University of Alberta

SIMULTANEOUSLY SEARCHING WITH MULTIPLE ALGORITHM SETTINGS: AN ALTERNATIVE TO PARAMETER TUNING FOR SUBOPTIMAL SINGLE-AGENT SEARCH

by

Richard Valenzano

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

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

Jonathan Schaeffer, Computing Science

Nathan Sturtevant, Computing Science

Andy Liu, Mathematical and Statistical Sciences

Vadim Bulitko, Computing Science

Abstract

Many single-agent search algorithms have parameters that need to be tuned to get the best performance. Typically, the parameters are tuned offline, resulting in a generic setting that is supposed to be effective on all problem instances. However, though the settings found by tuning will exhibit strong average performance over the test set, it can be shown that parameter settings that are problem-instance specific can result in substantially reduced search effort. We consider the use of dovetailing as a way to deal with this issue. Dovetailing is a procedure that performs search with multiple parameter settings simultaneously. In this thesis, we present results testing the use of dovetailing with the weighted A*, weighted IDA*, weighted RBFS, and BULB algorithms on the sliding tile and pancake puzzle domains. Dovetailing will be shown to significantly improve weighted IDA* with which it commonly improves run-time by several orders of magnitude. It will also generally enhance the performance of weighted RBFS. In the case of weighted A* and BULB, dovetailing will be shown to be ineffective when used with these algorithms. Dovetailing is also trivially parallelizable and we will demonstrate that the use of this procedure decreases the search time in all considered domains.

Acknowledgements

I would like to thank my supervisors, Jonathan Schaeffer and Nathan Sturtevant, for providing me with guidance and wisdom while we were undertaking this project. Their support was both invaluable in its contribution to this thesis, but also in my development as a researcher and scientist. I am also grateful for the advice and information I received during my discussions with Ariel Felner, Holger Hoos, and Akihiro Kishimoto, and the committee for their helpful comments and feedback. I am also glad to have had the opportunity to collaborate with Karen Buro on some early work on the statistical analysis of search domains, and thank her very much for her contributions.

Finally, I would also like to thank my family and friends for their love and support while I was working on this research. Whenever I stumbled, there was always someone around to help pick me up, and for that, I am eternally grateful.

Table of Contents

1 Introduction **1**

2 Background **4**
2.1 Formalization of Single-Agent Search . . . 4
2.2 Search Terminology . . . 6
2.3 The Best-First Search Algorithm . . . 8
2.4 The Iterative Deepening A* (IDA*) Algorithm . . . 9
2.5 The Recursive Best-First Search (RBFS) Algorithm . . . 10
2.6 Weighted Heuristics . . . 13
2.7 Test Domains . . . 14
2.7.1 Sliding Tile Puzzle . . . 14
2.7.2 Pancake Puzzle . . . 15
2.8 Weighted A* (WA*) . . . 17
2.8.1 Bound on Solution Quality . . . 17
2.8.2 WA* in the Sliding Tile and Pancake Puzzles . . . 21
2.9 Weighted IDA* (WIDA*) . . . 22
2.9.1 Bound on Solution Quality . . . 22
2.9.2 Effect of w on the Iterations of WIDA* . . . 23
2.9.3 Deficiencies of WIDA* . . . 25
2.9.4 WIDA* in the Sliding Tile and Pancake Puzzles . . . 26
2.10 Weighted RBFS (WRBFS) . . . 27
2.10.1 WRBFS in the Sliding Tile and Pancake Puzzles . . . 27
2.11 Beam Search and BULB . . . 28
2.11.1 Beam Search . . . 28
2.11.2 Beam Search with Limited Discrepancy Backtracking (BULB) . . . 29
2.11.3 Properties of BULB . . . 31
2.11.4 BULB and the Sliding Tile and Pancake Puzzles . . . 31
2.12 Chapter Summary . . . 32

3 Configuration Selection and Dovetailing **34**
3.1 Batch Tuning . . . 35
3.2 Deficiencies in Batch Tuning . . . 35
3.3 Per-Instance Tuning . . . 37
3.3.1 Issues with Per-Instance Tuning . . . 38
3.4 Dovetailing for Single-Agent Search . . . 40
3.4.1 Dovetailing and Memory . . . 41
3.4.2 Dovetailing and Diversity . . . 41
3.5 Related Work . . . 42
3.6 Chapter Summary . . . 43

4 Dovetailing Over the Main Parameter Spaces of Search Algorithms **44**
4.1 Experimental Design . . . 44
4.2 Dovetailing over Weights in WA* . . . 47
4.2.1 Dovetailing over Weights with WA* on the Sliding Tile Puzzle . . . 48
4.2.2 Parallel Dovetailing over Weights with WA* on the Sliding Tile Puzzle . . . 50
4.2.3 Dovetailing over Weights with WA* on the Pancake Puzzle . . . 53
4.3 Dovetailing and WIDA* . . . 55
4.3.1 Dovetailing over Weights with WIDA* on the Sliding Tile Puzzle . . . 56
4.3.2 Dovetailing over Weights with WIDA* on the Pancake Puzzle . . . 60
4.4 Dovetailing and WRBFS . . . 66
4.4.1 Dovetailing over Weights with WRBFS on the Sliding Tile Puzzle . . . 66

4.4.2 Dovetailing over Weights with WRBFS on the Pancake Puzzle 69
4.5 Dovetailing over Beam Widths in BULB 72
4.5.1 Dovetailing over Weights with BULB on the Sliding Tile Puzzle 72
4.5.2 Dovetailing over Weights with BULB on the Pancake Puzzle 75
4.6 Chapter Summary 78

5 Dovetailing Over Operator Orderings **79**
5.1 Dovetailing over Operator Orderings in WA* 81
5.1.1 Comparing Parallel Dovetailing over Operator Orderings with WA* to wPBNF 83
5.2 Dovetailing over Operator Orderings in WIDA* 85
5.3 Dovetailing over Operator Orderings in WRBFS 88
5.4 Dovetailing For Optimal Problem Solving 91
5.5 Dovetailing over Operator Orderings in BULB 94
5.6 Chapter Summary 96

6 Conclusion **97**
6.1 Removing Duplicates From Candidate Sets 99
6.2 Analyzing the Distribution of Work Performed by a Starting Configuration Set . . . 100
6.3 Finding Effective Candidate Set Sizes 101
6.4 Finding Effective Candidate Sets of Size k 103
6.5 Contributions and Closing Remarks 106

Bibliography **108**

List of Tables

4.1 The Number of Nodes Expanded by WIDA* with 4 Configurations on 3 4×4 Sliding Tile Puzzles. 45
4.2 The Speedup of wPBNF and the Average Speedup of Parallel Dovetailing on 43 4×4 Sliding Tile Puzzle Problems. 51

5.1 The Speedup of wPBNF and the Average Speedup of Parallel Dovetailing over Operator Orderings on 43 4×4 Sliding Tile Puzzle Problems. 84
5.2 Dovetailing over Operator Ordering in WIDA* Compared to the Best Single Order on the 5×5 Sliding Tile Puzzle. 86
5.3 Dovetailing over Operator Ordering in WIDA* Compared to the Best Single Order on the 16 Pancake Puzzle. 87
5.4 Dovetailing over Operator Ordering in WRBFS Compared to the Best Single Order on the 4×5 Sliding Tile Puzzle. 89
5.5 Dovetailing over Operator Ordering in WRBFS Compared to the Best Single Order on the 16 Pancake Puzzle. 90
5.6 Dovetailing over Operator Ordering for Optimal Problem-Solving. 92

6.1 A Summary of Speedup Results for Dovetailing Over the Main Parameter Space of Algorithms. 97
6.2 A Summary of Speedup Results for Dovetailing Over Operator Orderings. 98

List of Figures

2.1 The Best-First Search Algorithm. 8
2.2 The Iterative Deepening A* Search Algorithm. 9
2.3 Tree with non-monotonic cost function. 11
2.4 The Recursive Best-First Search Algorithm. 12
2.5 Example State in the 3×3 Puzzle and the Neighbours of this State. 14
2.6 The Goal State for the 3×3 Puzzle. 15
2.7 Example State in the 4 Pancake Puzzle and the Neighbours of this State. 16
2.8 Example Domain. 18
2.9 WA* on $1,000$ 4x4 Sliding Tile Problems 21
2.10 WA* on $1,000$ 14 Pancake Tile Problems 22
2.11 Different Search Trees for Different Weights. 25
2.12 WIDA* on $1,000$ 4x4 Sliding Tile Problems 26
2.13 WIDA* on $1,000$ 14 Pancake Problems 26
2.14 WRBFS on $1,000$ 4x4 Sliding Tile Problems 27
2.15 WRBFS on $1,000$ 14 Pancake Tile Problems 28
2.16 The BULB Algorithm. 30
2.17 BULB on $1,000$ 4×4 Sliding Tile Problems. 31
2.18 BULB on $1,000$ 14 Pancake Sliding Tile Problems. 32

3.1 A Comparison of the Configuration with the Best Average Performance over 100 4x4 Sliding Tile Problems to an Oracle. 37
3.2 The number of nodes expanded by WIDA* on two 15-puzzle problem when using weights 2 through 25 incremented by 0.25. 39
3.3 The number of nodes expanded by WRBFS on two 15-puzzle problem when using weights 2 through 25 incremented by 0.25. 39

4.1 Dovetailing over Weights in WA* on the 4×4 Sliding Tile Puzzle. 49
4.2 WA* on 1000 4×5 Sliding Tile Puzzles 49
4.3 Dovetailing over Weights in WA* on the 4×5 Sliding Tile Puzzle. 50
4.4 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WA* on the 4×5 Sliding Tile Puzzle. 52
4.5 Dovetailing over Weights in WA* in the 14 Pancake Puzzle. 53
4.6 WA* on $1,000$ 16 Pancake Puzzles 54
4.7 Dovetailing over Weights in WA* on the 16 Pancake Puzzle. 54
4.8 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WA* on the 16 Pancake Puzzle. 55
4.9 WIDA* on $1,000$ 4×5 Sliding Tile Puzzles 56
4.10 Dovetailing over Weights in WIDA* on the 4×5 Sliding Tile Puzzle. 56
4.11 WIDA* on $1,000$ 5×5 Sliding Tile Puzzles 57
4.12 Dovetailing over Weights in WIDA* on the 5×5 Sliding Tile Puzzle. 58
4.13 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WIDA* on the 5×5 Sliding Tile Puzzle. 58
4.14 Dovetailing over Weights in WIDA* on the 6×6 Sliding Tile Puzzle. 59
4.15 The Performance of Different Starting Configuration Sets on the 4×5 Sliding Tile Puzzle. 61
4.16 Dovetailing over Weights in WIDA* on the on the 14 Pancake Puzzle. 62
4.17 WIDA* on $1,000$ 16 Pancake Puzzles. 62
4.18 Dovetailing over Weights in WIDA* on the on the 16 Pancake Puzzle. 63
4.19 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WIDA* on the 16 Pancake Puzzle. 63
4.20 WIDA* with 3 Different Heuristics on $1,000$ 16 Pancake Puzzles. 65

4.21 The Ratio of the Average Performance of Dovetailing over Weights in WIDA* to the Single Best Weight with 3 Different Heuristics on the 16 Pancake Puzzle. . . . 66
4.22 Dovetailing over Weights in RBFS on the 4×4 Sliding Tile Puzzle. 67
4.23 WRBFS on $1,000$ 4×5 Sliding Tile Puzzles. 67
4.24 Dovetailing over Weights in RBFS on the 4×5 Sliding Tile Puzzle. 68
4.25 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WRBFS on the 4×5 Sliding Tile Puzzle. 69
4.26 Dovetailing over Weights in RBFS on the 14 Pancake Puzzle. 70
4.27 WRBFS on 1000 16 Pancake Tile Puzzles. 70
4.28 Dovetailing over Weights in RBFS on the 16 Pancake Puzzle. 71
4.29 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WRBFS on the 16 Pancake Puzzle. 71
4.30 BULB on $1,000$ 5×5 Sliding Tile Puzzles. 73
4.31 Dovetailing over Weights in BULB on the 5×5 Sliding Tile Puzzle. 73
4.32 BULB on $1,000$ 6×6 Sliding Tile Puzzles. 74
4.33 Dovetailing over Weights in BULB on the 6×6 Sliding Tile Puzzle. 74
4.34 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in BULB on the 6×6 Sliding Tile Puzzle. 75
4.35 Dovetailing over Weights in BULB on the 14 Pancake Puzzle. 75
4.36 BULB on $1,000$ 16 Pancake Puzzles. 76
4.37 Dovetailing over Weights in BULB on the 16 Pancake Puzzle. 77
4.38 The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in BULB on the 16 Pancake Puzzle. 77

5.1 Dovetailing over Operator Ordering in WA* on the 4×5 Sliding Tile Puzzle. . . . 82
5.2 Dovetailing over Operator Ordering in WA* on the 16 Pancake Puzzle. 82
5.3 Dovetailing over Operator Ordering in WIDA* on the 5×5 Sliding Tile Puzzle. . . 85
5.4 Dovetailing over Operator Ordering in WIDA* on the 16 Pancake Puzzle. 87
5.5 Dovetailing over Operator Ordering in WRBFS on the 4×5 Sliding Tile Puzzle. . 89
5.6 Dovetailing over Operator Ordering in WRBFS on the 16 Pancake Puzzle. 90
5.7 Dovetailing over Operator Ordering in BULB on the 6×6 Sliding Tile Puzzle. . . 95
5.8 Dovetailing over Operator Ordering in BULB on the 16 Pancake Puzzle. 95

6.1 Log-Log Plot of the Cumulative Distribution of Work over WIDA* Configurations Subtracted From 1 on a 4×4 Sliding Tile Puzzle, With and Without Dovetailing. . 101
6.2 A Comparison of Regular Dovetailing and UCB1 Dovetailing on the 5×5 Sliding Tile Puzzle. 106

Chapter 1

Introduction

Consider the problem of finding a route from one location in a city to another when navigating with a map. Before any travel can begin, a route for travel (or at least a partial route) must be found. Unfortunately, there may be a large number of candidate partial paths that must be considered before a complete route is found.

In this example, a single agent has the task of path finding. In general, all agents — whether they be living or artificial — are faced with some number of tasks to perform. In order to complete these tasks, agents must develop plans for action.

The effectiveness with which an agent plans is evaluated in terms of two different measures. The first is the time required to find a plan that will complete the given tasks. The second is the cost of the plan, the metric for which will be a function of the agent's objectives. In the case of navigation, the possibilities for plan cost include distance travelled if the agent is to find a short path, or the expected travel time if the agent is to find a quick path (*ie.* when navigating in a map, such an agent would prefer the use of highways over city streets).

In many domains, there is additional information that can be used to inform and thereby speed up planning. For example, when navigating between locations in Edmonton, Canada, it is reasonable to initially disregard routes through distant locations such as Madagascar. Moreover, if the destination is east of the initial location, the first routes to consider are naturally those that initially proceed eastward (if such paths exist).

Such information can be used to build *heuristics* which estimate the cost of the remaining path to the goal from any area in the domain. The field concerned with the development of heuristics and the construction of algorithms that use heuristics for planning is called *single-agent search*. In this field, planning is performed with a heuristic guided search in the space of candidate partial plans.

Single-agent search remains an important field for research due to the large number of real-world applications in which search algorithms have proven to be effective problem-solving techniques. These applications include autonomous robot navigation, in which heuristics are used to guide pathfinding [42]; DNA sequence alignment [33]; and, computer games [43].

There is a class of single-agent search algorithms that will provably find the lowest cost solution

provided that the heuristic value of any area of the domain is guaranteed to never overestimate the true distance from that part of the domain to the goal. Unfortunately, as problem domains grow larger, these optimal algorithms will often take too long to find a solution. To address this issue, suboptimal single-agent search algorithms have been developed which sacrifice solution quality for a decrease in search time. These algorithms are ideal when a solution (often near optimal) is needed quickly.

When constructing a single-agent search system, it must first be determined if suboptimal solutions will suffice or if optimal solutions are desired. Another consideration is if planning will precede execution or if the system is to work in real-time, with planning and execution being interleaved. The system designer is then faced with decisions regarding the proper selection of an algorithm and a heuristic function for the given domain(s). There are also often subtle choices such as tie-breaking (the order in which equally promising candidate paths are considered) that can similarly affect search speed. With all of these possible choices, properly recognizing and evaluating all of the necessary design decisions is a vital aspect of building an effective search system.

In the case of suboptimal problem solving, additional options arise as almost all applicable algorithms involve some kind of parameterization. For example, in the weighted variants of A*, IDA* [30], and RBFS [31], the value of the weight parameter must be set. Similarly, beam-search variants like BULB [18] and Beam-stack search [45] require the selection of a beam width. Parameterization also occurs in the class of best-first search variants like KBFS [16], MSC-WA* [27], and MSC-KWA* [19] in which a system designer can adjust the number of nodes expanded in parallel, the number of nodes to commit to, and the combination of these two ideas, respectively.

Any adjustment of these parameter values can change both the solution quality and the search speed. In a few cases, there are theoretical results that indicate how changing a parameter will affect the search (such as the bounds on solution quality in weighted A*, weighted IDA*, and weighted RBFS [31]). Unfortunately, it is much more common that a significant amount of pre-computation is needed so as to determine the relationship between a parameter and these two metrics. In practice, parameter values are tested offline on a set of training examples through a process commonly referred to as *parameter tuning*. The single parameter setting that satisfies any given constraints on solution quality and exhibits the best average performance is then used in all future searches.

Parameter tuning customizes a search algorithm for each specific problem domain. As such, the results of this expensive process cannot effectively be transferred across domains — a fact that is of particular concern when designing general search systems such as automated heuristic search planners. In these systems, general heuristics are used to guide a suboptimal search algorithm. In general, planners such as HSP [4] commit to a single parameter value that will hopefully be effective over a diverse class of problems.

Parameter tuning also suffers from another deficiency: there is no guarantee that a tuned value will perform well on each individual problem. Tuning only finds the setting that has the best average

performance on the training set. On a per problem basis, there can be other parameter values which significantly outperform the tuned setting. As we will demonstrate in Section 3.2, it is often the case that if a search system could properly select the correct parameter setting for each problem, the planning speed over a number of problems can be greatly improved.

In this end, we will consider the *dovetailing* procedure as an approach to this problem. Dovetailing involves running several independent instances of an algorithm — each of which has a different set of parameter settings — at the same time by interleaving the execution of the instances. The procedure is also trivially parallelizable. This aspect of the algorithm is important due to the increasing availability of multi-processor machines. As such, we will also analyze the performance of the algorithm when used in this fashion.

The main contributions of this thesis can be summarized as follows:

1. The single-agent search algorithms of WA*, WIDA*, WRBFS, and BULB are described, and the weaknesses of WA*, WIDA*, and WRBFS as suboptimal search algorithms are demonstrated. The behaviour of these algorithms in the sliding tile and pancake puzzles as a function of the weight parameter will also be shown. Similarly, the behaviour of BULB in these domain as a function of the beam width will be demonstrated.

2. Dovetailing is explored as an approach to proper parameter selection. Dovetailing is shown to significantly enhance WIDA* in the domains of the sliding tile puzzle and the pancake puzzle. The parallel version of dovetailing is also shown to exhibit massive improvements in search time when used with this algorithm. In the case of WRBFS, dovetailing without any pre-computation is shown to improve the speed of the algorithm in the sliding tile puzzle domain, and offer comparable performance in the pancake puzzle domain. Dovetailing with WRBFS over operator orderings will also improve the quality of the solutions. We will also demonstrate that parallel dovetailing offers an effective parallelization of this algorithm.

3. The sequential version of dovetailing will be shown to decrease the search speed of BULB and WA*, but improve the quality of the solutions when considered over operator orderings. The reasons for this behaviour are discussed. The parallel version of dovetailing will also be shown to offer the best performance of all known parallelizations of WA* when higher weights are used in the sliding tile puzzle domain.

Chapter 2

Background

In Chapter 1, the idea of single-agent search was introduced through the use of the navigation example. In this chapter, the single-agent search problem is formalized. The optimal algorithms A*, IDA*, and RBFS are then introduced along with the notion of weighted heuristics. Two problem domains, the sliding tile puzzle and the pancake puzzle, are then described, and the behaviour of the weighted algorithms WA*, WIDA*, and WRBFS are demonstrated in these domains. Additional properties of these algorithms are also considered. The chapter then concludes with a description of the beam-search variant BULB and some experimental results for this algorithm in the aforementioned domains.

2.1 Formalization of Single-Agent Search

Underlying every single-agent search problem is a *graph*, the definition of which is given below.

Definition 2.1.1. A graph G is defined by two sets: the vertex set V and the edge set $E \subset V \times V$. The graph given by a vertex set V and edge set E will often be denoted as $G(V, E)$.

A graph $G(V, E)$ is said to be *undirected* if for every pair v_1,$v_2 \in V$, $(v_1, v_2) \in E \Leftrightarrow (v_2, v_1) \in E$. If this condition does not hold, the graph is said to be *directed*. Each edge e — whether it be directed or undirected — will have an associated *cost*, denoted $c(e) \in \mathbb{R}$ or $c(u, w)$ where $e = (u, w)$. In single-agent search, the cost of any edge e is assumed to be larger than some real-valued constant $\epsilon > 0$. In the case of undirected graphs the cost of an edge is the same regardless of in which direction it is traversed.

