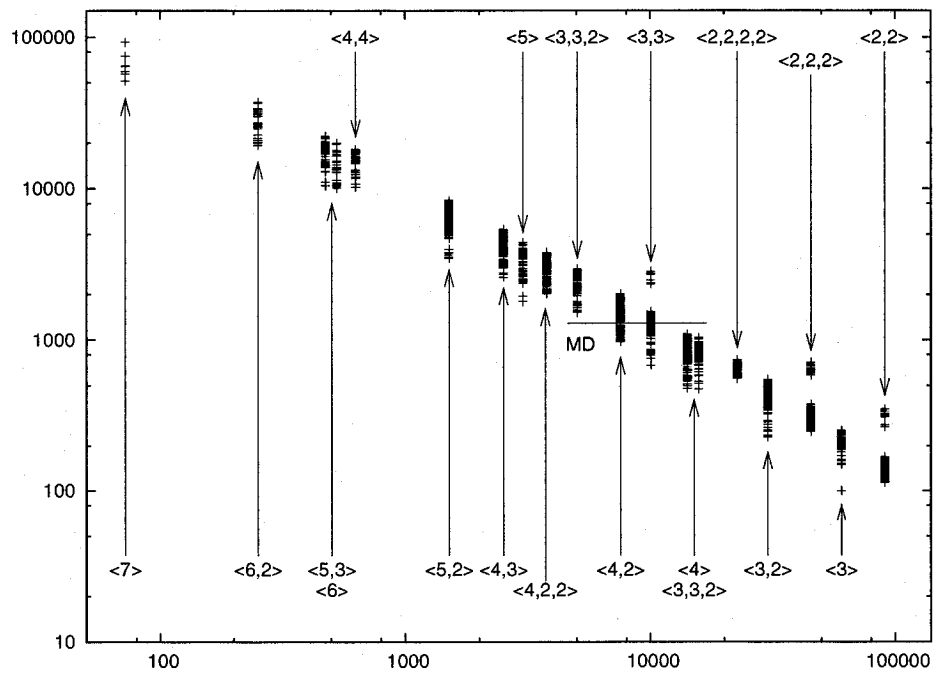


Figure 3.4: Pattern Database Size and Range of Nodes Expanded for IDA*.

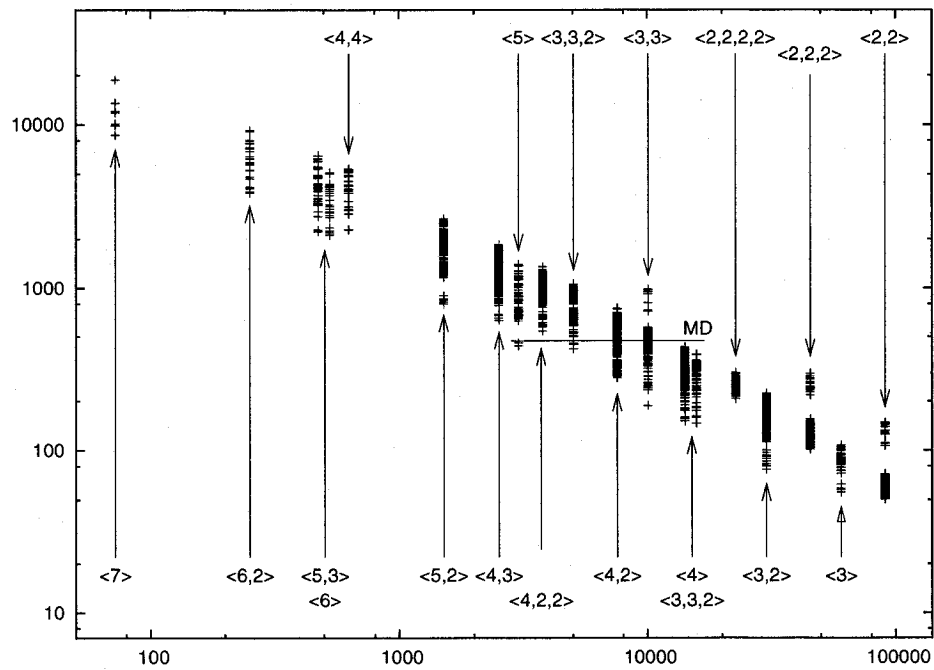


Figure 3.5: Pattern Database Size and Range of Nodes Expanded for A*.

the performance ranges are not always normally distributed as one would expect. For example, the pattern databases with granularities $\langle 5, 2 \rangle$, $\langle 5 \rangle$, $\langle 2, 2, 2 \rangle$ and $\langle 2, 2 \rangle$ show obvious discontinuities. In the case of the granularity $\langle 5, 2 \rangle$ and $\langle 5 \rangle$ pattern databases there are a few pattern databases which are clearly better than the majority. In fact, this group of pattern databases is composed of almost entirely the same instances for both IDA* and A*. On the other hand, in the case of the granularity $\langle 2, 2, 2 \rangle$ and $\langle 2, 2 \rangle$ pattern databases there is a group of databases which perform significantly worse than the majority and again, this group is the same for both IDA* and A*. While it is clear from Figure 3.3 that the average performance is very predictable from the size of the pattern database, from Figures 3.4 and 3.5 we can also tell that the best performing pattern databases of each granularity show much more discrepancy. It is still generally true that the larger the size of a pattern database the fewer the nodes expanded with the best pattern database of that size, but there are more counter examples. In fact, the best pattern databases with granularity $\langle 2, 2, 2, 2 \rangle$ (size 22,680) expand 47% more nodes with A* and 22% more nodes with IDA* than the best of the granularity $\langle 4 \rangle$ pattern databases (size 15,120).

Our estimate of time is nodes expanded, which is an implementation independent measure. We also have examined the trade-off with actual CPU time³. Figure 3.6 shows the size of the pattern database (x -axis) and the average number of CPU ticks spent on solving a single problem instance (y -axis). Both axes are on a logarithmic scale. This plot corresponds to the exact same runs as depicted in Figure 3.3. There are also two IDA* lines. IDA*-1 corresponds to the time it took to expand all nodes (ignoring goal nodes) on the last iteration while IDA*-2

³measured in CPU “ticks”

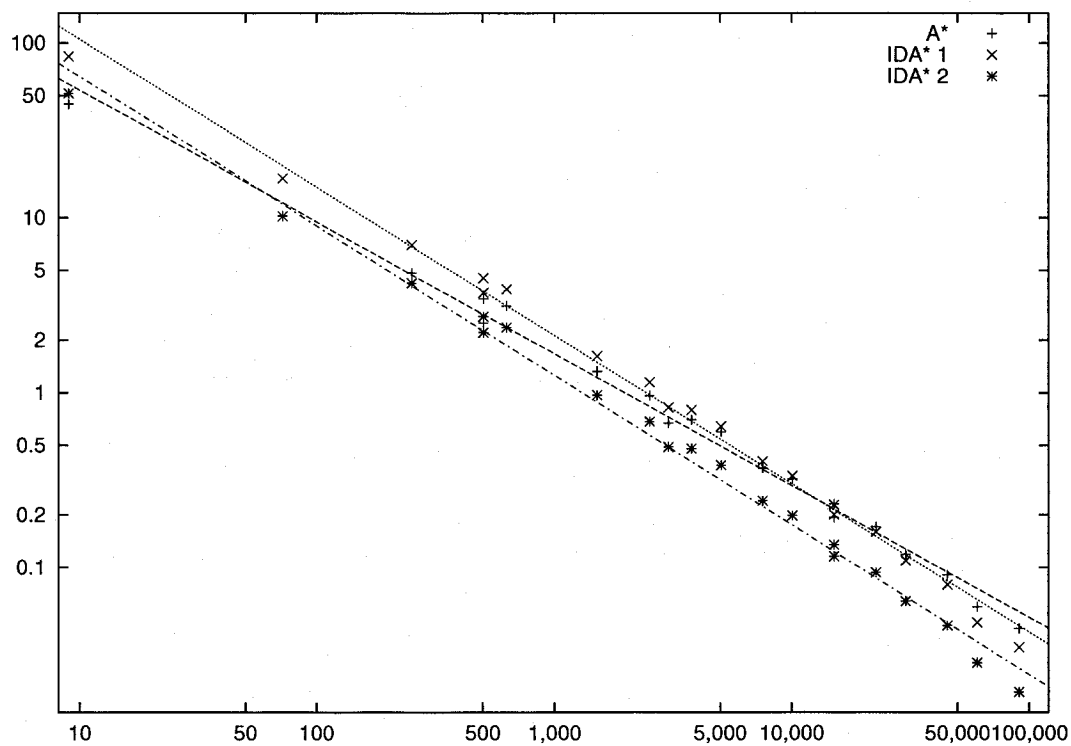


Figure 3.6: Size of Pattern Database (x -axis) vs. CPU ticks (y -axis)

shows the time to find the first goal node including all necessary iterations (as it would be run in practice to solve a problem). In essence, the IDA*-1 datapoints are the counterparts of the IDA* experiments in Figure 3.3. Since CPU time is a practical measure, we included the experiments that measure the actual time it would take to solve the problems.

As expected, the IDA* and A* lines are closer together than in Figure 3.3, since A* node expansions are more expensive (the nodes are expanded in priority order and there is duplicate detection). Unfortunately one cannot fairly compare the algorithms (A* and IDA*) with respect to speed by measuring CPU time. The actual running times very much depend on implementation details. We have made an effort to optimize both so we can perform the experiments on this large scale. We can, however, point out some observations which are not the artifacts of implementation. First, IDA* can find a goal much faster than expanding the last f -level in its entirety, even when we count all f -levels. This corroborates Korf *et al.*'s [KRE01] premise that the last iteration dominates IDA*'s performance. Second, the IDA* and A* regression lines have different slopes which are almost the same as in Figure 3.3. This suggests that nodes expanded is indeed a good measure to estimate search performance. It is also interesting, that in these experiments, the IDA*-1 and IDA*-2 lines have almost identical slopes which indicates that the number of nodes on the last f -level is also a good measure to compare performance of IDA* when it is stopped after finding the first goal node.

3.3 High-performance Pattern Databases

To account for the performance differences within pattern databases of the same size, we experimentally evaluated three statistics. The experiments were performed with the test set T_2 , since we want to predict performance on randomly selected states from the entire state space.

The three ranking statistics we considered were

- $K\&R(\phi)$: the formula derived by Korf and Reid in [KR98] (and somewhat refined in [KRE01]).
- $ave(\phi)$: the average of the heuristic values stored in the pattern database.
- $h_0(\phi)$: the average of the heuristic values associated with the states in the test set T_2 .

The $K\&R$ formula predicts the number of nodes expanded $E(s)$ for start state s on the last iteration of IDA* from the number of nodes and distribution of heuristic values over the brute-force search tree (BFST).

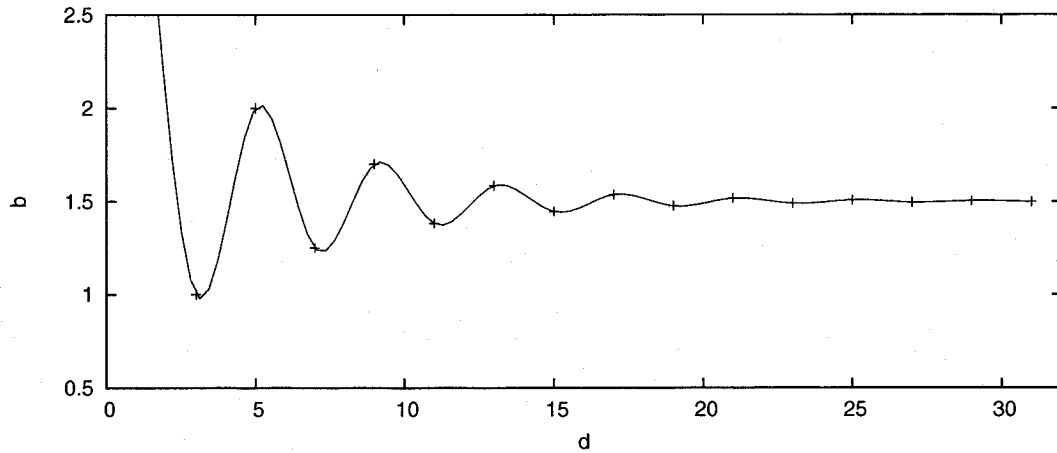
$$E(s) = \sum_{i=1}^d N(i, s) P(i, s, d - i) \quad (3.4)$$

$N(i, s)$ is the number of nodes in the BFST rooted at s on level i , d is the depth of the goal and $P(i, s, d - i)$ is the cumulative distribution of heuristic values on the i th level. $P(i, s, d - i)$ is the probability that a node's h -value in the BFST rooted at s at level i is less or equal to $d - i$. The correctness of this formula is given in [KRE01]⁴. To use the formula, we have to be able to estimate d , $N(i, s)$ and $P(i, s, d - i)$.

⁴The proof is given for the more general case where the operators are not unit cost and the

We will use $d = 22$ since this is the average distance to the goal in the entire state space and also in our test set.

The estimation of $N(i, s)$ will be different for A^* and IDA^* since their brute-force search trees are different. Edelkamp *et al.* [EK98, Ede01b] introduced a model which can be used to determine the number of nodes on a given level of the IDA^* BFST when the states can be grouped into branching equivalence classes. This model can also incorporate some form of duplicate node detection. For example, the branching factor of the BFST can significantly be reduced if the “grandparent” nodes are not generated. This does not effect the solution since no shortest path would include an operator followed by its own inverse. The model can also be used to calculate the asymptotic branching factor assuming that it converges to an equilibrium. In the case of the 3×3 puzzle, the branching factor actually converges to different values on even and odd depths. The convergence on the odd levels is immediate, however on even depths the convergence is slow as shown in Figure 3.7. Their model, however, can be used to efficiently obtain the exact number of nodes when the recursions corresponding to the model are solved by dynamic programming. This is what we do to get $N(i, s)$ exactly for IDA^* . For A^* , the brute-force search tree is a spanning tree over the state space and its shape is determined by the position of the blank. There are three different topology BFSTs for A^* corresponding to start states with the blank being in the middle, the corner or at the edge position. Since the 3×3 space is small, we can calculate the three different BFSTs. For larger problems, however, this approach summation is over discrete cost levels i . Since, in the general case, even the number of cost levels is hard to estimate, we only use the formula to predict performance when the operators are all of cost 1 and therefore cost and depth levels are the same.

Figure 3.7: d vs. b at Even Levels in the 3×3 Puzzle

is unrealistic. Therefore we will use the estimate

$$N(i, s) = b^i$$

The asymptotic branching factor for IDA* is $\sqrt{3}$ which we can derive from Edelkamp *et al.*'s model⁵. Since A* eliminates more duplicate nodes than IDA*, b must be smaller than $\sqrt{3} = 1.7321$. It also must be larger than $\log_{31} 181,440 = 1.4779$ because there are only a few new nodes generated at the last few levels (the diameter of S is 31 and $|S| = 181,440$). We chose the value $b = 1.60$ ⁶.

We will estimate $P(i, s, d - i)$ with a single distribution. Korf *et al.* in [KR98, KRE01] suggested that the *equilibrium* distribution would be an appropriate substitution. The equilibrium distribution is the distribution of heuristic values at large depth, assuming that the distribution converges to stable values. If the equi-

⁵While there are two different branching factors at even and odd depths, the single value which best estimates branching is $\sqrt{3}$, which is one of the eigenvalues of the branching matrix [Ede01b].

⁶We also evaluated other values between 1.4779 and 1.7321 and found that the choice of b in this range does not affect conclusions.

librium distribution cannot be derived analytically, Korf *et al.* suggests probing. In our experience, there is an excessively large number of probes needed before the distribution shows convergence and therefore it is not practical for our purposes. On the other hand, the distribution of heuristic values in the pattern database is easy to calculate. Moreover it is also the same as the overall distribution of heuristic values in the entire state space since each abstract state has exactly $|S|/|S'|$ preimages. In the case of A^* , this distribution is also the distribution over the brute-force search tree, since no duplicates are generated (our heuristics are monotone).

Therefore we will use the formulæ

$$E_{IDA^*}(s) = \sum_{i=1}^{22} N(i, s) P_{PDB}(22 - i) \quad (3.5)$$

$$E_{A^*}(s) = \sum_{i=1}^{22} 1.60^i P_{PDB}(22 - i) \quad (3.6)$$

to estimate number of nodes expanded with IDA^* and A^* .

We used the three statistics, $K\&R(\phi)$, $ave(\phi)$ and $h_0(\phi)$, to rank the pattern databases. The average number of nodes expanded ($E(\phi)$) over the test instances in T_2 with the pattern database was used to establish the true performance order. Our statistics also establish a predicted performance order which we compared to the true order of nodes expanded. To be able to interpret our results numerically, we defined a rank correlation measure. Let $R(\phi)$ be the predicted *rank* of performance for abstraction ϕ . $R(\phi)$ is an integer between 1 and the number of abstractions. If the same performance is predicted for ϕ_1 and ϕ_2 then their ranks are also the same. Let the proportion of incorrectly ranked pairs M (for

“misses”) be defined as

$$M(\Phi, \mathbf{E}, R) = \frac{\# \text{ of pairs } \langle \phi_1, \phi_2 \rangle \text{ s.t. } E(\phi_1) > E(\phi_2) \text{ but } R(\phi_1) \leq R(\phi_2)}{\text{total } \# \text{ of pairs } \langle \phi_1, \phi_2 \rangle} \quad (3.7)$$

Φ is the set of abstractions and $\mathbf{E}$ is a vector where each slot contains $E(\phi)$ when a particular $\phi \in \Phi$ is used. M ranges from 0 (perfect ranking) to 1 (reverse order). That is, we count it as a miss if we ranked ϕ_1 to be better than ϕ_2 but the search with the pattern database associated with ϕ_1 expands more nodes than searching with the one associated with ϕ_2 on the test instances of T_2 . An alternative measure is $M(\Phi, \mathbf{E}, R, t)$ where t is a tolerance given as a percentage. In this case, we count the pair $\langle \phi_1, \phi_2 \rangle$ as a miss if $E(\phi_1) > (1 + t)E(\phi_2)$. Our rank correlation then is defined as

$$\rho(\Phi, \mathbf{E}, R, t) = 2 \left(\frac{1}{2} - M(\Phi, \mathbf{E}, R, t) \right) \quad (3.8)$$

$t = 0$ gives $M(\Phi, \mathbf{E}, R)$. The value of ρ is in $[-1, +1]$. It is $+1$, if there are no misses and -1 if all pairs are misses. A value 0 indicates that there is no correlation between the rank prediction R and the actual performances $\mathbf{E}$. With a sufficiently high choice of t , no pairs are misses. Our rank correlation is different from the classical Spearman rank correlation, which is distance based in rank space. In our formula, $E(\phi)$ is not converted to rank as it is usually done in Statistics. Our primary concern is that with small tolerance t our ranking techniques should only make a few mistakes.

First we discuss our ranking experiments with IDA*. The $K\&R$ formula (Equation 3.4, page 57) was originally derived to predict the number of nodes expanded on the last iteration of IDA* in the worst case. The two substitutions we made are using the distribution from the pattern database and to set $d = 22$ (Equation 3.5, page 60). Since we have also performed searches with T_1 we can evaluate what

difference it makes to know d exactly. In Figure 3.8 the $K\&R$ prediction is on the x -axis and the actual number of nodes expanded on the last iteration are on the y -axis. Every data point corresponds to the average number of nodes expanded with the test set with one particular pattern database with granularity $\langle 3, 3, 2 \rangle$. The results in the figure correspond to two experiments (with T_1 and with T_2). When T_1 is used the only substitution is using the distribution of heuristic values from the pattern database as opposed to that of over the brute-force search tree. As expected, when d is known exactly, the prediction is more accurate (which is clear from Figure 3.8). The rank correlations are 0.9245 with the set T_1 and 0.7909 with T_2 . In both cases, however, the formula underestimates nodes expanded (much more so with T_2). This indicates that the distribution of heuristics in the pattern database and over the brute-force search tree are different. In particular, IDA^* expands duplicate nodes and it is expected that nodes with smaller h -values will be re-expanded more often. Hence our substitution P_{PDB} undersamples the low h -values in the brute-force search tree. Nonetheless, it is clear from the plots that $K\&R$ can distinguish between poor and good pattern databases in this experiment.

Figures 3.9 and 3.10 show the plots of the $ave(\phi)$ and $h_0(\phi)$ statistics (x -axis) vs. the actual number of nodes expanded with T_2 . It is clear from both figures that they both rank the abstractions relatively well, and for these databases, the ranking looks even better than the $K\&R$ ranking (Figure 3.8, page 63). The rank correlations are 0.84967 for $ave(\phi)$ and 0.86149 for $h_0(\phi)$ which are higher than the rank correlation 0.7909 for $K\&R(\phi)$.

Table 3.2 shows our rank correlations. The values in the square braces are the rank correlations with 5% tolerance. The columns, $\rho(\Gamma, R)$, correspond to the

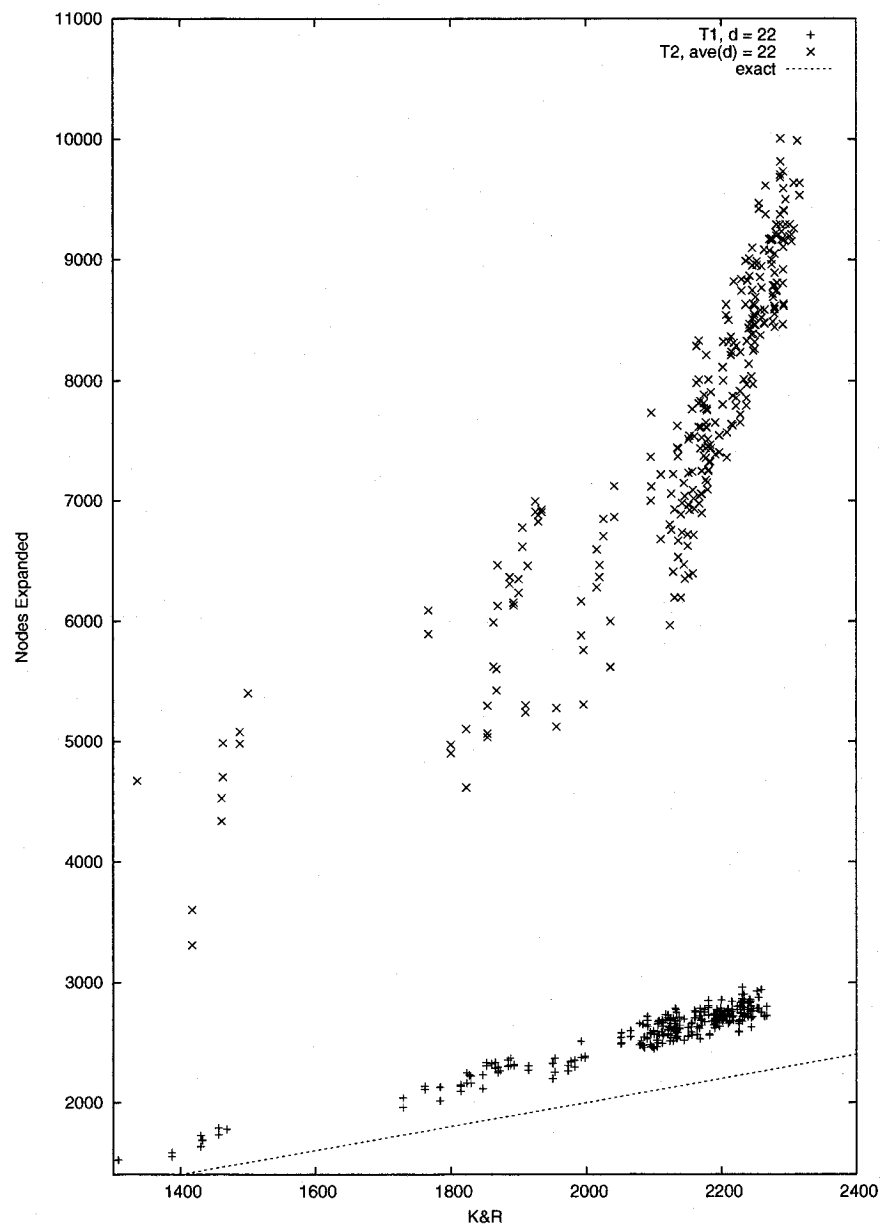


Figure 3.8: K&R Prediction vs. Actual Number of Nodes Expanded with IDA*.

Γ	$\rho(\Gamma, \text{K\&R})$ [5%]	$\rho(\Gamma, \text{ave})$ [5%]	$\rho(\Gamma, h_0)$ [5%]
$\langle 7 \rangle$	1.0000 [1.0000]	1.0000 [1.0000]	0.8571 [1.0000]
$\langle 6, 2 \rangle$	0.9365 [0.9735]	0.8783 [0.9577]	0.8677 [0.9524]
$\langle 5, 3 \rangle$	0.8519 [0.9506]	0.8818 [0.9494]	0.8727 [0.9506]
$\langle 6 \rangle$	0.8783 [1.0000]	0.9206 [0.9683]	0.8942 [0.9683]
$\langle 4, 4 \rangle$	0.8824 [0.9866]	0.8992 [0.9664]	0.8756 [0.9664]
$\langle 5, 2 \rangle$	0.8570 [0.9388]	0.8661 [0.9490]	0.8564 [0.9451]
$\langle 4, 3 \rangle$	0.8105 [0.9273]	0.8653 [0.9582]	0.8531 [0.9548]
$\langle 5 \rangle$	0.9026 [0.9714]	0.8909 [0.9610]	0.8818 [0.9571]
$\langle 4, 2, 2 \rangle$	0.7534 [0.8782]	0.8577 [0.9596]	0.8523 [0.9555]
$\langle 3, 3, 2 \rangle$	0.7909 [0.9210]	0.8497 [0.9621]	0.8615 [0.9686]
$\langle 4, 2 \rangle$	0.7785 [0.8890]	0.8277 [0.9322]	0.8243 [0.9302]
$\langle 3, 3 \rangle$	0.8140 [0.9254]	0.8354 [0.9414]	0.8419 [0.9480]
$\langle 4 \rangle$	0.8360 [0.9313]	0.8907 [0.9652]	0.8807 [0.9586]
$\langle 3, 2, 2 \rangle$	0.7582 [0.8859]	0.8141 [0.9310]	0.8278 [0.9430]
$\langle 2, 2, 2, 2 \rangle$	0.7784 [0.9062]	0.8037 [0.9260]	0.8234 [0.9421]
$\langle 3, 2 \rangle$	0.7836 [0.8950]	0.8343 [0.9366]	0.8360 [0.9400]
$\langle 2, 2, 2 \rangle$	0.7920 [0.8956]	0.8347 [0.9285]	0.8419 [0.9350]
$\langle 3 \rangle$	0.8896 [0.9649]	0.9221 [0.9896]	0.8935 [0.9805]

Table 3.2: Rank Correlation for All Pattern Databases with IDA*.

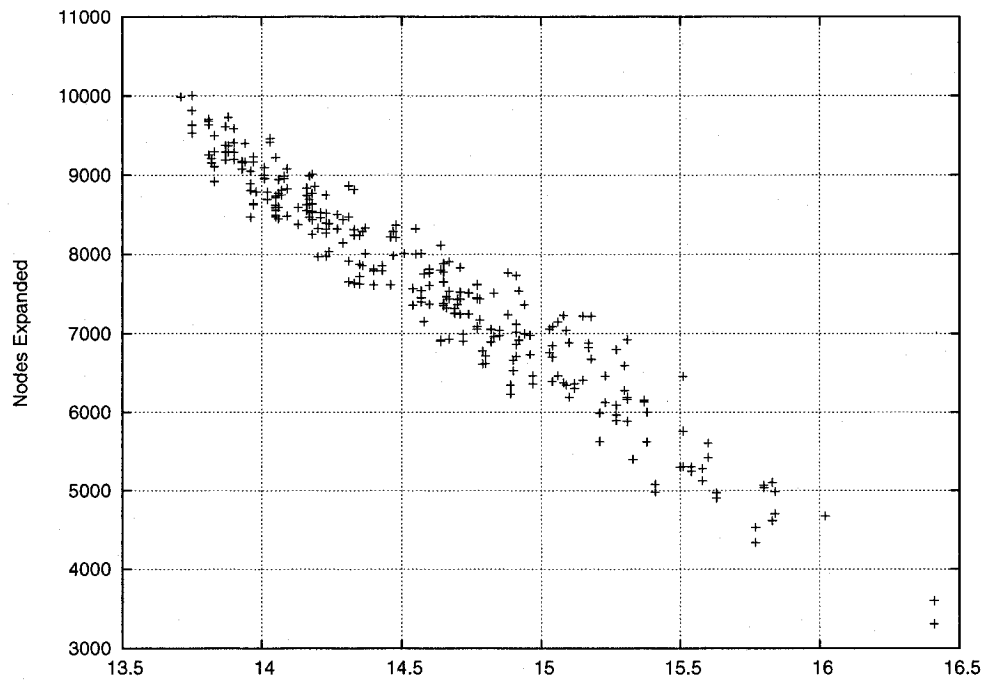


Figure 3.9: $ave(\phi)$ vs. Actual Number of Nodes Expanded with IDA*.

correlations (Equation 3.8) with the ranking method R with all pattern databases of granularity Γ and with the states in T_2 . For each granularity, the highest correlation value is shown in bold font. It is clear that all three statistics – K&R, ave and h_0 – are good to rank performance well. With few exceptions, the 5% tolerance the correlations are very accurate (over 0.93). The $ave(\phi)$ and $h_0(\phi)$ statistics, however, rank better than $K\&R(\phi)$. These results are encouraging since it seems that simple statistics are suitable to predict performance differences without performing actual searches.