While the techniques considered in this thesis can be applied to either directed and undirected graphs, the experiments will only be performed on domains that correspond to undirected graphs with a finite vertex set such that the edge set does not contain edges of the kind (v, v). As such, an edge (v_1, v_2) will refer to both (v_1, v_2) and (v_2, v_1). If $(v_1, v_2) \in E$, v_1 will be said to be *adjacent* to v_2 or a *neighbour* of v_2.

An ordered sequence of vertices $P = \{v_1, ..., v_k\}$ is called a *path* from v_1 to v_k in a graph $G(V, E)$ if every pair of consecutive vertices in P is adjacent. The cost of P, denoted $C(P) \in \mathbb{R}$,

will be the sum of all the edges connecting consecutive vertices in P. We will also only consider finite paths in which if v_i, v_j are in P, $v_i \neq v_j$ unless $i = j$ (ie. there are no cycles in the P).

The main task of a single-agent search system is to find paths between sets of vertices. As such, single-agent search algorithms are usually only applicable to problem-solving in discrete spaces. In order to apply single-agent search techniques to a problem like navigation, the map must first be discretized. It should be noted that the problem of modeling problem spaces as finite graphs is beyond the scope of this thesis. When evaluating a single-agent search algorithm, it is only the algorithm's performance in the model that is of concern.

As these models can be arbitrarily large, the vertex set may not fit into memory. As such, problem-solving on large graphs is usually performed on an *implicit* representation if possible. In an *implicit* representation, the vertices are usually referred to as *states* and the vertex set is called the *state space*. States can be thought of as annotated vertices. For example, a state in a navigation problem will be annotated with the location on the map that the vertex corresponds to.

The edge set in an implicit representation is described by the successor function $succ$. For any state s the successor function will return the set of states in S that are neighbours of s. Formally, this function has S as its domain and the power set of S as its range. As we are only considering undirected graphs, if $s_2 \in succ(s_1)$ then $s_1 \in succ(s_2)$.

If the successor function merely contains for each state s a list of adjacent states, the representation is no more compact than the full graph. Instead, the edge set of most single-agent domains is expressed in terms of a set of operators O, some subset of which can be applied in any state. The successor function will find the set of operators $O' \subseteq O$ that are applicable in any state s, and generate the $|O'|$ neighbours of s each found by executing a different $o \in O'$ in s. For example, in a simple navigation example, the operators may entail proceeding one step in one of the four cardinal directions: north, east, south, and west. At any location on a map, the applicable operators will be those directions in which there are roads.

With these notions, it is now possible to define a single-agent search problem:

Definition 2.1.2. A single-agent search problem is determined by a state space S, an initial state $s_i \in S$, a successor function $succ$, a function $h : S \to \mathbb{R}$ called the *heuristic function*, and a boolean function $G : S \to \{1, 0\}$. The task is to find a path P that starts with s_i and ends in a state s_g such that $G(s_g) = 1$, if such a path exists. The only constraint on h is that for any state s such that $G(s) = 1$, $h(s) = 0$.

Let T denote the set of all solution paths, formally described as $P \in T$ if and only if P is a path from s_i to some state s_g such that $G(s_g) = 1$. The problem is said to be an optimal single-agent search problem if the only acceptable paths are those with the minimum cost. The set of minimum cost paths is denoted by T_{opt} and is defined as $P \in T_{opt}$ if and only if $P \in T$ and $C(P) = \min_{P' \in T} C(P')$. As there exists an $\epsilon > 0$ such that all edge costs are at least as large ϵ,

we are guaranteed the existence of this minimum. The cost of any path in T_{opt} is called the optimal cost and is denoted by C^*.

Usually the heuristic function is used to guide the problem-solving process. Intuitively, the value of $h(s)$ is an estimate on the cost of a path from s to a goal state. Several properties of heuristic functions will be described in the next section.

The function G is called the goal test function and defines the desired conditions. By this definition there may be more than one goal state. For example, the goal of navigation may be either to reach a specific supermarket or go to one of several supermarkets. Despite this general definition, all experimentation in this thesis will be on domains that only have a single goal state denoted s_g. We leave experimentation with domains that have multiple goal states for future work.

2.2 Search Terminology

Before describing any algorithms, it is first necessary to introduce a number of terms. Central to most single-agent search algorithms is the *node* data structure. A node n consists of a state s and additional search information about s. Specifically, the node records the current candidate path from s_i to s. This path is usually the shortest such path found thus far. The path is often stored recursively, with each node holding a pointer to another node p that contains the state s_p that is immediately prior to s in the path from s_i to s. The node p is said to be the *parent* of s, and s is said to be a *child* of p. The state within a node will be referred to as $n.state$ and the parent will be referred to as $n.parent$. The node n_i containing s_i will not have a pointer to any other nodes.

Each node n also has a number of costs associated with it. The *g-cost* of a node n, denoted $g(n)$, will be the cost of the stored path from s_i. The *h-cost* of n is the heuristic cost of n. This value corresponds to the heuristic estimate of the cost of traversing from n to a goal node.

For any state s, there will be at most one node n in memory at any time. If a node n is stored in memory, and a path from the initial node n_i to n is found that is shorter than the stored path, the parent and *g*-cost of n is updated. Note, due to the correspondence between nodes and states, we will often use the same notation for nodes and states interchangeably. For example, $h(n)$ will be used to refer to the *h*-cost of n. In reality, $h(n)$ will be shorthand for $h(n.state)$.

Similarly, a path will often be used to refer to a sequence of nodes such that consecutive nodes are neighbours (*ie.* the corresponding states are adjacent). However, when we refer to a path P, we will not necessarily be referring to a path being considered during the execution of any algorithm. Though some algorithm may store all the nodes in P in memory at some time, it may be considering alternative paths to each of the nodes in P instead of the path of P itself. As such, the g-cost of some node n in P may not necessarily be equal to the cost of the portion of P from the beginning of the path to n. Also note, that all paths will be assumed to start at the initial node n_i unless otherwise specified.

The notion of a *node expansion* of a node n is also fundamental to most single-agent search

algorithms. As a first step of node expansion, the goal test is applied to n. If n is not a goal node, the successor function is used to find the neighbours of n. For each neighbour c, a new node, n_c, is constructed which has c as its state. The parent of n_c is set as n. These new nodes are said to be *generated*.

This expansion of nodes will be used to construct a *search tree* in which the initial node forms the root of the tree and below any node n are the children of n. If the successor function of a domain is expressed in terms of $b \in \mathbb{N}$ operators where $b > 0$, the number of nodes generated during any node expansion will be at most b. The value b will be called the *brute-force branching factor* of the domain. If the search tree is constructed to a depth d, the number of nodes in the tree will therefore have $O(b^d)$ nodes. If every node in a domain has exactly b children, the domain will be said to have a *uniform branching factor*.

Where $O = \{o_1, ..., o_b\}$ is the ordered set of operators, the successor function constructs a list L of states by checking each operator in order for applicability. If an operator is found to be applicable to the current state, the corresponding child is constructed and appended to L. Therefore, the order of elements in L will depend on the order of operators in O. In general, a static ordering is set before search begins and operators are checked for applicability in this order. As this *operator ordering* can significantly impact the speed of the search, the ordering used in all experiments will be reported.

The order in which nodes in the search tree are expanded will also depend on the heuristic function. The heuristic functions that are used in practice are always somewhat inaccurate as otherwise problem-solving would be trivial. One such metric for the effectiveness of a heuristic function and an algorithm is the *effective branching factor* [35]. For a solution found at depth d of a problem with a search tree containing M nodes, the effective branching factor b' is calculated as the solution to the equation $1 + b' + b'^2 + ... + b'^d = M$. The effective branching factor can be estimated by solving some small problems. Note that as b' becomes smaller, the search tree decreases in size and the search becomes more efficient.

For the purposes of theoretical analysis, it is often useful to consider the perfect heuristic function h^*, which returns the distance of the shortest path from s to the nearest goal state for any state $s \in S$. This perfect heuristic function can also be used to define additional heuristic properties. A heuristic function h is said to be *admissible* if for any state s in the state space, $h(s) \leq h^*(s)$. Intuitively, a heuristic function is admissible if it never overestimates the distance to the goal. A heuristic function h is also said to be consistent if for any adjacent states m and n, $|h(m) - h(n)| \leq c(m, n)$. It is easily shown that if a heuristic function is consistent, it is necessarily admissible. However, the converse of this statement is not true.

Note, any heuristic function h considered in the remainder of this thesis will have arbitrary properties (*ie.* it may or may not be admissible), unless otherwise specified. The only guaranteed condition is that the heuristic value of any goal state will be 0.

2.3 The Best-First Search Algorithm

One popular approach to single-agent search problems is the best-first search algorithm. In this algorithm, search is guided by a cost function F. The algorithm is shown in Figure 2.1. This figure is based on the pseudocode presented in "AI: A Modern Approach" [40]. Many of the properties found in this section can also be found in the work of Ira Pohl [37] although they may have appeared in other papers before that one.

BestFirstSearch(Initial State s_i):

```
1:  CLOSED ← empty set
2:  OPEN ← empty set
3:  Construct initial node n_i with state n_i.state = s_i and n_i.parent = null
4:  Insert n_i into OPEN
5:  loop
6:    if OPEN is empty then
7:      return no solution exists
8:    n ← node in OPEN with the lowest cost value of F(n)
9:    if n is a goal node then
10:     return path from n_i to n
11:   generate children nodes C = {c_1, ..., c_k} of n
12:   for all c ∈ C do
13:     if ∃m ∈ OPEN, m.state = c.state and F(m) > F(c) then
14:       m.parent ← c.parent
15:     else if ∃m ∈ CLOSED, m.state = c.state and F(m) > F(c) then
16:       m.parent ← c.parent
17:       remove m from CLOSED and add m to OPEN
18:     else if ∀m ∈OPEN∪CLOSED, m.state ≠ c.state then
19:       Add c to OPEN
```

Figure 2.1: The Best-First Search Algorithm.

The algorithm iteratively considers a set of partial candidate paths. Each iteration involves the selection of the most promising partial path P and the expansion of the deepest node on P. The cost, as given by F, of the deepest node on a path P' will determine how promising P' is. These deepest nodes are contained in the *OPEN* set. The selection of a path P corresponds to the selection of the deepest node n on P given by line 8 of Figure 2.1. The expansion of n will then add new candidate paths to memory, each of which consists of P and a successor of n.

Also note that a node n' is in the *CLOSED* set if it has already been expanded. By maintaining this set, it is possible to prune duplicate paths to the same node and to reconstruct the solution path once a goal node is found. These nodes are never removed from the *CLOSED* list and added to the *OPEN* list unless a new path is found to n' that is shorter than any previous such path.

Regardless of the cost function used, the algorithm will be *complete*: the algorithm will terminate with a solution if one exists. However, the choice of cost function F will determine the style of search performed. If $F(n)$ is set to the number of nodes in the path from the initial node to n, the search will be a breadth-first search. If $F(n) = g(n)$, the search will be a Djikstra's Search [40].

Alternatively, consider the function f, the value of which is referred to as the *f-cost* of n:

$$f(n) = g(n) + h(n).$$

If the cost function F is set to be equal to this function f, the result is the *A** algorithm.

The A* algorithm is guaranteed to find the optimal solution if the heuristic function is admissible. However, it is limited by its memory requirements. At any depth d, A* will store $O(b'^d)$ nodes where b' is the effective branching factor. This generation of an exponential number of nodes can occur even when using very accurate heuristics [23]. As such, the A* algorithm can be problematic when used in large domains.

2.4 The Iterative Deepening A* (IDA*) Algorithm

IDA* was developed so as to overcome the memory requirements of A*. An outline of the more general algorithm *Depth-First Iterative Deepening (DFID)* which uses an arbitrary cost function F is shown in Figure 2.2. The IDA* algorithm is a variant of DFID in which the cost function used is the same f-cost function described in Section 2.3. The description of the algorithm and discussion found in this section are based upon the work of Rich Korf [30].

DFID(Initial State s_i):
1: Construct initial node n_i with state $n_i.state = s_i$ and $n_i.parent = null$
2: $threshold \leftarrow F(n_i)$
3: **loop**
4: $threshold \leftarrow$ **RecursiveDFID**(n, $threshold$)
5: **if** solution has been found **then**
6: **return** solution extracted from recursion stack

RecursiveDFID(Node n, $threshold$):
1: $threshold_{next} \leftarrow \infty$
2: **if** n is a goal node **then**
3: **return** with the solution
4: generate children nodes $C = \{c_1, ..., c_k\}$ of n
5: **for all** $c \in C$ **do**
6: **if** $F(c) \leq threshold$ **then**
7: $threshold_{next} \leftarrow \min(threshold_{next}$, **RecursiveDFID**($c$, $threshold$))
8: **else if** $F(c) > threshold$ **and** $F(c) < threshold_{next}$ **then**
9: $threshold_{next} \leftarrow F(c)$
10: **return** $threshold_{next}$

Figure 2.2: The Iterative Deepening A* Search Algorithm.

The IDA* algorithm has a global threshold that is initially set as the heuristic value of the initial state. The algorithm then iteratively performs a series of depth-first searches that are limited to only expanding nodes with an f-cost at most as large as the threshold. If a solution is found during an iteration, the search ends and the solution is extracted from the recursion stack. If an iteration completes without having found a solution, the threshold is increased to the minimum node f-cost

seen during the iteration that exceeded the threshold. As such, it can be guaranteed that on the next iteration, at least one new node will be expanded.

In DFID, only the nodes along the current depth-first search path and the neighbours of nodes on this path are stored in memory. There will be no node updates since as soon as the depth-first search backtracks past a node, any information related to that node is lost. As such, the g-cost of a node n is simply the cost of the single path from n_i to n that is held in memory.

Like A*, IDA* is a complete algorithm that is guaranteed to find the optimal solution if one exists, given an admissible heuristic function. Also notice that IDA* can re-expand the same node multiple times since every iteration is a proper subset of every subsequent iteration. However, if there is some constant $c > 1$ such that for all iterations the number of nodes expanded is approximately c times the number expanded in the previous iteration, IDA* will asymptotically expand the same number of nodes as A*. Intuitively, this effect occurs because the work done in the last iteration will dominate the amount of work done in all previous iterations.

Unfortunately, there are domains in which IDA* can be ineffective. If the nodes in the search tree have a large number of unique f-cost values, the increase in the number of nodes expanded from one iteration to the next will by small. In the worst case, only a single new node is expanded for each new iteration. In this case, the number of nodes expanded by IDA* will actually be $O(N^2)$ where N is the number of nodes expanded by A*.

IDA* will also have problems in domains in which there are many cycles with a small length in the underlying graph. This issue occurs since there is no duplicate detection in IDA*. In these domains, the same state may be expanded multiple times as part of different paths within the same iteration.

Despite these deficiencies, IDA* remains a useful algorithm since it is a *linear-space algorithm*. An algorithm is said to be linear-space if the memory requirement of the algorithm at any time is $O(d)$, where d is the depth of the search. This is because the only nodes stored in memory are the d along the path currently being explored and the neighbours of these nodes, of which there are at most b. As such, IDA* can often solve problems in much larger domains than A* (which is usually limited by its large memory requirements).

2.5 The Recursive Best-First Search (RBFS) Algorithm

While IDA* has proven to be an effective linear-space search algorithm, it can exhibit odd behaviour if the heuristic function is such that the cost function is not monotonically increasing. A cost function F is said to be *monotonically increasing* if and only if for nodes n and m where m is a child of n, $F(m) \geq F(n)$. It is easy to show that the cost function f is monotonic if and only if the heuristic function is consistent.

In Figure 2.3, a small part of a search tree is depicted in which the heuristic function is not consistent. Each circle represents a node and the value inside each circle corresponds to the heuristic

value of the node. A line between nodes indicates that the nodes are neighbours, with the one above the other being the parent. Assume that all edge costs are 1 and that if two nodes both satisfy the cost threshold, the leftmost node is expanded first. This ordering of node expansions is caused by the operator ordering. The f-cost of each node is shown beside the node.

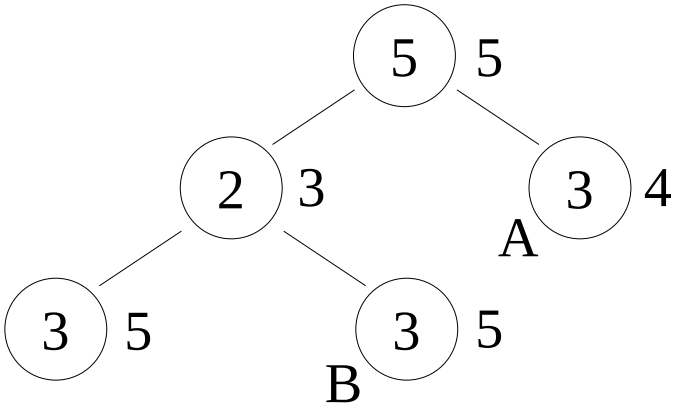


Figure 2.3: Tree with non-monotonic cost function.

The threshold on the first iteration will be set to 5, and so all vertices shown will be expanded. Due to the operator ordering, the node labelled "B" will be expanded before the node labelled "A." However, "A" has a lower f-cost and hence should be considered more promising. While node "A" would be expanded prior to "B" in an A* search, this will not necessarily be the case in an IDA* search.

The RBFS algorithm was developed as a linear-space algorithm that would address this problem. In this section, the algorithm is described based upon the original RBFS paper [31]. The algorithm is shown in Figure 2.4 for any cost function F. Note that in the rest of this thesis, RBFS will be used to refer to the use of this algorithm where the cost function is the f-cost function used in A*.

The call to the RBFS algorithm is made to `MainRBFS`. This algorithm merely calls the recursive function `RecursiveBestFirstSearch` on the initial node with a cost bound of infinity and a lower bound equal to the f-cost of the initial node. The main component of the algorithm is the recursive function.

The algorithm was designed so as to ensure that if a node n is generated for the first time and a node m with $f(m) > f(n)$ is generated for the first time after n, then n will be expanded before m. This property is enforced through the use of the upper bound parameter. The upper bound records the lowest f-cost among all nodes that have been generated but not expanded. When the recursive function is called on any node n, no node with an f-cost greater than the upper bound in the subtree below it will be expanded. If all nodes below n have a larger f-cost, the value of the smallest f-cost exceeding the bound is returned so that any search in other parts of the tree will have this additional constraint on the search.

MainRBFS(Initial State s_i):

Construct initial node n_i with state $n_i.state = s_i$ and $n_i.parent = null$
RecursiveBestFirstSearch(n_i, ∞, $F(n_i)$)

RecursiveBestFirstSearch(Node n, Upper Bound B, Lower Bound l):

if $F(n) > B$ **then**
  return $F(n)$
if n is a goal node **then**
  return solution extracted from recursion stack
if n has no children **then**
  return ∞
generate children nodes $C = \{c_1, ..., c_k\}$ of n
for all $c \in C$ **do**
  if $F(n) < l$ **then**
    $Value(c) \leftarrow \max(l, F(c))$
  else
    $Value(c) \leftarrow F(c)$
$best \leftarrow \arg\min_{c \in C} Value(c)$
while $Value(best) \leq B$ **and** $Value(best) < \infty$ **do**
  if $|C| > 1$ **then**
    $bound_{second} \leftarrow \min_{c \in \{C - best\}} Value(c)$
  else
    $bound_{second} \leftarrow \infty$
  $Value(best) \leftarrow$ **RecursiveBestFirstSearch**($best$, $\min(B, bound_{second}), Value(best)$)
  if a solution was found **then**
    return found solution
  $best \leftarrow \arg\min_{c \in C} Value(c)$
return $V(best)$

Figure 2.4: The Recursive Best-First Search Algorithm.

The $Value$ of any node is an estimate of how promising a node is. For a node n, this value is initally set as the f-cost of n. $Value(n)$ is then updated to the value of the most promising node that exceeds the upper bound in the subtree below n. Intuitively, this update occurs because of the deficiencies in the heuristic function, and it is only through new knowledge found during exploration of the search tree that better estimates on how promising a node are can be determined.

The purpose of the lower-bound is to limit the re-expansion of nodes. A complete description of how this parameter achieves this improvement is beyond the scope of this thesis, but can be found in the original paper on RBFS.

RBFS shares many theoretical properties with both A* and IDA*. For example, the order in which nodes are expanded for the first time during an RBFS search will be the same as the order in which nodes are expanded by A* (aside from differences due to tie-breaking). RBFS is also a complete algorithm that is guaranteed to find the optimal solution if one exists and the heuristic function is admissible. Moreover, RBFS only requires memory linear in the depth of the search and, under certain conditions which are out of the scope of this thesis, will asymptotically expand no more nodes than A*.

Unfortunately, RBFS also shares many of the same deficiencies as IDA*. In domains in which there are many cycles, the lack of memory for duplicate detection can cause RBFS to exhibit poor performance. Similarly, if every node in the search tree has a unique f-cost, RBFS will again expand $O(N^2)$ nodes, where N is the number of nodes expanded by A*.

2.6 Weighted Heuristics

While A*, IDA*, and RBFS are guaranteed to find a solution if the heuristic function is admissible, the time for problem-solving (and the space requirements in the case of A*) can be very large. When suboptimal solutions will suffice, it is often possible to speed up the search in exchange for a decrease in solution quality. The most common way to do so is to use *weighted heuristics*. This strategy was first proposed by Ira Pohl [37].

A weighted heuristic is constructed by multiplying an admissible heuristic function h by a constant factor $w \in \mathbb{R}$, where $w \geq 0$. The resulting heuristic function $h'(n)$ that is then used to guide search is given by $h'(n) = wh(n)$. The cost function used therefore reduces to

$$f(n) = g(n) + h'(n) = g(n) + wh(n).$$

Notice that if $w = 1$ then $h(n) = h'(n)$. Moreover, if $w \leq 1$, the value of $h'(n) = wh(n)$ is guaranteed to be less than the value of the perfect heuristic and hence is admissible. As such, we will not consider weight values in this range for suboptimal search. Also note that in the remainder of this thesis we will use WA*, WIDA*, and WRBFS to denote the use of A*, IDA*, and RBFS respectively with weighted heuristics.

2.7 Test Domains

In the following sections, we will analyze the behaviour of several suboptimal algorithms, including those weighted variants introduced above. Before doing so, we will introduce two problem domains so that the performance of these algorithms can be demonstrated through experimentation.

2.7.1 Sliding Tile Puzzle

The $M \times N$ sliding tile puzzle is a standard test domain for single-agent search algorithms. Each state consists of a matrix of M columns and N rows. $MN - 1$ of these locations contain a tile, each labelled with one of the unique integers from 1 to $MN - 1$. The other location is empty. The available actions involve sliding one of the tiles adjacent to the empty location into that empty location. The original tile location before sliding will be empty after the slide. As tiles cannot slide in a diagonal direction, the four operators are `up`, `down`, `left`, and `right`, which correspond to sliding a tile upward into the empty location, downward into the empty location, leftward into the empty location, and rightward into the empty location. Unless otherwise specified, all experiments are performed with the operator ordering $O = \{$`down`, `right`, `left`, `up`$\}$. The cost of each of these operators is 1, but not all operators are applicable in every state. For example, if the empty location occurs in the upper-left corner, the only applicable operators will be `left` and `up`. Figure 2.5 shows a 3×3 puzzle state, and the neighbours of the state.

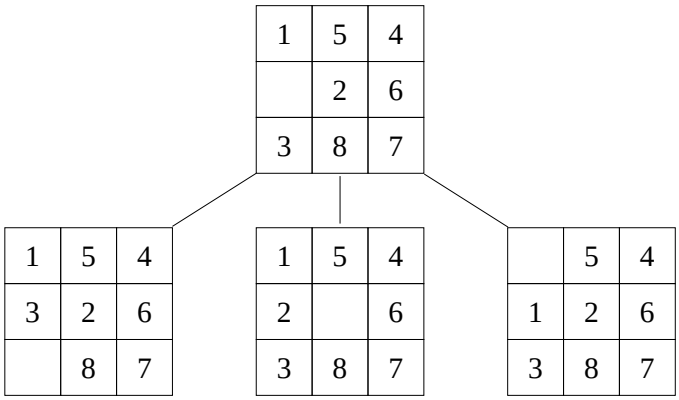


Figure 2.5: Example State in the 3×3 Puzzle and the Neighbours of this State.

A problem in this domain involves finding a sequence of operators that transform some initial state into some desired goal state. In all experiments below, the goal state will remain constant. In this static goal state, the upper-left corner position will be empty, and the remaining tiles will be in consecutive order when read from left-to-right and top-to-bottom. For example, the goal of the 3×3 puzzle is shown in Figure 2.5.

The sliding-tile puzzle is an instance of a permutation puzzle in that any state can be represented as a permutation of the positive integers from 0 to $MN - 1$. In this representation, if a number i

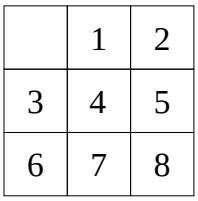


Figure 2.6: The Goal State for the 3×3 Puzzle.

is in position k of the permutation, then the tile marked i will be in column $(k \mod M)$ and row $\lfloor k/M \rfloor$. The permutation location containing 0 corresponds to the empty location. For example, the top state in Figure 2.5 is represented by the permutation $[1, 5, 4, 0, 2, 6, 3, 8, 7]$.

This representation shows that the number of states in the $M \times N$ sliding tile puzzle domain is $(MN)!$. However, the graph corresponding to this domain consists of two distinct, but equally sized connected components. As such, there is only a path from any node to the desired goal for half of the states in the domain. A state is said to be *solvable* if it is in the same connected component as the static goal. All experiments in this thesis were performed on solvable states. Whether a state is solvable or not can be determined with a parity test that will not be described here.

Let $column(s, i)$ and $row(s, i)$ be the column and row in state s in which tile i occurs, respectively. The heuristic function we will use in our experiments is called the Manhattan distance. For each tile i, this function calculates the horizontal and vertical grid distances from the current state s to the goal state s_g. Formally, this distance is calculated as follows:

$$h_{manhattan}(s) = \sum_{i \in \{1,2,\ldots,MN-1\}} (|column(s,i) - column(s_g,i)| + |row(s,i) - row(s_g,i)|).$$

This heuristic function is both admissible and consistent. Also notice that the difference between the heuristic value of a parent node and any of its children will be exactly 1 or -1. This is because every operator will correspond to a shift of the row or column of only one tile.

2.7.2 Pancake Puzzle

The N pancake puzzle domain is another common test domain for single-agent search algorithms. In this domain, there are a stack of N pancakes, each of a unique size. Each operator in this domain is denoted by one of the numbers in the sequence `2`, `3`, ..., `N`. The application of an operator k involves flipping the top k pancakes and therefore inverting their order. All $N - 1$ operators are applicable in each state. Unless otherwise specified, the operator ordering used in all experiments will be $O = \{$`N`, `N-1`, ..., `2`$\}$. In Figure 2.7, an example 4 pancake puzzle state and the neighbours of this state are shown.

As in the sliding tile puzzle, a single static goal state will be used in all experiments in the pancake puzzle domain. In this goal state, the pancakes will be stacked from top to bottom in ascending order of pancake size.

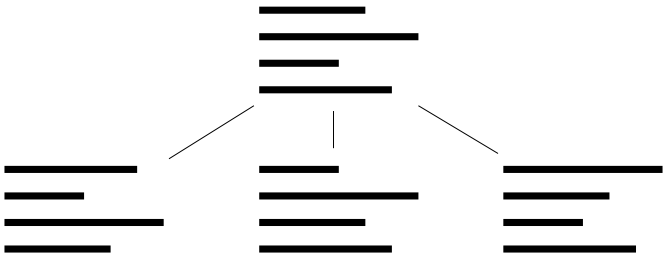


Figure 2.7: Example State in the 4 Pancake Puzzle and the Neighbours of this State.

The N pancake puzzle is also a permutation puzzle with each permutation being of size N. For this representation, each pancake will be uniquely labelled with one of the integers from 0 to $N-1$ such that the pancake labelled i is the ith smallest pancake. For example, the state depicted at the top of Figure 2.7 can be represented with the permutation $[1, 3, 0, 2]$. This representation clearly demonstrates that the number of states in this domain is $N!$. Note, all pancake puzzle states are solvable regardless of the goal chosen.

The heuristic functions that will be used for the N pancake puzzle are based on pattern databases [12]. Pattern databases are built upon a particular kind of abstraction of a permutation puzzle domain. Consider a permutation state space S where each permutation is of size N. The abstraction is built upon the idea of a *pattern*, which is a subset of the values in the permutation. A pattern will be denoted $< t_1, ..., t_k >$ where $k < N$ and each t_i is a unique value found in the permutation.

Let the state space of the abstract version of S be denoted S_a. In the abstract version of any state s, the $N-k$ symbols that do not occur in the pattern will be indistinguishable. Intuitively, this transformation can be thought of as replacing these $N-k$ symbols in the permutation representation of s by a *don't care* element denoted by $\square$.

For an example of such an abstraction, we consider the abstraction given by the pattern $< 1, 2, 5 >$ on a permutation puzzle with 6 values. Consider a state s determined by the permutation $[4, 5, 2, 0, 1, 3]$. The permutation of the abstract state s_a corresponding to s will be given by $[4, \square, \square, 0, \square, 3]$. Note that state s' given by $[4, 1, 5, 0, 2, 3]$ will also be abstracted to s_a. As such, the function that transforms regular states into abstract states, denoted $F_a : S \leftarrow S_a$, is one-to-one but not onto.

The only constraint on the selection of a pattern is that if an operator is applicable in a regular state, it is also applicable in the abstract state. Moreover, if s' is a neighbour of s in S, then $F_a(s')$ is a neighbour of $F_a(s)$ in S_a. Under these conditions, the length of a path between any two abstract states is guaranteed to not overestimate the distance between any two corresponding regular states. In the pancake puzzle, any pattern will guarantee this condition. In the sliding tile puzzle, the empty space must be included in the pattern for this condition to be satisfied.

The purpose of the abstract space is to have a state space on which the exact distance from any state s_a to the abstract goal g_a can be calculated. This is possible since the abstract space has size

$\binom{N}{k}k!$, where k is the number of tiles in the pattern. If k is small enough, the entire abstract state space can be stored in memory. This allows for a pre-processing step by which for any state $s_a \in S_a$, the minimum distance between s_a and g_a can be calculated and stored in a table called a *pattern database*.

The pattern database is then used during a search in S as follows: for any state s, $F_a(s)$ is found, and the distance from $F_a(s)$ to g_a is returned by the pattern database. This distance is then used as the heuristic value of state s. The resulting heuristic function is both admissible and consistent.

There are a number of additional ways in which pattern databases can be used. These include leveraging the symmetry [12] or the duality of permutation puzzles [17]. While these variants are not used in this thesis, we will often build heuristics by maximizing over multiple pattern databases [24]. In this variant, a number of pattern databases are built, each with a different pattern. The resulting heuristic is then the maximum of the values returned by each of the individual pattern databases. The resulting heuristic remains admissible and consistent.

2.8 Weighted A* (WA*)

Having introduced the two test domains described above, we can now analyze the performance of several suboptimal search algorithms, of which WA* is the most commonly used. The effectiveness of WA* was first reported by Ira Pohl [37] but it has also been studied extensively by other researchers.

Part of the reason for the popularity of this algorithm is the fact that there is a proven bound on the suboptimality of the solutions found. In this section, the proof of this bound is reproduced and experiments are given that show WA* in practice.

2.8.1 Bound on Solution Quality

Below we will prove Theorem 2.8.5 which states that a WA* search will find a solution with cost at worst w times the optimal solution cost. This bound was first shown for a related algorithm by Ira Pohl [38], however below we reproduce a proof similar to the one given by Davis *et al.* [13]. This is not the simplest known proof of this bound, however it is instructive in that it demonstrates which set of solution paths are candidates for WA* to return and a similar approach will be used to prove the bound on the solution quality of WIDA* in Section 2.9.1. The following proof includes statements that are more general, such as Lemmas 2.8.1, 2.8.2, and 2.8.3 which are applicable to any arbitrary heuristic function h. For a simpler proof of Theorem 2.8.5, see the aforementioned paper by Ira Pohl.

Before reproducing this proof, some notation must be introduced. First, recall that any path P will be assumed to start at the initial node n_i of some problem. Where n is a node on path P (denoted $n \in P$), let $C(P, n)$ be the cost of the portion of the path from n_i to n. $C(P, n)$ is not necessarily the g-cost of n since there may be shorter paths to n than the one taken by P. For a path

P and a node $n \in P$, we will define $f(P, n) = C(P, n) + h(n)$ and denote the *M-cost* of a path P, $M(P)$ as follows:

$$M(P) = \max_{n \in P} f(P, n).$$

To illustrate these ideas consider Figure 2.8 which contains a sample graph with all edges being of unit-cost. Each node is labelled with the node name. The number inside each node is the heuristic value. Note that the heuristic function is constructed from an admissible heuristic with a weight of 3. There are five solution paths in the figure: $P_1 = \{n_i, a, b, c, d, n_g\}$, $P_2 = \{n_i, e, f, g, h, i, n_g\}$, $P_3 = \{n_i, j, g, h, i, n_g\}$, $P_4 = \{n_i, e, f, k, l, m, n, n_g\}$, and $P_5 = \{n_i, j, g, f, k, l, m, n, n_g\}$. Notice that only P_1 and P_3 are optimal.

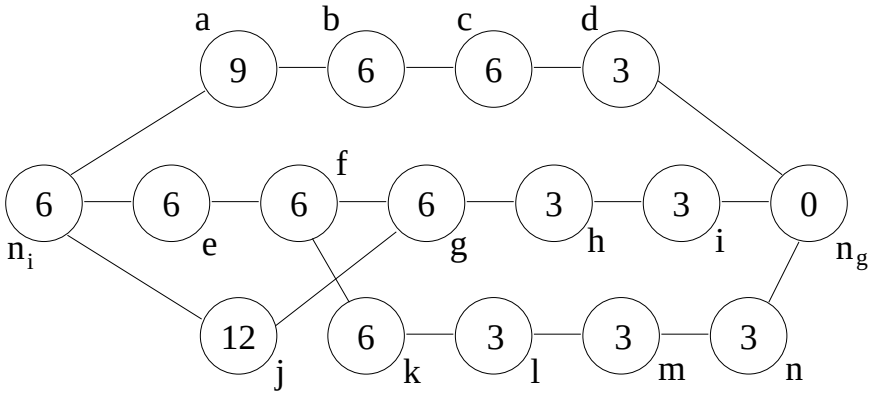


Figure 2.8: Example Domain.

Consider path P_2. The value of $f(P_2, n)$ for each $n \in P_2$ is 6, 7, 8, 9, 7, 8, and 6 from left to right. Therefore, $M(P_2) = 9$. Notice that there is a shorter route to node g than is taken by P_2. This means that the f-cost of node g is not necessarily 9, and may be 8. At any time during the search, the actual value will depend on which paths to g have been discovered thus far. As such, for any path P, $M(P)$ is not necessarily the largest f-cost of nodes on P.

Recall that T and T_{opt} denote the set of all solution paths and the set of all optimal solution paths for a problem, respectively. Q will now be defined as follows:

$$Q = \min_{P \in T} M(P)$$

Q_{opt} will be defined similarly, except the minimization is only over paths in T_{opt}. Since $T_{opt} \subseteq T$, necessarily $Q \leq Q_{opt}$.

Now, let T_Q be the set of solutions with *M*-cost Q and let C_Q^{min} be the minimum cost of any solution in T_Q. We can now define $T_Q^{min} = \{P \mid P \in T_Q, C(P) = C_Q^{min}\}$. Intuitively, T_Q^{min} is the set of lowest cost paths in T with an *M*-cost of Q. Note, if T is non-empty, T_Q and T_Q^{min} will also necessarily be non-empty.

In the case of the graph in Figure 2.8, $T_{opt} = \{P_1, P_3\}$ and $T = \{P_1, P_2, P_3, P_4, P_5\}$. Since $M(P_1) = 10$, $M(P_2) = 9$, $M(P_3) = 13$, $M(P_4) = 9$, and $M(P_5) = 13$ then $Q = 9$ and

$Q_{opt} = 10$. Moreover, $T_Q = \{P_2, P_4\}$. As P_2 is shorter than P_4, then $C_Q^{min} = C(P_2) = 6$ and $T_Q^{min} = \{P_2\}$.

Finally, when the tth node is to be expanded during an A* search, $U(t)$ will be used to denote the largest value of $\min_{n \in OPEN} f(n)$ seen thus far. For example, at $t = 1$ only n_i is on the $OPEN$ list, and $U(1) = f(n_i)$. Let n_c be the child of n_i with the smallest f-cost. If $f(n_c) > f(n_i)$, then $U(2) = f(n_c) > U(1)$. However, if $f(n_c) \leq f(n_i)$, then $U(2) = U(1)$. Clearly, U is monotonically increasing with t.

Similar statements to Lemmas 2.8.1, 2.8.2, and 2.8.3 are found in the works of Davis *et al.* [13] and Bagchi and Mahanti [3], however the proofs are left to the reader. Below, these lemmas are proven in full as is the proof on the bound on solution quality given by Davis *et al.* [13].

Lemma 2.8.1. *For any single-agent search problem with an arbitrary heuristic function h, and any $P \in T$, $C(P) \leq M(P)$.*

Proof. The final node in $n_g \in P$ will be a goal node. In this case, $f(P, n_g) = C(P, n_g) + 0 = C(P)$. By the definition of function M, $f(P, n_g) \leq M(P)$ and so $C(P) \leq M(P)$. □

The following lemma will show that A* will be simultaneously considering all paths in T_Q^{min}.

Lemma 2.8.2. *After $t - 1$ node expansions of an A* search, if $U(t) \leq Q$ then for all $P \in T_Q^{min}$, there is at least one node n on the $OPEN$ list such that $n \in P$ and $g(n) = C(P, n)$.*

Proof. This proof is by induction. The base case occurs when no nodes have been expanded. At this point, only n_i is in the $OPEN$ list. As n_i is on all solution paths, $f(n_i) \leq Q$ and so necessarily $U(1) \leq Q$. As well, n_i will have g-cost of 0 which satisfies the fact that $C(P, n_i) = 0$ for any $P \in T$. Therfore the statement is true in the base case.

Assume the statement is true after N nodes have been expanded, $U(N + 1) \leq Q$, and n is the $N + 1$st node to be expanded. Prior to expanding n, any path $P \in T_Q^{min}$ has at least one node $m \in P$ such that m is on the $OPEN$ list with $g(m) = C(P, m)$ by the induction hypothesis. It is now necessary to show that after expanding n, m is unchanged or a new node $n' \in P$ is on the $OPEN$ list.

Consider the situation for which $m = n$. Let q be the node immediately after m in P. The path found through m to q will have cost $C(P, q)$ by the induction hypothesis. q will necessarily be a child of m and will either have never been seen before; already reside on the $OPEN$ list; need to be removed from the $CLOSED$ list and added to the $OPEN$ list; or, already be in the $CLOSED$ list with a smaller g-cost than $C(P, q)$. If it is this last case, let R be the path from n_i to q with a g-cost less than $C(P, q)$. A new path P' can be constructed which begins with path R and continues along the same sequence as P after q. The new path P' will clearly have a lower cost than P. Since $U(N + 1) \leq Q$, all nodes r on R wil be such that $f(R, r) \leq Q$ and $M(P) \leq Q$, then $M(P') \leq Q$. Together, these facts contradict the fact that $P \in T_Q^{min}$. Therefore, q cannot already be on the $CLOSED$ list with a smaller g-cost.

As such, q must be a new successor, already reside in the $OPEN$ list, or be moved from the $CLOSED$ list to the $OPEN$ list with an updated g-cost. In any of these cases, the g-cost assigned to q will be $C(P, n) + c(n, q)$. Therefore, $g(q) = C(P, q)$ and the statement is satisfied for P.

Now consider the situation in which $m \neq n$. The only way that m will be affected in any way by the expansion of n is if m is a successor of n and the g-cost of m is updated. However, by a similar construction as above, a path P' can be found with $M(P') = Q$ and $C(P') < C(P)$. This contradicts the fact that $P \in T_Q^{min}$ and so the statement is satisfied for this situation. As these cases cover all possible situations for $P \in T_Q^{min}$, the inductive step is complete. $\square$

With the previous lemma, it is now possible to show that the only solutions that will be found by an A* search, regardless of whether the heuristic is admissible or not, are those in the set T_Q. Combined with Lemma 2.8.1, we have the following statement.

Lemma 2.8.3. *For a single-agent search problem with an arbitrary heuristic function h, any solution path P found by A* will satisfy the inequality $C(P) \leq Q$.*

Proof. Note that no solution path will by found by A* until there have been at least t node expansions such that $U(t + 1) \geq Q$ since for any $P' \in T$, there is some node $n \in P'$ such that $f(n) \geq Q$ (since $M(P') \geq Q$). n will only be expanded when it is the node in $OPEN$ with the smallest f-cost.

As n_i is on all solution paths, $f(n_i) \leq Q$ and so $U(1) \leq Q$. Now, assume that $M(P) > Q$ and let n be the first node on P such that $f(P, n) = M(P)$. Before n can be expanded, U must increase to $M(P)$. By Lemma 2.8.2, for any path $P' \in T_Q^{min}$ there will be some node $n' \in P'$ on the $OPEN$ list while $U(t) \leq Q$. Since U starts with a value at most Q, U will not be able to increase beyond Q until every node on P' has been expanded. If this is the case, the algorithm will return P' instead of P, which is a contradiction. Therefore $M(P) \leq Q$. Since $M(P) \geq Q$ by definition, $M(P) = Q$. By Lemma 2.8.1, this gives us $C(P) \leq Q$. $\square$

The following lemma bounds the value of Q by a function of the optimal solution cost where the heuristic function is bounded.

Lemma 2.8.4. *For a single-agent search problem with an arbitrary heuristic function h, if for all s, $h(s) \leq wh^*(s)$ for some $w \geq 1$, then $Q \leq wC^*$.*

Proof. Consider any optimal path $P \in T_{opt}$. Since the perfect heuristic value of the initial node will by definition be equal to C^*, $f(n_i) \leq wC^*$. Any node $n \in P$ will satisfy the following:

$$f(P, n) \leq C(P, n) + wh^*(n) \leq C(P, n) + w(C^* - C(P, n)) \leq (1 - w)C(P, n) + wC^* \leq wC^*$$

since $w \geq 1$. Therefore, $M(P) \leq wC^*$. As P is an arbitrary path in T_{opt}, $Q_{opt} \leq wC^*$. Since $Q \leq Q_{opt}$, $Q \leq wC^*$. $\square$

With these lemmas, it is now possible to bound the solution quality of any path found by WA*.