Next we evaluate the same ranking functions with A*. Figures 3.11, 3.12 and 3.13 plot the statistics $K\&R(\phi)$, $ave(\phi)$ and $h_0(\phi)$ (x -axis) vs. the actual number of nodes expanded (y -axis) with T_2 , A* and the $\langle 3, 3, 2 \rangle$ granularity pattern databases. The counterparts for these figures for IDA* are Figures 3.8, 3.9 and

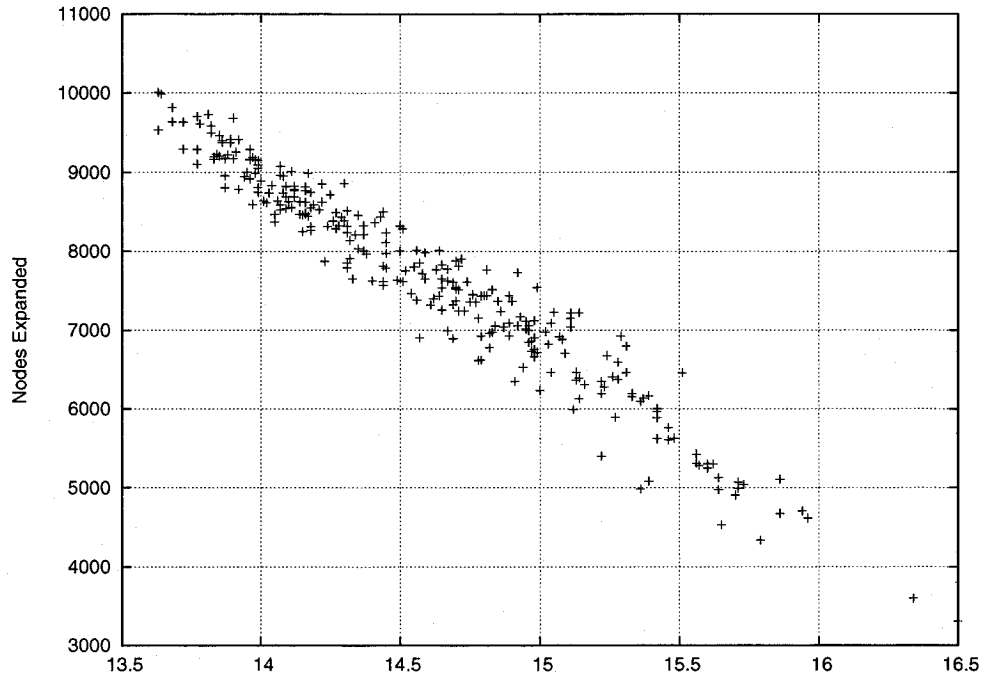


Figure 3.10: $h_0(\phi)$ vs. Actual Number of Nodes Expanded with IDA*.

3.10. The most outstanding difference is that with our substitutions, the $K\&R$ ranking is better for A^* than for IDA* with the $\langle 3, 3, 2 \rangle$ pattern databases, even though the formula was developed for IDA*. Table 3.3 shows the actual correlation values for all pattern databases. It is evident that $K\&R$ for A^* is better than the $ave(\phi)$ and $h_0(\phi)$ statistics for most granularities. In fact, in all cases, $K\&R(\phi)$ ranks better for A^* than for IDA* (compare the second columns in Tables 3.2 and 3.3). For IDA* (Equation 3.5) we use the exact number of nodes at level i and estimate the distribution of heuristic values over the brute-force search tree with the distribution in the pattern database. On the other hand, for A^* (Equation 3.6) the distribution is exact but we estimate the number of nodes at level i with b^i . This further suggests that a good estimate for the distribution of h -values over the brute-force search tree is important to rank the performance

Γ	$\rho(\Gamma, \text{K\&R})$ [5%]	$\rho(\Gamma, \text{ave})$ [5%]	$\rho(\Gamma, h_0)$ [5%]
$\langle 7 \rangle$	1.0000 [1.0000]	1.0000 [1.0000]	0.9286 [1.0000]
$\langle 6, 2 \rangle$	0.8995 [0.9577]	0.9365 [0.9841]	0.9101 [0.9788]
$\langle 5, 3 \rangle$	0.9078 [0.9610]	0.8974 [0.9688]	0.8545 [0.9390]
$\langle 6 \rangle$	0.8783 [0.9471]	0.9101 [0.9577]	0.9153 [0.9683]
$\langle 4, 4 \rangle$	0.9160 [0.9731]	0.8924 [0.9462]	0.8521 [0.9193]
$\langle 5, 2 \rangle$	0.9214 [0.9722]	0.8509 [0.9270]	0.8328 [0.9136]
$\langle 4, 3 \rangle$	0.9123 [0.9818]	0.8366 [0.9297]	0.8188 [0.9135]
$\langle 5 \rangle$	0.9351 [0.9909]	0.8688 [0.9338]	0.8688 [0.9338]
$\langle 4, 2, 2 \rangle$	0.8854 [0.9688]	0.8355 [0.9246]	0.8143 [0.9111]
$\langle 3, 3, 2 \rangle$	0.8989 [0.9870]	0.8911 [0.9665]	0.8718 [0.9634]
$\langle 4, 2 \rangle$	0.9040 [0.9731]	0.8034 [0.8986]	0.7886 [0.8886]
$\langle 3, 3 \rangle$	0.9081 [0.9823]	0.8566 [0.9472]	0.8444 [0.9436]
$\langle 4 \rangle$	0.9354 [0.9884]	0.8286 [0.9048]	0.8112 [0.8981]
$\langle 3, 2, 2 \rangle$	0.8686 [0.9714]	0.8532 [0.9554]	0.8523 [0.9562]
$\langle 2, 2, 2, 2 \rangle$	0.8604 [0.9722]	0.8744 [0.9777]	0.8670 [0.9755]
$\langle 3, 2 \rangle$	0.8748 [0.9660]	0.8491 [0.9488]	0.8447 [0.9484]
$\langle 2, 2, 2 \rangle$	0.8584 [0.9603]	0.8683 [0.9643]	0.8638 [0.9607]
$\langle 3 \rangle$	0.9390 [0.9987]	0.9325 [0.9961]	0.9078 [0.9753]

Table 3.3: Rank Correlation for All Pattern Databases with A*.

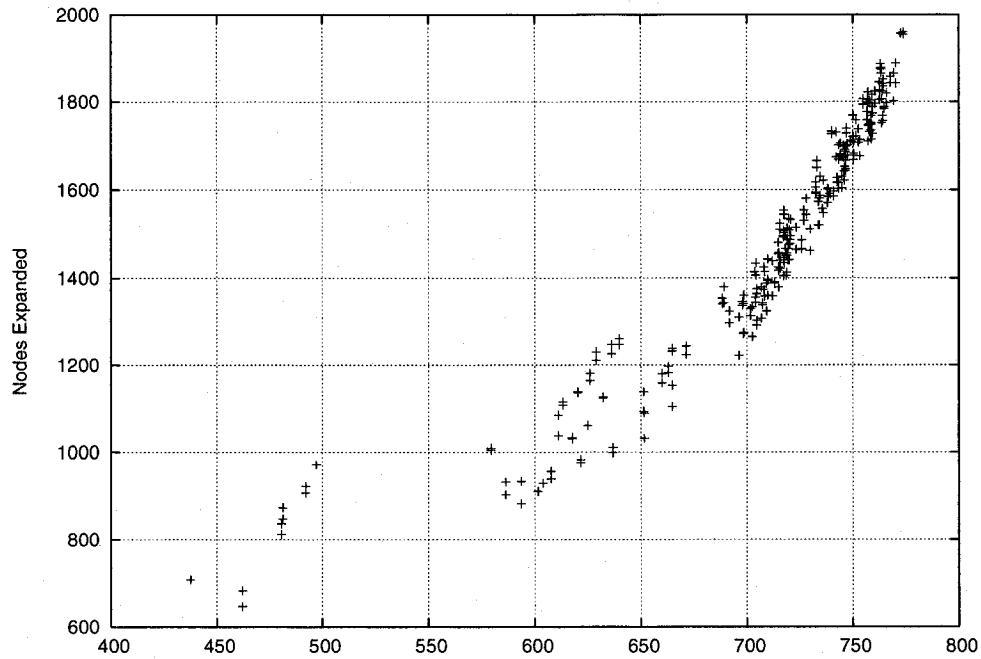


Figure 3.11: $K\&R(\phi)$ vs. Actual Number of Nodes Expanded with A^* .

of pattern databases accurately.

3.4 Chapter Summary

In this chapter we have experimentally established a predictable search-speed trade-off and showed that simple statistics from the pattern databases are good to rank search performance for IDA^* and A^* . While our experiments were performed with the 3×3 puzzle, we also have experimental evidence from other state-spaces [HH99b, HH00, HH04] (although on a much smaller scale) which corroborate these observations. The space-time trade-off and the merit of the average value of a heuristic to characterize the heuristic itself were first conjectured by Korf [Kor97], although, to our knowledge, we were the first who performed experiments with

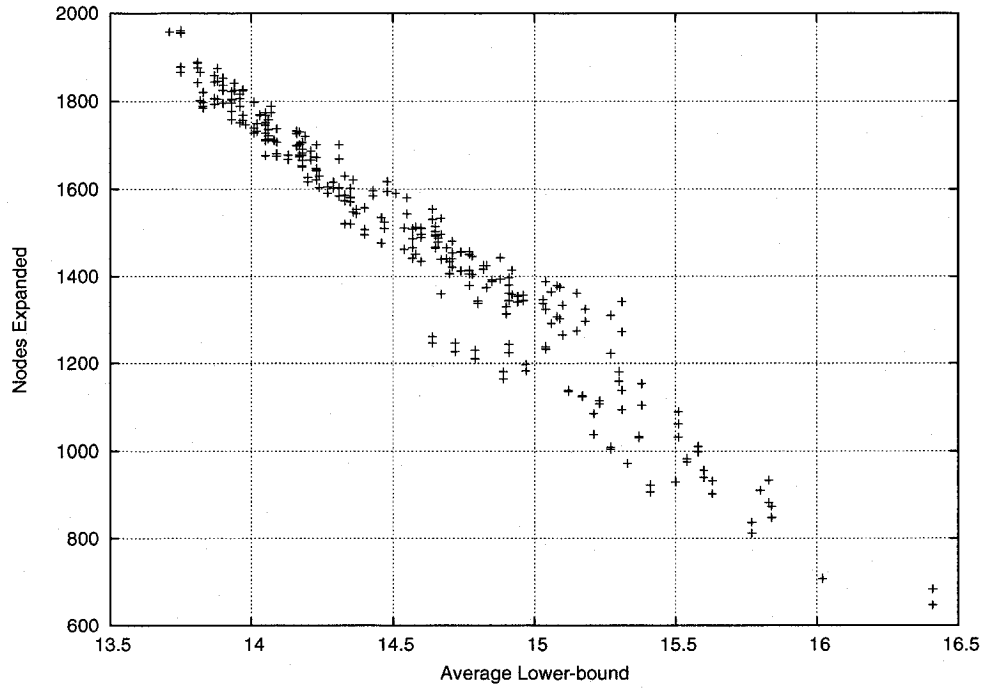


Figure 3.12: $ave(\phi)$ vs. Actual Number of Nodes Expanded with A^* .

a large number of heuristics to confirm these conjectures. These results clearly suggest that to achieve the best search performance, one should strive to build the largest pattern database which fits in memory and also try to optimize the $K\&R(\phi)$, $ave(\phi)$ and $h_0(\phi)$ statistics for the pattern database corresponding to the abstraction ϕ .

In this thesis we will make use of the space-time trade-off result directly. In Chapters 4 and 5 we derive abstractions automatically that yield the largest pattern databases which fit in our memory limits. It is more difficult to optimize the $K\&R(\phi)$, $ave(\phi)$ and $h_0(\phi)$ statistics directly (without building the pattern database). In Chapter 4, for some state spaces we will be able to design domain abstractions that avoid giving low heuristic values for the start states we need to solve. In Chapter 5, we will perform greedy local searches in the space of domain

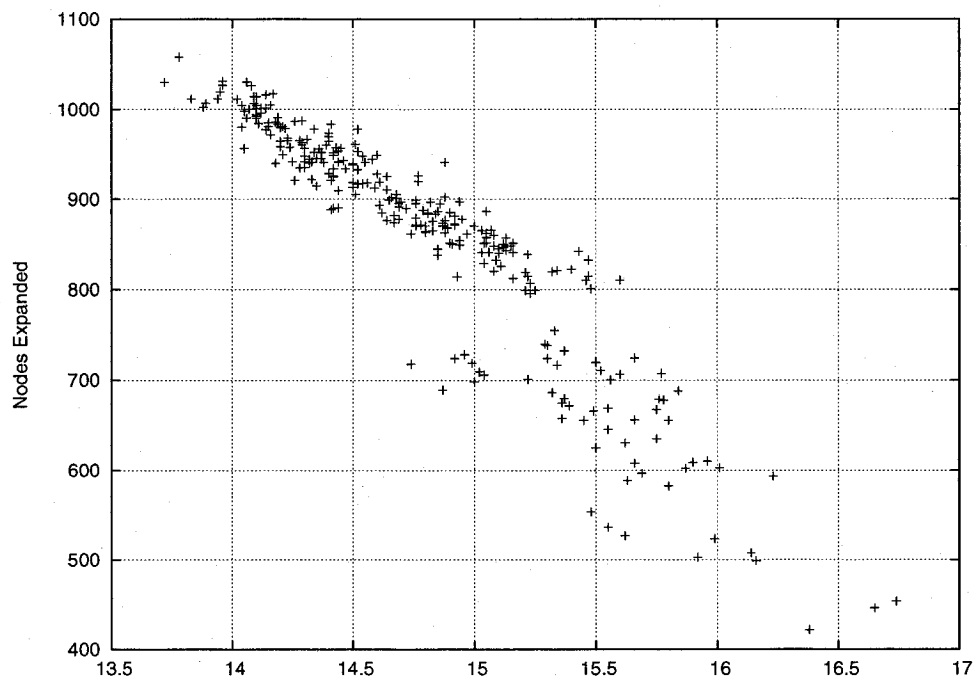


Figure 3.13: $h_0(\phi)$ vs. Actual Number of Nodes Expanded with A*.

abstraction to find those that will yield high heuristic values, which also means that the average and h_0 values are also likely to be higher.

Chapter 4

Searching for Macro Operators

With *macro-search* a solution can be derived or it can be decided that problem is not solvable in constant time. A solution is composed of a sequence of precalculated *macro-operators* which are retrieved from the *macro-table*. Each macro-operator in a solution sequence solves a single subgoal. We assume that the subgoal sequence is already chosen and our task is to find the macro-operators that correspond to these subgoals. We present a novel way of searching for macro-operators. For the first time, we use automatically generated heuristics to find optimal-length macro-operators. In fact, due to the speed-up achieved by using heuristic search, we show that we can reduce the number of subgoals and obtain shorter solutions. We use the Rubik's Cube to demonstrate our techniques. For this puzzle, a 44% improvement of the average solution length was achieved over macro-tables built with previous techniques. Macro-search is also an application where we can showcase automatically generated heuristics. Each subgoal represents a different search space and therefore a different heuristic is needed. Since

we have a way of generating heuristics automatically our method in many cases is more effective but not less general than building macro tables with previous techniques.

4.1 Subgoals and Macros

In macro-search, the solution path is constrained to pass through states which satisfy subgoal conditions. A state is represented by a vector of state variables

$$\mathbf{x} = \langle x_1, x_2, \dots, x_k \rangle$$

A subgoal corresponds to a subset of the state variables being set to their goal values (fixed). For example, a subgoal could be described as “ x_1 , x_4 and x_7 must be fixed”. Therefore a subgoal represents a set of states which satisfy the condition g_i . First, a sequence of subgoals is selected. If a state u satisfies the subgoal g_i we write $u \in g_i$. It is required that if $u \in g_i$ then $u \in g_j$ for all $j > i$. The macro-table stores a set of macro-operators for each subgoal. The macro-operators (or macros for short) are operator sequences in terms of the original operators which achieve a subgoal state with a single application. Suppose there are k subgoals associated with the macro-table M and we use it to obtain a solution for the problem s . The solution is always a sequence

$$\langle m_1, m_2, \dots, m_k \rangle$$

where m_i is a macro retrieved from the macro-table such that

$$(m_1 \circ m_2 \circ \dots \circ m_j)(s) \in g_j$$

M has a full set of applicable macro-operators which are retrieved on demand. M is usually implemented as a hash-table of macros. Given a state $u \in g_i$, $m_{i+1} = M[u]$ is the applicable macro-operator such that $m_{i+1}(u) \in g_{i+1}$.

Let us consider the 2×2 puzzle as an example. Our state space representation is as follows.

$$\langle \underbrace{x_1}_{\text{top-left}}, \underbrace{x_2}_{\text{top-right}}, \underbrace{x_3}_{\text{bottom-left}}, \underbrace{x_4}_{\text{bottom-right}} \rangle$$

In state $s = \langle 3, 1, 0, 2 \rangle$, the tile 3 is at the top-left position and the blank (label 0) is in the bottom-left corner. The goal state is $\langle 1, 2, 3, 0 \rangle$. The original operators are U , L , R and D for moving a tile adjacent to the blank “up”, “left”, “right” or “down”. We chose the following subgoal sequence

1. Bring the blank tile into the bottom-right corner.
2. Bring tile 1 into the top-left corner.
3. Bring tile 2 into the top-right corner.
4. Bring tile 3 into the bottom-left corner.

The entries for the macro-table corresponding to subgoal 1 are shown in Figure 4.1. The blank can be in any of the four possible positions (shown as the dark tile). The $\bullet$'s covering the tiles indicate that for solving this subgoal it is irrelevant how the other tiles are arranged. To solve the first subgoal for state $s = \langle 3, 1, 0, 2 \rangle$, we retrieve the macro

$$M[\langle 3, 1, \mathbf{0}, 2 \rangle] = \langle L \rangle$$

This corresponds to the blank positioned in the bottom-left corner (indicated by the bold 0). Applying L to s , we obtain $s_1 = \langle 3, 1, 2, 0 \rangle$ where the blank is in

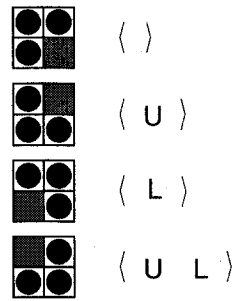


Figure 4.1: Macro-table Entries Corresponding to Subgoal 1.

its goal position in the bottom right corner. We now proceed to satisfy the next subgoal. The macro-table entries for subgoal 2 are shown in Figure 4.2. In all

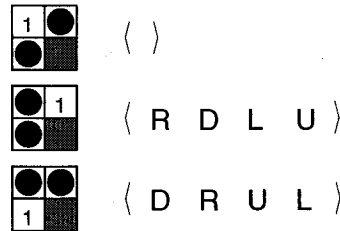


Figure 4.2: Macro-table Entries Corresponding to Subgoal 2.

entries the blank tile is in the bottom-right corner, because it is assumed that the macros will only be applied to states which satisfy subgoal 1. The next subgoal is to fix tile 1.

$$M[\langle 3, 1, 2, 0 \rangle] = \langle R D L U \rangle$$

which applied to s_1 brings it to the goal state $\langle 1, 2, 3, 0 \rangle$. In fact, for the 2×2 puzzle every state that satisfies goal 1 *and* goal 2 must be the goal state so there is no need for any more macro entries. The solution is $\langle L R D L U \rangle$ (the concatenation of $\langle L \rangle$ and $\langle R D L U \rangle$).

While this is a very simple example, it is sufficient to point out some of the issues with macro-search. First, it is no coincidence that the first subgoal is to fix the

blank. The position of the blank alone determines which operators apply. We assume that the given subgoal sequence is correct but as this example shows, the subgoal sequence in general may not be any sequence of state variables. Second, the solution sequence we derived for s , namely $\langle L R D L U \rangle$, is suboptimal. In this case, however, the L is followed by R in the sequence. These two operators are inverses of each other and therefore they can be canceled. Post processing local search can often shorten the solution obtained from the macro-table but it is not guaranteed that it results in an optimal solution.

4.2 Macro-search

The premise of macro search is that it is cheaper to solve the subgoals individually than to solve the problem as a whole optimally. It is important that we gain insight how this conjecture can be true. The state spaces corresponding to our choice of subgoals for the 2×2 puzzle are depicted in Figure 4.3. The first state space S_1 which corresponds to the first subgoal g_1 is much smaller than the original state space. Therefore we expect that searches in this smaller space are also faster. The state space S_2 (for subgoal g_2), however, is exactly as large as the original state space (Figure 2.1, Page 13). In fact, usually it is the case that the search space for the last subgoal is as large as the original state space. Where do the savings come from? In the figure, we framed the states which are the entries in the macro-table for subgoal 2. The goal state is identified by the arrow. The entry states are 4 operator moves away from the goal, which is also 2 moves less than the furthest state from the goal. For macro-search to be appropriate for a state space, the entries in the last few subgoals, which are consequently

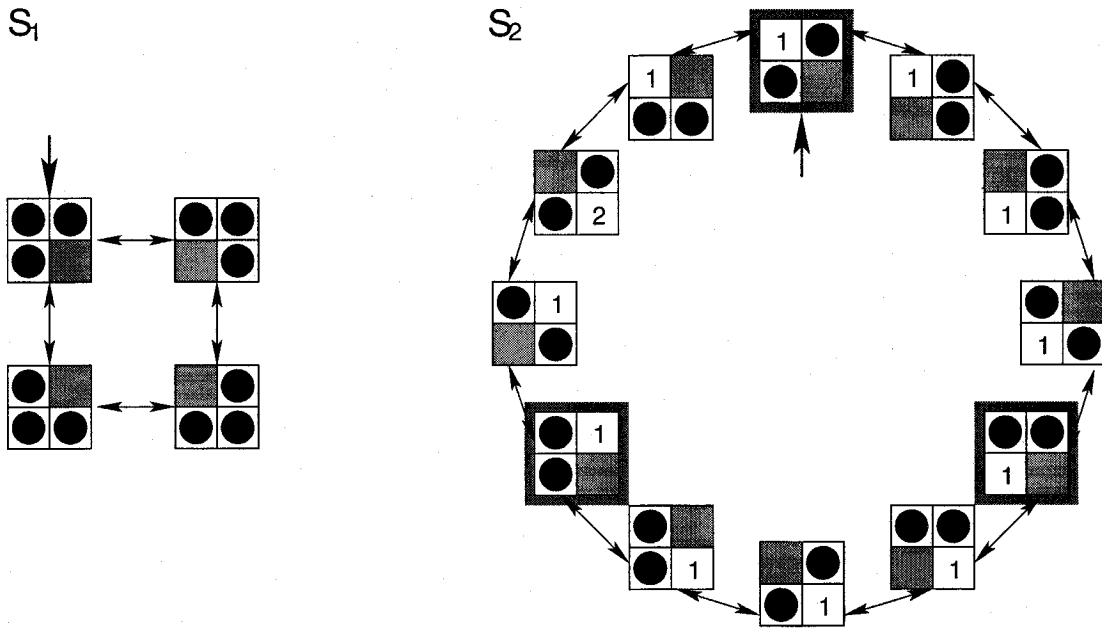


Figure 4.3: State-spaces Corresponding to Subgoals 1 and 2.

associated with larger state spaces, must be much closer to the goal than most states. In our example, it is clear, that macro search would not save us anything. On the other hand macro-search is quite appropriate for our test application, the Rubik's Cube and larger versions of the $n \times n$ puzzle.

A macro is *optimal* if no shorter operator sequence exists which solves the given subgoal. Since the solutions obtained by macro-search are usually not optimal, it is not that important that the macros are themselves optimal. Of course, the shorter the macros the shorter the solutions.

4.2.1 Uninformed Search

Traditionally macro-operators were found by uninformed search techniques. *Macro composition* is a technique developed by Sims [Sim70]. Suppose macros m_1 and

m_2 have the same invariant state variables. The idea is to compose m_1 with m_2^{-1} to obtain a new macro $m_1 m_2^{-1}$ which has extra invariants. This composition technique is integral to the Schreier-Sims method [But91], which is also one of the most important algorithms in Computational Group Theory. The *point stabilizer chain* of permutation groups is analogous to a macro-table where the macro-operators are permutations. In the original algorithm, however, these permutations are not stored as compositions in terms of the original generators. The data structure is used for calculating the order of groups and to decide group membership. In essence, the macros are generators of a subgroup which correspond to a subgoal. Nonetheless, it is straightforward to modify this algorithm such that it “remembers” how the generators were obtained. The solutions derived this way are usually much worse than optimal. The Schreier-Sims method also requires that state space is a permutation group.

Partial-match, bi-directional search [Kor85c, Kor83a] is a breadth-first expansion from the start and the goal state simultaneously. When the two search frontiers meet, an optimal macro has been found. The drawback of this method is that the size of the search frontier is exponential in search depth and therefore it is not applicable to search for long macros.

4.2.2 Heuristic Search

To find macros, we are going to use the IDA* [Kor85b] algorithm with automatically generated pattern database heuristics. Since IDA* only uses memory proportional to search depth, we can use practically all our memory to store pattern databases. Since the values in the pattern database H_i for subgoal g_i also

provide valid lower bounds to search in g_j for $j > i$, we will use the lower bound value

$$h(u) = \max(H_{j-1}[u], H_j[u])$$

for state u to search in the state space associated with g_j .

4.3 Reducing the Number of Subgoals

In previous works with macro-operators [Kor83a, Kor85c], each subgoal corresponds to fixing one state variable at a time. However, it is easy to see that a subgoal just as well could be chosen such that it requires a subset of the variables to be fixed. With the 2×2 puzzle example we showed that in general the subgoal sequence may not be any sequence of the state variables. In that case the blank had to be positioned first so it is known which operator applies. On the other hand it is clear that if

$$G = \langle g_1, g_2, \dots, g_k \rangle$$

is a valid subgoal sequence then merging any consecutive sequence of subgoals which retains the overall order is also a valid subgoal sequence. In other words, we can always choose a consecutive subsequence of subgoals $\langle g_i, g_{i+1}, \dots, g_j \rangle$ and make it a single subgoal.

If the number of subgoals is reduced, we expect the solution to be shorter. In fact, if all macros are optimal and G_2 is derived from G_1 with subgoal merging, then a solution corresponding to G_2 cannot be longer than the one corresponding to G_1 . Subgoal merging, however, will come with a price. First, the state spaces corresponding the subgoals are likely to be larger. Second, the number of entries

for a single subgoal will explode. Suppose that our merged subgoal is to fix state variables $\langle x_1, x_2, \dots, x_n \rangle$ where $x_i \in D_i$. Assume that these state variables are independent. If solved individually, then the total number of macro entries for the n subgoals is

$$\sum_{i=1}^n |D_i|$$

but when solved together

$$\prod_{i=1}^n |D_i|$$

macros are needed. Fortunately, independence does not always apply, but the trend is clear. Macro amortization is potentially a problem if more than one state variable is fixed in a single subgoal. If there is a large number of macros associated with a subgoal, an individual macro will only be rarely used. We suggest three solutions.

1. Choose the number of state variables so the number of macros needed is manageable.
2. When the number of macros is large for a subgoal, do not precalculate them but search for them on demand and then store them so they do not have to be searched for again.
3. If the number of macros is excessively large, solve the subgoal with search and do not store the macro.

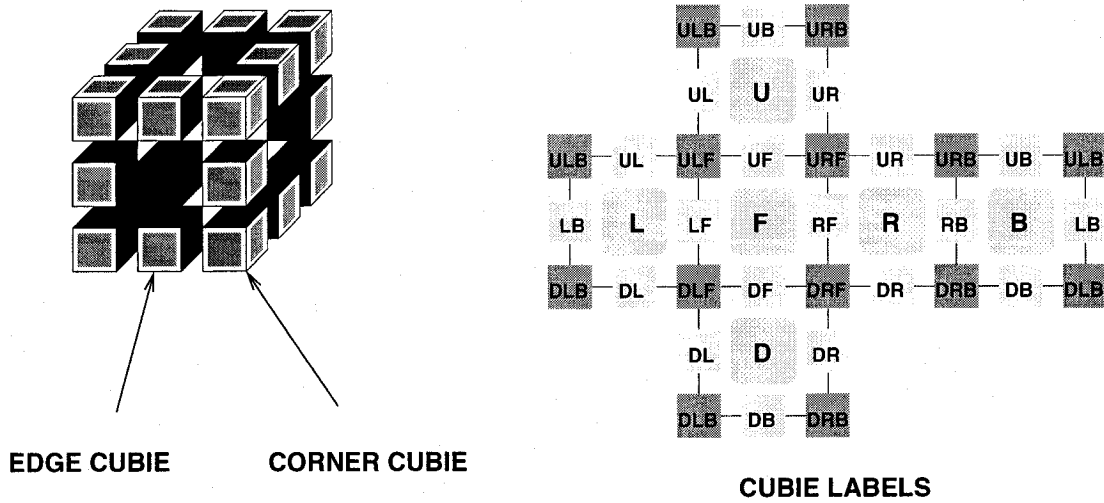
The latter two will result in less predictable search performance. On the other hand, if we can solve a larger subgoal quickly, then searching on demand could be worth the additional search time. In our case, we can generate heuristics automatically and therefore we can perform fast searches. Instead of storing the

macros for a subgoal when the number of macros is very large, we store the pattern database which we search with. We will demonstrate this with the Rubik's Cube where we build a macro-table where some of the subgoals are solved with on-demand heuristic search while the rest are solved by retrieving the precalculated macros.

4.4 The Rubik's Cube

The Rubik's Cube is a well known combinatorial puzzle often used as a benchmark problem for new algorithms. Optimal solutions (a dozen or so) were only recently obtained [Kor97]. Finding short solutions quickly still remains a challenge. This puzzle was also among the ones Korf considered in his original macro work [Kor83a] and it still fascinates many people¹. It has also been solved in many different ways and therefore we can benchmark our approach. The cube has six colored faces with nine colored stickers each. The moves are the quarter turns of the faces. The goal of the game is to bring the same color stickers back on each face after the puzzle has been scrambled. Structurally the cube is made up of twenty six little blocks called *cubies*. There are eight corner cubies each with three different colored stickers, twelve edge cubies with two colored faces and six center cubies with one color sticker each (Figure 4.4). The center cubies do not move with respect to each other, hence they should not be part of an encoding of the space. The space has $4.3 \cdot 10^{19}$ states. The quarter turn operators U , L , R , F , B and D correspond to a clock-wise quarter turn of the *up*, *left*, *right*, *front*,

¹International Rubik's Cube solving championships are still being held and the "Cube Lover's" news group is still currently active.

Figure 4.4: The *Rubik's Cube*

back and *down* faces respectively. We count two consecutive quarter turns of the same face in the same direction as a single move ($B B = B^2$). The edge and corner cubies are labeled as shown in Figure 4.4.

Our state representation corresponds to four vectors

$$\begin{aligned}
 v_1 &= \left\langle \begin{array}{cccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_2 &= \left\langle \begin{array}{cccccccccccc} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 & y_9 & y_{10} & y_{11} & y_{12} \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle \\
 v_3 &= \left\langle \begin{array}{cccccccc} z_1 & z_2 & z_3 & z_4 & z_5 & z_6 & z_7 & z_8 \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_4 &= \left\langle \begin{array}{cccccccccccc} w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 & w_9 & w_{10} & w_{11} & w_{12} \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle
 \end{aligned}$$

v_1 and v_2 are indexed by the positions of the corner and edge cubies respectively. The domains of the state variables x_i and y_i are also the labels of the corner and edge cubie labels. For example, if $x_3 = DLF$, then the cubie labeled DLF is in

the position *ULF*. v_3 and v_4 are the orientations of the corner and edge cubies. The corner cubies can be oriented in 3 different ways (labels 0, 1 or 2) and the edge cubies in 2 ways (labels 0 or 1). In the goal state, all cubie labels in v_1 and v_2 are at their home positions and all orientations in v_3 and v_4 are 0. In our encoding, each quarter twist changes the orientation of an edge cubie, while the corner cubie orientations change according to the rules depicted in Figure 4.5. For example, if the corner cubie is at the *ULF* position (shown by the bold

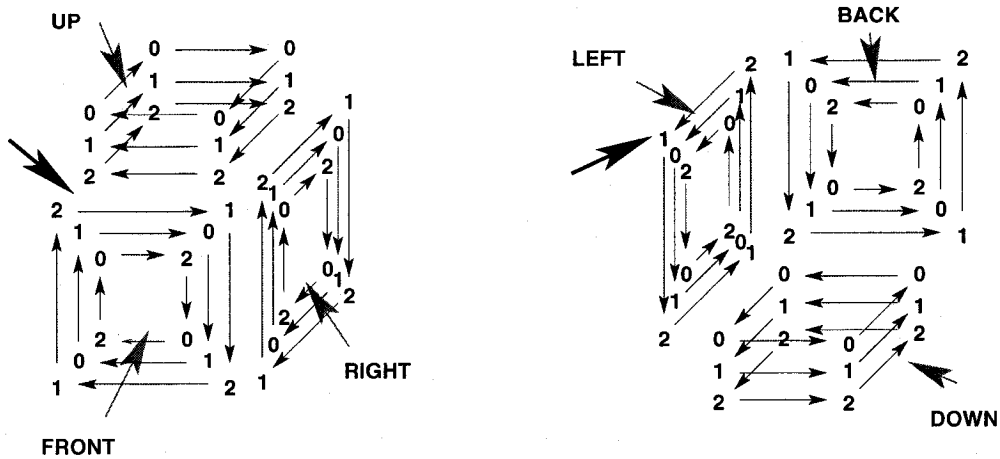


Figure 4.5: Corner-cubie Orientations.

arrow) currently with orientation 2 then turning the front face clockwise changes the orientation to 1, turning the up face does not change the orientation and twisting the left face clockwise changes its orientation to 0.

4.5 Domain Abstractions

First, we show an actual example of a subgoal. Consider the subgoal $\langle x_6, z_6, y_{12}, w_{12} \rangle$ where the positions and the orientations of cubies *RF* and *DRF* are to be fixed and all top face cubies were fixed for previous subgoals. Figure 4.6 shows the goal

state (left) and a start state (right) for a macro that needs to be calculated for this subgoal. The cubies to be fixed are identified by the arrows. The dark balls

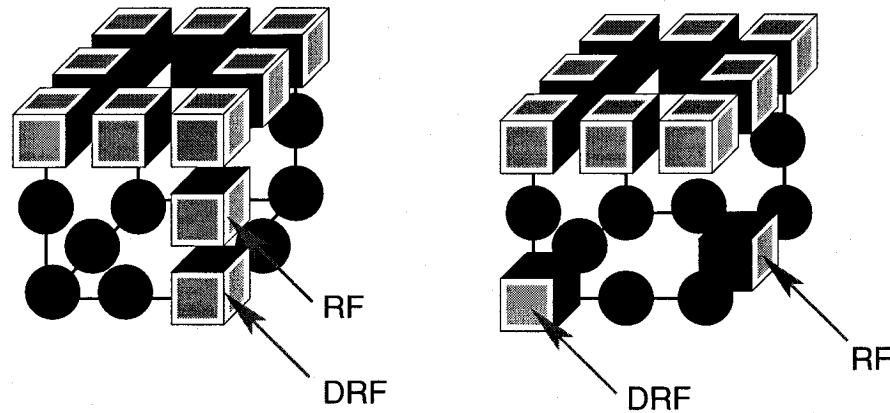


Figure 4.6: Subgoal States for the Rubik's Cube.

represent the cubies which will be fixed by the macros of subsequent subgoals. In our state representation the goal state for this macro operator is

$$\begin{aligned}
 v_1 &= \left\langle \begin{array}{cccccccc} ULB & URB & ULF & URF & \bullet & DRF & \bullet & \bullet \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_2 &= \left\langle \begin{array}{cccccccccccc} UB & UL & UR & UF & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & RF \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle \\
 v_3 &= \left\langle \begin{array}{cccccccc} 0 & 0 & 0 & 0 & \bullet & 0 & \bullet & \bullet \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_4 &= \left\langle \begin{array}{cccccccccccc} 0 & 0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 0 \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle
 \end{aligned}$$

and the start state is

$$\begin{aligned}
 v_1 &= \left\langle \begin{array}{cccccccc} ULB & URB & ULF & URF & DRF & \bullet & \bullet & \bullet \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_2 &= \left\langle \begin{array}{cccccccccccc} UB & UL & UR & UF & \bullet & \bullet & RF & \bullet & \bullet & \bullet & \bullet & \bullet \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle \\
 v_3 &= \left\langle \begin{array}{cccccccc} 0 & 0 & 0 & 0 & \bullet & 2 & \bullet & \bullet \\ ULB & URB & ULF & URF & DLF & DRF & DLB & DRB \end{array} \right\rangle \\
 v_4 &= \left\langle \begin{array}{cccccccccccc} 0 & 0 & 0 & 0 & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1 \\ UB & UL & UR & UF & DF & DL & DR & DB & LB & LF & RB & RF \end{array} \right\rangle
 \end{aligned}$$

Since we have 4 vectors of state variables in a state representation all with different domains, we need 4 abstractions to derive the abstract states. ϕ_1 , ϕ_2 , ϕ_3 and ϕ_4 are individually applied to v_1 , v_2 , v_3 and v_4 . The corresponding domains D_1 , D_2 , D_3 and D_4 also include the $\bullet$ which is used for the cubie labels and orientations which will be fixed by the macros of later subgoals. The scope of ϕ_1 and ϕ_2 are the entire state vectors v_1 and v_2 . On the other hand, ϕ_3 and ϕ_4 act on the individual variables in v_3 and v_4 .

One of our aims is to generate domain abstractions which induce the largest possible pattern databases that fit in memory. As we showed in Chapter 3, the larger the pattern database the faster the searches. Therefore we will start with the identity abstraction and will make it successively coarser until the pattern database it induces fits within our memory limits. Initially $D'_i = D_i$. Our domain abstraction generator applies one of the following rules to make the abstraction less fine-grained.

1. Choose two non $\bullet$ labels from D'_1 and merge them into one.

2. Choose two non $\bullet$ labels from D'_2 and merge them into one.
3. Choose one non $\bullet$ label from D'_3 and assign it the label $\star$
4. Choose one non $\bullet$ label from D'_4 and assign it the label $\star$

The $\star$ label, from the orientation point of view, is like the $\bullet$. The operators do not change the $\star$ orientation. We keep $\bullet$ and $\star$ separate to distinguish the ones ($\star$) that belong to this subgoal. For example, a few iterations of the rules above could yield the domain abstractions

x	ULB	URB	ULF	URF	DRF	$\bullet$
$\phi_1(x)$	A	A	B	B	DRF	$\bullet$

x	UB	UL	UR	UF	RF	$\bullet$
$\phi_2(x)$	A	A	UR	A	RF	$\bullet$

x	ULB			URB			ULF			URF			DRF			•
	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	
$\phi_3(x)$	0	1	2	★	★	★	0	1	2	★	★	★	0	1	2	●

x	UB	UL	UR	UF	RF	$\bullet$
	0	1	0	1	0	1
$\phi_2(x)$	$\star$	$\star$	$\star$	$\star$	0	$\bullet$

After each application of a rule we determine the size of the resulting abstract space and if it is still too large then we keep making the abstractions coarser.

In our initial experiments we observed that our abstraction generation rules

should not select the labels and the orientations of the cubies to be fixed for the subgoal. If these are abstracted, then the heuristic value obtained for the start state could be very low or often zero for the last few subgoals. For example, consider the subgoal where all cubies are fixed except for two corner cubies. If the abstraction assigns the same label to these two cubies and also abstracts their orientations, then the initial lower bound $h_0(s)$ will be 0 for all macro start states because $\phi_i(g) = \phi_i(s)$ where g is the goal state. From the experiments in Chapter 3 we know that the higher the h_0 value the faster the search with IDA*. Therefore we definitely want to avoid low h_0 values and our abstraction generator will not select the domain labels of the state variables fixed for the subgoal².

4.6 Results

For the experiment, we built 3 macro tables. Table $T_{18,1}$ is a macro table with 18 subgoals (the order of these subgoals is listed in Table 4.1 and corresponds to our preference to solve the cube by hand). T_6 is a macro table with six subgoals. The subgoals of T_6 (Table 4.2) were obtained by merging consecutive subgoals from $T_{18,1}$. The first subgoal of T_6 is to fix all cubies on the "up" face at once. Rather than precomputing and storing in a table the 96 million macros required for this subgoal, we instead solve each instance of it, as needed, by a heuristic search with pattern databases. $T_{18,2}$ is also a macro table with 18 subgoals. It uses the same subgoal sequence as Korf did in his original work [Kor83a]. At the time, he could not calculate all the optimal macros (because of small memory computers) and obtained 7 by macro composition. We replaced these 7 by the optimal macros

²unless it is necessary to fit the resulting pattern database in memory.

found by our method. We have also set a bound of $N = 2,000,000$ for the size of a pattern database. For the initial subgoals we used only one pattern database. Subsequent subgoals used the pattern database from the preceding subgoal as well. The heuristic values were combined by the *max* function to guarantee admissible heuristic values.

We also performed post processing on the solution paths. The first form of reduction can be applied for every macro table, the second is specific to the Rubik's Cube. If the macro returned for subgoal g_i ends with $\dots o_m, o_{k-x}, \dots, o_{k-1}, o_k$ and the macro retrieved for subgoal g_{i+1} starts with $o_k^{-1}, o_{k-1}^{-1}, \dots, o_{k-x}^{-1}, o_n$, then the concatenated sequence should have the inverses which cancel removed: $\dots, o_m, o_n, \dots$. The other reduction comes from the fact that operators turning opposite faces in the Rubik's Cube do not move common cubies. If the concatenated path contains a subsequence of the form $\dots, U, D, U, \dots$ or $\dots U, D, U^{-1}, \dots$ then these can be reduced to $\dots, U^2, D, \dots$ and $\dots, D, \dots$ respectively because U and D are opposing faces.

We generated 10,000 random instances of the Rubik's Cube by random walks of 500 moves each. These problem instances were then solved by the macro tables $T_{18,1}$, $T_{18,2}$ and T_6 . The lengths of the solutions for the test instances are summarized by the histograms in Figure 4.7. It is clear from these histograms that the longest solutions with the 6 subgoal macro table are as good as the shortest solutions with the 18 subgoal macro table. It would be worthwhile to compare these results to the optimal solutions but today even with the latest methods this is impossible. It is conjectured [Kor97] that the longest shortest path (diameter) between two states is no more than 20 moves and the vast majority of states are 18 moves away from the goal state. In light of these numbers, we estimate that

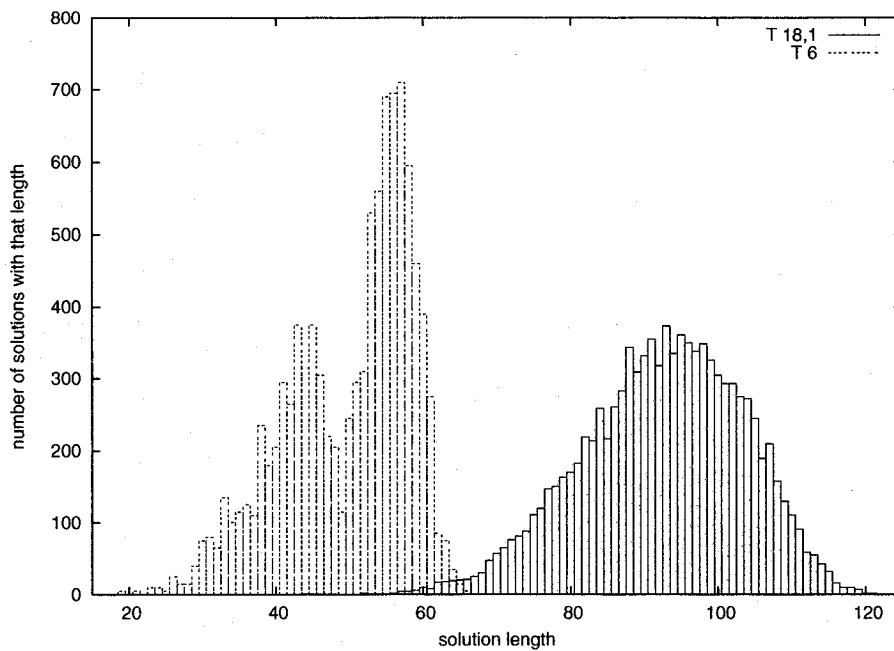


Figure 4.7: Histograms of solution lengths obtained for 10,000 random cube states

our solutions with T_6 are on average 2.5 times and in the worst case 3.5 times of optimal. With $T_{18,1}$ the solutions are 5 times longer on average and in the worst case 7 times the optimal length. The average solution length on these 10,000 random instances for T_6 is 50.3 and for $T_{18,1}$ it is 89.8. $T_{18,1}$ performs slightly worse than $T_{18,2}$, whose solutions average 82.7 moves.

T_6 's histogram seems to have two humps. We believe that this is due to the fact that there are many very short macros for the second, third, fourth and fifth subgoals. If the cubies are in a "lucky" arrangement they will be fixed with significantly fewer moves than the average.

Tables 4.1 and 4.2 show the detailed statistics corresponding to the searches for the macros of $T_{18,1}$ and T_6 . Table 4.1 also gives the combined results for those subgoals which were merged to form single subgoals in T_6 . The entries in the first column are the individual subgoals. For example, in Table 4.1, the first subgoal

subgoal	# of macros	longest macro	average macro	nodes expanded
<i>URF</i>	24	2	1.54	200
<i>URB</i>	21	3	2.05	8,300
<i>ULB</i>	18	4	2.22	40,400
<i>ULF</i>	15	6	3.87	117,000
<i>UF</i>	24	8	5.33	187,200
<i>UL</i>	22	8	5.27	171,600
<i>UR</i>	20	8	5.40	156,000
<i>UB</i>	18	8	5.22	142,000
total:	162	47	30.90	822,700
<i>LF</i>	16	9	5.50	128,000
<i>LB</i>	14	9	5.92	122,400
<i>DL</i>	12	7	4.33	92,600
total:	42	25	15.70	344,000
<i>RF</i>	10	9	6.40	91,200
<i>RB</i>	8	10	6.75	141,600
total:	18	19	13.15	232,800
<i>DF</i>	6	7	5.33	46,800
<i>DB</i>	7	11	7.00	286,600
total:	13	18	12.33	333,400
<i>DRF</i>	12	12	8.25	6,117,600
<i>DRB</i>	9	12	8.22	6,081,000
total:	21	24	16.47	12,198,600
<i>DLF</i>	2	13	13.00	94,303,000
grand total:	258	146	101.6	108,234,500

Table 4.1: Macro table $T_{18,1}$: Statistics

subgoal	# of macros	longest macro	average macro	nodes expanded
<i>URF, URB, ULB, ULF, UF, UL, UR, UB</i>	96,282,624 <i>not stored</i>	~ 13	9.30	$\sim 250,000$ <i>for each solution</i>
<i>LF, LB, DL</i>	2,688	11	8.84	109,524,000
<i>RF, RB</i>	80	12	8.45	5,526,200
<i>DF, DB</i>	24	12	7.75	3,875,000
<i>DRF, DRB</i>	108	13	10.73	1,914,102,800
<i>DLF</i>	2	13	13.00	94,303,000
grand total:	2,902	74	58.07	2,127,331,000

Table 4.2: Macro table T_6 : Statistics

is to fix the cubie URF , the second subgoal is to fix the cubie URB while leaving the previously fixed cubies (namely URF) intact. For T_6 the first subgoal is to fix the entire upper face (the 8 cubies listed in the first row and first column of Table 4.2). The second column entries are the number of macros needed for the subgoal. For example, the number of macros needed for subgoal URB in $T_{18,1}$ is 21 (second row, second column) because cubie URB can occur in 21 different places and orientations once the previous subgoals are satisfied. For T_6 , the first subgoal would require 96,282,624 macros, which we do not store or pre-calculate. The third column lists the longest of the macros for the subgoal and column four gives the average macro-length. The fifth column is the total number of nodes expanded to calculate these macros. Table 4.1 has rows with label **total**, which are simply the totals for the subgoals just above. These can be directly compared to the rows of Table 4.2, because they total those subgoals which are merged to form single subgoals in T_6 . For example, for the three subgoals LF , LB and DL in Table 4.1, the total number of macros is 42, the total of the longest macros is

25 and the total of the average of macro lengths is 15.7 moves. The corresponding subgoal in Table 4.2 is $\{LF, LB, DL\}$. For the composite subgoal, the number of macros needed is 2,688. The longest of these macros is 11 and the average macro length is 8.84 moves. The grand total column for the tables show the totals for all subgoals. The projected average performance of $T_{18,1}$ (grand total row, fourth column in table 4.1) is 101.6 moves and it is 58.07 moves for T_6 . These are longer than the experimentally obtained averages. The reason is the path post-processing we described earlier.

$T_{18,1}$ has 258 macros of which 18 are trivial. T_6 has 2,902 macros stored of which only 5 are trivial. A trivial macro has length 0 and corresponds to an already fixed subgoal. Each subgoal has exactly one of these. We counted the trivial macros as well, because it can happen that a subgoal is fixed by chance and therefore should be included in the averages.

Both the projected and the experimentally determined worst case for T_6 (74 and 68) are much better than the average performance of $T_{18,1}$ (and $T_{18,2}$). The improvement in the average solution length is 44%. These reductions speak for themselves, however we must also consider at what cost they were achieved.

First we consider memory used in the searches. It took 13 different pattern databases with 15 million entries stored altogether to build the macros for $T_{18,1}$. At any time there were no more than three pattern databases in memory with 5 million entries combined. We needed 11 pattern databases to build T_6 and similarly, we never used more than three of these at a time. All macros were obtained by IDA* [Kor85b], hence no memory additional to the pattern databases was required during the searches.

In comparison to the partial-match, bi-directional search method [Kor83a, Kor85c], we can show significant savings in memory usage. The longest of Korf's macros are of length 13. For these, a bi-directional search must reach level 6, where the number of nodes in the search tree is over 10 million. Keeping in mind that the search frontier must be in memory and that for finding each macro the search trees are different, heuristic search with reusable pattern databases is a clear winner. We also can limit our memory usage (we never had more than 5 million entries in the pattern databases used for a single search and we also imposed a 2 million entry limit for a single pattern database). We do not need more memory to search for longer macros. Korf's bidirectional approach, however, needs memory exponential in search depth.

To obtain the 258 macros of $T_{18,1}$, a total of 108 million nodes were expanded, while 2 billion nodes were expanded to build the 2,902 macros of T_6 . Building $T_{18,1}$ took an hour of CPU time, while over a day of CPU time was needed to build T_6 ³.

The memory required to obtain a solution with $T_{18,1}$ is to store the 258 macros, T_6 has 2,902. This is negligible. On the other hand, T_6 uses two pattern databases with a combined size of 300,000 entries to search for the first subgoal. We could store much larger pattern databases, however, we felt that this should count as part of the macro-table and therefore smaller pattern databases are more appropriate. However, we believe that even these requirements are reasonable with modern computers.

With respect to the speed of obtaining a solution, $T_{18,1}$ and $T_{18,2}$ are the clear

³With a 300MHz clock speed computer

winners. It takes a constant number of operations to derive a solution and the post processing is linear in the number of moves of the solution. T_6 performs a heuristic search to fix the first 8 cubies and this on average takes 250,000 node expansions. While this search takes about a second with our implementation, it is orders of magnitudes slower than $T_{18,1}$ and $T_{18,2}$.

Our primary goal, however, was to significantly reduce the solution lengths. We believe that the additional time required to build the macros and the initial search to fix the first subgoal are well worth the effort.

From Tables 4.1 and 4.2, we can also determine where the savings come from. The first 8 subgoals combined in $T_{18,1}$ are represented by a single search in T_6 . The projected worst case for these subgoals is 47 moves with $T_{18,1}$ and 15 moves with T_6 and the predicted averages are 30.9 ($T_{18,1}$) and 9.30 (T_6) moves without post processing.

The difference in the predicted averages for the first 8 cubies is 21 moves and hence this one big subgoal merge accounts for almost 50% of the total savings alone. Then we merged 3 subgoals (LF , LB and DL) into one. The projected expected savings is 7 moves (from 15.7 to 8.84). The rest of the subgoal combinations show similar ratios, except there is less of an explosion in the number of macros.

There are two other methods we can compare our approach to. The two extremes are the algebraic solutions calculated by the Schreier-Sims method [Sim70] and the optimal solutions obtained by Richard Korf [Kor97]. Our implementation of the Schreier-Sims method needs only 8 minutes to build a stabilizer chain⁴ (the

⁴This was implemented in the interpreted language MIT Scheme, while our macro implementation is in C++

Group Theory equivalent of macro tables). The average solution length is around 455 moves. An expensive local search path improvement suggested by Osterlund [Ost] shrinks these paths to an average of 89 moves. This is almost the same as $T_{18,1}$. However, the Schreier-Sims method and Osterlund's path improvement rely on the fact that Rubik's Cube is a group and therefore are less general than Korf's macro search.

The optimal solutions obtained by Korf [Kor97] required expanding 3 billion states for optimal solutions of length 16 and close to a trillion states for optimal solutions of length 18. He used three pattern databases with 172 million entries combined. The long solutions take days of CPU time individually. In essence, any one of the optimal searches, even for the short solutions, requires expanding many more states and much more memory than obtaining all macros in both macro tables combined and solving the 10,000 test instances.

4.7 Chapter Summary

In this chapter we demonstrated that macro-operators can be obtained efficiently with automatically generated heuristics. The speedup over uninformed techniques is so great it also allows us to reduce the number of subgoals by merging consecutive ones. This results in dramatically reduced solution lengths at reasonable additional cost. Macro-search is a very well suited application for our pattern database technique, since each subgoal represents a different search space and therefore automatically derived heuristics are preferable. On our test problem, Rubik's Cube, we achieved 44% improvement in solution length. Unlike bidirec-

tional partial-match, IDA* with automatically generated heuristics can run within user defined memory limits.

Chapter 5

Application to the Sequential Ordering Problem

In this chapter we use pattern databases to solve the Sequential Ordering Problem (SOP), which is a well studied problem in scheduling and in Operations Research. Our interest in the SOP is that it is an optimization problem with $O(n^2)$ costs. In all previous pattern database work (except [CMT81]¹) the operators associated with the problems had unit costs. We will describe a novel way of building pattern databases. The domain abstractions will be obtained by greedy local search and we will show that in some cases the ones we derive this way could result in substantially more accurate lower bounds than the best ones chosen from a random pool. We will also introduce two novel search algorithms which are improvements of Depth-first Branch and Bound (DFBB).

¹While Christofides *et al.* used memory stored lower bounds first to solve routing problems (with operators having varying costs), our derivation of the abstract space is substantially different from their work.

5.1 The Sequential Ordering Problem

The Sequential Ordering Problem is a version of the Asymmetric Traveling Salesman Problem where precedence constraints must also be observed. Let G be a graph with n vertices with $v_0, v_{n-1} \in G$ chosen to be the start and terminal vertices. The aim is to find the cheapest cost path through G which starts in v_0 and terminates in v_{n-1} while visiting each vertex only once without violating precedence constraints on the vertices. The vertices v_0 and v_{n-1} may be two labels for the same vertex and in this case the solution corresponds to a Hamiltonian circuit instead of a Hamiltonian path.

The SOP is a model for many industrial applications [Asc95, GD00] and despite significant research focusing on this problem, there are still SOP instances with only 40-50 vertices which have not yet been solved optimally², although there are instances (for example the ones derived from stacker crane schedules) with over 100 vertices that could be solved easily. Compare this to the thousands of vertices Symmetric Traveling Salesman Problems (STSP) (like the famous “Tour of Germany”³) and several hundred vertex Asymmetric TSPs (ATSP) which can be solved optimally today. Large STSPs can be solved with branch and cut because there are large classes of necessary linear inequalities and good separation heuristics. The ATSPs use the Assignment Problem (AP) lower bound which can be calculated in polynomial time (with the so called *Hungarian method* [Kuh55]) and often it is also accurate [Zha03] (there is a phase transition related to the

²The TSPLIB [TSP] has a collection of unsolved instances of SOPs

³The “Tour of Germany” is a symmetric TSP of 15,112 cities. The optimal solution is posted at <http://www.math.princeton.edu/tsp/d15sol/> and is referred to as `d15112.tsp` in the TSPLIB [TSP]

number of digits used for the costs). However, once precedence constraints are introduced, these techniques are no longer adequate to solve the same size problems. The challenges with the IP formulation include encoding the precedence constraints as linear inequalities and identifying the ones which are facet inducing (separation and lifting). Some classes of inequalities have been implemented by [AJR00] and shown to be valid for the SOP. It is also still unknown how to incorporate precedence constraints into the AP lower bound and therefore the AP bound is usually very inaccurate. It is striking how the introduction of additional constraints renders state-of-the-art TSP solvers inadequate to solve some hard instances of the SOP. The best known technique ([AJR00]) could solve a class of larger SOPs (100+ vertex) based on actual stacker crane schedules, but the same program was also unable to solve some smaller (44 vertex) SOPs which were obtained by adding random precedences to ATSP instances from the TSPLIB. On some problems, the gap between a lower and upper bound was quite significant (eg. lower bound 65,497 and upper bound 83,005 for p43.4). Because of this, we expected that our automatically generated lower bounds used with state-space search could be as good or even outperform previous attempts on SOPs that look for optimal solutions.