Theorem 2.8.5. *If a solution path P is found by a WA* search with weight w, $C(P) \leq wC*$.*

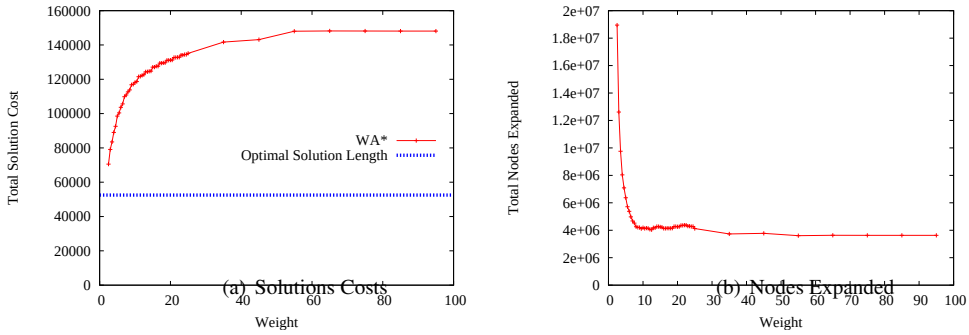
Proof. Let h denote an admissible heuristic and consider some constant $w \geq 1$. For any node n, $wh(n) \leq wh^*(n)$ is satisfied. This means that for a WA* search with weight w (which uses heuristic function $h' = wh$), $Q \leq wC*$ by Lemma 2.8.4. By Lemma 2.8.3, $C(P) \leq Q$ and so $C(P) \leq wC*$. $\square$

With this bound it is possible to control the suboptimality of a solution. However, as will be shown below, WA* generally significantly outperforms these bounds. Also note that the bound given by Theorem 2.8.5 applies with any arbitrary heuristic function h where for all nodes n, $h(n) \leq wh^*(n)$.

2.8.2 WA* in the Sliding Tile and Pancake Puzzles

In this section, we demonstrate the behaviour of WA* by experimenting with the algorithm in the 4×4 sliding tile and 14 pancake puzzle domains. Due to the space requirements of A*, the number of nodes stored during problem solving on any single problem was limited to a million states.

For the sliding tile puzzle, the test set used consisted of $1,000$ randomly generated solvable 4×4 puzzle states. The total optimal cost of all $1,000$ problems was found using IDA* and is $52,522$. All the weights in the set $W_1 = \{1.0, 1.5, 2.0, 2.5, ..., 25.0\}$ were tested as were the weights in the set $W_2 = \{35, 45, ..., 95\}$. The results for the sliding tile experiments are shown in Figure 2.9. Due to the memory restriction, WA* was only able to solve 241, 897, and 996 of the $1,000$ problems with the weights of 1, 1.5, and 2, respectively. As such, these data points have been omitted from Figure 2.9. Also note that a dotted line is shown in Figure 2.9(a) depicting the total optimal solution length over all $1,000$ problems.

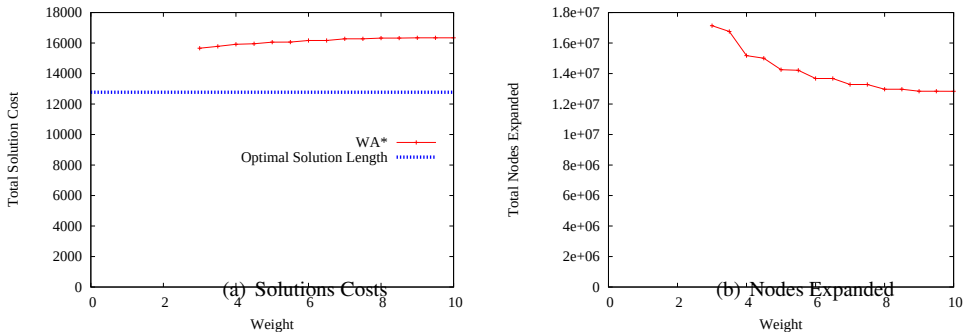


(a) Solutions Costs (b) Nodes Expanded

Figure 2.9: WA* on $1,000$ 4x4 Sliding Tile Problems

For the pancake puzzle, the test set is composed of $1,000$ randomly generated puzzle states. The total optimal cost of all $1,000$ problems was found using WIDA* and is $12,775$. The heuristic function used is given by the $< 0, 1, 2, 3, 4, 5, 6 >$ pattern database. The results for WA* are shown in Figure 2.10. Due to the memory limit, WA* was unable to solve all problems with the smaller weights. For the weights of 1, 1.5, 2, and 2.5 only 459, 883, 987, and 993 of the $1,000$ problems were solved, respectively. As such, these weights have been omitted from the figure. Also note that

for this puzzle, all WA* searches with a weight of 10 or greater produced identical results. The data points corresponding to weights larger than 10 have also been omitted from the the figures.



(a) Solutions Costs (b) Nodes Expanded

Figure 2.10: WA* on $1,000$ 14 Pancake Tile Problems

In both domains, the solution quality found by WA* during the experiments outperforms the guaranteed upper bound by a large margin. As an example, consider the weight of 3 finds solution paths that are on average 1.5 and 1.2 times more than the optimal cost in the sliding tile and pancake puzzles, respectively. Moreover, with the weight of 10, the factor of suboptimality is only 2.3 and 1.3 in these puzzles, respectively. Figures 2.9(a) and 2.10(a) show that in these domains the solution quality actually plateaus instead of growing linearly with the weight value.

Figures 2.9(b) and 2.10(b) show that as the weight increases, the search time generally decreases. Part of the reason for this behaviour is the fact that in both of these domains, there is guaranteed to be a solution in the subtree below any node n in the search tree. As such, taking a more greedy approach by increasing the weight is an effective strategy. Note, that the amount of improvement in search speed gained from increasing the weight diminishes for larger weight values.

2.9 Weighted IDA* (WIDA*)

WIDA* is another well-known suboptimal search algorithm in which a weighted heuristic is used in IDA*. In this section, we will show that WIDA* has the same bound on solution quality as WA*. Several other properties of this search technique will be highlighted, including several of the deficiencies of the algorithm. Finally, we will demonstrate the behaviour of WIDA* experimentally.

2.9.1 Bound on Solution Quality

We have been unable to find a proof for a bound on the quality of solutions found by WIDA* in the search literature. However, in the work of Davis *et al.* [13], a bound is found for an algorithm called A^+_δ which shares properties with both WA* and WIDA*. Below, we adjust this proof to make it applicable to WIDA*, using much of the same notation as was used in Section 2.8.1.

The first lemma is analogous to Lemma 2.8.3. In this proof, we will use the notation $threshold(i)$ to refer to the value of the global threshold on the ith iteration. The $threshold$ function takes on a similar role as the function U did in Lemma 2.8.3.

Lemma 2.9.1. *For any single-agent search problem with a heuristic function h, any solution path P found by IDA* will satisfy the inequality $C(P) \leq Q$.*

Proof. Assume P is found on the ith iteration. P cannot be found until $threshold(i) \geq Q$ since there is some node $n \in P$ with $f(P, n) \geq Q$ and n will not be expanded during an examination of P until $threshold(i) \geq f(P, n)$. Also notice, that if an iteration k starts such that $threshold(k) > Q$, then all nodes on all paths in T_Q can be expanded, and so a solution will necessarily be found before the iteration completes. Below we will prove that $threshold(i)$ must be exactly equal to Q.

Assume some iteration k begins with $threshold(k) > Q$ without having an iteration $k' \in \{1, ..., k-1\}$ such that $threshold(k) = Q$. As $threshold(1) = h(n_i) = f(P, n_i) \leq Q$, $k > 1$. Due to the depth-first nature of IDA*, every path P' will be examined up until the first node $m \in P'$ is found such that $f(P', m) > threshold$. As the next threshold is set as the smallest such value, the f-cost of a node that was generated but not expanded during iteration $k-1$ was $threshold(k)$.

For any $P_Q \in T_Q$, P_Q was not found in iteration $k-1$. However, during this iteration some node $n_Q \in P_Q$ must have been generated but not expanded. As $M(P_Q) = Q$, necessarily $f(P_Q, n_Q) \leq Q \leq threshold(k)$. This contradicts the choice of $threshold(k)$, and so iteration k cannot have $threshold(k) > Q$ unless an earlier iteration had a threshold of exactly Q.

Since the threshold value is also strictly increasing and $threshold(i) \geq Q$, then $threshold(i) = Q$. Therefore, $M(P) = Q$. By Lemma 2.8.1, $C(P) \leq Q$. □

It is now possible to prove the bound on WIDA*.

Theorem 2.9.2. *If a solution path P is found by a WIDA* search with weight w, $C(P) \leq wC*$.*

Proof. Let h denote an admissible heuristic and consider some constant $w \geq 1$. For any node n, $wh(n) \leq wh^*(n)$ is satisfied. As such, by lemma 2.8.4 $Q \leq wC*$ for WIDA* with weight w. By lemma 2.9.1, $C(P) \leq Q$ and so $C(P) \leq wC*$. □

While WIDA* and WA* both have the same bound on solution quality and are both limited to returning solutions from T_Q, in practice, the solution quality found by WIDA* is generally worse than that found by WA*. This will be shown experimentally in Section 2.9.4. However, we will first consider several other properties of WIDA*.

2.9.2 Effect of w on the Iterations of WIDA*

The first obvious question when analyzing WIDA* is how the search trees examined during any iteration i changes when different weights are used. In the case of the first iteration, it can be shown that larger weights will always examine a larger search tree. This idea will be formalized in the following theorem:

Theorem 2.9.3. *Consider the first iteration of two WIDA* searches in which the search does not terminate when a solution is found and the same admissible heuristic function h is being weighted in both searches. The first search will be performed with weight w_1 and the second with weight $w_2 > w_1$. If a node n is expanded by the WIDA* search with weight w_1 during the first iteration, then it is also expanded by the WIDA* search with weight w_2.*

Proof. The proof is by induction. The base case is the node n_i. As the threshold used in the first iteration will be $w_1 h(n_i)$ and $w_2 h(n_i)$ for the weight w_1 and w_2 searches, rescpectively, clearly this node will be expanded by both searches.

Now consider any path P of length N such that both WIDA* searches expand the first $N-1$ nodes along P. Let n be the Nth node on P. If n is not expanded by the weight w_1 search, the statement is vacuously true for n. Let $f_{w_1}(n')$ refer to the f-cost of node n' when searching with weight w_1, and define $f_{w_2}(n')$ analogously. If n is expanded by the weight of w_1, then $f_{w_1}(n) \leq w_1 h(n_i)$. As such, $g(n) + w_1 h(n) \leq w_1 h(n_i)$. This leads to the following algebraic manipulation:

$$\begin{aligned} g(n) + w_1 h(n) &\leq w_1 h(n_i) && (2.1)\\ g(n) &\leq w_1(h(n_i) - h(n)) && (2.2)\\ g(n) &\leq w_2(h(n_i) - h(n)) && (2.3)\\ g(n) + w_2 h(n) &\leq w_2 h(n_i) && (2.4)\\ f_{w_2}(n) &\leq w_2 h(n_i) && (2.5) \end{aligned}$$

Line 2.3 is possible since $w_2 \geq w_1$ and $h(n_i) - h(n) \geq 0$. Therefore, n will also be expanded by the search with weight w_2. □

While the set of nodes expanded by any weight during the first iteration is guaranteed to be a superset of the set of nodes expanded by any smaller weight during the first iteration, this is not the case in subsequent iterations. An example of this behaviour is shown in Figure 2.11. In this figure, a partial subtree is shown from left to right where each circle represents a node, the initial node is the farthest leftward, and the lines represent edges. The number inside the node is the heuristic value of the node. The number above the node is the f-cost of the node for a search with a weight of 1, and the number below the node is the f-cost of the node for a search with a weight of 2. The initial thresholds are set to the f-costs of the initial node. The nodes in bold correspond to the first iteration for both weights. In this case the first iterations will be exactly the same.

The nodes labelled "A" and "B" will be generated but not expanded during the first iteration. On the second iteration, the thresholds for the second iteration will be set to 12 and 21 for weights 1 and 2, respectively. Now consider "B" and all its descendents. The search with a weight of 2 will expand all the nodes depicted, while the weight of 1 will expand all of these nodes except node "C".

The fact that a larger weight will expand a larger set of nodes is unsurprising. However, consider node "A". This node will be expanded by the search with a weight of 1, but will *not* be expanded by

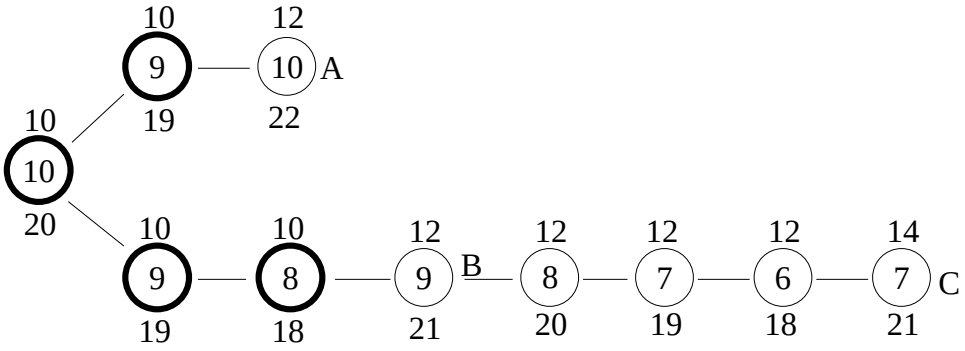


Figure 2.11: Different Search Trees for Different Weights.

the search with a weight of 2. As such, on any iteration that is not the first, larger weights are not guaranteed to search a superset of the nodes expanded by smaller weights.

2.9.3 Deficiencies of WIDA*

The main issue with WIDA* is that the search can become stuck searching large parts of the search tree without any heuristic guidance. For example, consider a domain with a uniform branching factor b in which all operators have a cost of 1. Let the admissible heuristic function being weighted be denoted h and assume h is consistent. In such a domain, the heuristic value of a child n_c can be at most 1 more than the heuristic value of the parent n_p. In this case, $f(n_c) = g(n_c) + wh(n_c) = g(n_p)+1+w(h(n_p)+1) = g(n_p)+wh(n_p)+w+1 = f(n_p)+w+1$. As well, the heuristic value of a child n_c can be at most 1 less than the heuristic value of the parent n_p. By a similar calculation, it can be shown that in this case, $f(n_c) = f(n_p) - (w-1)$.

Assume the current threshold is H and that a node n is to be expanded. As the f-cost of any node can be at most $w+1$ greater than the parent, all nodes to a depth $\lfloor (H-f(n))/(w+1) \rfloor$ below n will be expanded during the current iteration unless a solution is found first. The minimum size of this subtree will be $Z = b^{\lfloor (H-f(n))/(w+1) \rfloor}$. The key observation is that in this subtree, no pruning will occur and WIDA* will be forced to perform a depth-first search with no heuristic guidance.

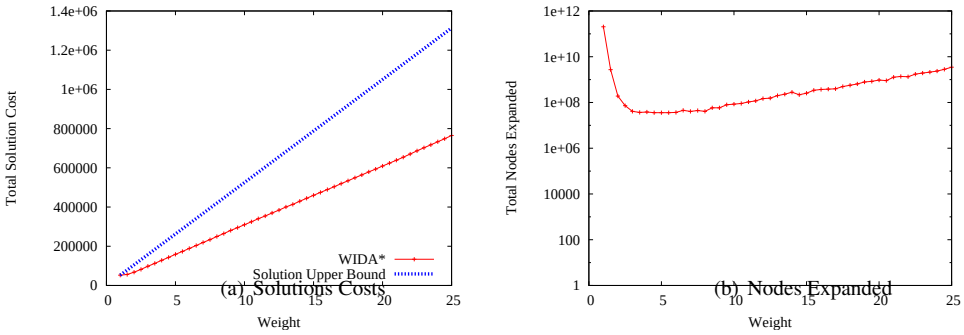
If $h(n)$ is low, the value of $H - f(n)$ will most likely be high and consequently so will Z. Therefore, if a heuristic leads the search into an area of the state space with low heuristic values but which is not actually near the goal, WIDA* must expand a large number of nodes before it can backtrack to n. Even if there is a goal near n, WIDA* may still have to expand a large number of nodes before finding it since the search has no guidance in the search tree of size at least Z.

In an extreme example of this behaviour, assume it is the first iteration ($H = wh(n_i)$) and all d moves that lead to n have decreased the heuristic value by 1. The f-cost of n is then $f(n) = d + w(h(n_i) - d) = d(1-w) + wh(n_i)$. The minimum depth below n to which all nodes must be expanded is therefore $\lfloor (wh(n_i) - [d(1-w) + wh(n_i)]))/(w+1) \rfloor = \lfloor d(w-1)/(w+1) \rfloor$. Therefore the minimum number of nodes in the subtree below n that can be expanded during the

current iteration is exponential in d and increases as w increases.

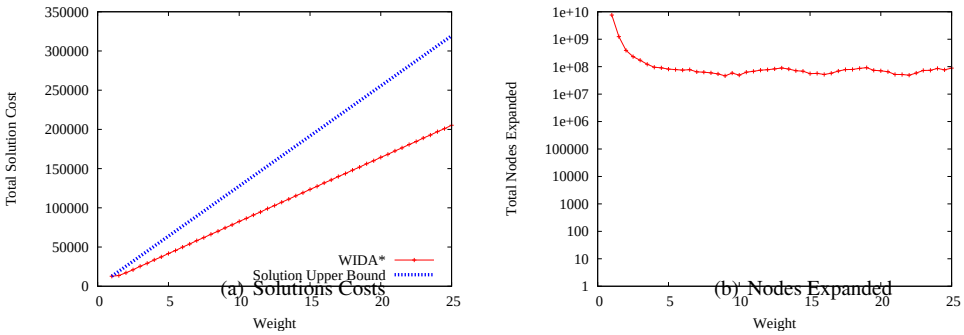
2.9.4 WIDA* in the Sliding Tile and Pancake Puzzles

The WIDA* algorithm was also tested on the $1,000$ 4×4 sliding tile puzzle problems, and the $1,000$ 14 pancake problems that WA* was tested on. For both puzzles, the weights tested were those from the set $W_1 = \{1.0, 1.5, 2.0, 2.5, ..., 25.0\}$. The results of these experiments are shown in Figures 2.12 and 2.13.



(a) Solutions Costs (b) Nodes Expanded

Figure 2.12: WIDA* on $1,000$ 4x4 Sliding Tile Problems



(a) Solutions Costs (b) Nodes Expanded

Figure 2.13: WIDA* on $1,000$ 14 Pancake Problems

In both puzzles, IDA* was able to solve all problems optimally unlike WA*. Figures 2.12(a) and 2.13(a) demonstrate that while WIDA* outperforms the solution cost bounds guaranteed by Theorem 2.9.2 (which are shown in the figure), the solution quality still degrades linearly. The slopes of the total solution cost found by WIDA* are approximately $30,325$ and $8,176$ for the sliding tile and pancake puzzles, respectively. Note, the slopes of the upper bound lines are given by the optimal total solution costs of $52,522$ for the 4×4 puzzle and $12,775$ for the 14 pancake puzzle.

Figures 2.12(b) and 2.13(b) also show that both puzzles, the total number of nodes expanded hits a minimum as the weight increases. However, in the sliding tile puzzle, increasing the weight too far beyond this minimum causes the search efficiency to degrade quickly. This behaviour is not

apparent in the pancake puzzle, in which the average search effort remains mostly stable beyond this minimum. The minimum is hit at a weight of 5 for the 4×4 puzzle, although most of the weights between a weight of 3 and 7 perform similarly. The minimum is hit at a weight of 9 for the 14 pancake puzzle although all weights from 4 to 25 perform similarly.

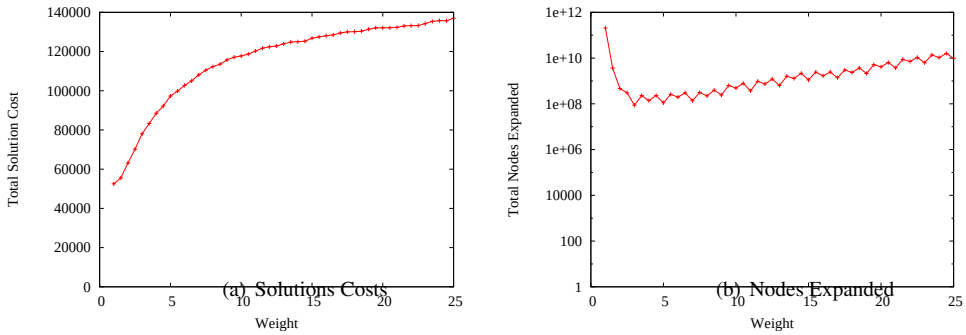
2.10 Weighted RBFS (WRBFS)

WRBFS also has the same bound on solution quality as the other two weighted algorithms. While this will not be proved formally, the argument is based on the fact that RBFS will expand nodes in the same order as A* aside from differences due to tie-breaking. As such, RBFS will necessarily only find solutions in T_Q, each of which will be at worst w times the length of the optimal solution.

In practice, WRBFS generally outperforms this bound and finds solutions with similar quality to those found by WA*. This behaviour will be demonstrated in the two test domains.

2.10.1 WRBFS in the Sliding Tile and Pancake Puzzles

In Figures 2.14 and 2.15 the results are shown for the experiments involving the use of WRBFS for problem-solving on the $1,000$ 4×4 puzzles and $1,000$ 14 pancake puzzles, respectively. Like IDA*, RBFS was able to solve all problems in both problem sets optimally.