5.2 State Space

To demonstrate our problem formulation and solution for the SOP, let us consider an example routing problem. A helicopter is sent off from shore to visit six oil rigs (as depicted in Figure 5.1). Its purpose is to deliver fresh crew to the rigs and to bring back workers to shore for vacation. It also must transfer a driller

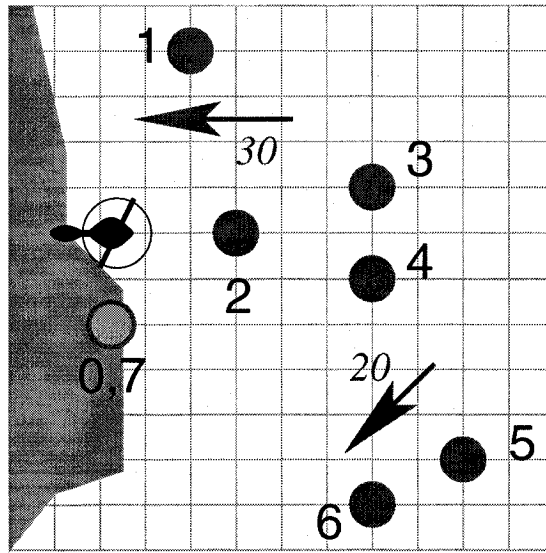


Figure 5.1: A Helicopter Routing Problem

from rig 5 to rig 2, an engineer from rig 2 to rig 3, an electrician from rig 4 to rig 3 and pick up supplies at rig 6 and deliver them to rigs 2 and 4. To simplify matters, let us also assume that with respect to payload and seating capacity the helicopter can visit the oil rigs in any order. There is also wind over the sea as indicated by the arrows in Figure 5.1. As a result, between a pair of rigs a and b , the cost of flying the leg ab could differ from that of ba .

To model this problem as SOP, we treat the rigs and the landing pad as vertices in a graph G . In fact, the landing pad is also the source and destination vertex and therefore it will be represented by two vertices (0 and 7) in G .

The precedences correspond to a graph P with the same vertices as G . In P , the edge uv represents the precedence constraint that vertex v must precede vertex u in a feasible tour. The redundant edges are removed, if there are edges uv and vw in P then the edge uw is redundant as the corresponding precedence is implied. The precedences corresponding to our example are depicted in Figure

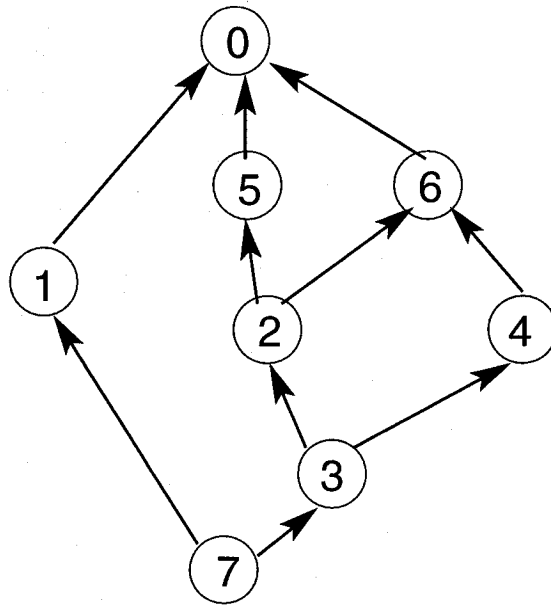


Figure 5.2: Precedence Graph for the Helicopter Routing Problem

5.2. The start vertex 0 must precede all other vertices and the terminal vertex 7 is preceded by all other vertices. The original edge costs of G are given by the cost matrix below.

$$C = \begin{pmatrix} \times & 7 & \times & \times & \times & 12 & 7 & \times \\ \succ & \times & 3 & 6 & 7 & 13 & 12 & 6 \\ \succ & 2 & \times & 4 & 4 & \succ & \succ & \times \\ \succ & 5 & \succ & \times & \succ & \succ & \succ & 7 \\ \succ & 6 & 3 & 2 & \times & 4 & \succ & \times \\ \succ & 11 & 6 & \times & 4 & \times & 4 & \times \\ \succ & 10 & 4 & \times & 7 & 5 & \times & 6 \\ \succ & \succ & \succ & \succ & \succ & \succ & \succ & \times \end{pmatrix} \quad (5.1)$$

The entries marked $\succ$ correspond to the precedences. $C[i][j] = \succ$ if vertex j must precede vertex i in a feasible tour. If $C[i][j] = \times$, then the edge $v_i v_j$ cannot be taken. The SOP instances posted in the TSPLIB [TSP] are given in a similar

format. -1's encode precedences but there are no $\times$ entries. There are two reasons for having missing edges ($\times$ entries) in the cost matrix. First, some transitions may be forbidden in the original model (*eg.* there is no road connecting two cities). Second, the precedence constraints may logically exclude some of the edges. These constraint violating edges, however, can be readily obtained from the precedence graph which can be calculated efficiently by dynamic programming.

A SOP state in our representation corresponds to a partial completion of the tour. For example,

$$u = 0 \rightarrow 1 \rightarrow 5 \rightarrow 6$$

is a partial tour starting in vertex 0 and currently terminating in vertex 6. Its cost is 24. It is also feasible⁴ with respect to the precedences defined in C (Equation 5.1, page 100). The successor states each add a new vertex to the tour branching from the last visited vertex. The resulting states (below) must also be feasible.

$$\left\{ \begin{array}{ll} \Delta(u) & \text{total cost} \\ 0 \rightarrow 1 \rightarrow 5 \rightarrow 6 \xrightarrow{A} 2 & 28 \\ 0 \rightarrow 1 \rightarrow 5 \rightarrow 6 \xrightarrow{7} 4 & 31 \end{array} \right\}$$

To evaluate whether adding a vertex to a partial tour u is feasible and to obtain the cost of the resulting state v , we only need to know the vertices already included in u and its last vertex. From this perspective, the order of already visited vertices in the tour is irrelevant. For computational reasons, in our actual state representation, we record the last vertex in the partial tour and the set of states which are still available. The vertices already part of the tour can be inferred

⁴the vertices included in the partial tour satisfy the precedence constraints

from this. For example, in state

$$u = \underbrace{(2, 3, 4, 7)}_{\text{available}} \underbrace{[6]}_{\text{last}}$$

the vertices 2, 3, 4 and 7 are still available and the partial tour currently terminates in vertex 6. From this, we also know that the vertices 0, 1 and 5 are already visited. We also define the functions *av* and *curr* over states which return the list of available vertices and the current last vertex in the partial tour respectively. For the example state *u* above,

$$\begin{aligned} av(u) &= (2, 3, 4, 7) \\ curr(u) &= 6 \end{aligned}$$

The SOP state space $S(n)$, with this representation is a lattice with the start state at the apex and the goal state at the bottom. The lattice has n levels corresponding to adding a vertex to a partial tour. Edges connect only adjacent levels. The start state is at level 0. Our state representation implicitly encodes level information: the number of vertices available subtracted from $n - 1$ is the level where the state belongs. The state-space lattice corresponding to our example is depicted in Figure 5.3.

The full lattice not considering precedences has

$$|S(n)| \leq \left(\sum_{i=0}^{n-3} \binom{n-2}{i} (n-2-i) \right) + 2 \quad (5.2)$$

states. At level $1 \leq l \leq n - 2$, $n - 2 - l$ vertices are available from the $n - 2$ possible vertices, since vertices 0 and $n - 1$ are reserved. The last visited vertex in the partial tour could be any of the not chosen l vertices from the $n - 2$. Setting $i = n - 2 - l$, Equation 5.2 follows. There are at most $(n - 2)!$ complete tours

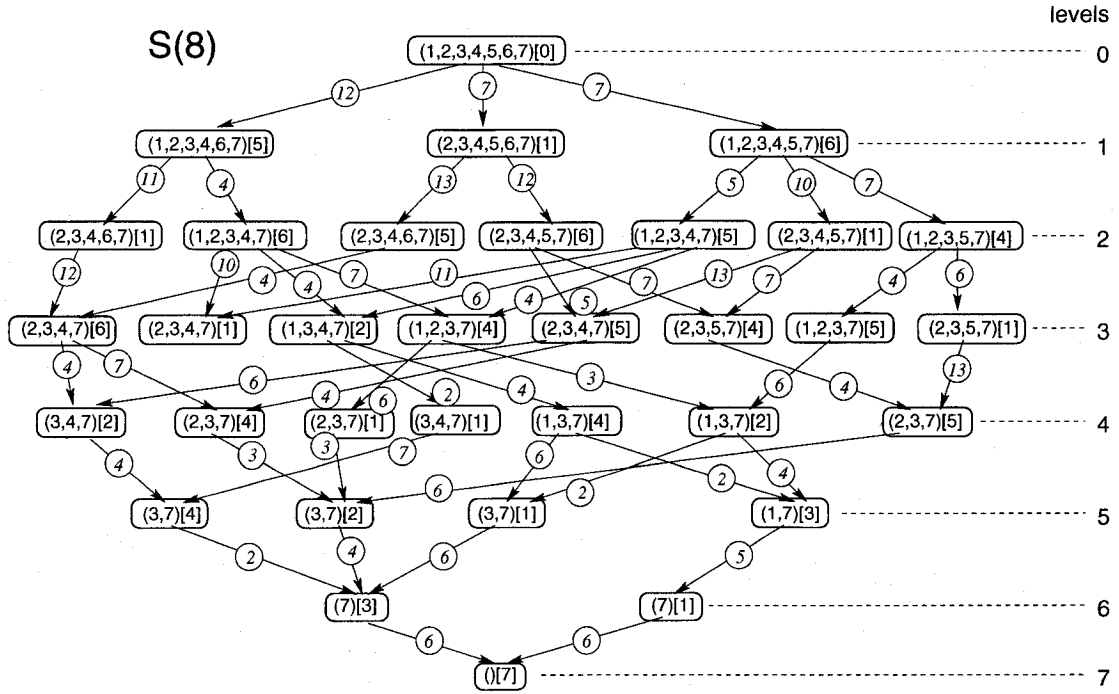


Figure 5.3: State-space Lattice for the Helicopter Routing Problem

through S ; some are infeasible because of the precedence constraints. $|S(n)|$ is significantly smaller than $(n-2)!$. For example, for a 30 vertex SOP,

$$\begin{aligned}
 |S(30)| &= 3,758,096,386 \quad // \text{ states in } S \\
 28! &= 304,888,344,611,713,860,501,504,000,000 \quad // \text{ paths through } S
 \end{aligned}$$

5.3 Lower Bounds

5.3.1 The Abstract Space

The lower bounds will be derived from state space abstractions. $\Phi : S \rightarrow S'$ is a function which assigns states and edges of S to those in S' . Φ also preserves the level structure of S . Figure 5.4 illustrates the conceptual relationship between

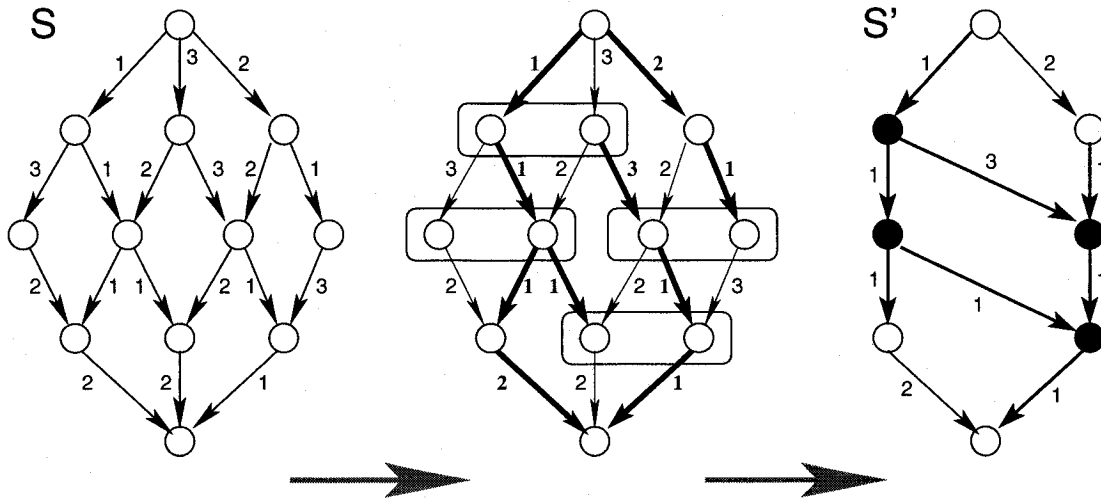


Figure 5.4: State Space Lattice Abstraction

the original and abstract lattices. The action of Φ on the states of S is to cluster states from the same level and assign each group of states a unique state in S' . Φ also picks the cheapest cost edges between the resulting clusters and only these edges will be retained in S' . It is easy to see that Φ induces a state-space homomorphism as defined in Section 2.4 and therefore the cost of the cheapest path from s' to g' in S' is a lower bound on the cost of the cheapest path from s to g in S . In the previous chapters, each pattern database was built from a single abstraction. For the SOP, we will use more than one abstraction to build a single pattern database⁵. Each domain abstraction will correspond to one level of the SOP state space lattice.

Let S_i and S'_i represent the i th level of S and S' respectively.

$$\Phi = \langle \phi_0, \phi_1, \dots, \phi_{n-1} \rangle$$

is a vector of *domain abstractions* indexed by the levels of S . Each domain

⁵or equivalently, we can think of having a different pattern database for each level

abstraction induces one level in the abstract state space lattice S'

$$\phi_i : S_i \rightarrow S'_i$$

The number of states on level S'_i depends on the granularity of ϕ_i . Coarse granularity abstractions yield narrower levels and fine grain abstractions result in wider levels. Without loss of generality, we will use the integers 0 to $n-1$ for the original domain and 0 to $\phi_i(n-1)$ for the target domains. Our domain abstractions will also preserve the uniqueness of the start vertex and the terminal vertex which are 0 and $n-1$ in the original domain. This way the vertex label is also an index into the cost matrix (indexed from 0 to $n-1$). For example, consider the two domain abstractions ϕ_4 and ϕ_5 for our helicopter routing example (Figure 5.1).

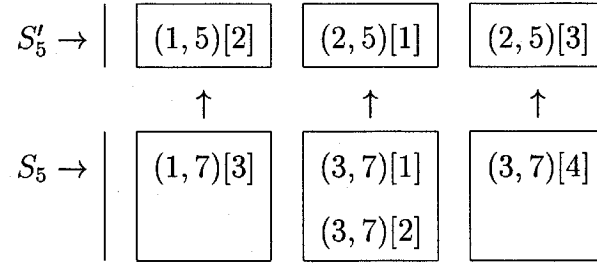
$$\begin{array}{c|cccccccc} v & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline \phi_4(v) & 0 & 1 & 1 & 1 & 2 & 3 & 3 & 4 \\ \phi_5(v) & 0 & 1 & 1 & 2 & 3 & 3 & 4 & 5 \end{array} \quad (5.3)$$

ϕ_4 and ϕ_5 cluster the states of S_4 and S_5 (Figure 5.3) respectively. The resulting abstract states are shown in Figures 5.5 and 5.6. The action of Φ on the states

$$\begin{array}{c} S'_4 \rightarrow \left| \begin{array}{|c|c|c|c|} \hline (1, 1, 4)[1] & (1, 1, 4)[2] & (1, 1, 4)[3] & (1, 2, 4)[1] \\ \hline \end{array} \right. \\ \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \\ S_4 \rightarrow \left| \begin{array}{|c|c|c|c|} \hline (2, 3, 7)[1] & (1, 3, 7)[4] & (2, 3, 7)[5] & (3, 4, 7)[1] \\ (1, 3, 7)[2] & (2, 3, 7)[4] & & (3, 4, 7)[2] \\ \hline \end{array} \right. \end{array}$$

Figure 5.5: The States of $S'_4 = \phi_4(S_4)$

of S is purely a syntactic relabeling of the state representation. Note that the vertices of the original SOP graph do not get merged by domain abstractions. Instead, some vertices receive identical labels which make them indistinguishable in

Figure 5.6: The States of $S'_5 = \phi_5(S_5)$

the abstract space. For example, in the abstract state $\phi_4((2, 3, 7)[1]) = (1, 1, 4)[1]$ there are two vertices available with label 1 and both must be taken to complete the tour.

The action of Φ is fairly trivial on the states. To complete the abstraction of Φ we must specify its action on the edges. Let us consider two adjacent levels of the abstract lattice S' . These are the result of the action of two potentially different domain abstractions. Let D be the domain of original vertex labels (always $\{0, \dots, n-1\}$) and D_i and D_{i+1} the domain of labels at levels i and $i+1$. $S'_i = \phi_i(S_i)$ and $S'_{i+1} = \phi_{i+1}(S_{i+1})$, where $\phi_i : D \rightarrow D_i$ and $\phi_{i+1} : D \rightarrow D_{i+1}$. In our example, (Equation 5.3)

$$\begin{aligned}
\phi_4 : \{0, 1, 2, 3, 4, 5, 6, 7\} &\rightarrow \{0, 1, 2, 3, 4\} \\
\phi_5 : \{0, 1, 2, 3, 4, 5, 6, 7\} &\rightarrow \{0, 1, 2, 3, 4, 5\}
\end{aligned}$$

There are $|D_i|$ labels on S'_i and $|D_{i+1}|$ labels on S'_{i+1} . Let $\phi_i^{-1}(s'_i)$ denote the preimages of s'_i on S_i , that is,

$$\phi_i^{-1}(s'_i) = \{s_i | \phi_i(s_i) = s'_i, s_i \in S_i\}$$

The edges which get merged by ϕ_i and ϕ_{i+1} are those with entries

$$C[\phi_i^{-1}(\text{curr}(s'_i))][\phi_{i+1}^{-1}(\text{curr}(s'_{i+1}))]$$

in the original cost matrix. Therefore the cost matrix $C_{i,i+1}$ that assigns the edge costs between S'_i and S'_{i+1} is a $|D_i| \times |D_{i+1}|$ matrix. Of the merged edge costs, we have to choose the one with the minimum cost. However, as in Equation 5.1, the cost matrix also contains precedence information and non-existent edges ($\succ$ and $\times$ entries) which we also have to handle. First, we introduce some notation and then define our rules of deriving the edge costs in the abstract state space. Let v_i be the label assigned to the last vertex in the abstract partial tour s'_i and let V_i be the preimages of v_i under ϕ_i .

$$\begin{aligned} v_i &= \text{curr}(s'_i) && \text{label of the current last vertex in } s'_i \in S'_i \\ V_i &= \phi_i^{-1}(v_i) && \text{labels which map to } v_i \text{ under } \phi_i \end{aligned}$$

Figure 5.7 shows a cluster on adjacent levels induced by the domain abstractions. Every abstract state whose currently last vertex in the corresponding partial tour

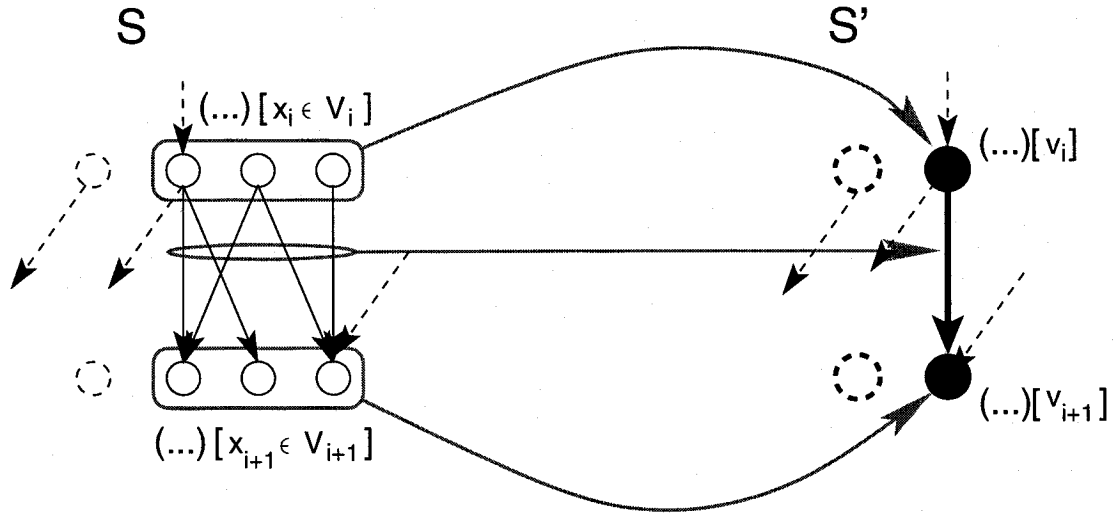


Figure 5.7: Edges Connecting State Clusters

is v_i has preimages in S with a currently last vertex in V_i . In Figure 5.7, the state clusters in S become the single states (filled nodes) in S' . Therefore the edge between the two abstract states (shown as the bold arrow) corresponds to the

entries $C[V_i][V_{i+1}]$ in the original cost matrix. In the abstract cost matrix that describes the edges between S'_i and S'_{i+1} , these entries will be represented by a single edge cost. Since the domain abstractions can vary from level to level in S'_i , we need a different cost matrix to specify the connectivity between each adjacent pair of levels.

$$C' = \langle C'_{0,1}, C'_{1,2}, \dots, C'_{n-2,n-1} \rangle$$

C' is a vector of cost matrices. Note that $C'_{i,i+1}$ is not necessarily a square matrix. The rows are indexed by the codomain of ϕ_i and columns correspond to the codomain of ϕ_{i+1} .

The abstract cost matrices must result in abstract spaces that guarantee the connectivity and cost requirements of a State Space Homomorphism (Definition 14, page 21). For a SOP G , with cost matrix C , state-space lattice S and domain abstractions $\Phi = \langle \phi_0, \phi_1, \dots, \phi_{n-1} \rangle$, the following rules will be applied to obtain the vector of abstract cost matrices C' .

1. If all edge costs $C[V_i][V_{i+1}]$ are $\succ$ then $C'_{i,i+1}[v_i][v_{i+1}] = \succ$

The entry $\succ$ represents a precedence constraint, $C[a][b] = \succ$ if b must precede a . Since, in this case, all vertices in V_{i+1} precede all vertices in V_i in a feasible tour, v_{i+1} will precede v_i in all feasible abstract tours.

2. If all edge costs $C[V_i][V_{i+1}]$ are $\times$ then $C'_{i,i+1}[v_i][v_{i+1}] = \times$

This means that there are no edges from any of the preimages of v_i to any of the preimages of v_{i+1} in G and therefore there should be no edge from v_i to v_{i+1} in an abstract tour.

3. If the entries of $C[V_i][V_{i+1}]$ are all $\succ$ and $\times$ then $C'_{i,i+1}[v_i][v_{i+1}] = \times$

In this case we know that there are no edges from any of the vertices of V_i to any of V_{i+1} in G but we cannot preserve the precedence constraints as it is not true that *all* of V_{i+1} precedes *all* of V_i .

4. If there is at least one numerical entry in $C[V_i][V_{i+1}]$ then $C'_{i,i+1}[v_i][v_{i+1}]$ is the minimum of the numerical entries ignoring the $\succ$ and $\times$ entries

Since there is at least one edge from the set of vertices V_i to the set of vertices V_{i+1} , we must preserve this link. We choose the minimum of the numerical entries.

Theorem 4 *Applying the above four rules to derive the abstract cost matrices for $S' = \Phi(S)$, it is guaranteed that the optimal cost of completing $s' \in S'$ is a lower bound on the optimal completion cost of $s \in S$.*

Proof 5.1 (Theorem 4) Suppose s is at level l in S . Let $p = \langle v_l, v_{l+1}, \dots, v_{n-1} \rangle$ be any precedence observing completion of s . The cost of this completion is

$$c(p) = \sum_{i=l}^{n-2} C[v_i][v_{i+1}]$$

We show that the path $p' = \langle \phi_l(v_l), \phi_{l+1}(v_{l+1}), \dots, \phi_{n-1}(v_{n-1}) \rangle$ in S' is a completion of s' with cost $c(p') \leq c(p)$.

Since p is a feasible path, we know that the entries $C[v_i][v_{i+1}]$ are not $\succ$ and not $\times$. The cost of the corresponding abstract path p' is

$$c(p') = \sum_{i=l}^{n-2} C'_{i,i+1}[\phi_i(v_i)][\phi_{i+1}(v_{i+1})]$$

While we do not know the exact values of $C'_{i,i+1}[\phi_i(v_i)][\phi_{i+1}(v_{i+1})]$, we know that they were created by applying rule 4 only. We know this, because only rule 4 applies when there is at least one real edge (non $\succ$ and non $\times$) between the preimage states clusters. The cost of this edge, according to rule 4, is the cheapest of the edge costs between the preimage vertex clusters. The costs $C[v_i][v_{i+1}]$ belong to these clusters, so $c(p') \leq c(p)$.

We still have to prove that there are no precedence constraints introduced in $C'_{i,i+1}$ which would make p' constraint violating in S' . For p' not to exist, there must be a precedence constraint for some $l \leq i \leq n-2$ which prohibits taking the $\phi_l(v_l) - \phi_{l+1}(v_{l+1})$ edge in p' ; namely $C'_{k,k+1}[\phi_k(v_{i+1})][\phi_{k+1}(v_i)] = \succ$, where $l \leq k \leq n-2$. Only rule 1 introduces precedence constraints into $C'_{i,i+1}$. If there is a $\succ$ entry created in $C'_{i,i+1}$, then all of $\phi_k^{-1}(v_{i+1})$ must precede all of $\phi_{k+1}^{-1}(v_i)$; the precondition for rule 1. Since p is precedence observing in S , in p v_i precedes v_{i+1} and therefore rule 1 does not apply to the edges of p . $\square$

Using our rules, every feasible path is preserved but some feasible abstract paths do not have feasible preimages.

Let us demonstrate our rules with an actual example. Consider the cost matrix C (Equation 5.1) and the domain abstractions ϕ_4 and ϕ_5 (Equation 5.3). First, we show which edges are going to be merged by the abstractions.

$$\begin{array}{c}
 \begin{array}{cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 \phi_5 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & 0 & 1 & 1 & 2 & 3 & 3 & 4 & 5
 \end{array} \\
 \begin{array}{c}
 \phi_4 \\
 0 \rightarrow 0 \\
 1 \rightarrow 1 \\
 2 \rightarrow 1 \\
 3 \rightarrow 1 \\
 4 \rightarrow 2 \\
 5 \rightarrow 3 \\
 6 \rightarrow 3 \\
 7 \rightarrow 4
 \end{array}
 \begin{array}{|c|c|c|c|c|c|}
 \hline
 \times & \boxed{7 \times} & \times & \boxed{\times 12} & 7 & \times \\
 \hline
 \gamma & \boxed{\times 3} & \boxed{6} & \boxed{7 \ 13} & \boxed{12} & \boxed{6} \\
 \gamma & \boxed{2 \times} & \boxed{4} & \boxed{4 \ \gamma} & \gamma & \times \\
 \gamma & \boxed{5 \ \gamma} & \times & \gamma & \gamma & 7 \\
 \hline
 \gamma & \boxed{6 \ 3} & 2 & \boxed{\times 4} & \gamma & \times \\
 \hline
 \gamma & \boxed{11 \ 6} & \times & \boxed{4 \times} & \boxed{4} & \times \\
 \gamma & \boxed{10 \ 4} & \times & \boxed{7 \ 5} & \times & \boxed{6} \\
 \hline
 \gamma & \boxed{\gamma \ \gamma} & \gamma & \boxed{\gamma \ \gamma} & \gamma & \times
 \end{array}
 \end{array} \quad (5.4)$$

Equation 5.4 shows the edges that get merged as a result of domain abstractions ϕ_4 and ϕ_5 . ϕ_4 affects the rows (level 4) and ϕ_5 the columns (level 5). The entries which will be retained in the abstract cost matrix according to our four rules are shown in bold font. The resulting cost matrix is

$$C'_{4,5} = \begin{pmatrix} \times & 7 & \times & 12 & 7 & \times \\ \gamma & 2 & 4 & 4 & 12 & 6 \\ \gamma & 3 & 2 & 4 & \gamma & \times \\ \gamma & 4 & \times & 4 & 4 & 6 \\ \gamma & \gamma & \gamma & \gamma & \gamma & \times \end{pmatrix} \quad (5.5)$$

It is a 5×6 cost matrix, since there are 5 abstract labels at level 4 and 6 abstract labels at level 5 in S' .

Note that most of the precedence constraints got eliminated and in many cases high cost edges were wiped out by cheap ones. In Section 5.3.3 we will examine what considerations should be taken into account when selecting domain abstractions.

5.3.2 Calculating the Lower Bounds

While the correspondence between S and S' , as defined in the previous section, is conceptually clear, we must note that for realistic SOPs $|S|$ is so large that it cannot be enumerated explicitly and therefore we have to generate S' without generating S . The lower bounds will correspond to the γ_u values defined in Equation 2.12 in Theorem 1 (Page 27). First we need the function $\Delta_{\phi_{i+1}, \phi_i}^{-1}$ which generates the predecessors of abstract states. Because implementing this function is not straightforward and it is subject to issues not yet discussed, we delay its description until Section 5.3.4.

Mathematically, the distance to the abstract goal $\gamma(s'_i)$ is defined as

$$\gamma(s'_i) = \begin{cases} 0 & \text{if } s'_i = g' \\ \min \left(C'_{i,i+1}[\text{curr}(s'_i)][\text{curr}(s'_{i+1})] + \gamma(s'_{i+1}) \right) & \text{otherwise} \end{cases} \quad (5.6)$$

where

$$s'_i \in \Delta_{\phi_{i+1}, \phi_i}^{-1}(s'_{i+1})$$

We can modify Dijkstra's single-source all-shortest-paths algorithm [Dij59] to take advantage of the level structure of S' . Our algorithm only needs to consider

a pair of adjacent levels at a time in this expansion and therefore it does not need to keep in memory states expanded on previous levels in memory.

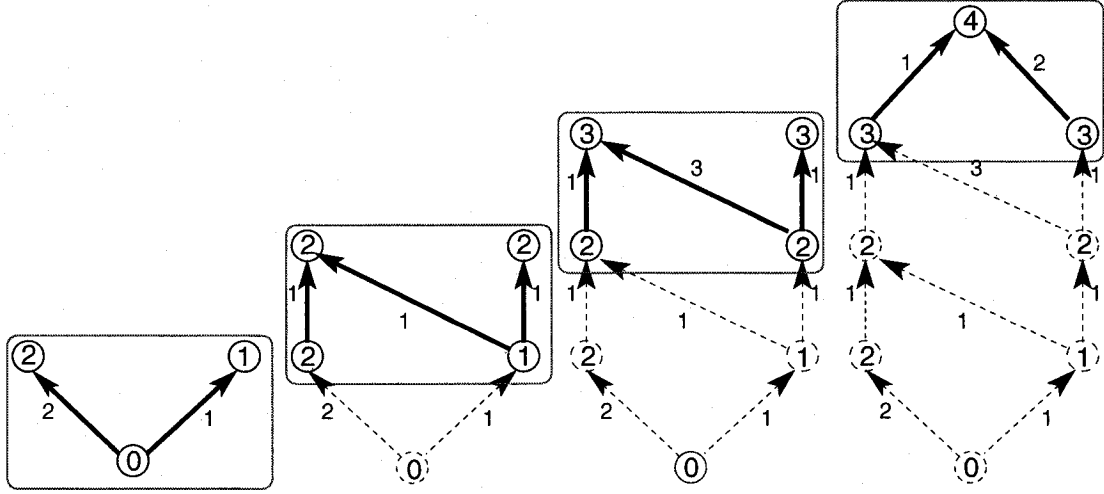


Figure 5.8: Expanding the Abstract State-space

Figure 5.8 shows how the abstract space S' is actually calculated by solving Equation 5.6. The expansion starts with the abstract goal state and proceeds by calculating the values of Equation 5.6 between adjacent levels. The values of γ are written in the nodes, while the edges $s'_{i+1}s'_i$ are labeled with the cost $cost(s'_i, s'_{i+1})$. The completed dynamic programming table is the pattern database of lower bounds we need. The running-time memory requirements of this algorithm is $O(width(S))$ since the γ values of already calculated levels can be written to disk.

Definition 21 (Width of Lattice) *The width of a lattice is the maximum number of states among all its depth levels.*

$$width(S) = \max_{S_i \in S} |S_i|$$

5.3.3 Choosing Domain Abstractions

An instance of a SOP is defined entirely by a single cost matrix. To calculate the lower bounds as described in the previous section, we also need to know the vector of domain abstractions Φ . Φ can be *any* vector of domain abstractions. We, however, are interested in domain abstractions that result in accurate lower bounds. From the results of Chapter 3 we expect that the larger the pattern database the better the lower bounds. Therefore, one of our goals is to calculate the abstractions that yield the largest S' that still fits within our memory limits. The number of abstract states on a given level in the abstract space lattice depends on two parameters. These are the *granularity* of the abstractions and the *precedence constraints*. Given the granularity Γ_{ϕ_i} we can calculate exactly the maximum number of abstract states on S'_i , not taking precedence constraints into account. Therefore, we are able to generate abstractions which yield abstract levels very close to the desired width w_i . There could be many domain abstractions with a certain granularity Γ . We also want to select the ones which yield better lower bounds. To do this, we perform a greedy local search to derive ϕ_i from ϕ_{i+1} and the states of S'_{i+1} . In other words, we do not start with a given Φ but calculate the abstractions ϕ_i , and from this $C'_{i,i+1}$, only when we encounter level i in the expansion. We start with ϕ_{n-1} being the identity map and initialize $C'_{n-2,n-1}$ to be the original cost matrix C . We also assume that we have a function $width(\phi_i)$ which returns the maximum number of abstract states on S'_i not considering the ones eliminated due to precedence violations⁶. Suppose the expansion of S' is currently at level $i+1$. At this stage of the algorithm, we have

⁶ $width(\phi_i)$ is actually calculated by a perfect minimal hash function which we describe in detail in Appendix B

calculated all levels $k > i$ of S' . We know $\gamma_{s'_{i+1}}$ and ϕ_{i+1} . To proceed to level i , we need ϕ_i . From that, we will be able to derive $C'_{i,i+1}$ and calculate $\gamma_{s'_i}$. The ϕ_i we are looking for has to be such that $\text{width}(\phi_i) \leq w_i$ and that it results in the highest possible $\gamma_{s'_i}$. The values of $\gamma_{s'_i}$ will depend on $C'_{i,i+1}$. Let $f(C'_{i,i+1})$ be a *goodness* measure of the cost matrix. To obtain higher $\gamma_{s'_i}$'s we want to optimize f

$$\text{opt}_{\phi_i} f(C'_{i,i+1}) \quad \text{s.t.} \quad \text{width}(\phi_i) \leq w_i \quad (5.7)$$

Since ϕ_{i+1} is already fixed at this level, the choice of ϕ_i will only affect the rows of the cost matrix $C'_{i,i+1}$. The space of domain abstractions is potentially large, therefore we will only consider a subset which is obtained by either *row merging* or *row splitting*. First we set $\phi_i = \phi_{i+1}$. If $\text{width}(\phi_i) > w_i$, then we have to make $\phi_i : D_i \rightarrow D_{i+1}$ more abstract. To do this, we consider merging exactly two labels l_1 and l_2 in D_{i+1} and assign these the same label l . In effect, this corresponds to merging the two rows with labels l_1 and l_2 in $C'_{i,i+1}$.

$$C'_{i,i+1} = \begin{pmatrix} & & \dots & & \\ & & & & \\ l_1 : & \succ & 3 & \dots & 0 \\ & & & & \\ & & & & \\ l_2 : & \succ & 2 & \dots & \times \\ & & & & \\ & & & & \end{pmatrix} \Rightarrow \begin{pmatrix} & & \dots & & \\ & & & & \\ l : & \boxed{\succ} & \boxed{3} & \dots & \boxed{0} \\ & & \boxed{2} & & \boxed{\times} \\ & & & & \\ & & & & \end{pmatrix} \quad (5.8)$$

Above we show a particular example where we apply the rules of our edge cost assignments. There are $O(|D_{i+1}|^2)$ such possible abstractions. It is possible that $\text{width}(\phi_i) > w_i$ and therefore we have to merge another two rows. We can keep repeating this process until we arrive at a ϕ_i that fits in memory. Similarly, it can also be the case that ϕ_i is too coarse at level i and we could fit a finer granularity

abstraction in memory. This would happen near the top of the state space lattice where the number of states on level i is smaller than at level $i + 1$. In this case we consider splitting labels of D_i . We choose a label $l \in D_i$ which has more than one preimage and assign one of these its own label l_2 . All other preimages will map to l_1 .

$$C'_{i,i+1} = \left(\begin{array}{c} \vdots \\ l : \begin{array}{|c|} \hline \succ \\ \hline \end{array} \quad \begin{array}{|c|} \hline 3 \\ \hline 2 \\ \hline \end{array} \quad \dots \quad \begin{array}{|c|} \hline 0 \\ \hline \times \\ \hline \end{array} \\ \vdots \\ \vdots \end{array} \right) \Rightarrow \left(\begin{array}{c} \vdots \\ l_1 : \succ \quad 3 \quad \dots \quad 0 \\ \vdots \\ l_2 : \succ \quad 2 \quad \dots \quad \times \\ \vdots \end{array} \right) \quad (5.9)$$

There are $O(\prod_{l \in D_{i+1}} |\phi_{i+1}^{-1}(l)|)$ abstractions that can be obtained by row splitting. With row merging and splitting the goodness measure f can be defined over the rows with labels l_1 and l_2 whose values we can readily obtain from the original cost matrix. Our f is a distance heuristic. When we merge rows, we want to *minimize* the distance between the original values in rows l_1 and l_2 (Equation 5.8) while in the case of row splitting, we want to *maximize* the distance between the values in the resulting rows l_1 and l_2 from the original values in row l . We considered the following three distance measures.

preserve precedences : choose l_1 and l_2 which minimizes (or for splitting, maximizes) the number of precedence constraints that get eliminated by the merge (or introduced by the split). If there is more than one best candidate, choose the one that optimizes the absolute error or the normalized error (see below).

absolute error : choose l_1 and l_2 which minimizes (or in the case of splitting, maximizes)

$$f(l_1, l_2) = \sum_{j=0}^{|D_{i+1}|-1} \left| C'[l_1][j] - C'[l_2][j] \right|$$

If $C'[l_1][j]$ or $C'[l_2][j]$ is $\succ$ or $\times$ the value of the difference is taken to be 0.

Some user defined penalty could also be added if a precedence constraint is eliminated by the merge or a bonus awarded if it is reintroduced by the split. In our implementation, the user provides the penalty (p) and bonus (b) values when $C'[l_1][j]$ and $C'[l_2][j]$ have the following value combinations (where c is a numerical entry)

merge		split	
$\times$ and $\succ$	$\times$ and c	$\times$ and $\succ$	$\times$ and c
p_1	p_2	$-b_1$	$-b_2$

The penalties and bonuses can also be functions of the values of the rows $C'[l_1]$ and $C'[l_2]$.

normalized error : similar to absolute error, but the absolute value of the difference is divided by the larger of the costs.

We will show in Section 5.5 that the absolute error heuristic is particularly effective for one of the unsolved TSPLIB problems.

5.3.4 Generating Predecessors in S'

We are given ϕ_i , ϕ_{i+1} and the cost matrix $C'_{i,i+1}$. The calculation of the states of S'_i are performed in three stages.

1. Generate $A_1 = \Delta^{-1}(s'_{i+1})$ pretending that $\phi_i = \phi_{i+1}$
2. From A_1 , which corresponds to ϕ_{i+1} , generate the abstract states corresponding to ϕ_i . Let this set be called A_2 .
3. From A_2 , eliminate the abstract states which have no preimages in S' .

This first step is relatively straightforward. For example, consider ϕ_5 in Equation 5.3 and the abstract state $s'_{i+1} = (1, 5)[2]$. Under ϕ_5 there are three predecessors of s'_{i+1} . These are

$$A_1 = \{(1, 2, 5)[1], (1, 2, 5)[3], (1, 2, 5)[4]\}$$

In the predecessors, $av(s'_i)$ includes the available labels $av(s'_{i+1})$ and the current label at level $i + 1$, $curr(s'_{i+1})$. The current vertices at level i are chosen from the vertex labels which are not available in s'_{i+1} . This excludes the label 0, unless $i = 0$.

The second step is to rewrite the states of A_1 so that they correspond to ϕ_4 which was either obtained as the result of merging or splitting. In either case, we can build a map M while performing the successive merges or splits such that in the case of row merging $M[x] = y$ is the unique label x becomes in ϕ_4 or for splits $M[x] = \{y\}$ returns the set of labels that x corresponds to in ϕ_4 . It then becomes straightforward to apply M to the states of A_1 and obtain the set A_2 .

The third step needs justification. Consider our rules (page 108) which are used to derive the cost entries in $C'_{i,i+1}$. In particular, rules 3 and 4 can potentially eliminate precedence constraints. Because of this, it is possible that the set A_2 includes states which have no preimages in S . Spurious states in our case do not

pose a memory problem because we preallocate arrays large enough to hold each possible permutation of abstract states for each level of S_i . However, as we have shown in Section 2.4, spurious states can be the cause of poor lower bounds as they can create cheap paths which are not refinable in the original state space. For the SOP, we can actually check if an abstract state has a preimage efficiently⁷. Elimination of spurious states can result in significant improvements in lower bounds, especially for SOP instances with a large number of precedences. For example, the best lower bound we could derive without checking for spurious states for the unsolved instance p43.4 from the TSPLIB [TSP] was 56,645, while with checking we could beat best previously known lower bounds even with very small pattern databases and we were also able to solve this instance optimally. Since this step is pivotal in our algorithm, we give a detailed description in Appendix A.

5.4 Search

Since the SOP state space is a directed acyclic graph (DAG), a depth first expansion always terminates. Since depth-first strategies are more memory efficient and we need memory to store the pattern database, our basic search strategy is Depth-first Branch and Bound (DFBB). However, for the SOP, basic DFBB can be greatly improved. We will discuss three novel enhancements. The first one is quite general. Its only requirement is that the state space can be searched in both directions. The other two rely on certain properties which, while less common,

⁷The actual complexity is $O(nk)$ where n is the number of vertices and k is the number of preimage vertices of the current vertex of the partial tour

are present in other problems as well.

5.4.1 Bidirectional Depth-first Branch and Bound

The first improvement comes from the observation that DFBB spends the vast majority of search refining the “tail” end of a solution. In other words, when considering the vertex order of successive suboptimal solutions, it is likely that the first few vertices are the same. This is the expected side effect of DFBB always expanding the most recently generated node. Figure 5.9 depicts how

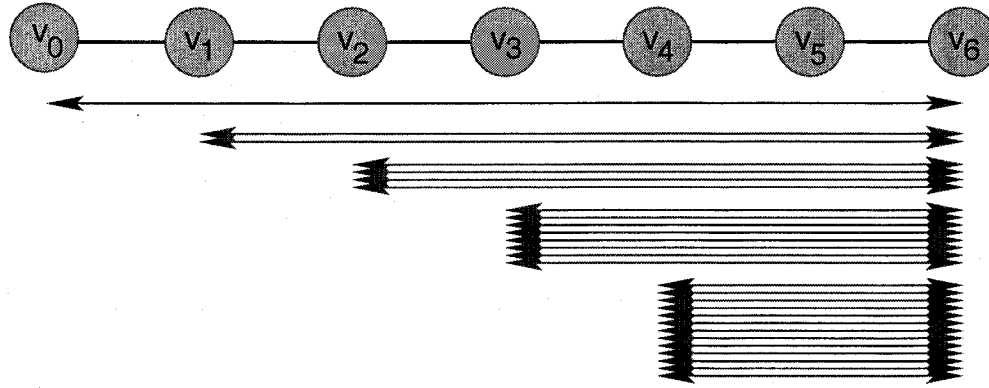


Figure 5.9: Vertex Order Changes in the Incumbent

changes in the vertex order of the incumbent solution are likely to occur. In successive incumbents, most frequently only the suffix vertices get shuffled. The prefix vertices change much less frequently, as indicated by the number of arrows. In our initial experiments we also observed that often a major improvement in the cost bound requires a change in a vertex closer to the start (searching under another larger branch of the search tree). Therefore, we wanted to force changes in the prefix earlier on to have faster convergence of the pruning bounds to the optimal solution.

When the space can be searched in both directions with DFBB, another DFBB can be initiated in the other direction which uses the incumbent not only as a bound but also its actual vertex order. In other words, the DFBB search in the other direction is *seeded* with the incumbent, such that it is the first solution found. We would like to point out that this can be done simply by ordering the successors obtained by Δ so that the vertices of the incumbent at their respective depths are put first on the first descent.

The two requirements of using Bidirectional DFBB (BiDFBB) are being able to implement Δ^{-1} and being able to derive a lower bound.

For the SOP specified by cost matrix C , we can simply mirror the costs about the top-right bottom-left diagonal⁸, and the resulting cost matrix, $C^\dagger$, is the definition of the *reversed* SOP.

For example, for the helicopter routing cost matrix C (Equation 5.1)

$$C^\dagger = \begin{pmatrix} \times & 6 & \times & \times & 7 & \times & 6 & \times \\ \gamma & \times & 4 & \gamma & \gamma & \gamma & 12 & 7 \\ \gamma & 5 & \times & 4 & \gamma & \gamma & 13 & 12 \\ \gamma & 7 & 4 & \times & \gamma & 4 & 7 & \times \\ \gamma & \times & \times & 2 & \times & 4 & 6 & \times \\ \gamma & 4 & 6 & \times & 4 & \times & 4 & \times \\ \gamma & 10 & 11 & 6 & 5 & 2 & \times & 7 \\ \gamma & \gamma & \gamma & \gamma & \gamma & \gamma & \gamma & \times \end{pmatrix} \quad (5.10)$$

Therefore, we can use our automatic technique to build pattern databases to search in either direction. Our BiDFBB switches directions after expanding L

⁸Transpose about the *other* diagonal

nodes or after finding a new incumbent (whichever happens first). The incumbents are passed from one direction to another, hence the solutions will be successively cheaper and, in our experience, will also converge faster to the optimal solution. Note that only one of the pattern databases has to be kept in memory. The pattern database corresponding to the other search direction can be dumped to the hard disk and read back⁹ when the search again changes direction.

5.4.2 Recursive Depth-first Branch and Bound

There is another property of the SOP which can be exploited to speed up DFBB. Let $z = \langle v_0, v_1, \dots, v_{n-1} \rangle$ be a suboptimal solution. If we select *any* subsequence of consecutive vertices $z_{i,j} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ from z , then $z_{i,j}$ is a feasible solution to a smaller SOP whose cost matrix $C_{i,j}$ corresponds to the entries of the original cost matrix C

$$C_{i,j} = \begin{pmatrix} C[v_i][v_i] & C[v_i][v_{i+1}] & \dots & C[v_i][v_j] \\ C[v_{i+1}][v_i] & C[v_{i+1}][v_{i+1}] & \dots & C[v_{i+1}][v_j] \\ & & \dots & \\ C[v_j][v_i] & C[v_j][v_{i+1}] & \dots & C[v_j][v_j] \end{pmatrix}$$

Recursive DFBB (RecDFBB) exploits this property the following way. Let L_i be a user specified limit on how many expansions the user is willing to tolerate for solving the subtree at level i with the current lower bound. When this limit is reached, the current search is abandoned and the completion of s_i is solved as an independent SOP. The subproblem SOP has fewer vertices (and therefore it is a much smaller problem) so we expect to generate more accurate lower bounds.

⁹In our implementation it is a simple memory dump as our pattern databases are contiguously allocated. There is no parsing or building a data structure.

Moreover, we also have the cost of the best currently known solution. When the subtree is solved independently, we are only looking for solutions whose costs are actually better. In most cases it takes much less time to prove that no solution exists whose value is less than this new bound than actually finding the optimal solution to the subproblem.

5.4.3 Level-wise Transposition Tables

DFBB does not keep track of states already reached and therefore some states are reached many times during search. From the representation

$$s_i = (v_{l+1}, v_{l+2}, \dots, v_{n-1})[v_l]$$

we know that the vertices $v_0, \dots, v_l$ have been used, but we only know that v_0 was the start vertex and that this partial tour ends in v_l . The different precedence observing combinations of $v_1, \dots, v_{l-1}$ are all valid partial tours represented by s_i and in the worst case all of them could be traversed in the search. These different paths to s_i are often referred to as transpositions in the search literature. To reduce the number of transpositions, we can utilize *transposition tables* to store the cost of the cheapest cost of reaching s_i so far. If the algorithm encounters s_i with a higher cost, then s_i does not have to be expanded again. The drawback of keeping transposition tables is the memory overhead. Since we use memory to store pattern databases, we would have to consider if it is worth to trade some of it to store transposition tables instead. In the case of the SOP, it turns out that there is a justifiable answer. We have a perfect hash function which operates on S'_i (see Appendix B). The hash on S is just a special case of this where ϕ maps each vertex to itself. Therefore we also have a perfect minimal hash function to

index the transposition tables associated with a particular level. There are no transpositions on levels 0, 1 and 2. The size of level S_i is given by Equation 5.2. Therefore, we can calculate how much memory we would need exactly to store transpositions at levels 3 and below. Since re-expansions at the levels closer to the start (larger subtrees) are the most expensive, we will utilize transposition tables for these shallower levels only. This memory is taken away from the pattern database. In essence, the n levels of the pattern database memory is split as follows. Levels 1 and 2 used for pattern databases. For x consecutive levels, starting at level 3, all memory is used for transposition tables. The remaining $n - x - 2$ levels are used for pattern databases again. The x levels used for transpositions are full (no domain abstractions) therefore x is usually between 2 and 4, depending on the size of the problem.

5.4.4 Local Search and Perfect Hashing

Depth-first branch and bound finds feasible paths of decreasing cost. Since these costs are used as upper bounds to prune branches, it is very important to find cheap tours early. BiDFBB is one generic approach. For the SOP, there is a well known heuristic tour improvement method, based on the TSP 3-opt [LK73], which can be used to improve the solution quality of a feasible solution (also employed by [AJR00, GD00, SM03]). Whenever a new incumbent is found, we run this 3-opt tour improvement heuristic.

The lower bound $h(s_i)$ for a state is obtained by looking up the value $H[\phi_i(s_i)]$ in the pattern database H . In our case H is implemented as a simple array. We were able to derive a perfect hash function I that returns the unique index of the

state s'_i in H . I is a perfect minimal hash function which is calculated together with ϕ_i . Its time complexity is $O(|av(s'_i)|)$ where $|av(s'_i)|$ is the number of *distinct* labels in the *available list* of the representation of s'_i . Since this hash function plays a very important role (it is also used in calculating the width of S'_i , without expanding S'_i), we describe it in Appendix B.

5.5 Experimental Evaluation

In this section we describe the experiments we performed to evaluate our pattern databases and algorithms. We were interested in the following questions.

- Can we solve or improve bounds for unsolved instances of the SOP from the TSPLIB?
- How do we compare to the state-of-the-art solver (Ascheuer *et al.*'s branch and cut [AJR00])
- Which error measure should we optimize with our greedy local search to obtain the best pattern databases?
- How much does doubling the pattern database size speed up search?
- Does the constrainedness of the SOP affect our algorithms?
- How do our improved algorithms compare to plain DFBB?

5.5.1 Test Sets

Norbert Ascheuer collected a number of challenging instances of the SOP which can now be downloaded from the TSPLIB [TSP]. Some are actual schedules from industry, while others are instances of ATSPs from the TSPLIB to which random precedences were added. After some initial experimentation, we realized that our technique would only be able to compete on instances that have no more than 40–50 vertices. Therefore we have only performed detailed experiments with the following problems from Ascheuer’s problem sets:

- stacker crane schedules: rbg019a – rbg050c (which excludes the problems rbg068a – rbg378a, of which rbg285a is the largest solved instance).
- TSPLIB ATSP instances with random precedences: p43.1–4, ry48p.1–4, and ft53.1–4. The naming convention is $\langle \text{name} \rangle \langle n \rangle . \langle c \rangle$, where n is the number of edges in a solution (which is one fewer than the number of vertices) and c is an indicator of the number of constraints. $c = 1$ instances have $\frac{n}{4}$, $c = 2$ have $\frac{n}{2}$, $c = 3$ have n while $c = 4$ instances have $2n$ potential precedence relationships. Our set excludes the problems ft70.1–4 and krol124p.1–4, because they were too large (71 and 125 vertices).

All of the TSPLIB instances with 2–4 precedences were unsolved despite the fact that some have only a few vertices (44–54)¹⁰. These unsolved problems, however, are far too large to run extensive experiments, so we also created a testbed of 16 smaller problems. Each problem is a SOP over 30 vertices¹¹ and

¹⁰We were able to solve the single instance p43.4

¹¹3,758,096,386 states without precedences

in each experiment we ran the algorithms to completion. These 16 SOPs are derived from 4 base problems. The first one was obtained by choosing 30 points on a 500×500 integer grid. The edge costs correspond to the rounded Euclidean distances between these points. We also picked one third of the edges randomly and added random noise within $\pm 50\%$ of the edge cost (to make the cost matrix asymmetric). The second SOP was generated exactly in the same manner but we selected the 30 points on a 6×6 grid. This latter only has edge costs in the interval $[0, 8]$. The edge costs for the third and fourth SOPs were generated by choosing uniform random integers from $[0, 1000]$ and $[0, 10]$ respectively. Next, we generated four random precedence graphs. We applied the constraints implied by these to each of our four base problems and obtained SOPs with 107 (1), 160 (2), 210 (3) and 253 (4) precedence constraints. These include the 57 trivial precedences which are due to the fact that the start and terminal vertices are designated. For ease of reference, we named these 16 test problems suggestively. 500 $\times$ 500-2 is the SOP whose vertices are chosen from the 500×500 grid and to which the second precedence graph was applied. Therefore its cost matrix has 160 precedence constraints.

The two control parameters we supply to calculate the lower bounds are the maximum number of abstract states on a single level w and the goodness function f which is optimized for the abstractions. In this discussion we use the maximum width as the measure of the size of pattern database as it is directly proportional to the total memory requirements (compare the columns “Maximum Width” and “Actual Size” in Table 5.2, page 135). We have considered four different error functions. These are the three measures we discussed in Section 5.3.3, namely “Preserve Precedences”, “Absolute Error” and “Normalized Error”, and

we also included “Random” which picks the labels to be merged entirely randomly. “Random” is used only as a benchmark for our domain abstractions which were derived by heuristic local search.

5.5.2 Error Functions

In all of our experiments we found that “Absolute Error” (AE) generates the most accurate lower bounds, even when the penalty value for eliminating a precedence constraint is zero. Applying a good penalty value would often tighten the bounds. Unfortunately, at this time, we have no insight how to derive the actual penalty values. Based on “trial and error” experiments we can conclude that the lower bounds are sensitive to different penalty values (could make the bounds both better and worse) and 0 is a relatively “safe” value to use. In one particular experiment we found that minimizing AE clearly outperforms the other error measures and in some cases it generates almost 100 times higher lower bounds than randomly chosen abstractions. The lower bound of the start state, h_0 , is also a lower bound of the optimal solution. In this experiment, we obtained significantly higher h_0 ’s, even with small pattern databases, than the best posted value of 69,569. This problem is referred to as p43.4 in the TSPLIB. We later solved this previously unsolved problem. Our best h_0 is 81,585 (which is 17% higher than the best previously known lower bound). Table 5.1 shows the results of this experiment. h_0 is the lower bound of the start state, $\text{ave}(h)$ is the average of the values in the pattern database and Time is the CPU time in seconds needed to build the pattern database. The values in bold are higher than the best lower bound previously reported for this problem. For each size, we generated

Max Width	Random			Absolute Error		
	h_0	ave(h)	Time	h_0	ave(h)	Time
500	400	402.90	0.57	53,965	53,704.10	0.57
1,000	435	668.63	0.69	54,110	47,218.00	0.62
2,000	490	830.25	0.80	80,690	59,256.60	0.77
4,000	510	848.48	0.85	54,110	50,163.50	0.92
8,000	545	1,219.24	1.00	54,115	52,356.10	1.13
16,000	580	1,631.24	1.15	54,000	50,853.30	1.17
32,000	630	2,146.16	1.46	80,890	65,315.20	1.29
64,000	655	2,624.48	1.69	54,045	52,732.60	2.14
128,000	680	2,934.33	2.13	81,055	60,713.50	2.24
256,000	700	3,618.05	2.50	81,140	77,298.50	2.85
512,000	710	4,925.33	3.30	81,110	69,782.70	2.63
1,024,000	955	5,000.84	4.05	81,320	74,026.30	3.71
2,048,000	965	7,841.39	5.82	81,350	65,885.20	4.48
4,096,000	1,130	8,218.48	8.95	81,385	70,395.50	6.23
8,192,000	1,265	9,704.65	18.43	81,585	70,473.30	12.87

Table 5.1: Generating Lower Bounds for p43.4

20 random pattern databases and took the highest h_0 , the highest $\text{ave}(h)$ and reported the time for the shortest calculation in the “Random” columns. A row in the AE columns correspond to a single pattern database of the given size which optimizes the AE error with no penalty. It is clear from the results that even the best values chosen from 20 random databases are far worse than the one built optimizing AE. In fact, our smallest AE pattern database has much higher lower bounds than the best of 20 random ones which are 16,384 larger. While these large discrepancies were not typical in our experiments, our results suggest that experimentation with the error measure for a particular SOP could be well worth the effort. This experiment also reveals an interesting phenomenon that we encountered on several occasions. The AE pattern database with width 2000 is better than many of the larger ones optimizing AE (column h_0 under “Absolute Error” for width 2,000 in Table 5.1) and its h_0 (80,690) is also significantly better than the best previously known lower bound (69,569). At this time we have no explanation other than the abstractions generated seem to preserve the costs between the levels particularly well.

We would also like to draw attention to another interesting observation. We would expect that it would take approximately twice as much CPU time to build a pattern database twice as large. From Table 5.1 (columns with label “Time”), however, it is clear that it often takes much less than double the time to build a double size pattern database. The reason is that the larger the pattern database, the more of the original precedences are preserved and therefore the actual number of entries in the double size pattern database is actually less than the entries in the half size pattern database multiplied by 2. We allocate memory to store all possible states in the pattern database, even for those which cannot be reached

due to precedence violations. When there are many precedence constraints, the large pattern databases are much sparser than the small ones.