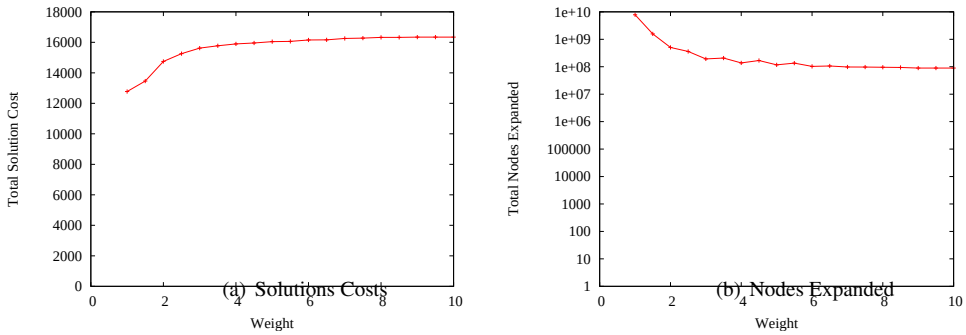
(a) Solutions Costs (b) Nodes Expanded

Figure 2.14: WRBFS on $1,000$ 4x4 Sliding Tile Problems

In terms of solution cost, WRBFS has similar behaviour in both domains. Specifically, the solution quality found by WRBFS is very similar to that found by WA*.

In terms of solution effort, WRBFS actually performs quite differently on both domains. In the sliding tile puzzle, the behaviour of WRBFS is similar to that of WIDA* in that the relationship between the value of the weight and the search effort is concave. However, WRBFS is actually outperformed by WIDA* on every single weight value, despite the fact that WRBFS was designed so as to avoid some of the deficiencies of WIDA*.

Even when comparing the algorithms at their peak performance on the 4×4 sliding tile puzzle, WIDA* significantly outperforms WRBFS. The weight value for which WRBFS requires the least



(a) Solutions Costs (b) Nodes Expanded

Figure 2.15: WRBFS on $1,000$ 14 Pancake Tile Problems

amount of total nodes expanded over all problems is the weight of 3. As mentioned in Section 2.9.4, the weight that required the least amount of total nodes expanded over the problem set is the weight of 5.5, which expanded 2.6 times fewer total nodes than WRBFS did with a weight of 3. This difference in the performance of the two algorithms is magnified when one considers the behaviour reported in the original RBFS paper that the running time needed per node expansion of WRBFS is greater than the running time needed per node expansion of WIDA*.

In the 14 pancake puzzle, the performance of WRBFS is quite similar to WIDA*. In WRBFS all weights of value at least 10 produce an identical search, at which point a minimum is reached. In WIDA*, the performance is quite stable for weights of size at least 4. WIDA* again outperforms WRBFS on all weights with the only exception being the weight of 19 for which the number of nodes expanded by WRBFS is slightly less than the number expanded by WIDA*. However, the amount by which WIDA* generally outperforms WRBFS is much smaller. When comparing the peak performance of these algorithms, WIDA* expands 1.9 times fewer nodes than WRBFS.

2.11 Beam Search and BULB

Another approach to suboptimal search is *beam search*. In this section, the general beam search algorithm will be described. Enhancements that make beam search into the complete algorithm known as BULB are then offered, and the behaviour of this algorithm is shown in experimentation. Aside from the experiments, this section is based on the work of Furcy and Koenig [18].

2.11.1 Beam Search

Traditional beam-search is designed for unit-cost domains. The main data structure of a beam search is the *beam*. Each beam is a container for at most B nodes, all at the same depth. B is referred to as the *beam width* or *beam size*.

The algorithm begins with the construction of the initial beam $beam_0$ which holds only the initial node n_i. n_i is then expanded, and the successors of n_i are sorted. At most B successors with the

lowest heuristic value then form the next beam: $beam_1$. Similarly, the construction of an arbitrary beam $beam_k$ requires the expansion of all nodes in $beam_{k-1}$. The generated nodes are then sorted and the best B nodes that do not already exist in any of the other beams are then used to form $beam_k$. If a goal node is found during the expansion of $beam_{k-1}$, the solution of length k is extracted from the beams and returned.

The process of constructing new beams is continued until a goal solution is found or some memory limit L is hit. As at most B nodes at any depth are ever considered, beam-search is an incomplete algorithm.

2.11.2 Beam Search with Limited Discrepancy Backtracking (BULB)

BULB is a variant of beam search that includes backtracking. Aside from the beam width parameter B, BULB takes in a limit on the number of states that can be stored in memory at any time, denoted L. By adding backtracking, BULB becomes a complete algorithm provided that there exists a solution that has a length of at most L/B.

When the set of successors S_{beam_j} of a beam $beam_j$ are generated and sorted, S_{beam_j} can be divided into a number of *slices* denoted $B_1, B_2, ..., B_k$ where $(k-1)B \leq |S_{beam_j}| < kB$. In the ordering of S_{beam_j}, B_i holds the nodes in positions $(i-1)B$ to $iB-1$. For example, B_1 holds the top B nodes ordered by heuristic value.

The BULB algorithm is shown in Figure 2.16. One of the main concepts behind this algorithm is the notion of a *discrepancy* developed by Harvey and Ginsberg [21]. As extended by Furcy and Koenig, a discrepancy occurs when instead of using B_1 in the construction of the next slice, some slice B_j is used, where $j > 1$. For example, consider some $beam_j$ whose list of successors can be split into 3 slices: B_1, B_2, and B_3. Instead of constructing $beam_{j+1}$ out of the nodes in B_1 as traditional beam search would, the new beam can be constructed from the nodes in B_2. When a beam is constructed in this way, a discrepancy is said to occur. Note, if the new beam is created with B_3, we still say only a single discrepancy has occurred.

The BULB algorithm runs iteratively with an increasing limit on the number of discrepancies that can be used. Each iteration corresponds to one pass through the loop beginning at line 5 of the `BULB` algorithm. The value of the discrepancy limit d on the first iteration will be 0. At any time during an iteration with limit d, no more than d beams in memory can have been constructed using some slice other than the first.

The order in which discrepancies are selected to occur is given by *limited discrepancy backtracking*. Backtracking begins whenever the memory limit is reached without having found a solution. At this point, beams will be removed until some $beam_j$ is found such that $beam_{j+1}$ was constructed using a discrepancy. Where B_i was last used to construct $beam_{j+1}$, B_{i+1} will be used this time. This behaviour is caused by line 10 of the `RecursiveBulb` which iterates through the discrepancies. If no such B_{i+1} exists, $beam_{j+1}$ will be constructed without a discrepancy, using B_1 (line

BULB(Initial State s_i, Beam Size B, Memory Limit L):

$d \leftarrow 0$
$BEAMS \leftarrow$ empty set of beams
Construct initial node n_i with state $n_i.state = s_i$ and $n_i.parent = null$
Construct $beam_1$ with n_i being the only node on it and add $beam_1$ to set $BEAMS$
loop
  RecursiveBulb($BEAMS$, B, L, d)
  if a solution was found **then**
    return found solution
  $d \leftarrow d + 1$

GetSuccessors($BEAMS$):

Generate successors S_{beam_k} of deepest beam $beam_k \in BEAMS$
Remove all Duplicates from S_{beam_k} and any nodes already in some beam in $BEAMS$

RecursiveBulb($BEAMS$, B, L, d):

$S_b \leftarrow$**GetSuccessors**($BEAMS$)
if S_b is empty **then**
  return without solution
else if $(|S_b| + \sum_{beam_j \in BEAMS} |beam_j|) > L$ **then**
  return without solution
else if S_b contains a goal node **then**
  return solution extracted from $BEAMS$
if $d > 0$ and $|S_b| > B$ **then**
  $num \leftarrow \lceil |S_b|/B \rceil$
  for all $j \in \{2, 3, ..., num\}$ **do**
    Sort S_b and find jth slice B_j. Delete all other nodes in S_b
    Construct new beam $beam_{k+1}$ consisting of nodes in B_j
    RecursiveBulb($BEAMS$, B, L, $d - 1$)
    if a solution was found **then**
      return found solution
    Remove $beam_{k+1}$ from $BEAMS$
    $S_b \leftarrow$**GetSuccessors**($BEAMS$)
Sort S_b and find 1st slice B_1. Delete all other nodes in S_b
Construct new beam $beam_{k+1}$ consisting of nodes in B_1
RecursiveBulb($BEAMS$, B, L, d)
if a solution was found **then**
  return found solution
Remove $beam_{k+1}$ from $BEAMS$
return without solution

Figure 2.16: The BULB Algorithm.

18). Note, if backtracking reaches the first beam and the second beam was constructed without a discrepancy, then the number of discrepancies is incremented and a new iteration will begin.

This scheme is designed so as to first reconsider decisions made near the root of the tree where the heuristic function is expected to be the least accurate. This behaviour is caused by line 8 of the `RecursiveBulb` procedure which uses available discrepancies as soon as possible. Also notice, that as there are no discrepancies allowed on the first iteration, this iteration will proceed identically to a traditional beam search.

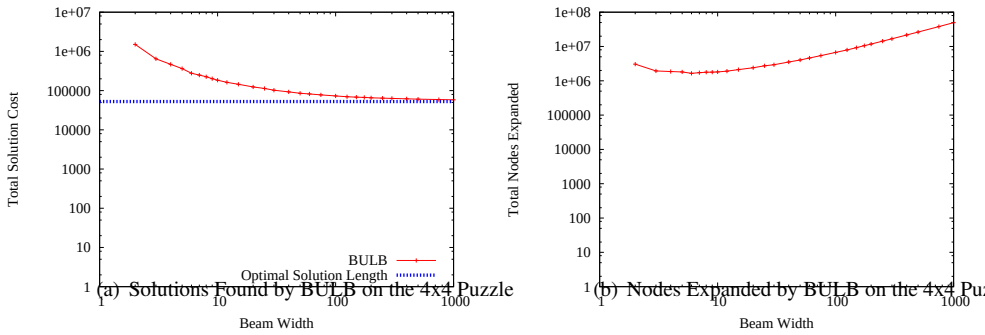
2.11.3 Properties of BULB

The main advantage that BULB has over standard beam-search is that BULB is complete. This property is ensured by the limited discrepancy backtracking.

Unfortunately, unlike WA*, WIDA*, and WRBFS, there are no guaranteed bounds on the sub-optimality of the solution returned by a BULB search. However, even without such guarantees, there are clear trends. As the beam size of a beam search approaches 1, the search degenerates into a greedy search. As the beam size approaches infinity, the search will degenerate into a breadth-first search. As such, beam search (and subsequently BULB) will tend to increase the solution quality with the beam size.

2.11.4 BULB and the Sliding Tile and Pancake Puzzles

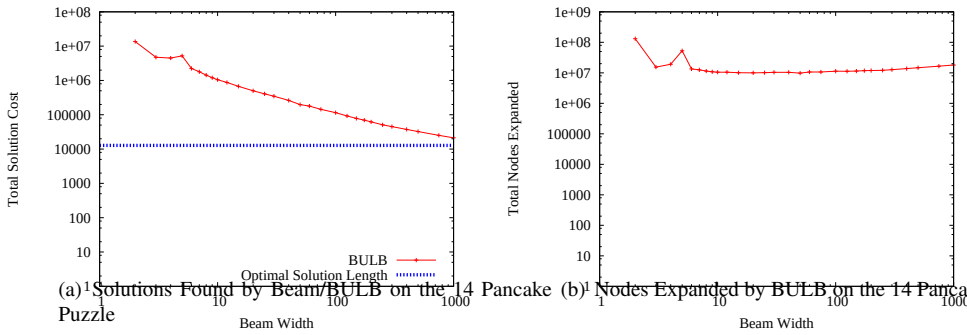
BULB was tested on both the 4×4 sliding tile and 14 pancake test sets that the other algorithms were tested on. A variety of beam widths from 2 to $1,000$ were considered. The value of L for all of these experiments was set at $50,000$. The results are seen in Figures 2.17 and 2.18. Notice that in both figures all axes are in logarithmic scale.



(a) Solutions Found by BULB on the 4x4 Puzzle (b) Nodes Expanded by BULB on the 4x4 Puzzle

Figure 2.17: BULB on $1,000$ 4×4 Sliding Tile Problems.

In the case of the 4×4 sliding tile puzzle, the algorithm never had to backtrack under the given memory constraints. As such, the results are the same as would be found by a standard beam search. The only beam width for which backtracking was necessary on the 14 pancake puzzle was that of 2.



(a) Solutions Found by Beam/BULB on the 14 Pancake Puzzle (b) Nodes Expanded by BULB on the 14 Pancake Puzzle

Figure 2.18: BULB on $1,000$ 14 Pancake Sliding Tile Problems.

In both puzzles, BULB exhibited the expected behaviour of finding poor quality solutions with small beam sizes and high quality solutions with large beam sizes. This effect was most prominent in the sliding tile puzzle for a beam size of 2 with which the average solution found was almost 29 times greater than the optimal solution length. In comparison, the beam size of 1000 found solutions only 1.11 times greater than the optimal length.

In the case of search effort, BULB reaches a minimum in the sliding tile puzzle at a beam width of 6. The minimum for the pancake puzzle was actually hit at a beam size of 2. Figures 2.17(b) and 2.18(b) show that the search effort grows significantly beyond these minimum beam widths.

2.12 Chapter Summary

This chapter began with the formalization of the single-agent search problem in Section 2.1. The algorithms of A*, IDA*, and RBFS were then introduced in Sections 2.3, 2.4, and 2.5 respectively. These algorithms are capable of finding the optimal solution paths for any single-agent search algorithm provided that the heuristic function is admissible.

Unfortunately, these algorithms often require an excessive amount of problem-solving time or have large memory requirements. One solution to this problem is the use of inadmissible heuristics which often find solutions quicker at the expense of solution quality. The idea of weighting heuristics so as to achieve an inadmissible heuristic function was then introduced in Section 2.6.

In order to demonstrate how different weights affect the search speed and solution quality found by the aforementioned algorithms, the $M \times N$ sliding tile puzzle and N pancake puzzle domains are introduced as test beds. A set of experiments are performed in these domains with the weighted variants of A*, IDA*, and RBFS in Sections 2.8, 2.9, and 2.10, respectively. These sections also include theoretical analysis of these weighted algorithms. Of particular importance is that if the weight on a admissible heuristic used in these weighted algorithms is $w \geq 1$, then the cost of any solution found will be at most w times the optimal solution cost.

The chapter concludes with a description of the beam search algorithm, as well as the complete

variant known as BULB. The performance of this algorithm in the sliding tile and pancake puzzles is then demonstrated.

Chapter 3

Configuration Selection and Dovetailing

The problem of finding an effective set of parameters for an algorithm is not specific to single-agent search. In this chapter, we examine some of the previous approaches to this problem, discuss their applicability to suboptimal search algorithms, and then describe an alternative strategy called *dovetailing*. Instead of committing to a single set of parameters, dovetailing involves the simultaneous consideration of multiple settings. The chapter will conclude with a description of the properties of dovetailing and related work.

Before continuing, it is necessary to introduce some notation, much of which is based on the work of Hutter *et al.* [26]. Let us first consider the idea of a *configuration*. A configuration θ for an algorithm a will refer to the set of all design decisions made for the implementation and application of a. These design decisions include those concerning heuristic function selection, random number generator, seeds, operator ordering, parameter values, etc. Each of these choices may have a numerical domain or be selected from a set (such as operator ordering). The instance of a with configuration θ will then be referred to as $a(\theta)$.

In many of the experiments, all of the configurations in a finite set will share many of the same design choices. For the sake of simplicity, these static choices will be omitted when describing the different configurations. For example, if θ_1 and θ_2 are two configurations of WIDA* that differ only in the weight parameter w, the configurations will be written as $\theta_1 = \{w = w_1\}$ and $\theta_2 = \{w = w_2\}$, where w_1 and w_2 are the values of the weight values in each configuration, respectively.

For any search problem p, we can now let $exp(a, \theta, p)$ denote the number of node expansions required by $a(\theta)$ during problem solving on p. We will also define the *batch results* of $a(\theta)$ over a problem set P as the total number of nodes expanded over all problems in P by $a(\theta)$. Formally, this definition is given by the following:

$$batch(a, \theta, P) = \sum_{p \in P} exp(a, \theta, p).$$

3.1 Batch Tuning

A natural approach to configuration selection is to evaluate the performance of a number of configurations on a test set and select the configuration with the best average performance. This strategy will be referred to as *batch tuning*. In general, batch tuning involves finding the configuration with the smallest batch value over a test set. Note, while any variation of batch tuning is an offline procedure, it may still require an immense cost in terms of time.

The simplest version of batch tuning involves testing all possible combinations of design choices. In the literature, this strategy is referred to as *full factorial design* [26]. Unfortunately, this approach is generally intractable due to the size of the design choice space.

An alternative approach to batch tuning is local search. Traditionally, this has been a manual process that begins with some initial configuration. The researcher then adjusts the configuration in hopes of improving performance. The perturbation of the configuration is mainly based on intuition and experience. The process continues until either no further improvement is found or some resource limit (such as time) is reached. The configuration with the best average performance is then used for future searches.

In the work of Hutter *et al.*, this local search procedure is automated [26]. The resulting procedure is called Iterative Local Search in the parameter space. In this procedure, the informed perturbation is replaced with the random selection of a neighbour of a configuration, where the neighbourhood relation is defined by the user. The natural definition of this relation, and that used in practice, is such that two configurations are considered neighbours if the two differ by exactly one design choice. For each new iteration, the configuration considered is the neighbour that improves in performance over the perturbed configuration (if such a configuration exists). Ocassionally, the algorithm will jump to a completely random configuration so as to avoid becoming stuck in a local minimum. A faster variant of this algorithm, called Focused Iterative Local Search in the parameter space, is also considered. We do not describe this algorithm in detail here.

3.2 Deficiencies in Batch Tuning

As described in Chapter 1, parameter tuning is both algorithm and domain specific. This behaviour can be seen in the experimental results presented in Chapter 2. For example, compare Figures 2.14 and 2.15 which show that WRBFS can have widely different behaviour in different application domains. As such, the offline process of parameter tuning needs to be performed independently for each individual domain.

Another issue with parameter tuning is that even within the same domain, a setting which exhibits strong performance on any one problem is not guaranteed to exhibit strong performance on all problems. Even the single parameter setting that has the lowest average amount of search effort may perform poorly on a number of individual problems.

In order to demonstrate this behaviour, consider the following example: suppose that for any set of configurations Θ, there exists a system that could indicate exactly which of the configurations in Θ required the least amount of search effort in solving some problem p. This system will be referred to as a *configuration oracle*. When solving p, the amount of search effort needed when informed by such an oracle will be denoted by $oracle(a, \Theta, p)$. This value is given by the following formula:

$$oracle(a, \Theta, p) = \min_{\theta \in \Theta} exp(a, \theta, p).$$

Similarly, $oracle(a, \Theta, P)$ will denote the total number of nodes expanded over an entire problem set P by the oracle, formally expressed as follows:

$$oracle(a, \Theta, P) = \sum_{p \in P} oracle(a, \Theta, p).$$

Now recall the experiments performed in Section 2.9.4 on the 4×4 sliding tile puzzle with the WIDA* algorithm and the weight set $\{1.0, 1.5, 2.0, ..., 24.5, 25\}$. Each of these weights forms a different configuration, the set of which we will refer to as W. A similar set of experiments were also performed on a set of 100 4×4 puzzles. This test set is composed of the solvable puzzle instances used in the original IDA* paper [30]. This test set will be used a number of times in the remainder of this thesis and will be referred to as the *Korf test set* from now on. The configuration in W which had the lowest number of total nodes expanded over all problems in this test set is the configuration with the weight of 5.5. This configuration expanded a total of $2,563,731$ nodes.

Testing these configurations involves the calculation of exp(WIDA*,w, p) for each problem p and configuration $w \in W$. Once this data has been collected, it is trivial to calculate $oracle$(WIDA*,W, p). The value of exp(WIDA*,$\{w = 5.5\}, p$)$/oracle$(WIDA*, W, p) is then an indication of how close the weight of 5.5 is to the best weight in W on problem p. If the value of this ratio is 1, then the weight of 5.5 is the best for the problem of all weights in the set. In Figure 3.1(a), for each of the 100 problems, the value of this ratio is shown. Problems are numbered in ascending order of nodes expanded on that problem by the weight of 5.5. Note the logarithmic scale of the y-axis.

Let $\theta_{5.5}$ denote the configuration $\{w = 5.5\}$. Notice that $\theta_{5.5}$ is the best configuration in W for only 3 of the 100 problems and that for 48 problems there exists a configuration in W that requires 10 times fewer node expansions than $\theta_{5.5}$. If an oracle was available it would only expand a total of $74,987$ nodes which corresponds to an expansion of 34 times fewer nodes than is done by the configuration of $\theta_{5.5}$. These facts demonstrate that there is the potential to significantly outperform the single configuration found by batch tuning if configurations could be correctly selected on a problem-by-problem basis.

The same experiment was replicated for each of WRBFS, WA*, and BULB and the results are shown in Figures 3.1(b), 3.1(c), and 3.1(d) respectively. The weight sets used for WRBFS and WA* were the same as the candidate weight set used in WIDA*, and the best weights for these algorithms

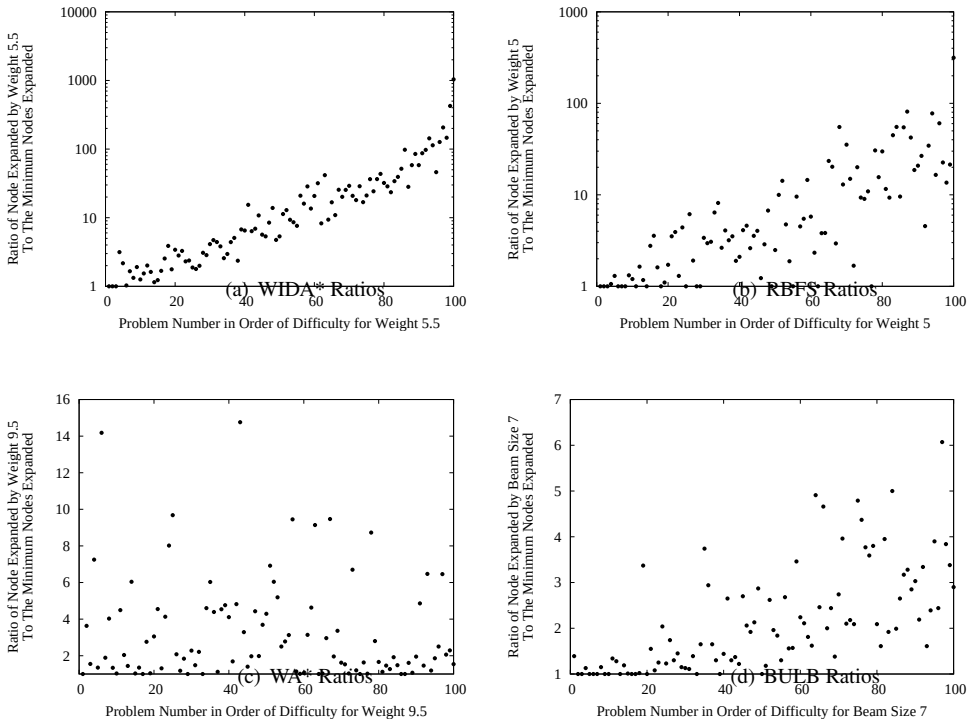


Figure 3.1: A Comparison of the Configuration with the Best Average Performance over 100 4x4 Sliding Tile Problems to an Oracle.

was found to be 5, and 9.5 respectively. The candidate beam size set is the same as that used in Section 2.11.4, with the best beam size found being 7. Note that the different figures have different scales for the y-axes.

WRBFS shows very similar behaviour as WIDA* in that the problems that were difficult for the weight of 5 were beat the most by other weights. While a similar trend is evident for BULB, the effect is not as pronounced. On the other hand, the problems which were hardest for WA* with a weight of 9.5 were not necessarily significantly outperformed by other weights.

Also notice that the effectiveness of the oracle is greatly dependent on the algorithm being used. For example, the weight of 5 was only outperformed by the oracle by a factor of at least 10 on 30 problems. The oracle was even less effective in WA* and BULB. With WA*, the oracle only outperformed the weight of 9.5 by at least a factor of 10 on 2 problems. In BULB, the oracle never outperforms the beam width of 7 by a factor of 10 on any of the problems tested.

3.3 Per-Instance Tuning

These results suggest there is potential for the use of problem-by-problem configuration selection or *per-instance tuning*. In this strategy, problem solving on a problem p begins with an information collection stage and then a configuration is selected specifically for p.

The EUREKA system is one such example of the use of per-instance tuning in single-agent search [11]. EUREKA was designed to address the fact that different problems should be solved optimally using different search parallelization techniques. For each problem, the EUREKA system builds a custom parallel version of IDA*. The system does so by collecting statistics during a breadth-first expansion of 100,000 nodes. These statistics are then fed into a decision-tree that builds a parallel IDA* instance by selecting between various methods of task distribution, load balancing, and node ordering. The decision tree is trained using a set of problem instances, each annotated with the combination of techniques found to be most effective for that problem.

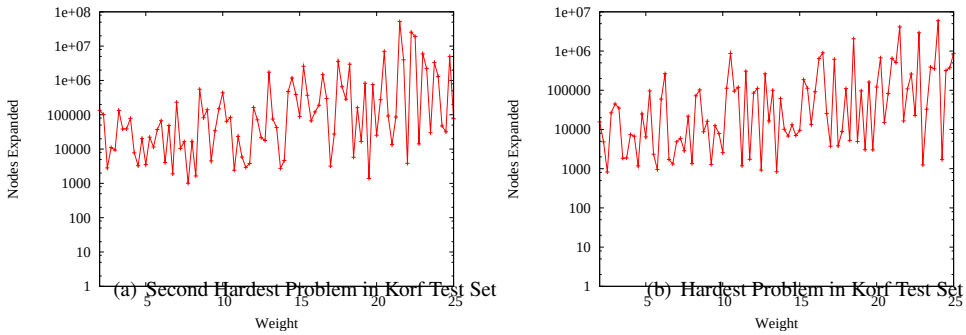
Per-instance tuning has also been successfully applied in other fields. For example, in local search SAT solvers, there is often a noise parameter which determines how often the solver makes random decisions as opposed to heuristically suggested decisions. In Auto-Walksat, a number of initial iterations are used to estimate a particular invariant which is used to find a good value for this noise parameter [36]. Other examples include the work of Horvitz *et al.* [25] which considers determining a per-problem restart policy for constraint satisfaction problems and SAT solvers, and the work of Lee and Bulitko [32] which considers the use of genetic algorithms as a way to improve the development of policies for the automatic construction of image recognition systems through the use of a classifier. A more complete description of these works is beyond the scope of this thesis.

3.3.1 Issues with Per-Instance Tuning

While the results in Section 3.2 suggest that the use of per-instance tuning in suboptimal search algorithms has significant potential, there are several issues that must be overcome before the development of any such system. The main requirement for such a system is the construction of a set of features that do well to predict the configuration to use. Finding such a set is a difficult problem, particularly if the system is expected to generalize across multiple domains.

The performance of the weighted linear-space algorithms of WIDA* and WRBFS demonstrate an additional problem. Consider Figure 3.2 which shows the nodes expanded by WIDA* when using weights in the set $\{2.0, 2.25, 2.5, 2.75, 3.0, ..., 24.75, 25.0\}$ on two 4×4 sliding tile puzzle problems. These are the two hardest problems in the Korf test set for IDA*. The figure demonstrates that the number of nodes expanded is not necessarily a smooth function of the weight. As small changes to the weight value can result in drastic changes in search effort, even very small mistakes in classification can result in poor performance. Moreover, as the peaks and valleys over the two problems do not correspond, the figure offers further evidence as to the importance of proper parameter selection.

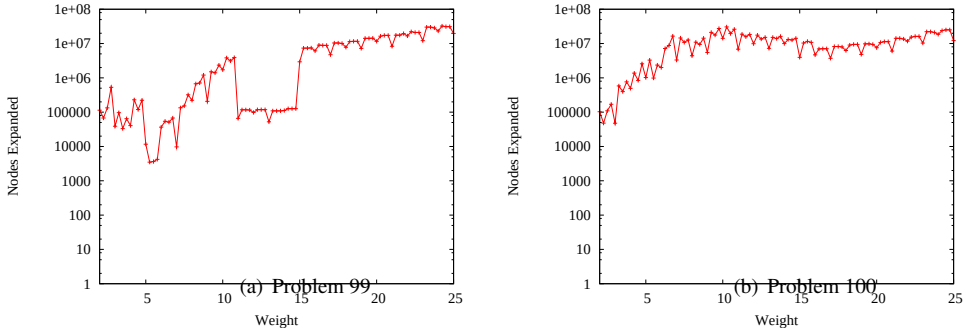
In Figure 3.3, the same experiment was run with WRBFS instead of WIDA*. While the function shape for problem 100 is smoother than that for WIDA*, there are many artifacts from the discrete nature of the domain in problem 99. Figure 3.3 also indicates that the selection of the weight set can be very important to such classification. In this case, the relationship between weight and work would appear much smoother if a coarser set of weights was initially used. This issue further



(a) Second Hardest Problem in Korf Test Set (b) Hardest Problem in Korf Test Set

Figure 3.2: The number of nodes expanded by WIDA* on two 15-puzzle problem when using weights 2 through 25 incremented by 0.25.

complicates any attempts for classification.



(a) Problem 99 (b) Problem 100

Figure 3.3: The number of nodes expanded by WRBFS on two 15-puzzle problem when using weights 2 through 25 incremented by 0.25.

While EUREKA [11] effectively uses a classifier for the selection of a configuration for parallel IDA*, several of the features used for classification were specifically designed to inform parallel configuration selection. For example, one of the features detected is the *imbalance* which is defined as a measure of how evenly the subtrees of the search tree are distributed. A second feature is the *iteration branching factor* which determines the ratio of subsequent IDA* iterations and is considered in more detail later in this thesis in Section 5.4. Both of these features are expected to significantly aid in the selection of a task distribution technique, but it is unclear if these features can inform the selection of a configuration for suboptimal single-agent search.

While per-instance tuning remains an interesting direction for future work — particularly through the use of a classifier as is done by EUREKA — it will first be necessary to resolve the issues regarding feature selection and the sensitivity of search to small changes in parameter values. As such, we will consider an alternative to both batch tuning and per-instance tuning in the next section.

3.4 Dovetailing for Single-Agent Search

In Section 3.2 it was shown that there is significant room for improvement over the use of any single parameter value as found by batch tuning. In Figures 3.2 and 3.3, we have also shown that the parameter values that require a near minimum amount of work on any single problem need not be similar. Together, these results suggest the idea of simultaneously searching the space with multiple parameter values so as to increase the probability that at least one of the values will show good performance on each problem instance. To this end, we consider the use of *dovetailing*.

Dovetailing is a strategy that takes as its input a problem p and a set of ordered pairs of search algorithms and configurations $A = \{(a_0, \theta_0), ..., (a_n, \theta_n)\}$ where for each i, θ_i is a configuration of algorithm a_i. The output of dovetailing is a solution to p. The set A is called an algorithm portfolio and each pair in A will be called a candidate algorithm.

For our purposes, we will make several simplifications. We will assume that each candidate algorithm performs the search in a series of steps and the work done during each step is comparable between algorithms. Unless otherwise stated, it will also be assumed that all candidate algorithms share the same base algorithm (*ie.* $a_0 = a_1 = ... = a_n$) and differ only in the configuration being used. As such, we will often refer to the input of dovetailing as being a *candidate set* of configurations for an algorithm a, instead of as a set of candidate algorithms.

Dovetailing is a technique by which a parallel algorithm is run on a single processor. Intuitively, dovetailing involves interleaving the work done by each algorithm. Formally, dovetailing consists of a number of rounds. Each round works as follows: each candidate algorithm will, in order, advance its search by a single step. If some candidate algorithm finds a goal on its turn, the solution found will be returned and dovetailing will stop. If a round completes without having found a solution, a new round begins. Note that during dovetailing, each of the candidate algorithms is performing a completely independent search. As such, there is no memory shared between configurations, and communication is restricted to messages indicating that a solution has been found on the current problem and the search should stop.

By having each algorithm advance by a single step during each round, dovetailing ensures that at all times, any candidate algorithm in A will have performed approximately as much work as any other. As such, the total problem-solving time taken by dovetailing on a problem p will be approximately $|A|$ times the problem-solving time of the candidate algorithm with the best performance on p.

We will also consider the parallelization of dovetailing, called *parallel dovetailing*, in which each of the candidate algorithms in A is assigned to one of $|A|$ processors each with its own memory. In parallel dovetailing, each processor will perform a completely independent search on a problem p. Communication is limited to messages indicating that p has been solved and processors should proceed to the next problem. As such, the time of a search taken by parallel dovetailing using $|A|$ processors will be approximately a factor of $|A|$ less than the time taken by dovetailing on a single

processor.

In all experiments in this thesis, the algorithms have been implemented so that each step advances the search by exactly one node expansion. Under our additional assumptions, the number of nodes expanded by dovetailing with an algorithm a over a candidate set of configurations Θ on a problem p will be at most $|\Theta| * oracle(a, \Theta, p)$. While parallel dovetailing will perform the same amount of total work over all processors as single processor dovetailing, the search time will be approximately reduced to the time needed by a single processor to expand $oracle(a, \Theta, p)$ nodes.

3.4.1 Dovetailing and Memory

Many of the properties of dovetailing will be related to the properties of the candidate configurations. For example, if each of the candidate algorithms has bounds on the solution suboptimality, the solution suboptimality of the dovetailing search will be the maximum of the individual bounds.

Similarly, the memory requirement of both parallel and single processor dovetailing is exactly the sum of the memory requirements of each of the individual algorithms. As such, dovetailing is problematic for memory intensive algorithms such as weighted A* except in parallel systems where each processor has its own memory. Systems of this type are said to have *distributed memory*.

3.4.2 Dovetailing and Diversity

If a search algorithm is misled by a heuristic it may spend a lot of time considering unneccessary areas of the state space. One approach to this issue is to expand multiple candidate paths in parallel so as to introduce diversity into the search. This is the strategy taken by beam searches and the KBFS algorithm [16]. In practice, diversity helps to decrease the probability of becoming stuck in a heuristic local minima or an area with many dead-ends.

Dovetailing will achieve diversity in search provided there is diversity in the behaviour of the candidate algorithms selected. If the algorithms all search the state space in a similar manner, any differences in search effort between candidate algorithms will be small. In these situations, any improvement made by an oracle will be overwhelmed by the cost of running multiple algorithms. For example, note that the worst case for dovetailing over k instances of an algorithm occurs when the candidate algorithms are identical, in which case dovetailing will take k times as much time as is necessary.

If the candidate algorithms do perform a diverse set of searches, there is an increased chance that at least one of these algorithms will avoid dead-ends or heuristic local minima. In this way, diversity in the candidate set allows for different configurations to overcome the weaknesses of others. It is this aspect of dovetailing that will lead to its strong behaviour in practice.

3.5 Related Work

As far as we know, dovetailing has only been previously considered for suboptimal search by Kevin Knight [28]. This work concentrates on Real-Time A* (RTA*) which is a real-time variant of A*. A key parameter for RTA* is the *lookahead value*. Increasing the lookahead value generally increases solution quality. Small lookahead values also tend to increase the stochasticity of the algorithm.

For small lookaheads, RTA* is often forced to randomly break ties between states. Knight observed that by dovetailing many instances of RTA* with a lookahead of 1, the solution quality increased dramatically over running a single instance with a lookahead of 1. Dovetailing over many instances with a lookahead of 1 also found solutions much quicker than having a single instance with a larger lookahead that achieved a similar average solution quality.

Note, in the work by Knight, the only difference in the configurations is the random number generator seed. In this thesis, we will be generalizing this idea and showing that dovetailing can be used in suboptimal search on a variety of parameters.

Parallel Window Search (PWS) is a parallel version of IDA* [39]. In this algorithm, each processor performs an IDA* search with a different cost threshold. These cost thresholds are selected so as to correspond to a consecutive set of IDA* iterations. When a processor completes its iteration, it begins again with the next smallest unexplored cost threshold. By simply returning the first solution found by any processor, PWS can be used to find suboptimal solutions.

PWS is a special case of parallel dovetailing as each processor can be thought of as being assigned an incomplete algorithm that performs a single iteration of IDA*, each with a different threshold. Since each iteration is a proper subset of all subsequent iterations, these searches will not be diverse. This lack of diversity explains why PWS with multiple processors outperforms a single-processor version of WIDA* (with weights selected so that the solution quality of the two algorithms is similar) in terms of search time but not in terms of total work.

Dovetailing is also related to the use of restarts in SAT solvers such as MiniSat [14] and Chaff [34]. These solvers perform a depth-first-like search where at each step, some variable is assigned a value. After a certain number of partial variable assignments are found to be invalid, the depth-first search restarts with all variables unassigned. Because SAT solvers can learn new constraints during search, restarts allow the decisions made near the root of the search tree to be more informed.

One of the motivations for restarts is the fact that it may take a long time to prove that choices made early in the depth-first search are poor. By restarting and instantiating with more information, this effect can be minimized. Instead of restarting the search, dovetailing approaches this problem by simultaneously searching with multiple algorithms so as to increase the probability that at least one of them will perform well — provided there is diversity in the candidate configurations.

Fast Downward is a automated planning system which uses a multi-valued state representation and a heuristic based on causal graphs [22]. Part of this system is the multi-heuristic best-first search. In this search, there are two $OPEN$ lists, each of which is ordered by a different heuristic.

The algorithm alternates between open lists when selecting nodes for expansion so as to overcome the deficiencies of each heuristic.

Both dovetailing and multi-heuristic best-first search simultaneously search the state-space with different choices. The main difference is that dovetailing completely separates the different algorithm variations while multi-heuristic best-first combines them. As multi-heuristic best-first search is a specific enhancement to A*, and we are interested in ideas that generalize across algorithms, we will not consider multi-heuristic best-first search again in this thesis.

In contrast to systems that select configurations on a per-problem basis, there are also those that dynamically alter the configuration at each step of algorithm execution. In this end, Russell and Wefald [41] considered estimating the amount of computation that may be necessary to improve upon the best move found at any time in real-time systems. In their paper, the authors develop a system which dynamically adjusts resource allocation based on this estimation. Alternatively, there is the work of Bulitko *et al.* [5] in which both subgoal selection and lookahead depth in an LRTA* search are adjusted dynamically based upon the recent history of the search. As these works perform configuration selection at a different level of algorithm execution, we will not consider them further.

3.6 Chapter Summary

This chapter began with a description of the notation necessary for the consideration of configuration selection. A first approach to this problem, batch tuning, is described in Section 3.1. Unfortunately, batch tuning suffers from the fact that the configuration which has the best average performance on a set a problems will often perform very poorly on several individual problems.

Another approach to configuration selection is per-instance tuning which is described in Section 3.3. While this remains an intriguing direction for future work, the approach is problematic since any such classifier will be both domain and algorithm specific. Moreover, the relationship between configuration settings and the search effort is not necessarily smooth which poses additional issues for classification. Despite these problems, the potential of per-instance tuning suggests that this approach remains a promising area for future work.

The strategy of dovetailing is then described in Section 3.4. This simple approach is an alternative to either batch and per-instance tuning and involves simultaneously searching with multiple configurations at once. Several properties of this approach are outlined. The chapter then concludes with a discussion of related work.

Chapter 4

Dovetailing Over the Main Parameter Spaces of Search Algorithms

This section will be concerned with dovetailing over configurations that only differ in the value assigned to the main parameter of each of the WA*, WIDA*, WRBFS, and BULB algorithms. The parameters of interest for these algorithms will be the weight in the case of the weighted algorithms, and the beam width in the case of BULB.

4.1 Experimental Design

Where Θ is a set of candidate configurations for an algorithm a, the number of nodes expanded during search when dovetailing over Θ on a problem p will be denoted by $dove(a, \Theta, p)$. The total number of nodes expanded over an entire problem set P will be denoted $dove(a, \Theta, P)$ and is calculated as follows:

$$dove(a, \Theta, P) = \sum_{p \in P} dove(a, \Theta, p).$$

Most of the experiments in this chapter and Chapter 5 were performed by simulating dovetailing as detailed below. To help demonstrate this procedure, we will also include an example.

When testing the performance of dovetailing on an algorithm a, a set of configurations Ω was initially selected. Ω will be referred to as the *starting configuration set*. For our example, the algorithm of interest will be WIDA*, and the starting configuration set will consist of 4 configurations: $\theta'_5 = \{w = 5\}$, $\theta'_{10} = \{w = 10\}$, $\theta'_{15} = \{w = 15\}$, and $\theta'_{20} = \{w = 20\}$. $\{w = j\}$ is defined as the configuration with the weight value set to j.

For some problem set P, the value of $exp(a, \theta, p)$ was found for each $p \in P$ and $\theta \in \Omega$ by running $a(\theta)$ on p. Table 4.1 shows this information for the 4 configurations in our example on the 3 easiset 4×4 sliding tile puzzle problems in the Korf test set. Notice that the number of nodes

	Solution Costs and Nodes Expanded By Each Configuration							
Problem	θ'_5		θ'_{10}		θ'_{15}		θ'_{20}	
	Cost	Nodes	Cost	Nodes	Cost	Nodes	Cost	Nodes
1	120	**3,326**	218	3,631	338	21,167	446	198,216
2	149	8,106	285	**2,984**	435	106,380	557	4,163
3	130	11,164	240	19,709	348	94,722	480	**1,279**
Totals	399	**22,596**	743	26,324	1,121	222,269	1,483	203,658

Table 4.1: The Number of Nodes Expanded by WIDA* with 4 Configurations on 3 4×4 Sliding Tile Puzzles.

expanded by the configuration which expanded the least number of nodes on each problem is shown in bold.

Once this data had been collected, dovetailing can be simulated for any set of candidate algorithms $\Theta \subseteq \Omega$. To perform a simulation of dovetailing with Θ, the configurations in Θ are given an order $\theta_1, \theta_2, ..., \theta_k$, where $|\Theta| = k$. For any problem p, the collected data was used to find the configuration θ_i such that $\theta_i = \arg\min_{\theta \in \Theta} exp(a, \theta, p)$ and where θ_i is the ith configuration in the ordering of the candidate sets. The exact number of nodes expanded when dovetailing over Θ with the above ordering is then given by:

$$dove(a, \Theta, p) = (exp(a, \theta_i, p) - 1) * k + i.$$

To see why this relation holds, consider the candidate set Θ' consisting of the two configurations of θ'_5 and θ'_{20}. The configuration ordering we will use is $\{\theta'_5, \theta'_{20}\}$. On the first problem, the configuration in the subset that expands the least number of nodes is θ'_5 which only expands $3,326$ nodes. During dovetailing, both configurations will each expand $3,326 - 1 = 3,325$ nodes without finding a solution. On the $3,326$th round of dovetailing, θ'_5 will expand a node, find the solution, and stop the dovetailing procedure. Since θ'_5 is before θ'_{20} in the ordering of the configurations, θ'_{20} will not perform a node expansion in this final round. In general, where the configuration that solves the problem is the ith in the ordering of the configurations, i nodes will be expanded during this last round. Therefore, the number of nodes expanded by dovetailing over the two configurations on this problem will be $(3,326 - 1) * 2 + 1 = 6,651$.