5.5.3 TSPLIB Problems

We did not manage to improve on the best known upper bounds (corresponding to heuristic solutions) of the TSPLIB instances but in one case we proved that the upper bound of 83,005 for the previously unsolved instance, p43.4, is also optimal¹². On the other hand, we did match the best reported upper bounds within 60 minutes of CPU time to the open problems p43.2, p43.3, p43.4, ry48p.2, ry48p.3, ry48p.4, ft53.2 and ft53.3. We ran BiDFBB and used a 3-opt to improve each incumbent found. We did not reach the best known upper bounds for ft53.4, p43.1, ry48p.1 and ft53.1. These ones with the .1 suffix are the least constrained and these were solved optimally by Ascheuer *et al.* [AJR00] but our solver did not reach the optimal values although we have tried more than one of our algorithms (BiDFBB, RecDFBB with and without transpositions tables) and we allowed for several hours of running time. The largest gap was in the case of ry48p.1, where we obtained 16,040 and the optimal value is 15,805. This particular problem was left running for a few days. Interestingly, the bound was found in a few minutes but our solvers could not improve on it. As we mentioned earlier, we did not attempt the problems ft70.1-4 and krol24p.1-4 because they were too large. Ascheuer *et al.* [AJR00] also have solved a number of larger

¹²which also contradicts the upper bound posted as 82,960 whose origin we could not trace. More than one solver has reached the value 83,005 but we were the first to prove it optimal. Both Ascheuer *et al.* [AJR00] and Gambardella *et al.* [GD00] report this value as an upper bound.

SOPs based on stacker crane schedules. They found these problems particularly “easy” even though many have well over a 100 vertices and contain many precedences. We could also solve the ones up to 50 vertices (rbg019a – rbg050c) but significantly slower than the branch and cut solver (hours instead of seconds). The stacker crane schedules, while posted in the TSPLIB, are not derived from TSPLIB ATSPs but correspond to actual stacker crane schedules. Ascheuer *et al.*’s branch and cut solver could easily manage the stacker crane problems (Table 1 in [AJR00]) but could not solve the much smaller more constrained instances which were derived from TSPLIB ATSPs (Table 4 in [AJR00]). Our solver found the stacker crane and TSPLIB instances equally hard. The following list is a summary of our achievements on the TSPLIB problems compared to Ascheuer *et al.*’s branch and cut.

- Problems Ascheuer *et al.* solved and we did not: p43.1, ry48p.1, ft53.1 and ft70.1
- Problems for which our solvers report the same upper bound: p43.2, p43.3, ry48p.2, ry48p.3, ry48p.4, ft53.2 and ft53.3 ¹³.
- Problems which our solver solved but Ascheuer *et al.*’s did not: p43.4.
- Still open problems for which Ascheuer *et al.* report cheaper costs: ft53.4.

As we mentioned earlier, we did not attempt ft70.1-4 and krol24p.1-4. Of these only ft70.1 is solved and the others are still open.

¹³It is not known exactly how much time it took the branch and cut to reach the actual upper bounds

We also realized that we can improve on the lower bound h_0 (which is the lower bound for the start state from the pattern database) just by running a simple depth-first search (without node ordering) with a cost cutoff set at the best known lower bound. We tried this with the lower bounds reported by [AJR00] for the open problems. Interestingly, we could also prove many of these lower bounds within 60 minutes but when we increased the cutoff bound just a little bit¹⁴, we could not find a better lower bound in reasonable time.

In the case of p43.4 we ran RecDFBB with 4 levels of transposition tables to completion. While it is the smallest unsolved problem, it is also the one which had the largest gap between best known upper and lower bounds. It took over 22 hours of CPU time before it terminated proving that 83,005 is indeed the optimal solution. Ascheuer *et al.* [AJR00] also reached the exact same solution cost but could not prove it is optimal.

5.5.4 Space-time Trade-off

In this section we investigate how increasing memory to store lower bounds affects search speed. To this end, we solved the 16 test SOPs, each with 7 different pattern databases of exponentially larger size. The size was measured in width, from 25,000 to 1,600,000 entries. For all these pattern databases, we optimized the “Absolute Error” measure in the abstract space. In these experiments we did not use the 3-opt (or any) heuristic tour improvement. Here we report only the details for 500×500-3, since it represents the general trend we observed for

¹⁴the values in these cost matrices often have a common divisor larger than 1, so we raised the cutoff by the greatest common divisor

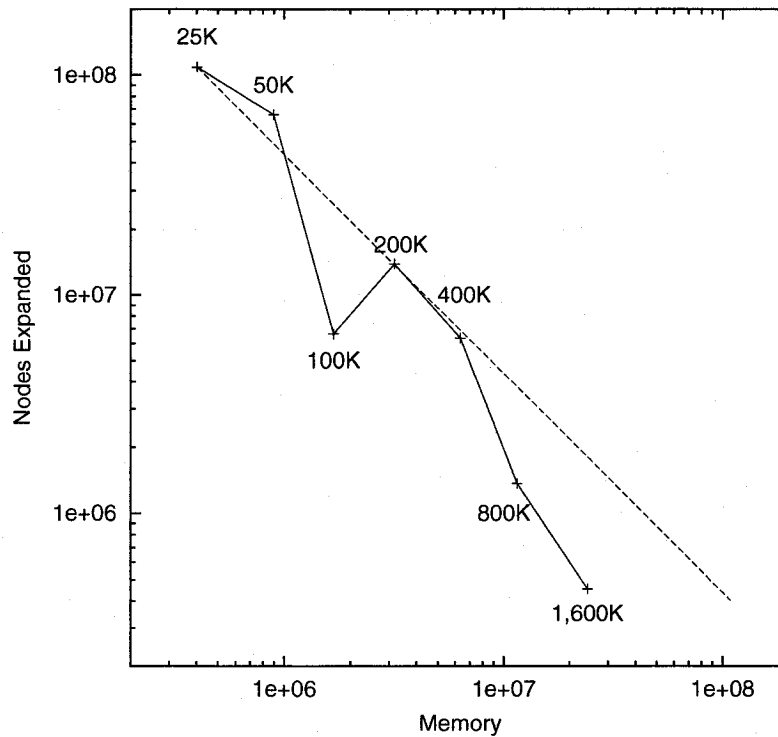


Figure 5.10: Nodes Expanded vs. Memory

the other test instances as well. Figure 5.10 plots the number of nodes expanded during search and the actual size of the pattern database, this time measured as total number of entries. The corresponding width is used to label the data points. Both axes are on a logarithmic scale. If the trade-off was 1-1, then the data points would be on a line with slope -1 (drawn as a dashed line). In fact, in this case it seems that the trade-off is even better. Doubling memory to store lower bounds results in less than half of the nodes expanded in the search. The notable exception is the pattern database with width 100K. This one clearly outperforms the one double its size and it is as good as the one four times larger. The explanation is given in the following section.

Max Width	Actual Size	h_0	First Solution	Nodes Expanded	Search (<i>sec</i>)	DB (<i>sec</i>)
25,000	402,043	2,577	4,183	108,618,722	261.61	0.11
50,000	895,986	2,905	4,655	66,025,147	160.84	0.14
100,000	1,680,769	3,023	5,048	6,621,231	19.16	0.72
200,000	3,178,762	3,065	5,433	13,920,570	40.68	1.33
400,000	6,339,234	2,975	4,183	6,339,234	19.23	2.14
800,000	11,517,316	3,292	4,375	1,367,800	4.54	3.62
1,600,000	24,147,235	3,109	4,695	452,692	1.80	5.10

Table 5.2: Results of Solving 500×500 3 with Different Databases

5.5.5 Cost Convergence

Since DFBB (and our improved versions) generates a sequence of decreasing cost solutions until the optimal is found, we are interested in how fast the cost of the incumbent solution converges to the optimal. Again, we use the results of 500×500-3 to demonstrate a trend we observed in all of our experiments. We use the same 7 pattern databases as in the space-time trade-off experiment in Section 5.10. The difference is that we also recorded the cost of each intermediate solution and when (CPU time) this solution was obtained. The results of this experiment are listed in Table 5.2. The column labeled “Actual Size” is the number of entries allocated for the pattern database (which includes unused entries that would correspond to precedence violating partial tours). The column “First Solution” is the cost of the first incumbent, “Nodes Expanded” is the total number of nodes expanded in solving the problem, “Search” is the time in seconds needed

to solve the problem and “DB” is the time to build the pattern database, also measured in seconds. The corresponding plot is shown in Figure 5.11, where the number of nodes expanded are on the x -axis and the solution cost is on the y -axis. The plot tells why the 100K width pattern database outperforms the one with width 200K. It is clear from Figure 5.11 that the database with width 200K

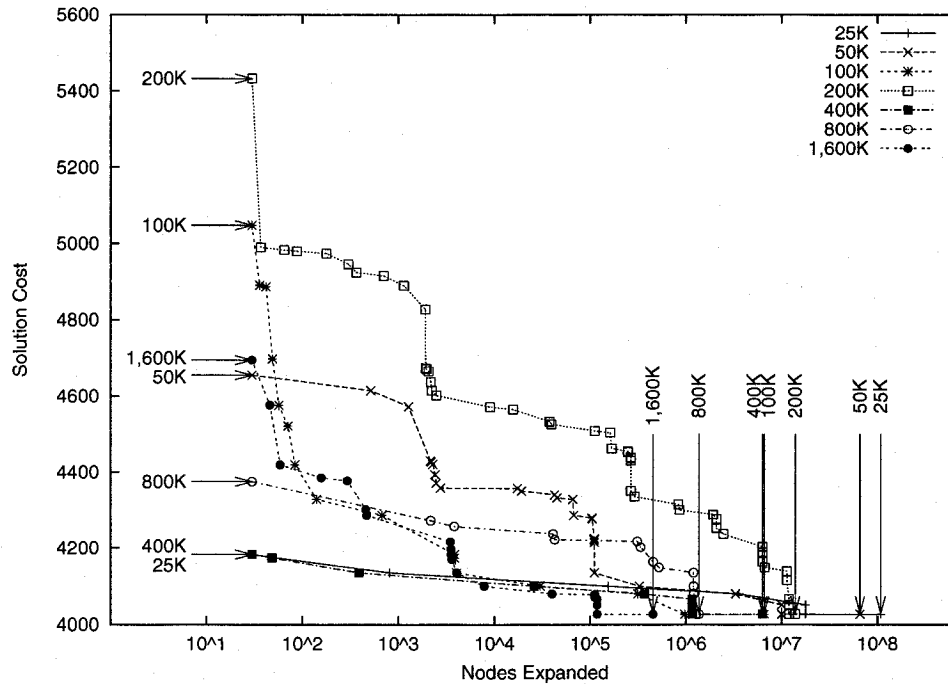


Figure 5.11: Solution Cost vs. Nodes Expanded for 500x500-3

starts with the costliest solutions and a lot of search effort is needed before the cheaper solutions are found. The value of the first solution found seems to be uncorrelated with the size of the pattern database (Table 5.2, columns “Actual Size” and “First Solution”). In this particular case, the smallest pattern database has the cheapest initial solution. The lower bounds play two roles in our search algorithm. They are used to prune branches from the search tree and to order the branches. The value of the first solution depends mostly on how the branches are

ordered. From the results of this experiment and other experiments we performed we cannot establish any relationship between the size of the pattern database and the quality of the initial solutions. We did compare to random ordering, which turned out to be consistently worse. However, to date, we found no telling statistics of the pattern database which would suggest cheap initial solutions.

While nodes expanded is an appropriate and exact measure to evaluate search performance, we also have to take into account the overhead of building pattern databases. Our results indicate (Table 5.2) that even with the overhead, in the same experiment, larger pattern databases resulted in faster searches. The order is almost the same as established by the number of nodes expanded, except that the pattern database with 100K width limit, with respect to CPU time, slightly outperforms the database whose maximum width is 400K.

5.5.6 Constrainedness vs. Search Speed

We are also interested in how adding more precedence constraints affect search performance. Ascheuer *et al.* [AJR00] observed that their branch and cut method is more effective for SOPs which have fewer precedence constraints. In particular they could solve all of the “pure” ATSP instances¹⁵ optimally. In the case of the TSPLIB problems with random precedence constraints (problems p43.1-4, ry48p.1-4, ft53.1-4, ft70.1-4 and kro1124p.1-4), they could easily solve the least constrained ones (suffix .1) but not the more constrained ones (Table 4 [AJR00]). Ascheuer, in his thesis [Asc95] (page 127), conjectures that probably the following two factors play a role in having less success with more constrained

¹⁵which are, in essence, SOPs with no precedence constraints

problems. First, “*the heuristics we are working with at the moment, do not handle the precedence structure properly*” and second, “*the TSPLIB problems are hard, i.e., more complex inequalities than the SEC [Subtour Elimination Constraints] are needed to solve them optimally*”. The limiting factors in branch and cut, besides the dimensionality, are the size of the pool of inequalities that have to be searched and the construction of the facet inducing inequalities. None of these are eased by more precedences. For us, more precedence constraints mean less branching in the search space and therefore fewer node expansions. On the other hand, it is more likely that more of the original constraints would be eliminated by the abstractions if there were more to begin with. Losing a significant portion of the original constraints could result in poor lower bounds. Our experiments indicate that in most cases the more constrained the SOP the faster our searches. Table 5.3 (page 145) shows the compiled results of our experiments with the 16 test SOPs. We chose 3 different size pattern databases; with maximum level widths of 50K, 250K and 1,250K states respectively. In all cases we minimized the “Absolute Error” measure when building the abstract spaces.

While we can easily solve these problems to optimality with large databases, we also used a small and a moderate size database as well to study how adding more constraints affects the quality of lower bounds with different size pattern databases. From our experiments we can establish that regardless of the size of the pattern database, in general, the more constrained the SOP the earlier the search terminates. The two exceptions are 0-1000-3 and 0-10-3. They both need more computation than their less constrained counterparts 0-1000-2 and 0-10-2. We believe we know the reason for this. 500×500 and 6×6 are derived from Euclidean base problems. When the “Absolute Error” is minimized, it is

likely that “physically close” vertices receive identical labels by the abstractions as their proximity to close vertices would carry the least absolute error. On the other hand, the edge costs of the 0-1000 and 0-10 problems are totally random. Therefore, for these, instead of optimizing “Absolute Error”, reducing eliminated precedences could be more appropriate. This conjecture is also supported by the h_0 values. In the case of 500×500-3 the lower bound for the start state is 72% of the optimal cost while it is only 40% for 0-1000-3 according to the smallest databases. When experimenting with different penalty values applied when a constraint is eliminated, we were able to bump up 0-1000-3’s h_0 to 81% of the optimal cost with the same size pattern database. Our results also show that having larger databases pays off. In the case of 500×500-1 allocating a 5 times larger database results in a 148 fold reduction in nodes expanded. Increasing this size yet another 5 times and there is another a 21-fold reduction in nodes expanded. While these very high ratios are not typical, our experiments indicate that in general the trade-off between memory and search speed favors adding more memory.

5.5.7 DFBB, RecDFBB and Transposition Tables

In this section we compare DFBB to RecDFBB and RecDFBB with transposition tables. For this experiment we used our 16 test SOPs. For RecDFBB, in all cases we set a tolerance limit of 300 times the width of the abstract space. When this limit is reached, a new pattern database is built for the sole purpose of solving the largest current subtree excluding the start state. The node counts reported in Table 5.4 for the recursive branch and bound experiments (column

marked Rec) include the nodes expanded in the subproblems as well. Because there is additional overhead of building more pattern databases, we also report the actual CPU times of the total runs. The column Rec+Trans lists the results of using recursive branch and bound and two levels of transposition tables (levels 3 and 4). The results suggest that in most cases recursive branch and bound outperforms plain branch and bound even with its additional overhead. Of the three algorithms, the statistics of the winner for the problem is shown in bold font. In some cases the improvements are negligible or there is an increase search effort. In other cases the improvements are significant. For example in the case of $500 \times 500-1$, the improvements are 7-fold and using transposition tables as well are almost 9-fold. For 0-1000-3 and 0-10-3 the improvements result in less search than using the 5-times larger pattern database (compare the Rec+Trans rows for 0-1000-3 and 0-10-3 in Table 5.4 to the same rows in the 1,250,000 column).

5.6 Additional Constraints

Our lower bounds are derived from state space abstractions. These are induced by domain abstractions which are simple syntactic patterns that group states. While we tested our technique with the classic SOP only, we believe it could also be effective to solve other constrained TSPs as well.

Here, we detail two variations of the SOP for which our technique could generate lower bounds. We do not have experimental results at this time to actually evaluate their effectiveness. However the fact that we can generate lower bounds with domain abstractions for these problems shows the generality of our approach.

5.6.1 Disjunctive Precedence Constraints

In the classical definition, the precedence is given explicitly. On the other hand one can imagine a more elaborate constraint structure where the constraint involves a choice. For example, “*one-of (3 and 4) must precede vertex 5*” or “*two-of (2, 3 and 4) must precede vertex 5*”. Once we add disjunction we can express arbitrary boolean constraints of precedences. The precedences must be in the form

(*boolean constraint of vertices*) must-precede vertex

It is easy to see that we can encode such constraints into the successor generator. Let s be a state representing the partial tour ending in the vertex with label v_c

$$s = (v_1, v_2, \dots, v_k)[v_c]$$

When we create the successors, we simply check that the precedence constraints are observed for v_c which is the right-hand side of the must-precede operator. In the abstract state, the domain abstractions must also be applied to the constraint definitions. For example, consider the constraint

two-of (2, 3 and 4) must-precede 5

and the domain abstractions

$$\phi_1(v) = \begin{cases} 2, & \text{if } v \in \{2, 3\} \\ v, & \text{otherwise} \end{cases}$$

$$\phi_2(v) = \begin{cases} 2, & \text{if } v \in \{2, 3, 5\} \\ v, & \text{otherwise} \end{cases}$$

Under ϕ_1 the constraint reads

two-of (2, 2 and 4) must-precede 5

and under ϕ_2

two-of (2, 2 and 4) must-precede 2

Since domain abstractions do not eliminate vertices just relabel them, checking the abstract constraints is still trivial from the state representation. With ϕ_1 , we check if the tour has used up two 2's or a 2 and a 4 labels. Otherwise the state is not generated. With ϕ_2 , if the successor's current vertex is 2, we still have to make sure that either two other 2's or a 2 and a 4 are already part of the tour.

5.6.2 Soft Precedence Constraints

We can also imagine that some precedences are cast in stone while others may be violated if some penalty is paid. It can easily be detected at the time the successors are generated if a penalty applies. The value of the penalty is added to the cost of the partial tour corresponding to the successor. This is also true in the abstract space. However, we will only be able to add the penalty if the uniqueness of the penalized edge is preserved. That is, when the labels of the source and end vertices are not abstracted. Otherwise only the minimum of the original penalties between the preimage vertices can be applied. However a well designed error measure could optimize preservation of penalties in the abstract space.

5.7 Chapter Summary

We have introduced a novel way of building pattern databases which uses different abstractions on each level of the SOP state space lattice. The actual domain abstractions were obtained by a greedy local search optimizing a user provided error measure. Our experiments indicate that optimizing “Absolute Error” results in the most accurate pattern databases and the lower bounds are substantially tighter than simply choosing the abstractions randomly.

We also have introduced several new variations of depth-first branch and bound. Recursive Branch and Bound (RecDFBB) takes advantage of the property of the SOP that every suffix of a feasible solution is a SOP itself. This SOP is much smaller and our pattern database technique can be used to solve this subproblem independently. We have experimentally shown that RecDFBB could substantially speed up search and it can often be much faster, even with its extra overhead, than plain DFBB. In addition, using a few levels of transposition tables can further speed up the search.

Bidirectional Branch and Bound (BiDFBB) exploits the fact that the SOP state space can be searched in both directions. This algorithm was designed to overcome the slow cost convergence of intermediate solutions. We used BiDFBB for some of the smaller unsolved TSPLIB instances and we were able to match previously reported best bounds in reasonable time. BiDFBB is not only applicable to the SOP, but for all state spaces which can be searched in both directions.

We experimentally confirmed that the larger the pattern database the faster the search. In fact, often the ratio of improvement in search speed far exceeds the ratio

of memory increase. Unfortunately we could not establish a similar relationship between the size of the pattern database and the cost of the initial sequence of solutions. To obtain better solutions early we can use BiDFBB or a local search, like 3-opt.

The pattern database technique is a general strategy to derive lower bounds for state space search. We believe that it can also be used with other variations of constrained ATSPs and possibly to solve other scheduling/optimization problems (we have described disjunctive and soft precedences). We also believe that our technique can be integrated with other SOP solvers. For example, instead of a simple 3-opt (or in addition), our algorithm could be seeded with the cheap solutions of the HAS-SOP (heuristic) solver of Gambardella *et al.* [GD00]. This way a good quality solution is guaranteed early. We also believe that the shortcomings of our pattern databases in finding cheap solutions early could be solved by Ascheuer's [AJR00] branch and cut. The solution to the LP relaxation is a fraction in $[0, 1]$ corresponding to each edge of the SOP. We could interpret these values as probabilities that tell how likely it is that the particular edge is included in the tour. We would sort branches by these probability values but still use our lower bounds for pruning.

SOP		Width					
		50,000		250,000		1,250,000	
		1,000 Nodes	Time (sec)	1,000 Nodes	Time (sec)	1,000 Nodes	Time (sec)
500×500	1	3,046,341	21,790	20,596	208.29	964	10.57
	2	212,656	1,132	8,761	52.63	692	5.27
	3	66,025	278.13	7,750	37.92	460	3.07
	4	27,796	127.25	9,288	49.80	488	3.17
6×6	1	9,022,057	66,451	1,775,135	16,058	141,987	1,570
	2	1,154,906	5,818	35,283	206.61	3,476	25.25
	3	26,579	133.14	9,509	57.88	814	5.63
	4	988	4.84	176	1.06	4	0.03
0-1000	1	360,391	2,773	34,060	336.05	17,091	184.29
	2	25,661	152.81	3,787	26.34	159	1.51
	3	266,526	1,243	70,435	376.39	8,936	60.89
	4	176	1.04	32	0.22	4	0.04
0-10	1	118,071	721.58	33,489	248.34	3,006	32.74
	2	468	3.06	202	1.48	54	0.52
	3	59,310	290.84	22,045	126.68	10,943	61.89
	4	7,788	40.01	895	5.33	43	0.32

Table 5.3: Nodes Expanded and CPU Times with Different Size Pattern Databases

SOP		Width = 50,000					
		Plain		Rec		Rec + Trans	
		<i>K</i> Nodes	Time	<i>K</i> Nodes	Time	<i>K</i> Nodes	Time
500×500	1	3,046,341	21,790	405,420	3,220	350,431	2,817
	2	212,656	1,132	351,017	2,113	315,387	1,922
	3	66,025	278.13	33,327	146.79	26,871	118.18
	4	27,796	127.25	56,153	248.63	44,182	198.21
6×6	1	9,022,057	66,451	18,950,825	134,032	12,250,761	101,120
	2	1,154,906	5,818	304,600	1,750	268,862	1,551
	3	26,579	133.14	26,579	135.48	21,293	108.86
	4	988	4.84	988	4.96	988	4.98
0-1000	1	360,391	2,773	347,409	2,732	317,975	2,506
	2	25,661	152.81	17,865	112.49	17,686	109.62
	3	266,526	1,243	51,707	230.21	48,021	212.11
	4	176	1.04	176	1.03	176	0.03
0-10	1	118,071	721.58	61,936	420.93	61,798	423.42
	2	468	3.06	468	3.11	468	3.13
	3	59,310	290.84	22,042	111.54	20,171	101.83
	4	7,788	40.01	7,788	40.95	7,786	40.93

Table 5.4: Nodes Expanded and CPU Times with DFBB, RecDFBB and RecDFBB with Transposition Tables

Chapter 6

Related Work

In this chapter we discuss related research which does not include work reviewed in previous chapters. The chapter is organized as follows. In Section 6.1 we examine how abstractions are used in the literature to speed up problem solving. In Section 6.2 we discuss the use and derivation of pattern databases. In Section 6.3 we briefly discuss analysis of search performance (much of it was detailed in Chapter 3). In Section 6.4 we describe other approaches to our two applications, macro-search and the Sequential Ordering Problem, and in Section 6.5 we examine other ways to use memory to speed up search.

6.1 Abstractions

The use of abstractions to guide search is a well known concept in Artificial Intelligence. In fact, the earliest attempts to solve problems faster by relaxing the original problem definition were used to solve planning problems. The planner

creates hierarchies of abstract spaces which are built from relaxations of the original definition. First the plan is solved in the most abstract level and then it is refined in the more concrete levels. Refinement in this context refers to extending or patching the abstract plan with operators to make it a valid plan in the original space. The classical planning systems GPS [NS61], ABSTRIPS [Sac74] and PRODIGY [Kno91] all use this technique to reduce search. The ALPINE [Kno94] *reduction* abstraction, however, is closer to our approach, since it results in abstract spaces which are smaller than the original. ALPINE too refines the abstract plan to obtain a solution. In contrast, we use the abstractions to derive lower bounds to search in the original space. In early work [Gas79, GS79, Pea84], often the effect of the abstraction was to add more edges to the state space graph. This happens when the constraints are relaxed or when macro-operators are added to the space. In these cases, the original space is *embedded* in the abstract space. Valtorta, however, proved [Val84] that every node expanded by blindly searching in the original space will also be expanded if the heuristic is computed by blindly searching in the abstract space. Therefore embedding abstractions are only useful if problems in the abstract space can be solved quickly and without exhaustive search.

Most search algorithms that find optimal solutions [HNR68, Kor85b, Kor93, LW66, Rus92, LW66] use lower bounds. These lower bounds are either hand-crafted or correspond to automatically derived solution costs in a simplified problem. Well known human designed lower bounds include the Manhattan distance heuristic for the $n \times n$ puzzle, the minimum weight spanning tree (MST) lower bound [HK70, HK71] for the Traveling Salesman Problem (TSP) and the assignment problem (AP) lower bound [CJMZ01] for the Asymmetric Traveling

Salesman Problem (ATSP). If the problem is formulated as an Integer Program then Lagrangian or Linear Programming (LP) relaxation can give accurate lower bounds. In Lagrangian relaxation [Fis81] the constraints appear as penalties in the objective function while the LP relaxation [Wol80] drops the integer constraints. The LP relaxation is particularly effective to solve large TSPs with the branch and cut algorithm [ABCC98]. The LP relaxation actually increases the dimension of the problem but there exist fast separation and lifting procedures which can deal efficiently with the large pool of constraints. The lower bounds mentioned above are the result of human intuition and their derivation requires a careful study of the original problem.

An alternative approach is to use automatic abstractions of the state space. Since Valtorta's [Val84] theorem applies to every abstraction, we are concerned with those abstractions that yield state spaces of smaller size. The use of homomorphic state spaces in Artificial Intelligence was first identified by Kibler [Kib82] and Banerji [Ban80]. The STAR abstraction technique of Holte *et al.* [HPZM96] groups neighboring states into abstract states. It is applicable if the state space is explicitly stored as a graph. The algorithm, Hierarchical A* [HPZM96] computes lower bounds on demand in this homomorphic abstract space. The relaxations suggested by Christofides *et al.* [CMT81] to solve routing problems also correspond to such homomorphisms. The abstractions are directly applied to the state representations. The ABSOLVER system of Priestitis [Pri93, MP89] uses a library of abstraction and speed-up transformations. This system discovered the center-corner heuristic for the Rubik's Cube which is a projection onto the corner-cubies. Later Korf [Kor97] included the center-corner heuristic in solving random instances of the Rubik's Cube for the first time. The pattern abstractions

introduced by Culberson and Schaeffer [CS94, CS98] are also homomorphisms. The idea is to mask some of the domain elements and therefore one *pattern* in the abstract space represents a group of states in the original state space.

Sometimes a lower bound can be tightened. One idea is to consider the constraints the abstraction violates in the original problem. For example, the Manhattan Distance allows the tiles to hop over one another. When this constraint is also taken into account [OHY92] a tighter lower bound can be obtained. Another standard technique is taking the maximum of lower bounds [CS98, Kor97, Her01]. It is also a lower bound and moreover it is never less accurate. Recently, however, lower bounds which can be summed were considered. Additive Lower bounds were used by Korf and Felner to derive optimal solutions to the 5×5 puzzle [KF02], by Felner, Korf and Hanan to solve multiple sequence alignment and the vertex cover problems [FKH03] and by Edelkamp [Ede01a] to obtain optimal plans for problems formulated in STRIPS [FN71]. Bonet and Geffner also used a sum of lower bounds which actually resulted in a non-admissible heuristic (not actually lower bound) but they did not insist on optimal plans. Their lower bounds would only be additive if the actual elements (atoms) the bounds correspond to were independent. Edelkamp [Ede01a] solved this problem for STRIPS by projecting the state spaces onto independent sets of propositions.

6.2 Pattern Databases

The term “pattern database” comes from [CS94] but the idea of storing lower bounds in a look-up table was independently developed in Artificial Intelligence

and Operation Research. To our knowledge the first work which uses precalculated lower bounds in state space search is due to Christofides, Mingozzi and Toth [CMT81]. They calculated dynamic programming tables of relaxed state spaces for constrained versions of the Traveling Salesman Problem and used the values (lower bounds) to search in the original state space. Culberson and Schaeffer [CS98] applied tile abstractions to the 15-puzzle and stored the lower bounds obtained from the abstract spaces in *pattern databases*. Frieditis' speed-up transformation "Precompute Lookup Table" in [Pri93] also results in pattern databases. Herbert Kociemba's "Two-phase Algorithm" [Koc] uses look-up tables of lower bounds to derive very short solutions to the Rubik's Cube.

Pattern databases have also been applied to many problem domains. These include puzzle solving [CS98, Kor97, KF02, JS99], STRIPS planning [Ede01a, Ede02], multiple sequence alignment [HTK03] and solving NP-hard optimization problems [FKH03, CMT81, MBR97, Her03a]. The pattern databases used in solving these applications are also different in some respects. Most of the above mentioned applications are domains where operators are of unit cost: puzzles [CS98, Kor97, JS99, Her01], STRIPS planning [Ede01a, Ede02] and the vertex cover problem [FKH03]. For the SOP and constrained TSPs [CMT81, MBR97, Her03a] there are potentially $O(n^2)$ costs where n is the number of cities. The abstract spaces corresponding to the multiple sequence alignment pattern databases are projections onto subsets of the sequences. Therefore there is no need to treat the costs as special. Similarly, the state space relaxations in [CMT81, MBR97] for constrained TSPs do not affect the cost matrix when calculating the dynamic programming table for the abstract state spaces. In this sense, our work clearly stands out for the SOP, since our abstractions are applied directly to the cost

matrix.

Most of the pattern databases simply store lower bounds. In this respect the work of Junghanns and Schaeffer on the Sokoban [JS99] is different as the pattern database is used to avoid deadlocks. Edelkamp's symbolic pattern databases [Ede02] also stand out as the abstraction that creates the pattern database is applied to the ordered binary decision diagram (OBDD) representation [Bry92] and not to the original STRIPS encoding of the problem.

The abstractions that induce the pattern databases in most previous work were handcrafted. For example the “fringe” and “corner” patterns for the 4×4 puzzle [CS98] and the pattern databases for the Rubik's Cube [Kor97] and the 5×5 puzzle [KF02] were picked by the authors. Edelkamp [Ede01a] uses an automatic selection procedure (based on bin-packing) to find pattern databases which come close to the given memory limit. In multiple sequence alignment [HTK03] the projection is always along the dimensions of the individual sequences that are aligned (therefore given). In [FKH03] dynamic partitioning of the tiles is considered and there is also mention of a gradient descent heuristic abstraction selection in [CMT81]. The hill-climbing search, however, that finds our abstractions is pivotal in our pattern databases derived for the SOP [Her03a] and it is not only driven by maximizing the size of the pattern database but it also seeks to tighten the lower bounds on each level. We have also shown that there are instances where pattern databases obtained by our greedy search could provide orders of magnitude tighter lower bounds than pattern databases of the same size which correspond to randomly chosen patterns.

6.3 Performance Analysis

The classical study of heuristics and search performance is due to Pearl [Pea84]. Korf *et al.* in [KR98, KRE01] criticize this and other previous analysis ([Poh77, Gas79]) which characterize heuristics in terms of their relative and absolute error and specifically in the context of the A* [HNR68] algorithm. If the error of a heuristic estimate (its difference from the true distance to the goal) is bounded by a constant then A*'s complexity is linear with respect to nodes expanded while if the error grows linearly with the true distance it will expand an exponential number of nodes. As Korf *et al.* point it out, the two problems with this analysis are the difficulty of estimating the heuristic's error and the A* assumption. A* with a monotone heuristic does not expand duplicates. On the other hand, memory limited algorithms, such as IDA* [Kor85b], DFBB [LW66], often expand a large number of duplicates.

The basis of the analysis of Korf *et al.* [KR98, KRE01] is the brute-force search tree (BFST). They estimate the number of nodes expanded on the last iteration of IDA* in the worst case. Considering only the last iteration is often appropriate since it dominates all previous iterations if the operators have the same cost. On the other hand, when there is a large number of different costs, an exponential number of iterations may be needed by IDA* to bump the cost bound to the cost of the goal. Holte *et al.* in [Hol03] also raise the issue that the number of iterations with IDA* even with uniform costs is very difficult to predict from the heuristic values. In particular, they show that a more accurate heuristic could result in more iterations than one which is less precise. We, however, believe that even with its limitations, estimating node expansions from the BFST offers much

insight in understanding heuristic search.

6.4 Applications

The two applications we considered in this thesis are finding macro-operators for the Rubik's Cube and solving the Sequential Ordering Problem. We also performed analysis with the 3×3 puzzle to gain insight into the performance of pattern databases. Finding optimal solutions for the $n \times n$ puzzle is shown to be NP-complete in [RW86]. Finding optimal solutions efficiently for the Rubik's Cube is still an open problem. The SOP is a special case of the TSP which is known to be NP-hard even in the Euclidean case [Pap77]. Therefore our approach of heuristic search is justified in attacking these problems. The $n \times n$ sliding tile puzzles and the Rubik's Cube have also been a driving force in improving and designing new search algorithms, [Kor85b, KT96, CS98, Sin81, Kor85c, Kor97, Her01] just to name a few.

6.4.1 Macro-operators

Adding macros to a space is often used in speed-up learning [FHN72, Iba85, Iba89]. This increases the branching factor and can result in the so-called utility problem [MS93] where there could be a very large number of operators applicable. Finkelstein and Markovitch [FM98] describe a framework that can be used to select high utility macro operators. In contrast, in Korf's work [Kor83a, Kor85c], the macro-operators totally replace the original operators and in the problems studied the utility problem does not apply. This is also our approach in this

thesis. In [Kor83b], Korf defines *serial decomposability* which is sufficient to guarantee that a set of macros can serialize a problem. Serializability refers to the existence of a subgoal ordering where once a subgoal is satisfied it does not need to be violated again. Traditionally macro operators were found by uninformed search. The stabilizer chain for permutation groups [DM96] can also be viewed as a macro table, where the generators of the subgroups are the macro-operators. In Computational Group Theory, the Schreier-Sims algorithm [Sim70, But91] is the most often used to build the stabilizer chain. Since the Rubik's Cube is a group, the algorithm can directly be adopted to build macro tables. The only modification needed is to also record the sequence in terms of the original generators corresponding to each generator of the subgroups. This would yield very long solutions. A greedy selection technique suggested by Osterlund [Ost] can reduce solution length substantially but these solutions are still much longer than the optimal solutions [Her01]. Osterlund also suggested a relatively expensive path post-processing technique that could reduce solution lengths to approximately that of the solutions found by Korf's macro tables [Kor85c]. Korf used bidirectional partial-match, an uninformed bidirectional breadth-first search, to find optimal macros. Due to the small memory in computers at the time, he could not find all optimal macros and calculated some by the composition technique suggested by Sims [Sim70].

6.4.2 The Sequential Ordering Problem

The SOP is a model for many real life applications, ranging from helicopter routing between oil rigs [PT91] to scheduling on-line stacker cranes in an automated

warehouse [Asc95]. Since it is an NP-hard optimization problem, in practice, greedy techniques are used to obtain feasible but usually suboptimal solutions. The HAS-SOP system of Gambardella *et al.* [GD00, GD97] is the most successful of these. It uses ant colony optimization as well as a novel tour construction technique based on the 3-opt edge exchange heuristic [LK73]. Optimal solutions to some instances of the SOP were obtained by Ascheuer *et al.* [AJR00] with a cutting plane technique. The SOP is modeled as an Integer Programming problem derived from the Asymmetric Traveling Salesman Problem (ATSP) polytope [Fis92, Fis95]. The challenge with the branch and cut approach is to find valid facet inducing inequalities that represent the additional precedence constraints [ALFES90]. Ascheuer *et al.* also employed heuristic tour constructions to derive actual solutions (choose edges of the tour for which the LP relaxation gives fractional values). Escudero *et al.* [EGM94] used a similar approach but the lower bounds were obtained by Lagrangian relaxation. State space search for constrained TSPs was first considered by Christofides, Mingozzi and Toth [CMT81] and later extended by Mingozzi, Bianco and Ricciardelli [MBR97]. The lower bounds used in the branch and bound are obtained as follows. The problem is formulated as a recursion in terms of the cost matrix, the state representation and a successor function. The abstractions (relaxations) are directly applied to the states and the relaxed recursion is solved by dynamic programming. The completed dynamic programming table corresponding to this relaxed recursion serves as a database of lower bounds. In fact, the q -path abstraction [CMT81] is very similar to our domain abstractions. The abstraction assigns integers q_i to each city and the abstract state is obtained by summing the q values in the state representation.

6.5 Using Memory to Speed-up Search

A* [HNR68] stores all nodes which have been reached during the search. Dynamic Programming [BD62] (for example, Dijkstra's Algorithm [Dij59]) uses memory to avoid recomputing the same recursion. Since the memory needs of these algorithms grow (usually exponentially) as problems get larger, there is a hardware imposed limit on the size of the problems they can solve. In contrast, the IDA* [Kor85b], DFBB [LW66], RBFS [Kor93] and other depth-first algorithms use memory proportional to search depth and therefore, with respect to memory at least, the size of the problem not an issue. In fact, there is a large amount of unused memory which could be used to speed-up search. MA* [CGAdS89] works like A* [HNR68] except when memory exceeds the given limit, the nodes with high f -values are removed from the open list (SMA* [Rus92] and ITS [GMN94] are much improved and simplified versions of MA*). Russell in [Rus92] performed memory and search speed trade-off experiments (similar to the ones in Chapter 3) and found that $t \approx m^{-1.33}$, where t is nodes expanded and m is the amount of memory given to SMA*. Kaindl *et al.* [KKLS95] investigate several strategies of using limited amount of memory during search. These include finding transpositions (eg. transposition tables [RM94]), ordering nodes (eg. the open list of A*), backing up more accurate heuristic values (eg. RBFS [Kor93], IE [Rus92] and MREC [SB89]) and storing search frontiers and perimeters (eg. BHPA [Poh71], perimeter search [DN94, Man95]). They compared node expansions and running time relative to that of IDA* and found that a bidirectional combination of A*-IDA* with transposition tables gave the best results. Since this study predates pattern databases, the best allocation of using a limited amount of memory still remains an open question. In our SOP experiments [Her03a, Her03b] we found

that some memory taken away from the pattern database and allocated to store transposition tables results in search speedup.

In this thesis we were not concerned with reducing memory to store pattern databases (except for perfect hashing), however, pattern databases can be compressed and sometimes this can even be achieved without loss of accuracy. In the original work of Culberson and Schaeffer [CS94] symmetries of the puzzle are used to fold the pattern databases. They also created a single higher performance pattern database which is obtained by taking values from other pattern databases. This latter technique is not guaranteed to lead to tighter lower bounds but it is shown to be very effective in practice by Newton and Holte [NH03].

Felner *et al.* [FMHK] show that lossy compressions are often very effective and the degradation in search performance is substantially smaller than the compression ratio. As a result one can store even larger pattern databases in the same amount of memory. In this thesis we did not consider compressions but we also have preliminary (unpublished) data suggesting that lossy compressions of large pattern databases can be very effective.

Chapter 7

Limitations

In this chapter, we examine limitations of our approaches to automatically generating high performance domain abstractions. First we discuss non-surjective state space homomorphisms, which could lead to unnecessarily large and poor quality pattern databases. Second we consider issues that could render some of the observations and techniques we studied less applicable. Finally, we touch upon open questions that are related to our work but we did not answer in the thesis.

7.1 Non-surjective Homomorphisms

We have encountered non-surjective state-space homomorphisms (Definition 18, page 27) with almost every problem we studied. We first discovered the phenomenon when domain abstractions resulted in larger abstract spaces for the Block's World planning problem [HH99a] than the size of the original space. In

the example presented in Chapter 2, the introduction of two blanks in the 2×2 puzzle lead to spurious states. The cause, however, in all cases of non-surjectivity is that the abstract space is calculated by a generator Δ_ϕ^{-1} instead of applying ϕ to the states of the original state space graph (as in STAR abstraction [HPZM96]). The condition for Δ_ϕ^{-1} to avoid non surjective homomorphisms for state space S , is

$$v \in \Delta(u) \iff \phi(u) \in \Delta_\phi^{-1}(\phi(v)) \quad (7.1)$$

where $u, v \in S$. To guarantee the connectivity requirement of a state-space homomorphism, only the implication

$$v \in \Delta(u) \implies \phi(u) \in \Delta_\phi^{-1}(\phi(v)) \quad (7.2)$$

must be satisfied. It is, however, not always straightforward to design Δ_ϕ^{-1} so it does not generate spurious states. In Chapter 5, the post processing that eliminates the spurious predecessors is pivotal to the algorithm's success. The best lower bound we were able to obtain without checking for spurious states for p43.4 was 56,645 which is significantly lower than the values obtained with checking (Table 5.1, page 129).

Next we give a description of a representation for the $n \times n$ puzzle where practically all domain abstractions lead to non-surjective abstractions. A state is represented by a vector which is indexed by the tile labels and the values correspond to the tile positions. We refer to this representation as the “dual” encoding. In our original representations, the positions are the indices and tile labels are the values. Figure 7.1 shows the indices we chose for the tile positions on the left and an actual state on the right. The state vector is composed of the state variables in this order

$$\left\langle \underbrace{x_1}_{\text{tile 1}}, \underbrace{x_2}_{\text{tile 2}}, \underbrace{x_3}_{\text{tile 3}}, \underbrace{x_4}_{\text{blank}} \right\rangle$$

1	2
3	4

indices

0	2
1	3

<3,2,4,1>

Figure 7.1: Dual Representation for the 2×2 Puzzle.

The state shown in Figure 7.1 is $\langle 3, 2, 4, 1 \rangle$. The operators that apply depend on the position of the blank, which in our dual representation, is the value of x_4 .

Consider the domain abstraction

$$\phi(x) = \begin{cases} 2, & \text{if } x = 2 \text{ or } x = 4 \\ 4, & \text{otherwise} \end{cases}$$

The operators in both the original and abstract spaces could be described as swapping the values of x_4 and x_i (where i is 1,2 or 3) when they have the settings shown below.

rule	Δ			Δ_ϕ^{-1}		
	x_4	x_i	action	x_4	x_i	action
#1	1	2	R	1	2	L
#2	3	4	R	3	2	L
#3	2	1	L	2	1	R
#4	4	3	L	2	3	R
#5	1	3	U	1	3	D
#6	2	4	U	2	2	D
#7	3	1	D	3	1	U
#8	4	2	D	2	2	U

The abstract operators 6 and 8 become irrelevant and can be deleted from Δ_ϕ^{-1} . The resulting state space is depicted in Figure 7.2. The operator numbers are indicated at the arrow tips. The ovals enclose the preimage states and the ones with dashed boundaries are the spurious states. It can easily be verified that every

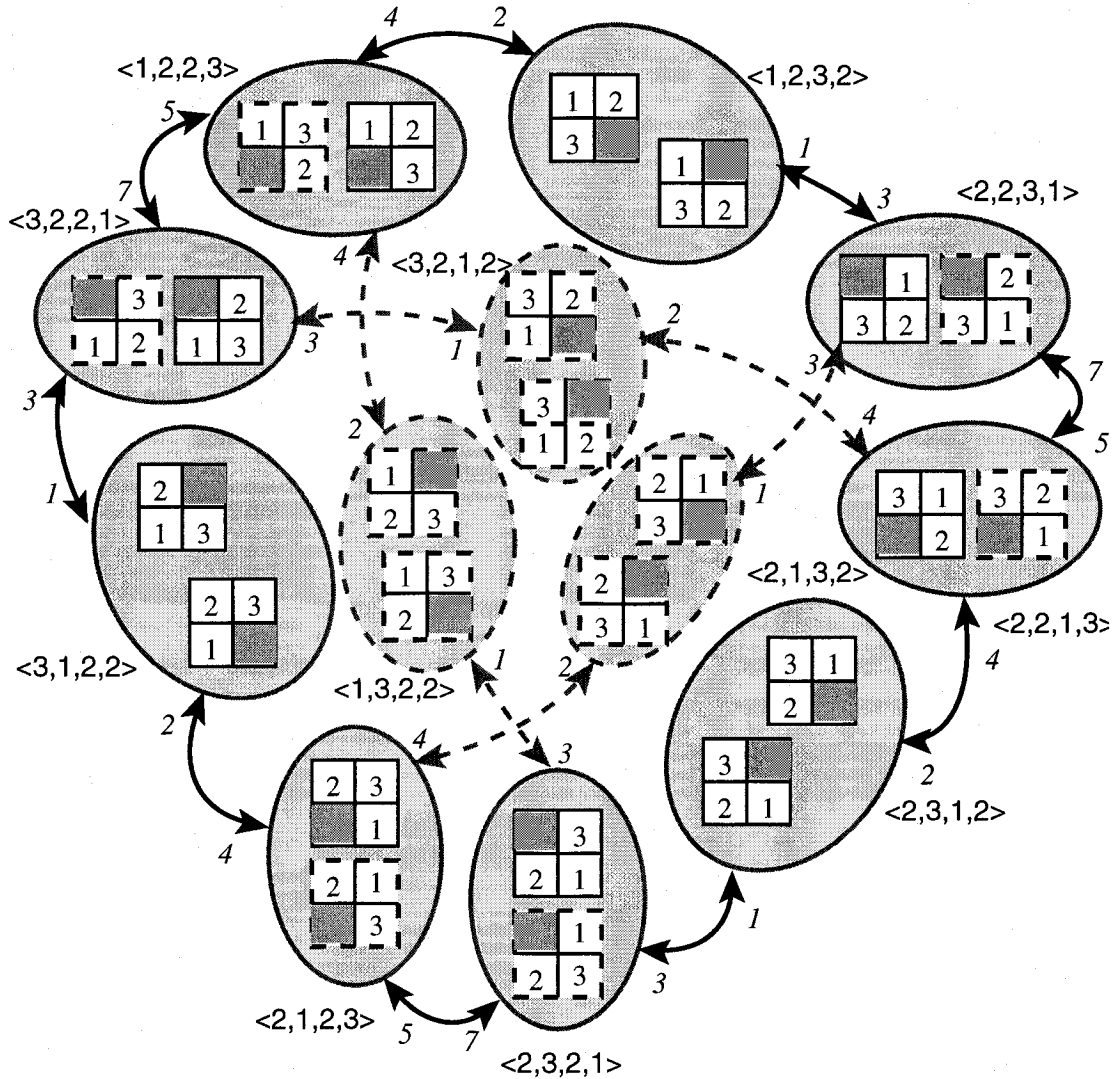


Figure 7.2: Abstract Space for the "Dual" Encoding of the 2×2 Puzzle.

domain abstraction with granularity $\langle 2 \rangle$ is non-surjective when Δ_ϕ^{-1} is obtained by simply relabeling the constants in the original operators. The abstract operators

which introduce spurious states are 1 and 7. In both cases the operators *could* correspond to moving the tile into the blank across a diagonal, which is illegal. On the other hand, the rules cannot be simply removed because the connectivity in the abstract space would not be preserved (some applications of these operators are required, as can be seen in Figure 7.2).

Whether a domain abstraction is surjective or not, it will induce a homomorphism and therefore our framework is general to derive lower bounds. Unfortunately non-surjective homomorphisms could be too large to be useful or they can result in unnecessarily inaccurate lower bounds. For example in Figure 7.2, the path between $\langle 2, 2, 3, 1 \rangle$ and $\langle 2, 3, 2, 1 \rangle$ is shortcut by the spurious state $\langle 2, 1, 3, 2 \rangle$. We do not have an automatic way to avoid non-surjective homomorphisms, but in this thesis we have shown that in some cases domain knowledge can be used to eliminate spurious states. In the case of the $n \times n$ puzzle it is sufficient that the domain abstraction preserves the uniqueness of the blank. In the case of the SOP, an efficient check can be performed. Nonetheless, the user should be aware of non-surjectivity.

7.2 Limitations in Approach

Pattern databases have been used to solve impressively large problems. One question is, how far can this approach scale up? Currently we can solve SOPs of 40-50 vertices optimally in reasonable time. Will computers have enough memory so we will be able to solve 100 vertex large SOPs with pattern databases? The state space corresponding to a 100 vertex SOP is approximately 10^{17} times

larger than the largest SOP we can currently solve and even with fast advancing technology we cannot expect that computer memory would grow on this scale. Using additive [KF02, FKH03] and multiple [NH03, HNF⁺04] pattern databases is often much more effective than using one single pattern database. However, we do not expect even these techniques to have enough power to solve much larger SOPs. In fact, this limitation is also true for the other state spaces we studied. The sliding tile puzzle is defined on an $n \times n$ grid, the generalized Rubik's Cube is made up of $n \times n \times n$ cubies (with $n = 3$ in the original puzzle) and the SOP is defined by an $n \times n$ cost matrix. The state spaces are exponential in n and unless memory grows at the same rate (which is highly unlikely) our technique cannot be used to solve our problems for much larger n . While this is slightly discouraging, the problems we deal with are *NP*-complete and many researchers believe that it will never be possible to solve these problems for substantially larger n efficiently. Despite this, optimal solutions are important. For one, they can be used to benchmark suboptimal algorithms. In the case of the SOP, small instances can still be models for realistic applications and judging from the often large gaps between the currently best known upper and lower bounds for relatively small unsolved instances [TSP] the best suboptimal heuristic techniques could still be far from optimal. In the case of the SOP our technique can also be used in an interesting local search. Since any segment in a feasible (but not necessarily optimal) solution is a SOP, we can choose consecutive segments of vertices and solve them optimally. We would perform this tour improvement after 3-opt and other cheaper local search techniques can no longer reduce the cost of the solution.

The ubiquitous way to build pattern databases is to expand the abstract state

space from the abstract goal state [CS98, Kor97, KF02, Her01, Her03a]. For the SOP (and in many optimization problems) there is always one unique goal state and therefore it is no limitation. On the other hand, can we search between any two states in the state space using the same pattern database? The answer to this question, in general, is certainly no. Interestingly, however, we can use the same pattern database to search between arbitrary states in the Rubik's Cube state space. Let g be the goal state for which the pattern database H_g is built in the Rubik's Cube state space. Suppose we change the goal state to be g_2 . Since all states in this state space can be expressed as permutations of one another, there exists a permutation p such that $p(g) = g_2$. For this problem, the operator closure from every state is also the same state space with the exact same cost structure, and therefore $H_g[p(s)]$ is a lower bound for a state s and goal state g_2 . Similarly, we can actually transform the whole macro table built for g (Chapter 4) to obtain a macro table for g_2 . On the other hand if the goal is replaced with a goal condition *cond* (a common case in planning problems) the pattern database approach is less likely to be applicable. If the number of states which satisfy *cond* is relatively small and can be computed explicitly, we can create a pseudo goal g which is connected to the states $cond(S)$ with 0 cost edges or in some cases (like the macro subgoal state spaces) we may be able to identify a state space where $cond(S)$ is unique. In general, however, pattern databases will not be reusable for problems where the goal state moves frequently.

In light of our results in Chapter 3, a straightforward approach to select good abstractions would be to generate a large number of domain abstractions which fill up memory, evaluate the statistics we suggested and pick the best. This "generate and pick" approach, however, is expensive. It maybe worth doing this if a large

number of problems are to be solved with the same pattern database over time. For our applications, the macro-table and the SOP, we believe that this approach would be ineffective. In the case of the SOP, we showed that randomly selected pattern databases are substantially less accurate than the ones we derived by local search in the space of domain abstractions (Table 5.1, page 129) and therefore the “generate and pick” approach would not even be applicable. Our insight and domain knowledge greatly improved the quality of the pattern databases in our applications.

Another inherent difficulty of our approach is that it requires the implementation of Δ_ϕ^{-1} , in addition to Δ . For problems where the action of the operators is to permute the values of state variables ($n \times n$ Puzzle and Rubik’s Cube), it is not an issue, because Δ_ϕ^{-1} is identical to Δ . In general, however, the implementations of Δ_ϕ^{-1} and Δ could be different (as it is the case for the SOP) and it may not be easy to automatically derive Δ_ϕ^{-1} from Δ . For example, our totally automatic system, PSVN, in general, cannot infer the unique inverse of an operator (Section 2.6). This could result in spurious states and therefore in less effective lower bounds.

7.3 Remaining Open Questions

We have shown a space-time trade-off and suggested statistics that were effective in identifying good pattern databases without search for a few state spaces¹. We cannot conclude from these experiments that these results hold for all state

¹we detailed the results of the 3×3 puzzle in this thesis and have published similar results for other state spaces in [HH99b] and in [HH04].

spaces and for other abstraction techniques as well. On the other hand, one can easily follow our methodology and perform the same type of experiments with a scaled down version of the problem to be solved. For example, we can reasonably expect that our conclusions drawn from the 3×3 puzzle experiments would also be valid for all $n \times n$ puzzles. Another limitation with our approach in this respect is that we took memory to be the size of the abstract space. With multiple [HNF⁺04, NH03] and additive [KF02, FKH03] pattern databases and with database compression techniques the trade-off could be quite different. Therefore the validity of our results for memory based heuristics in general is still an open question.

In this thesis we have studied pattern databases and shown that they are very effective for solving the problems we studied. On the other hand, the question of utilizing memory most effectively to speed up search is still open. Kaindl *et al.* in [KKLS95] compare a number of strategies, however, their study excludes using memory to store heuristics. The answer to the memory utilization question is probably even more complex in light of recent advancements, in particular additive [KF02, FKH03] and multiple [HNF⁺04, NH03] pattern databases. In the case of the SOP we have shown (Chapter 5) that using some memory taken away from the pattern database and use it to store transposition tables could be more effective. In this case, our approach was justified by the level structure of the state spaces which was also preserved in the abstract spaces (we traded the memory of the pattern database at the levels close to the start state). The best way of allocating memory to speed up search and even the features that play a role in making an informed decision are still open questions.

Chapter 8

Conclusions

8.1 Results and Reflections

The main contributions of this thesis were laid out in Chapters 3, 4 and 5. In Chapter 3 we experimentally established a space-time trade-off for the 3×3 puzzle. With very high correlation, $tm^\alpha \approx c$, where t is the number of nodes expanded, m is the size of the pattern database, c is some constant and α is a problem specific constant (in our experiments between 0.69 and 0.85). If α were higher than 1, then doubling memory results in fewer than half the nodes expanded. This trade-off was valid for both A* and IDA*. The result not only confirms the intuition that larger pattern databases result in faster searches, but also that if α is known (e.g. determined by performing experiments similar to ours), one can very accurately approximate the gains in increasing memory. The size of the abstract space, however, is not the only indicator of search performance. For the 3×3 puzzle there was a 2-3 fold difference in search performance between pattern

databases of the same size and often the best pattern database of a certain size outperformed an average one with substantially larger size. To account for this performance difference we performed experiments involving thousands of pattern databases and identified simple statistics which are effective in identifying high performance pattern databases without actual searches in the original state space. While our experiments were performed only with small state spaces (including the ones in [HH99b, HH04]) we believe that our conclusions are valid for scaled up versions of these spaces as well. At present, it is infeasible to repeat our experiments on a similar scale for larger spaces. We believe, however, that the number of heuristics involved in the experiments are sufficient to draw affirmative conclusions. As we pointed out in Chapter 7, state spaces could widely vary and while our results are informative, we cannot say with certainty that our results would be true for all problems. Therefore, we suggest to perform experiments similar to that of Chapter 3 with a scaled down version of the problem to be solved. In Chapters 4 and 5 we directly made use of the space-time trade-off by building the largest possible pattern databases that fit within our memory limits.

In Chapter 4 we showed that automatically derived domain abstractions are effective to search for optimal macro-operators. Traditionally macro-operators were found by uninformed search techniques. Since each subgoal represents a different search space, more than one heuristic is needed if heuristic search is to be used. Therefore searching for macro-operators was a natural candidate for putting automatic derivation of pattern databases to the test. Our heuristic to find good pattern databases was very simple. We greedily kept making the domain abstraction coarser until it resulted in a pattern database which fit in memory. We also have observed that for the last subgoals (which have the most invariants

and therefore are also associated with the largest state spaces and the longest searches) we can ensure that h_0 is relatively high which is expected to result in faster searches according to our experiments in Chapter 3. We also have made a number of novel enhancements. In previous work each state variable represented a single subgoal. Due to the faster searches achieved by using heuristic search, we were able to find macros for subgoals which are associated with more than one state variable. When the merge of subgoals resulted in an abundance of macros to find, we stored a pattern database instead of the macros to solve the subgoal with heuristic search. Subgoal merging resulted in substantial improvements in solution quality for the Rubik's Cube at reasonable additional cost. While the testbed was the Rubik's Cube, our improvements are generally applicable to build macro-tables for other state spaces. In particular, if there is a subgoal sequence seq for which a problem is serializable, the problem remains serializable for any subgoal sequence obtained by merging consecutive subgoals from seq . The only additional requirement is to be able to implement Δ_ϕ^{-1} which is relatively trivial for many benchmark problems to which macro-search has been applied.