On the second and third problems, θ'_{20} is the best configuration of the two in Θ'. By performing the same calculation as above, we find that dovetailing over these two configurations will expand $8,326$ and $2,558$ nodes on the second and third problems respectively. Therefore, dovetailing over these two configurations will expand a total of $17,535$ nodes on this 3 problem test set.

With regards to the solution costs found using dovetailing, it should be clear that this will depend on which configuration solves each problem. With the selected candidate set, the 3 problems will be solved with length 120, 557, and 480 respectively, for a total cost of $1,157$.

Notice that the difference between the number of nodes expanded by the best ordering of the

configurations in the candidate set and the worst ordering of the set will be exactly $k-1$ on any single problem. In our experiments, $dove(a, \Theta, p)$ will usually be many orders of magnitude larger than k and so the ordering of the configurations in the candidate set will have no significant impact on search speed. As such, we will not consider the problem of configuration ordering again in either this chapter or Chapter 5 except to note the ordering used. In this chapter, configurations are ordered in ascending value of the main parameter value being considered, which will either be the weight or beam width.

The purpose of testing through simulation is that it allows us to efficiently calculate the performance of dovetailing on a large number of candidate sets. Where there are n configurations in the starting configuration set, there are $\binom{n}{k}$ possible candidate sets of size k. In practice, where $n \leq 15$, we will simulate dovetailing on all $\binom{n}{k}$ sets. For larger initial starting configuration set sizes, the number of candidate sets tested is capped at $10,000$ per candidate set size k and the subsets tested are selected randomly.

For a set of n configurations and a configuration set size of k, the average number of nodes expanded over all $\min(\binom{n}{k}, 10,000)$ candidate sets tested will be recorded, as will the number of nodes expanded by the candidate sets that have the shortest and longest search times. Going back to our example with the starting configuration set that contains θ'_5, θ'_{10}, θ'_{15}, and θ'_{20}, let us consider all $\binom{4}{2} = 6$ possible candidate sets of size 2. The candidate set with the least amount of seach time is given by $\{\theta'_{10}, \theta'_{20}\}$ which expands $15,786$ nodes. The worst configuration set is $\{\theta'_{15}, \theta'_{20}\}$ which expands $53,217$ nodes. The average number of nodes expanded over all 6 configurations is $36,553$.

Most of the figures in this chapter will depict the number of nodes expanded by dovetailing as a function of the candidate set size. For each tested candidate set size k, the figures will show the performance of the best and worst candidate sets of size k and the average performance over all candidate sets of size k tested. Note, different sized puzzles will be tested with each of the algorithms considered due to the differences in the ability of each of the algorithms to scale to larger domains. For the largest puzzle in each domain tested with an algorithm A (in which a large number of configurations capably solved all problems), we will also depict the performance of A with individual configurations and with parallel dovetailing over some starting configuration set, as a function of the solution quality achieved.

Unfortunately, the large search time needed when problem-solving in the larger domains with WIDA* and WRBFS prevented the calculation of $exp(\text{WIDA*}, \theta, p)$ and $exp(\text{WRBFS}, \theta, p)$ for all configuration-problem pairs. For example, of the weights considered for WIDA* in the 6×6 sliding tile puzzle, only the weight of 5 solved all puzzles in the problem set of size 100 after a week of computation. As several of these weights were expected to require months of problem-solving time, large-scale simulation was deemed infeasible in this domain.

Instead, a restricted set of data was collected so that dovetailing could at least be simulated over the single candidate set containing all configurations in the starting configuration set. For all config-

urations $\theta \in \Omega$, the search on a problem p was bound to expand no more than $exp(\text{WIDA*}, \{w = 5\}, p)$ nodes. If θ was unable to do so, p was recorded as unsolved by θ.

However, where $exp(\text{WIDA*}, \theta, p) > exp(\text{WIDA*}, \{w = 5\}, p)$, the actual value of $exp(\text{WIDA*}, \theta, p)$ does not matter when performing a simulation of dovetailing over any candidate set containing $\{w = 5\}$. This is because $\{w = 5\}$, if not some other faster configuration, will solve p. Therefore, dovetailing over all configurations in Ω can still be simulated. Note, we have taken this approach so that it would be possible to compare the performance of dovetailing to the performance of the configuration with the lowest average search time on the problem set (which is the configuration with the weight of 5 in this case).

Recall that when using parallel dovetailing over a candidate set Θ, the search time on any problem p will approximately be $dove(a, \Theta, p)/|\Theta|$ when $|\Theta|$ processors are available. In general, we will only describe the speedup achieved from parallel dovetailing over all configurations in the starting configuration set. This value will be used to evaluate the effectiveness of dovetailing as a parallelization technique. The one exception is in our experiments with parallel dovetailing over WA* configurations on the 4×4 sliding tile puzzle. In this domain, we have found experimental results regarding another algorithm known as wPBNF which parallelizes WA* [7]. As such, we will examine the performance of parallel dovetailing more closely in this domain and compare it to this other system.

With regards to the problems used in the experiments, in general we will consider test sets that contain $1,000$ solvable problems. In certain experiments, we have limited the test sets to the size of 100 due to the excessive time needed for the calculation of $exp(a, \theta, p)$ with all configuration-problem pairs. The 100 problems selected are merely just those in the first 100 of the larger problem sets. The one exception is in the 4×4 sliding tile puzzle, in which some experiments will be performed using the Korf test set.

4.2 Dovetailing over Weights in WA*

As mentioned in Section 3.4.1, dovetailing over k configurations is not particularly appropriate for memory intensive algorithms like WA* due to the factor of k increase in memory requirements. In the case of parallel dovetailing, these memory issues go away if the system uses distributed memory.

There has been a number of investigations into the parallelization of A* search. These include Parallel Retracting A* [15], best-first search using parallel structured duplicate detection [46], and Parallel Best-NBlock-First Search (PBNF) [8]. However, there has been little investigation into the parallelization of WA*. To the best of our knowledge, the only consideration is by Burns *et al.* [7] in which the three methods mentioned above are extended and evaluated in their application to suboptimal search using weighted heuristics. The algorithm which performed the best in almost all domains tested is the weighted variant of PBNF, denoted wPBNF.

wPBNF works by abstracting the search space into node groups called *nblocks* such that for any

node n that is abstracted to an nblock b, the abstract version of any successor of n will either be in b or a neighbour of b in the abstract space. During search, each nblock has its own $OPEN$ and $CLOSED$ lists. Due to this abstraction, separate threads can expand nodes in separate nblocks without any communication as long as the nblocks do not share neighbours in the abstract space.

An nblock b is said to be *free* if b and all neighbours of b are not being worked on by any threads. Free nblocks are stored in a heap sorted by the f-cost of the next node to expand in the $OPEN$ list for that nblock. If an nblock becomes free, it is added to the heap.

When a thread is assigned to an nblock b, it will make at least m node expansions, where m is a parameter to the algorithm. At that time, the thread will continue expanding nodes in b until the f-cost of the next node to be expanded in the nblock at the top of the heap is less than the f-cost of the next node to be expanded in b. When this occurs, the thread will be assigned to the nblock at the top of the heap.

There are a number of details that ensure the completeness of the algorithm as well as several additional pruning techniques for use with weighted heuristics. We will not discuss those here. However, we will note that the performance of wPBNF has been demonstrated to scale well with the number of processors being used in several planning domains and in the grid pathfinding domain.

In the paper on wPBNF, the algorithm is tested on the easiest 43 problems in the Korf test set. In this domain, all of the algorithms show poor performance for larger weights and larger numbers of processors. While we have yet to perform a complete comparison of the algorithms, in Section 4.2.2 we will analyze the relative performance of wPBNF and parallel dovetailing on this domain based upon the information in this paper.

In the case of single-processor dovetailing, configurations in the starting configuration set will all have some memory limit L. The performance of dovetailing will then be compared to the performance of each of the individual configurations. While a fairer comparison would be between dovetailing over such candidate sets of size k and the performance of individual configurations with the memory limit of kL, the results below will demonstrate that even in such favourable conditions, dovetailing is a poor addition to WA* in the domains considered.

4.2.1 Dovetailing over Weights with WA* on the Sliding Tile Puzzle

Recall Figure 2.9 which showed the performance of WA* in terms of both search time and solution quality on $1,000$ 4×4 sliding tile puzzles. For the dovetailing simulations on both this puzzle and the 4×5 sliding tile puzzle considered later in this section, the starting configuration sets were selected so as to consist of the first 15 integer weights that successfully solved all $1,000$ problems under the memory limit of $1,000,000$ states. In the case of the 4×4 sliding tile puzzle, these weights are those integers in the range of 3 to 17 inclusive.

Figure 4.1 shows the performance of dovetailing over all subsets of this starting configuration set. Note, the number of nodes expanded by the weight with the lowest batch results — the weight

of 12 — is also shown in the figure. When using all 15 configurations, 5.8 times more nodes are expanded than are expanded by the weight of 12 alone. While the average solution quality found through dovetailing is closer to that found by the weight of 6, the number of nodes expanded is still 4.4 times larger than even this weight alone.

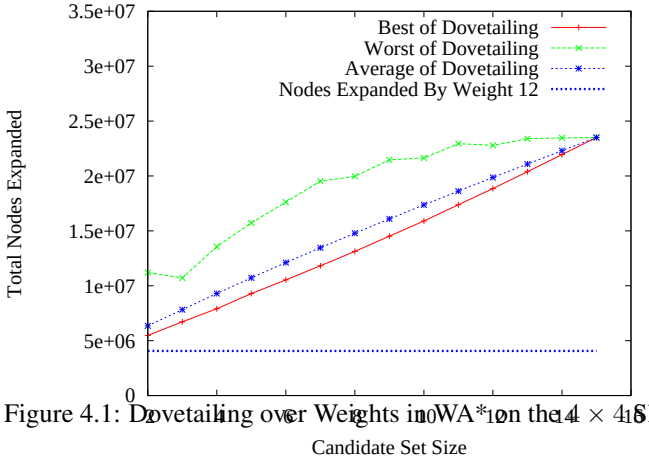


Figure 4.1: Dovetailing over Weights in WA* on the 4×4 Sliding Tile Puzzle.

Figure 4.2 shows the performance of WA* on $1,000$ 4×5 sliding tile puzzle problems. In the figure, the first 15 integer weights that completely solved all states in the problem set are shown. These weights are those at least as large as 5 and no larger than 19. Figure 4.3 shows the results of the dovetailing simulations in which these 15 weights are used to form a starting configuration set. The number of nodes expanded by the single weight with the best batch results — namely the weight of 18 — is shown in the figure.

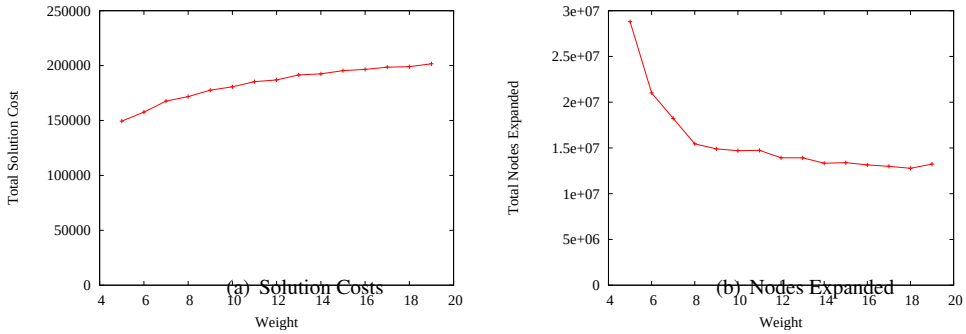


Figure 4.2: WA* on 1000 4×5 Sliding Tile Puzzles

The figure demonstrates that dovetailing remains an ineffective procedure even in the larger domain. Dovetailing over all 15 configurations requires 4.8 times as many node expansions as the best weight alone. Again, the average solution quality found is better than the weight with the best

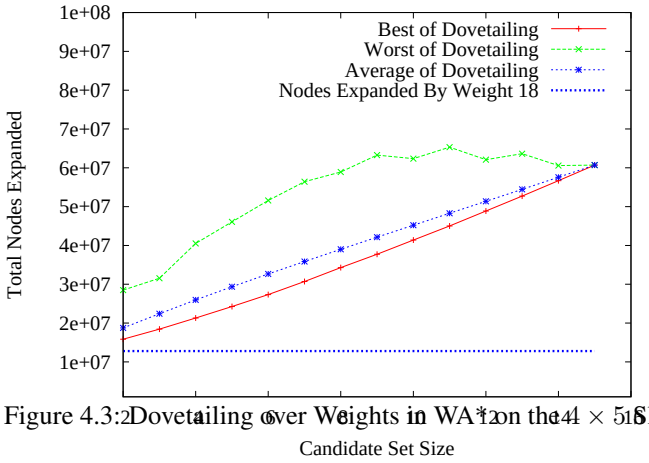


Figure 4.3: Dovetailing over Weights in WA* on the 4×5 Sliding Tile Puzzle.

batch results. The solution quality is more similar to the weight of 8, but the dovetailing procedure still requires the expansion of 3.9 times more nodes than this weight alone.

These results should be expected when considering Figure 3.1(c) which demonstrates that even an oracle does not significantly outperform the weight with the best batch value. This is due to the behaviour shown in Figures 2.9(b) and 4.2(b) which indicate that increasing the weight almost monotonically improves the search speed, and this is similarly true on individual problems. A higher weight causes the search to favour node expansions in the search tree that are deeper and in areas with a low heuristic value. As there is a solution below any branch in the search tree of these domains, this behaviour of WA* is favourable in these environments. The largest valued weight in the candidate set with therefore rarely be significantly outperformed by any other weight on a problem-to-problem basis.

4.2.2 Parallel Dovetailing over Weights with WA* on the Sliding Tile Puzzle

In this section, we will consider the performance of parallel dovetailing by first comparing this technique to wPBNF. The experiments in the wPBNF paper were performed with 16 GB of shared memory. The test set used consisted of 43 4×4 puzzle problems from the Korf test set that were solvable by A* with this memory limit.

We do not know the exact set of problems used in their experiments and the set of problems which satisfy this condition well depend on their implementation of A*, and so we will use the 43 problems that were found to be easiest to solve by IDA*. We suspect that this set is similar to the set used in the wPBNF experiments. Moreover, we restrict ourselves to these 43 problems since wPBNF is said to perform better in comparison to WA* on the harder problems. Including harder problems may therefore unfairly favour parallel dovetailing when it is compared to wPBNF.

The experiments with parallel dovetailing will also be performed through simulation and we

	Speedup With Different Numbers of Processors							
	wPBNF				Parallel Dovetailing			
Weight	2	4	5	8	2	4	5	8
2	0.37	0.62	1.34	1.46	6.42	9.0	9.85	11.7
3	0.74	0.62	0.9	0.78	1.71	2.4	2.6	3.12
5	0.6	0.76	0.72	0.64	1.58	2.22	2.43	2.88

Table 4.2: The Speedup of wPBNF and the Average Speedup of Parallel Dovetailing on 43 4×4 Sliding Tile Puzzle Problems.

will consider both the distributed and shared memory situations. The starting configuration set used consists of the 15 configurations, each with a different integer weight in the range of 2 to 16 inclusive. For the distributed memory experiments, each processor will be assumed to have enough memory to store $1,000,000$ nodes. In the case of shared memory, the total memory among processors will be $1,000,000$ nodes. Therefore, when collecting the initial data for the experiments with k processors, the searches were forced to expand no more than $1,000,000/k$ nodes. Note, this is a significant handicap when compared to the memory limits imposed upon wPBNF which we speculate allowed their system to hold at least $50,000,000$ at a time.

The behaviour of the algorithms is shown in Table 4.2. For wPBNF, the table shows for each weight w, the speedup factor achieved from the weight w wPBNF search when compared to a serial WA* search with weight w. The total time taken by wPBNF is only less than that taken by WA* if the speedup is greater than 1. Moreover, larger factors of speedup imply shorter search times. Note, these numbers are taken from the combination of a conference paper [7] and a workshop paper [6]. Where the numbers conflict, the results from the more recent conference paper are shown.

For parallel dovetailing, almost all the weights in the starting configuration set were able to solve all 43 problems even with only being able to store $125,000$ nodes (which corresponds to 8 processors sharing the $1,000,000$ node limit). The only exception was the weight of 2. However, any candidate set containing the weight of 2 will also contain some other weight which will be able to handle problems unsolved by 2. As such, the results in the table correspond to both memory architectures since the experiments for both distributed and shared memory performed exactly the same. For each weight w and number of processors k, the table shows the average speedup of parallel dovetailing over all $\binom{15}{k}$ possible candidate configurations when compared to a sequential WA* search with a weight of w. Note, the number of nodes expanded by parallel dovetailing does not change within the same column. The speedup changes because the number of nodes expanded by the weight it is being compared against changes.

The figure shows that wPBNF does not offer much speedup over serial WA*, particularly for larger weights and numbers of processors. The authors believe that in this domain the overhead of communication is not overcome for larger weights because WA* alone requires so few node expansions. However, parallel dovetailing over weights does improve upon the performance of WA*

even if the value of w is larger. It should be noted that the inclusion of high weights in the candidate sets causes the average solution quality to suffer. While the average solution quality does improve slightly as the number of processors increases, the average solution length is generally similar to that found with a weight 5 WA* search. When using 2 processors, the solution quality is 2% worse than the weight of 5. When using 8 processors, the solution quality is 5% better. However, this solution quality is also 23% and 16% worst than the weight of 2 when using 2 and 8 processors respectively.

We can also compare parallel dovetailing against the performance of the best configuration in the starting configuration set with a distributed memory architecture. We will do so in the 4×4 and 4×5 sliding tile puzzles with the $1,000$ puzzle problem sets. The starting configuration sets are the same as those considered in the dovetailing experiments at the beginning of this section. When using 15 processors, we find that the search time is decreased by 2.28 in the 4×4 sliding tile puzzle when compared to the single weight with the best batch results on a single processor.

In the case of the 4×5 sliding tile puzzle, consider Figure 4.4 which shows the performance of WA* and parallel dovetailing over WA* configurations for all candidate set sizes tested as a function of the solution quality. Each WA* point corresponds to the total number of nodes expanded over the entire problem set when using a single different configuration. Each dovetailing point corresponds to using a different candidate set size k, where each configuration is assigned to one of k processors. In the case of the parallel dovetailing points, we show the number of nodes expanded by each processor over the entire problem set. Since there is minimal communication between processors, this value will be proportional to the total run-time of parallel dovetailing.

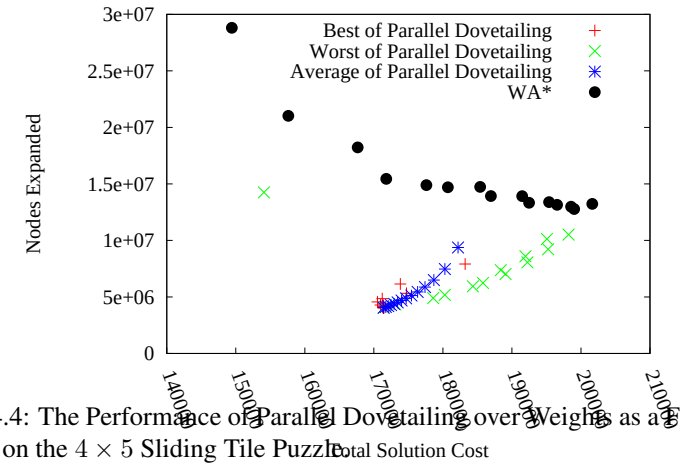


Figure 4.4: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WA* on the 4×5 Sliding Tile Puzzle

The figure shows that the average solution quality achieved by dovetailing actually outperforms many individual configurations, and that even parallel dovetailing over the worst of all considered candidate sets will usually be faster than any individual configuration alone. When parallel dove-

tailing with 15 processors over all 15 configurations, the speedup is by a factor of 3.18.

4.2.3 Dovetailing over Weights with WA* on the Pancake Puzzle

In Figure 2.10, the performance of WA* on the 14 pancake puzzle using the heuristic given by the $< 0, 1, 2, 3, 4, 5, 6 >$ pattern database was summarized. However, all weights greater than or equal to 10 performed exactly the same. Therefore, having multiple weights with a value of at least 10 in any candidate set will cause a duplication of search effort due to the lack of diversity. The fact that such performance can be seen in practice suggests that the selection of good candidate sets is an important problem. This issue will be discussed in Chapter 6.

Due to the equivalence in the search effort of different weight values, the starting configuration set selected for the dovetailing simulations was set as the 8 configurations that differ only in weight value, with each being assigned a different integer weight in the range of 3 to 10. While the larger weights in this set do perform similarly, they do not produce identical searches. However, as can be seen in Figure 4.5, the performance of dovetailing over these configurations in the 14 pancake puzzle is poor. For example, dovetailing over all 8 weights in the starting configuration set requires 6.4 times as many node expansions as the weight of 10 alone.