In Chapter 5 we used pattern databases to solve the Sequential Ordering Problem. While Christofides *et al.* [CMT81] had used lower bounds stored in memory to search in the original state space for optimization problems for the first time, our approach is substantially different. There are still many relatively small instances of the SOP which are still unsolved [TSP] and therefore we decided to try to derive better lower and upper bounds and to solve unsolved instances for the first time. We were able to solve one unsolved instance which is called p43.4 in the TSPLIB. It is a SOP with a relatively large number of precedence constraints and it has only 44 vertices. We were also able to match the best lower

and upper bounds for some other modest size (up to 54 vertices) unsolved SOPs. We made a number of novel improvements which were all essential to achieve our results. Our first novel contribution is a heuristic guided local search in the space of domain abstractions which finds the domain abstractions which induce the pattern databases. This technique derives much better quality pattern databases than the best picked from random ones with the same number of entries. We also invented two novel search techniques which are derived from Depth-first Branch and Bound (DFBB). Bidirectional Depth-first Branch and Bound (BiDFBB) is a general search technique applicable for all problems which can be searched by DFBB in both directions. BiDFBB was designed to overcome the poor quality of initial feasible solutions. While one can often use local search to improve a suboptimal solution (like a 3-opt for the TSPs), BiDFBB may also be effective in obtaining good quality initial solutions when a local search is not available. Recursive Depth-first Branch and Bound (RecDFBB) takes advantage of the fact that a segment of a solution is a SOP itself and, since we can generate lower bounds automatically, we can solve the suffix as an independent problem with its own heuristics when the search corresponding to the subtree with the global lower bounds is ineffective. In fact, in most cases, the recursive call does not need to solve the subproblem but to prove that no solution exists that would improve on the best known solution. RecDFBB, while less general than BiDFBB, is applicable for most problems derived from TSPs and for problems which exhibit this recursive property.

8.2 Future Work

We have mentioned limitations and related open questions in Chapter 7. Out of these, we are mainly planning to continue research in the direction of using pattern databases to solve very constrained *NP*-hard optimization problems. These, in particular, include constrained Vehicle Routing, scheduling and larger Sequential Ordering problems. From private discussions with experts who solve real optimization problems we gathered that realistic problems are even more constrained than the usual benchmark side constraints commonly considered in the Operations Research literature. The common ones are precedence constraints and time windows. In reality, however, some of the constraints can be considered soft (can be violated with some penalty) or often disjunctive. For example, labour laws prohibit truckers to drive more than a certain number of hours without a certain amount of rest, to drive faster than the posted speed limits or carry more cargo than a specified capacity. On the other hand when looking at the actual records it is often the case that the constraints are violated¹. In Section 5.6 we have hinted that we can generate lower bounds for soft and arbitrary boolean expressions of precedence constraints but we did not evaluate them due to the lack of benchmark problems. Our next course of action is to obtain real problems which involve many different and potentially unusual constraints which make it hard to derive nice mathematical models. In our experiments with the SOP we found that our technique is more effective to solve the more constrained instances. This is exactly the opposite for the branch and cut solvers, which are better at solving the less constrained SOPs. Since we were told that realistic scheduling problems

¹In fact, we were told by two independent research teams that the actual schedules used in industry almost always violate the constraints described by the companies.

are often far more constrained than the benchmark problems, heuristic search can be more effective to derive optimal or high quality solutions than branch and cut with the Linear Programming relaxations. We believe it is likely to be the case that we can encode unusual constraints more easily into a successor generator than an expert formulating the same constraints as linear inequalities. Even if the inequalities can be derived, branch and cut will only be effective if there are also good separation and lifting procedures that take the new inequalities into account. If the additional constraints substantially reduce branching, we can hope to solve much larger SOPs or similar scheduling and optimization problems. We will also consider using multiple pattern databases, database compression and better local searches in the space of domain abstractions to derive more accurate lower bounds. In particular, we considered only a subset of domain abstractions for our local search (merging and splitting columns) because it made the implementation of Δ_ϕ^{-1} easy (and efficient). The full space of domain abstractions, however, is much larger and therefore by considering all possibilities, we can hope to build more accurate pattern databases with the same number of entries.

8.3 Final Word

In this thesis we have shown that automatically generated lower bounds can be effective in solving difficult search problems. While all domain abstractions can yield lower bounds, our aim was to consider the large space of domain abstractions to build high performance databases. First we examined the search performance of pattern databases by performing experiments which involved a large number of searches and domain abstractions. These experiments confirmed

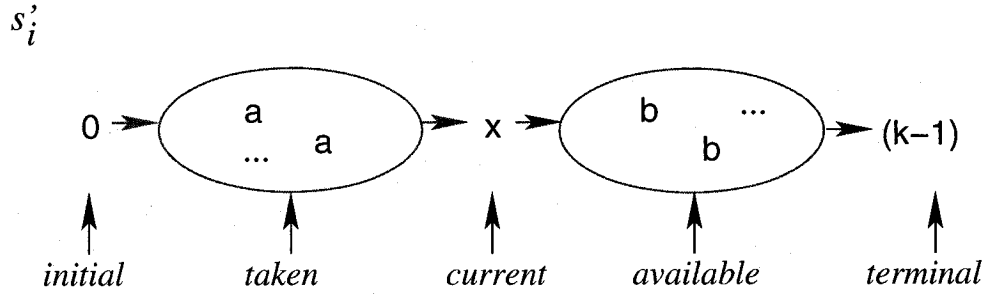
a very predictable space-time trade-off and revealed that there are simple statistics which can be used to identify good pattern databases. The two applications we considered to demonstrate our techniques were macro-search and the Sequential Ordering Problem. In macro-search there is a different search space for each subgoal and therefore the automatic generation of lower bounds is clearly preferable. In the case of the SOP, the derivation of tight lower bounds is still an open problem. For this application we performed local search in the space of domain abstractions to obtain high performance pattern databases. The lower bounds obtained by our greedy optimizer were far more accurate than the ones built from random abstractions, even when the pattern databases had the same size.

Appendix A

Checking for Spurious States

In this section we describe a check for spurious states after the predecessors of an abstract state have been generated by $\Delta_{\phi_{i+1}, \phi_i}^{-1}$ for the SOP. In particular, we check if an abstract state, s'_i , actually corresponds only to precedence violating partial tours. This can happen because $\Delta_{\phi_{i+1}, \phi_i}^{-1}$ uses the cost matrix $C_{i+1, i}$ to derive the predecessors. Since $C_{i+1, i}$ cannot preserve all the constraints of C (applying rules 3 and 4 could eliminate precedences, page 108), we will now check s'_i against the original precedences. We also have to do this efficiently, as this test is going to be performed for every abstract state. If we were to check all preimages for each abstract state separately, we would, in essence, expand the entire state space. We identified two necessary conditions which can be checked efficiently.

Consider the abstract state s'_i depicted in Figure A.1. Its representation is $s'_i = (... , b, b)[x]$. The corresponding partial tour must start at vertex 0, terminate in vertex $k - 1$ (when there are k abstract labels) and it is currently in vertex x . We also know that there are still two vertices with image b available. From $av(s'_i)$

Figure A.1: Abstract State s'_i .

we can also tell which vertices are taken, and suppose there are two vertices with label a already included in the partial tour. For s'_i to be feasible, (1) having two vertices with image a preceding a vertex with image x must be feasible and (2) two vertices with image b can come after a vertex with image x in a tour.

The maximum number of vertices with label a which can precede a vertex with label x is the number of non $\succ$ entries among the columns of C , where the row labels are the vertices that map to a and the column labels are the vertices that map to x . The columns are shown below.

$$C = \left(\begin{array}{c} \begin{array}{c} x \\ a \\ a \\ a \end{array} \quad \begin{array}{c} x \\ \succ \\ \dots \\ 3 \end{array} \quad \begin{array}{c} x \\ 4 \\ \dots \\ \succ \end{array} \quad \begin{array}{c} x \\ \times \\ \dots \\ 2 \end{array} \end{array} \right)$$

In the special case, when $a = x$, there can be one fewer a . This is a conservative measure, it is not always true that there could actually be these many a 's, however, it is certain that there cannot be more. Similarly, when two b 's are still available in s'_i with current vertex x , it is necessary that there are at least two non $\succ$ entries across the columns of C , where the column labels have image b and

the row labels have image x .

$$C = \begin{pmatrix} & b & b & b \\ x & \boxed{\succ} & \boxed{4} & \boxed{\times} \\ x & \boxed{\dots} & \boxed{\dots} & \boxed{\dots} \\ x & \boxed{3} & \boxed{\succ} & \boxed{2} \end{pmatrix}$$

To perform the check more efficiently, we precalculate the maximum number of a 's and b 's which can come before and after an x for all combinations of a , b and x . When the abstract space is expanded, the actual check for an abstract state s'_i with current vertex x is performed in $O(nk)$ time where n is the number of vertices and k is the number of preimage vertices of x .

Appendix B

Perfect Minimal Hash Function

Our perfect minimal hash function, I , over the abstract states plays many important roles. First, instead of a complicated hash table data structure, we can use a simple array of lower bounds H , where

$$h(s_i) = H[I_i(\phi_i(s_i))]$$

Not storing the abstract states and the fact that we do not need overflow data structures substantially reduce the amount of memory we need to store the pattern database. I_i is also used to determine the exact number of abstract states on level i . Therefore we can tell if our abstraction ϕ_i will fit in memory without expanding the level S'_i . The transposition tables on level i also use I_i as the index function.

Our I is not a closed form formula, but is obtained from dynamic programming tables.

I_i is a perfect minimal hash function over the states of $S'_i = \phi_i(S_i)$, not taking

precedences into account. This means that given a state on $s'_i \in S'_i$, $I_i(s'_i)$ returns a unique integer in the interval $[0, |S'_i| - 1]$, where $|S'_i|$ is the total number of states on S'_i including those which represent partial tours that violate precedence constraints.

Recall that s'_i is represented as a set of abstract labels corresponding to the vertices not in the partial tour and the label of the last vertex in the partial tour. The number of labels available in s'_i for an n vertex SOP is $n - i$. This includes the label $k - 1$ for the terminal vertex of the tour. Therefore we have $n - i - 1$ “wild card” labels.

$$s'_i = (\underbrace{\text{available labels}}_{n-i-1}, k-1)[v_c]$$

The $n - i - 1$ available labels are not unique since some labels could repeat. Suppose there are b different labels (excluding the labels 0 and $k - 1$ which are reserved for the labels of the start and terminal vertices). Therefore $b = k - 2$. To model this problem, imagine that we have b boxes. The box i , for $1 \leq i \leq b$, contains $|\phi_i^{-1}(i)|$ pebbles with the same colour. No two boxes have pebbles with identical colours and there is at least 1 pebble in each box. The labels of the abstract state correspond to the pebbles we pick from the boxes. In particular, we have to choose 1 pebble for the current vertex (v_c) and then another $n - i - 1$ pebbles from the remaining ones to represent the available vertex labels. To count the abstract states, we break down the problem into the these subproblems.

- choose 1 pebble from the $b = k - 2$ boxes
- choose $n - i - 1$ pebbles from the remaining ones

Subproblem 1 is clear. There are $k - 2$ choices. Subproblem 2 is much more

complicated. Our strategy is to enumerate the choices in lexicographic order. Let's consider a concrete example. We have 4 boxes ($b = 4$) and we have to choose 5 pebbles ($n - i - 1 = 5$). There are 3 pebbles in box 1, 2 pebbles in box 2, 5 pebbles in box 3 and 2 pebbles in box 4, as shown in Figure B.1. The choices

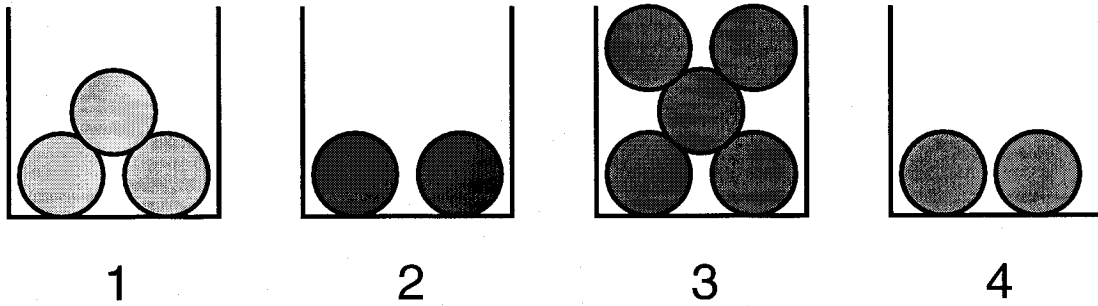


Figure B.1: Pebbles in Boxes.

in lexicographic order are shown in Table B.1. The entries correspond to how

Box						Box						Box						Box					
1	2	3	4	#		1	2	3	4	#		1	2	3	4	#		1	2	3	4	#	
0	0	3	2	0		0	2	3	0	8		1	2	1	1	16		2	2	0	1	24	
0	0	4	1	1		1	0	2	2	9		1	2	2	0	17		2	2	1	0	25	
0	0	5	0	2		1	0	3	1	10		2	0	1	2	18		3	0	0	2	26	
0	1	2	2	3		1	0	4	0	11		2	0	2	1	19		3	0	1	1	27	
0	1	3	1	4		1	1	1	2	12		2	0	3	0	20		3	0	2	0	28	
0	1	4	0	5		1	1	2	1	13		2	1	0	2	21		3	1	0	1	29	
0	2	1	2	6		1	1	3	0	14		2	1	1	1	22		3	1	1	0	30	
0	2	2	1	7		1	2	0	2	15		2	1	2	0	23		3	2	0	0	31	

Table B.1: Picking Pebbles in Lexicographic Order.

many pebbles were chosen from the box. For example, for the choice with #14, there is one pebble chosen from box 1, one pebble from box 2 and three pebbles

from box 3. The corresponding abstract state is

$$(1, 2, 3, 3, 3, 5)[v_c]$$

Label 5 is always available since it is the label of the terminal vertex, 0 is only available in the start state and v_c is 1, 2, 3 or 4. To assign the lexicographic order for a choice of pebbles we define a recurrence relation. Let $f(x, y)$ be the number of ways one can select y pebbles from the last boxes starting with box x . Box x contains n_x many pebbles. The recurrence is

$$f(x, y) = \sum_{i=\max(y-n_x, 0)}^y f(x+1, y-i)$$

with the initializations

$$f(x, y) = \begin{cases} 0, & \text{if } x = b \text{ and } y > n_x \\ 1, & \text{if } x = b \text{ and } y = n_x \\ 1, & \text{if } y = 0 \text{ (one way to select 0 pebbles)} \end{cases}$$

We can solve this recurrence with dynamic programming. The completed dynamic programming table T for our example is shown below.

			y						
	Boxes	n_x	x	0	1	2	3	4	5
$T =$	1, 2, 3, 4	3	1	1	4	10	18	26	32
	2, 3, 4	2	2	1	3	6	8	9	9
	3, 4	5	3	1	2	3	3	3	3
	4	2	4	1	1	1	0	0	0

The value $T[1][5] = 32$ is the total number of ways selecting 5 pebbles from the 4 boxes and, for example, $T[2][4] = 9$ is the number of ways one can pick 4 pebbles from boxes 2, 3 and 4.

Let vector $\mathbf{m}$ be a particular selection of pebbles from b boxes and let $\eta(\mathbf{m})$ be the index of the lexicographic order corresponding to the selection $\mathbf{m}$. For example, $\eta(\langle 2, 1, 2, 0 \rangle) = 23$ as shown in Table B.1.

$$\eta(\mathbf{m}) = \sum_{x=1}^{b-1} \left(\sum_{i=0}^{\mathbf{m}[x]-1} T[x+1][p(x)-i] \right) \quad (\text{B.1})$$

where $p(x)$ is the total number of pebbles chosen from the last boxes starting with box x

$$p(x) = \sum_{j=x}^b \mathbf{m}[j]$$

For the selection $\mathbf{m} = \langle 2, 1, 2, 0 \rangle$,

$$\eta(\mathbf{m}) = \underbrace{(T[2][5] + T[2][4])}_{x=1, p(x)=5, \mathbf{m}[1]=2} + \underbrace{T[3][3]}_{x=2, p(x)=3, \mathbf{m}[2]=1} + \underbrace{(T[4][2] + T[4][1])}_{x=3, p(2)=2, \mathbf{m}[3]=2} = 23$$

We need $k - 2$ such dynamic programming tables, one for each choice of v_c . We order them according to the label values of v_c . The hash value is obtained by calculating $\eta(\cdot)$ and offsetting this value by the maximum entries of the dynamic programming tables that precede the one labeled v_c .

Bibliography

- [ABCC98] D. Applegate, R. Bixby, Vasek Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume III:645–656, 1998.
- [AJR00] N. Ascheuer, M. Jünger, and G. Reinelt. A branch & cut algorithm for the asymmetric traveling salesman problem with precedence constraints. *Computational Optimization and Applications*, 17(1):61–84, 2000.
- [AJR01] N. Ascheuer, M. Jünger, and G. Reinelt. Solving the asymmetric traveling salesman problem with time windows by branch-and-cut. *Mathematical Programming*, 90(3):475–506, 2001.
- [ALFES90] N. Ascheuer, M. Grötschel L. F. Escudero, and M. Stoer. On identifying in polynomial time violated subtour elimination and precedence forcing constraints for the sequential ordering problem. In *Proceedings of the first Integer Programming and Combinatorial Optimization Conference, Waterloo, Canada*, pages 19–28, 1990.
- [Asc95] N. Ascheuer. *Hamiltonian path problems in the on-line optimization and scheduling of flexible manufacturing systems*. PhD thesis, Technical University of Berlin, 1995.
- [Bal65] E. Balas. An additive algorithm for linear programming in zero-one variables. *Operations Research*, 13:517 – 546, 1965.
- [Ban80] R. B. Banerji. *Artificial Intelligence: A Theoretical Approach*. North Holland, 1980.
- [BD62] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.
- [BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison & Wesley, 1998.

- [Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
- [But91] G. Butler. *Fundamental Algorithms for Permutation Groups*. Lecture Notes in Computer Science. Springer-Verlag, 1991.
- [CGAdS89] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(2):197–122, 1989.
- [CJMZ01] J. Cirasella, D. S. Johnson, L. A. McGeoch, and W. Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. *Algorithm Engineering and Experimentation (ALANEX)*, pages 32–59, 2001.
- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [CMT81] N. Christofides, A. Mingozzi, and P. Toth. State space relaxation procedures for the computation of bounds to routing problems. In *Networks*, volume 11, pages 145–164, 1981.
- [CS94] J. C. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta, 1994.
- [CS96] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, pages 402–416. Springer, 1996.
- [CS98] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(2):318 – 334, 1998.
- [Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [DM96] J. D. Dixon and B. Mortimer. *Permutation Groups*. Graduate Texts in Mathematics. Springer-Verlag, 1996.
- [DN94] J. F. Dillenburg and P. C. Nelson. Perimeter search. *Artificial Intelligence*, 65:165–178, 1994.
- [Ede01a] S. Edelkamp. Planning with pattern databases. *Proceedings of the 6th European Conference on Planning (ECP-01)*, pages 13–24, 2001.

- [Ede01b] Stefan Edelkamp. Prediction of regular search tree growth by spectral analysis. *KI 2001: Advances in Artificial Intelligence, Joint German/Austrian Conference on AI, Vienna, Austria, September 19-21, 2001*, pages 154–168, 2001.
- [Ede02] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS), Toulouse, France*, pages 274–283. AAAI, 2002.
- [EGM94] L. F. Escudero, M. Guignard, and K. Malik. A Lagrangian relax-and-cut approach for the sequential ordering problem with precedence constraints. In *Annals of Operations Research*, volume 50, pages 219–237, 1994.
- [EK98] S. Edelkamp and R. E. Korf. The branching factor of regular spaces. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 299–304, 1998.
- [FHN72] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251 – 288, 1972.
- [Fis81] M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27:1–18, 1981.
- [Fis92] M. Fischetti. Three facet-lifting theorems for the asymmetric traveling salesman polytope. In *Proceedings of the second Integer Programming and Combinatorial Optimization Conference*, pages 260–273, 1992.
- [Fis95] M. Fischetti. Clique tree inequalities define facets of the asymmetric traveling salesman polytope. *Discrete Applied Mathematics*, 56:9–18, 1995.
- [FKH03] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics, 2003. Submitted for publication.
- [FM98] L. Finkelstein and S. Markovitch. A selective macro-learning algorithm and its application to the $N \times N$ sliding-tile puzzle. *Journal of Artificial Intelligence Research*, 8:223–263, 1998.
- [FMHK] A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. To appear in AAAI-2004.

- [FN71] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [Gas79] J. Gaschnig. A problem similarity approach to devising heuristics: First results. *International Joint Conference on Artificial Intelligence*, pages 301–307, 1979.
- [GD97] L. M. Gambardella and M. Dorigo. HAS-SOP: Hybrid ant system for the sequential ordering problem. Technical Report IDSIA-11-97, IDSIA, 1, 1997.
- [GD00] L. M. Gambardella and M. Dorigo. An ant colony system hybridized with a new local search for the sequential ordering problem. *INFORMS Journal on Computing*, 12(3):237–255, 2000.
- [GMN94] S. Ghosh, A. Mahanti, and D. S. Nau. ITS: An efficient limited-memory heuristic tree search algorithm. In *Proceedings of twelfth National Conference of Artificial Intelligence, Seattle, USA*, pages 1353–1358, 1994.
- [Gre92] T. Gregorics. Megjegyzések a heurisztikus gráfkereső algoritmusokhoz. *Alkalmazott Matematikai Lapok*, 16:61–71, 1992.
- [GS79] G. Guida and M. Somalvico. A method for computing heuristics in problem solving. *Information Sciences*, 19:251–259, 1979.
- [Her00] I. T. Hernádvölgyi. Using pattern databases to find macro operators. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (Poster)*, page 1075, 200.
- [Her01] I. T. Hernádvölgyi. Searching for macro operators with automatically generated heuristics. *Advances in Artificial Intelligence - Proceedings of the Fourteenth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (LNAI 2056)*, pages 194–203, 2001.
- [Her03a] I. T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. In *Operations Research Proceedings, Selected Papers of the International Conference on Operations Research (OR 2003)*, pages 355 – 362, 2003.
- [Her03b] I. T. Hernádvölgyi. Solving the sequential ordering problem with automatically generated lower bounds. Technical report, University of Alberta, 2003. TR03-16.

- [HH99a] I. T. Hernádvölgyi and R. C. Holte. PSVN: A vector representation for production systems. Technical Report TR-99-04, School of Information Technology and Engineering, University of Ottawa, 1999.
- [HH99b] R. C. Holte and I. T. Hernádvölgyi. A space-time tradeoff for memory-based heuristics. *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 704–709, 1999.
- [HH00] I. Hernádvölgyi and R. Holte. Experiments with automatically created memory-based heuristics. *Abstraction, Reformulation and Approximation*, 1864:281–290, 2000.
- [HH04] I. T. Hernádvölgyi and R. C. Holte. Steps towards the automatic creation of search heuristics. Technical report, University of Alberta, 2004. TR04-02.
- [HK70] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138 – 1162, 1970.
- [HK71] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6 – 25, 1971.
- [HNF⁺04] R. Holte, J. Newton, A. Felner, R. Meshulam, and D. Furcy. Multiple pattern databases. In *14th International Conference on Automated Planning & Scheduling (ICAPS) (To appear)*, 2004.
- [HNR68] P.E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
- [Hol03] Robert C. Holte. Common misconceptions concerning Heuristic Search, 2003. Unpublished.
- [HPZM95] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. Technical Report TR-95-18, University of Ottawa, 1995.
- [HPZM96] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535, 1996.
- [HTK03] H. Hohwald, I. Thayer, and R. E. Korf. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, Acapulco, Mexico, pages 1239 – 1245, 2003.

- [Iba85] G. A. Iba. Learning by discovering macros in puzzle solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 744 – 749, 1985.
- [Iba89] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285 – 317, 1989.
- [JS99] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 570–575, 1999.
- [KF02] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1–2):9–22, 2002.
- [Kib82] D. Kibler. Natural generation of admissible heuristics. Technical Report TR-188, University of California at Irvine, July 1982.
- [KKLS95] H. Kaindl, G. Kainz, A. Leeb, and H. Smetana. How to use limited memory in heuristic search. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 236–242, 1995.
- [Kno91] C. Knoblock. Automatically generating abstractions for problem solving, 1991. PhD Thesis, CMU.
- [Kno94] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.
- [Koc] Herbert Kociemba. Two-phase algorithm to solve Rubik’s Cube. Unpublished.
- [Kor83a] R. E. Korf. *Learning to Solve Problems by Searching for Macro-Operators*. PhD thesis, Carnegie-Mellon University, 1983.
- [Kor83b] R. E. Korf. Operator decomposability: A new type of problem structure. *Proceedings of the National Conference on Artificial Intelligence (AAAI-83)*, pages 206–209, 1983.
- [Kor85a] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
- [Kor85b] R. E. Korf. Iterative-deepening A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 1034–1036, Los Angeles, California, 1985. Morgan Kaufmann.

- [Kor85c] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.
- [Kor93] R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
- [Kor97] R. E. Korf. Finding optimal solutions to Rubik’s Cube using pattern databases. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, 1997.
- [KR98] R. E. Korf and M. Reid. Complexity analysis of admissible heuristic search. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 305–310, 1998.
- [KRE01] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129:199–218, 2001.
- [KT96] R. E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1202–1207, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.
- [Kuh55] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.
- [LK73] S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.
- [LW66] E. Lawler and D. Wood. Branch and bound methods: A survey. *Operations Research*, 14:699–719, 1966.
- [Man95] G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.
- [MBR97] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations Research*, 45:365–377, 1997.
- [MP89] J. Mostow and A. Prieditis. Discovering admissible heuristics by abstracting and optimizing: A transformational approach. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 701–707, 1989.
- [MS93] S. Markovitch and P. D. Scott. The role of forgetting in learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 113 – 151, 1993.

- [NH03] J. Newton and R. C. Holte. Multiple pattern databases. Technical Report TR03-10, University of Alberta, 2003.
- [NS61] A. Newell and H. Simon. GPS, a program that simulates human thought. *Lernende Automaten*, 1961.
- [OHY92] A. Mayer O. Hansson and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63:207–227, 1992.
- [Ost] P. Osterlund. Private communication.
- [Pap77] C. H. Papadimitriou. Euclidean TSP is NP-complete. *Theoretical Computer Science*, 4:237–244, 1977.
- [Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley, 1984.
- [Poh71] I. Pohl. Bi-directional search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 6*, pages 127–140. Edinburgh University Press, Edinburgh, Scotland, 1971.
- [Poh73] I. Pohl. Genuine dynamic weighting and computational issues in heuristic problem solving. In *In Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73)*, volume 3, pages 12–17, Stanford, California, 1973.
- [Poh77] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. In E. W. Elcock and Donald Michie, editors, *Machine Intelligence 8*, pages 55–72. Ellis Horwood, Chichester, England, 1977.
- [Pri90] A. E. Prieditis. *Discovering Effective Admissible Heuristics by Abstraction and Speedup: A Transformational Approach*. PhD thesis, Rutgers University, 1990.
- [Pri93] A. E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.
- [PT91] W. Pulleyblank and M. Timlin. Precedence constrained routing and helicopter scheduling: Heuristic design. Technical Report RC17154, IBM, 1991.
- [RM94] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:701–709, 1994.

- [Rus92] S. Russell. Efficient memory-bounded search methods. *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 1–5, 1992.
- [RW86] D. Ratner and M. Warmuth. Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable. *Proceedings of the National Conference on Artificial Intelligence AAAI*, pages 168–172, 1986.
- [Sac74] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.
- [SB89] A. K. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 297 – 302, 1989.
- [Sim70] C. C. Sims. Computational methods in the study of permutation groups. *Computational Problems in Abstract Algebra*, pages 169–183, 1970.
- [Sin81] D. Singmaster. *Notes on Rubik's 'Magic Cube'*. Enslow Publishers, 1981.
- [SM03] Dong-Il Seo and Byung-Ro Moon. A hybrid genetic algorithm based on complete graph representation for the sequential ordering problem. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 669–680. Springer-Verlag, 2003.
- [TSP] TSPLIB unsolved and challenging instances of various Traveling Salesman Problems. <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.
- [Val84] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, pages 47–59, 1984.
- [Wol80] L. Wolsey. Heuristic analysis, linear programming and branch and bound. *Mathematical Programming Study*, 13:121–134, 1980.
- [Zha93] W. Zhang. Truncated branch and bound: A case study on the asymmetric TSP. In *Working Note of AAAI Spring Symposium: AI and NP-hard Problems*, pages 160–166, 1993.
- [Zha03] W. Zhang. Phase transitions of the asymmetric traveling salesman. In *18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1202 – 1207, 2003.

- [ZK96] W. Zhang and R. E. Korf. A study of complexity transitions on the asymmetric traveling salesman problem. *Artificial Intelligence*, 81:223–239, 1996.