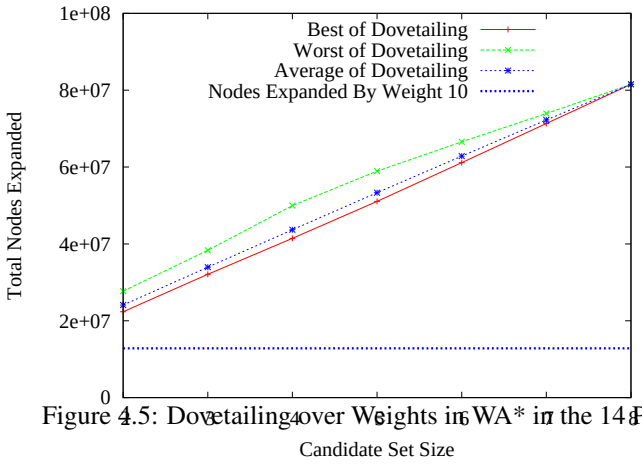
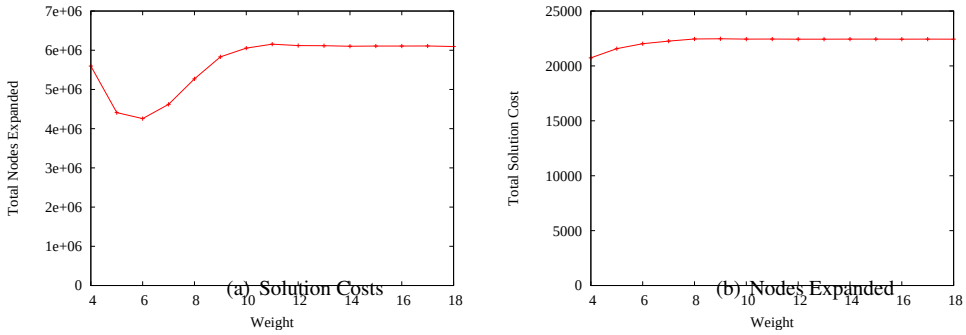


Figure 4.5: Dovetailing over Weights in WA* in the 14 Pancake Puzzle.

Concerning solution quality, the average length of solutions found when dovetailing over all 8 weights is similar to that found by the weight of 4 alone. Unfortunately, dovetailing over all 8 weights still requires 5.4 times as many node expansions as the weight of 4.

The performance of dovetailing over WA* configurations was also considered for the 16 pancake puzzle. The heuristic function used in these experiments is given by the maximization of the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ pattern databases. Figure 4.6 shows the performance of the first 15 integer weights found that successfully solved all $1,000$ problems in the test set. Note that in this case, WA* does not smoothly improve performance with the weight and actually finds a minimum

at the weight of 6.



(a) Solution Costs (b) Nodes Expanded

Figure 4.6: WA* on $1,000$ 16 Pancake Puzzles

For the 16 pancake puzzle, the starting configuration set includes the integer weights from 4 to 18, inclusive. The results of these simulation experiments are shown in Figures 4.7 which demonstrates that dovetailing remains an ineffective enhancement to WA* even in the larger 16 pancake puzzle. In this case, dovetailing over all 15 weights requires 9.7 times more search effort than the best single weight of 6, and 9.3 times more nodes than the weight of 5 alone, with which dovetailing finds the most similar average solution quality.

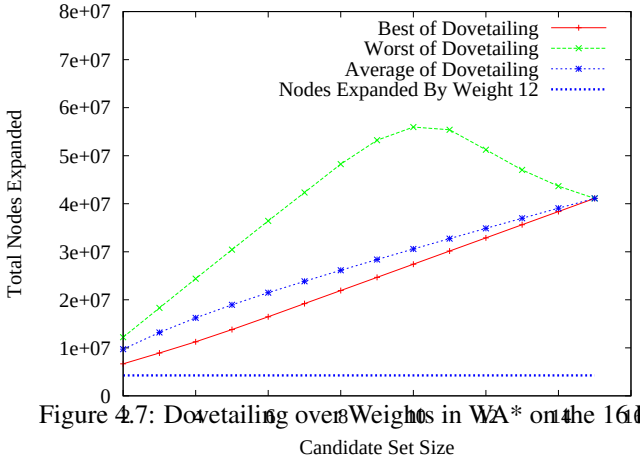


Figure 4.7: Dovetailing over Weights in WA* on the 16 Pancake Puzzle.

Figure 4.8 shows the performance of parallel dovetailing when compared to the individual configurations alone, and indicates that dovetailing achieves a solution quality that is significantly better than all but two of the configurations considered. These results indicate that only modest speedups are found when using parallel dovetailing over WA* configurations as a parallel search algorithm in the pancake puzzle domain. In the case of the 14 pancake puzzle, the speedup found is only 1.3 when using 10 processors. In the 16 pancake puzzle, the speedup is only 1.6 when using 15

processors.

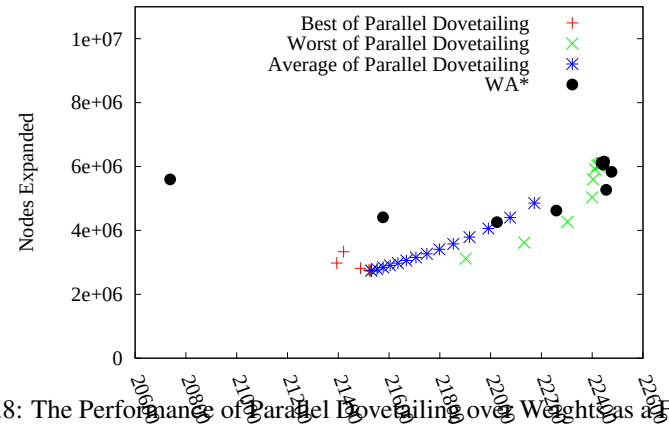


Figure 4.8: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WA* on the 16 Pancake Puzzle.

The increased ineffectiveness of dovetailing in the pancake puzzle domain when compared to the sliding tile puzzle domain may be related to the fact that solutions to pancake problems are much shorter than those for the sliding tile problems. As there is also guaranteed to be a solution under any branch in the search tree, the use of any high weight again proves to be an effective strategy.

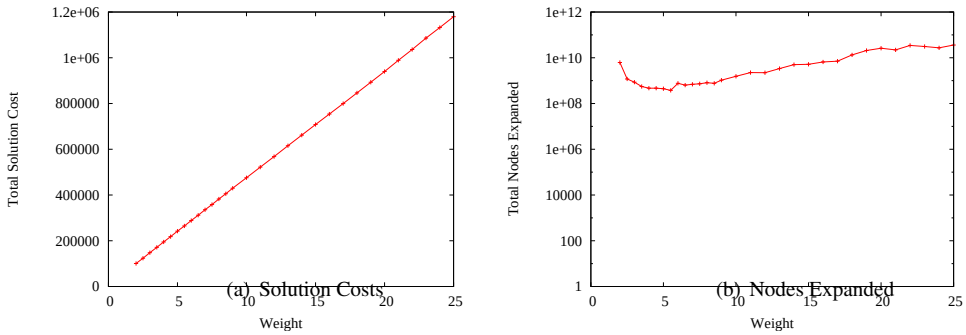
4.3 Dovetailing and WIDA*

In this section, we consider the effectiveness of dovetailing as an enhancement to WIDA*. As the memory requirements of WIDA* are small, comparisons can be made between the performance of single configurations to dovetailing without any concern as to the difference in resources used by the different problem-solving methods. It is only on excessively large candidate sets that memory may become an issue. However, it is expected that the overhead of simultaneously running so many instances will prohibit the use of dovetailing over WIDA* instances long before memory does. In this thesis, we will only be using candidate sets no larger than 24 and so this issue will not be considered further.

Below we will show that parallel dovetailing offers a very simple and effective form for parallelizing. While there has been much investigation into the parallelization of IDA* in the search literature, we are unaware of any work regarding parallelizing WIDA*. Additionally, we are similarly unaware of any consideration as to how well the techniques parallelizing IDA* generalize to the weighted variant of the algorithm. As such, we have not compared parallel dovetailing to any other parallel technique.

4.3.1 Dovetailing over Weights with WIDA* on the Sliding Tile Puzzle

As WIDA* has proven to be an effective algorithm for solving permutation puzzles, the experiments in this section will be on larger puzzles than those tested in Section 4.2.1. We will first consider the use of dovetailing in the 4×5 puzzle. Figure 4.9 shows the performance of WIDA* in terms of both search effort and solution quality in this environment. Note that in Figure 4.9(b), the y-axis is shown in a logarithmic scale.



(a) Solution Costs (b) Nodes Expanded

Figure 4.9: WIDA* on $1,000$ 4×5 Sliding Tile Puzzles

Figure 4.10 shows the behaviour of dovetailing when applied to this puzzle. The starting configuration set consists of the 15 configurations that differ only in that each is assigned a different integer weight in the range of 2 to 16. The number of nodes expanded by the best weight in the starting configuration set, that of value 5, is also shown in the figure.

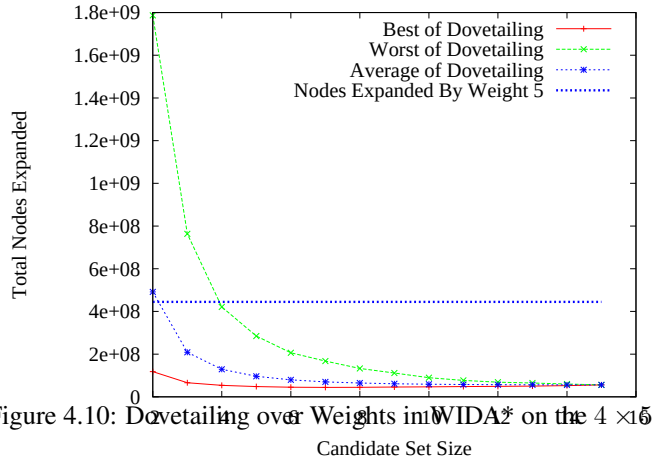


Figure 4.10: Dovetailing over Weights in WIDA* on the 4×5 Sliding Tile Puzzle.

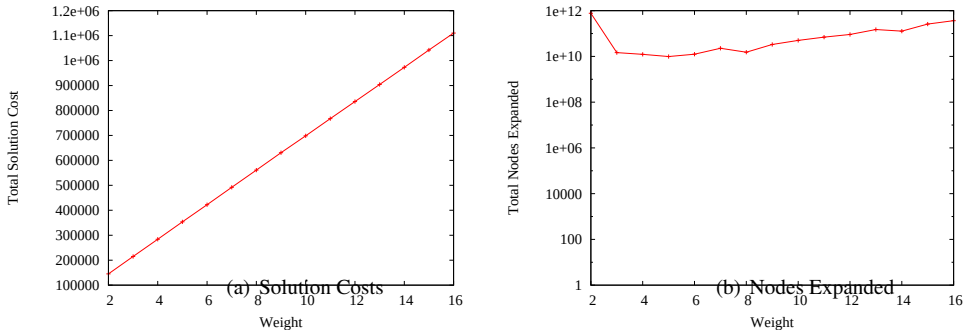
The figure indicates that dovetailing over WIDA* configurations that differ only in weight significantly improves over even the single best configuration alone. In this case, if the candidate set size is at least 4, even the worst of the $\binom{15}{4}$ candidates sets outperforms the weight 5 search. If

the candidate set contains all 15 configurations, the improvement in terms of search effort over the weight of 5 alone is by a factor of 7.9. With regards to solution quality, dovetailing over all 15 configurations results in a total cost that is similar to that found by the weight of 8.5.

Unlike WA*, in which the efficiency of search degraded when used with dovetailing, WIDA* shows significant speed increases when enhanced with this technique. This is because of the behaviour seen in Figure 3.1(a) which shows that the weight with the best average performance will often exhibit poor performance on a certain number of problems. This was not the case with WA*, in which increasing the weight generally improved search speed on a problem-by-problem basis.

These results indicate that in this domain, parallel dovetailing exhibits *super-linear speedup* in search time. A parallelization of an algorithm exhibits super-linear speedup when the use of k processors results in a speedup in the running time that is greater than k. In the case of parallel dovetailing over all 15 configurations in this starting configuration set on the 4×5 sliding tile puzzle, the speedup is by a factor of 119.21. Note that whenever a single-processor dovetailing outperforms the best configuration in the candidate set, then parallel dovetailing will necessarily be exhibiting super-linear speedup in that domain. This is true for any algorithm in all experiments in this thesis.

The performance of WIDA* on $1,000$ 5×5 puzzle problems is shown in Figure 4.11. In this case, the single weight with the fastest average search time is the weight of 5. Note that the y-axis in Figure 4.11(b) is also shown in a logarithmic scale.



(a) Solution Costs (b) Nodes Expanded

Figure 4.11: WIDA* on $1,000$ 5×5 Sliding Tile Puzzles

When the dovetailing simulations are performed on these 5×5 sliding tile puzzles while using the same starting configuration set as above, the performance improvements are even more impressive. These results are shown in Figures 4.12. The performance of parallel dovetailing, in which each configuration is assigned to a different processor, is shown in Figure 4.13. In this figure, each WIDA* point corresponds to a different single configuration run of WIDA*. Each dovetailing point corresponds to a different candidate set size. Notice that the scale of the y-axis in both figures is logarithmic.

In this puzzle, the candidate set sizes must be 4 and 6 respectively before the average and worst candidate sets exhibit better performance than the weight of 5 alone. However, when the candidate

set contains all 15 configurations, the improvement over the use of the single best configuration is greater than the improvement found in the 4×5 puzzle. Specifically, dovetailing over all 15 configurations performs 42.5 times fewer node expansions than the weight of 5 alone. The solution quality found by such a search is similar to the solution quality found by the weight of 9 alone. With regards to parallel dovetailing, when using all 15 configurations each on a different processor, the speedup is by a factor of 637.

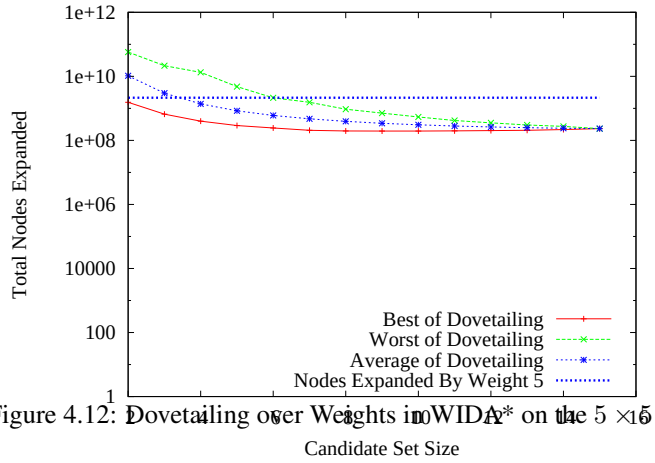


Figure 4.12: Dovetailing over Weights in WIDA* on the 5×5 Sliding Tile Puzzle.

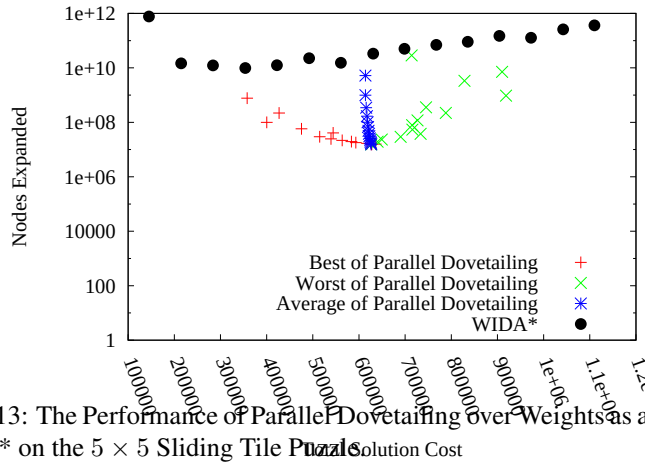


Figure 4.13: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WIDA* on the 5×5 Sliding Tile Puzzle.

Figure 4.13 shows that the candidate sets with the worst performance are those with poor solution quality. This is because these candidate sets generally consist mostly of high weights. While parallel dovetailing over any such candidate set will outperform all the weights in the set alone, there are too many problems in which all weights perform poorly. This causes these candidate sets to exhibit

slower search speeds when compared with other candidate sets.

Just as the effectiveness of dovetailing increased from the 4×5 sliding tile puzzle to the 5×5 sliding puzzle, a similar trend is seen when dovetailing over WIDA* configurations is considered for the 6×6 puzzle. This puzzle proved to be too large to collect all the necessary data needed to perform the simulation experiments even when the problem set was decreased to a size of 100. However, as described in Section 4.1, it is still possible to calculate the number of nodes expanded over the 15 candidate weights in the range from 2 to 16.

For each problem p in the problem set, Figure 4.14 shows the value of $dove(\text{WIDA*}, \Theta, p)$ and $exp(\text{WIDA*}, \{w = 5\}, p)$ where Θ is the set of 15 configurations described above. The configuration of the weight of 5 was the only configuration in the candidate set that successfully solved all 100 problems in the time allotted, and can therefore be considered the configuration with the best average search time. Note that the y-axis in Figure 4.14 is in a logarithmic scale and that problems in the set P are ordered in ascending value of $exp(\text{WIDA*}, \{w = 5\}, p)$.

In total, dovetailing over the 15 configurations expands 121 times fewer nodes than the configuration of $\{w = 5\}$ alone. For parallel dovetailing over the candidate set Θ, the factor of improvement in terms of search time is 1826. This result offers further evidence that the improvement offered by dovetailing over weight sets in WIDA* scales in the sliding tile puzzle domain along with the state space size.

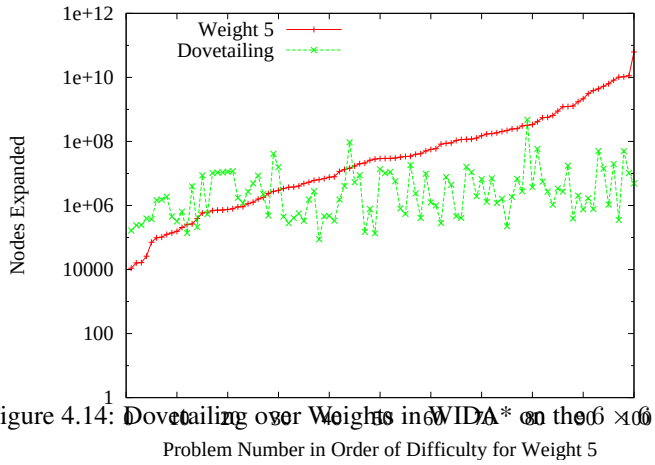


Figure 4.14: Dovetailing over Weights in WIDA* on the 6×6 Sliding Tile Puzzle.

Another question that arises from these results is as to how much this enhancement depends on the starting configuration set. In this end, we will consider dovetailing with 3 different starting configuration sets. The first will be the starting configuration set considered above — specifically, the set containing one configuration for each of the integer weights in the range from 2 to 16. This set will be referred to as the *consecutive integer set*. The *narrow set* will consist of configurations each with a unique weight value from the set $W_{narrow} = \{2, 2.5, 3, 3.5, ..., 8.5, 9\}$. The *wide set*

will consist of configurations, each with a unique weight value from the set W_{wide} where W_{wide} is the union of $W_{even} = \{2, 4, ..., 22, 24\}$ and $W' = \{7, 13, 21\}$. The wide set consists of all even weights in the range 2 to 24, as well as 3 additional equally spaced odd weights. Notice that all three starting configuration sets include exactly 15 configurations.

For any candidate set size k, Figure 4.15 shows the average performance over all $\binom{15}{k}$ possible candidate sets by each of the narrow, consecutive integer, and wide sets. The performance of the best weight alone of all those considered, namely that of weight 5.5, is also shown in the figure. The figure demonstrates that regardless of the starting configuration set, the average results achieved with dovetailing significantly outperforms the weight of 5.5 once the size of the candidate sets is large enough. Also notice that for larger candidate set sizes, the performance of the 3 starting configuration sets is quite similar.

When comparing the solution quality achieved by the different sets, the results are unsurprising: sets with a lower average weight outperform those with a higher average weight. However, for every single candidate set size, the narrow set also outperformed both the consecutive integer and wide sets in terms of nodes expanded. Similarly, the consecutive integer set outperformed the wide set on every candidate set size.

This effect is most pronounced for small candidate weight sizes because the average is skewed by a few bad candidate sets. For example, in the wide set, the candidate set of size 2 which had the worst performance was that of weights 21 and 22. As shown in Figure 4.9(b), these weights alone require a large number of node expansions. While dovetailing will allow weight 22 to compensate for some of the errors that weight 21 makes when navigating its search tree (and vice versa), there will still be too many problems in which both exhibit poor performance. In the narrow and consecutive integer sets, there are fewer weights with such poor performance and so this behaviour occurs less often.

However, even for the candidate set size of 15 the narrow set still outperforms the others. This behaviour can again be explained by the fact that the narrow set contains the highest number of weights with good individual performance. While the weights in this set are the closest together of any of the sets, there is evidently enough diversity in the search trees induced by these weights such that they complement one another well.

4.3.2 Dovetailing over Weights with WIDA* on the Pancake Puzzle

In this section, dovetailing over WIDA* configurations that differ only by weight will be applied to the 14 pancake puzzle and the 16 pancake puzzle. The heuristic functions used for these puzzles will be those given by the pattern database $< 0, 1, 2, 3, 4, 5, 6 >$, and the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ databases respectively.

The performance of WIDA* on the 14 pancake puzzle with the $< 0, 1, 2, 3, 4, 5, 6 >$ pattern database heuristic function has already been shown on in Figure 2.13. The results of the dovetailing simulations on the 14 pancake puzzle are shown in Figure 4.16. In this case, the average performance

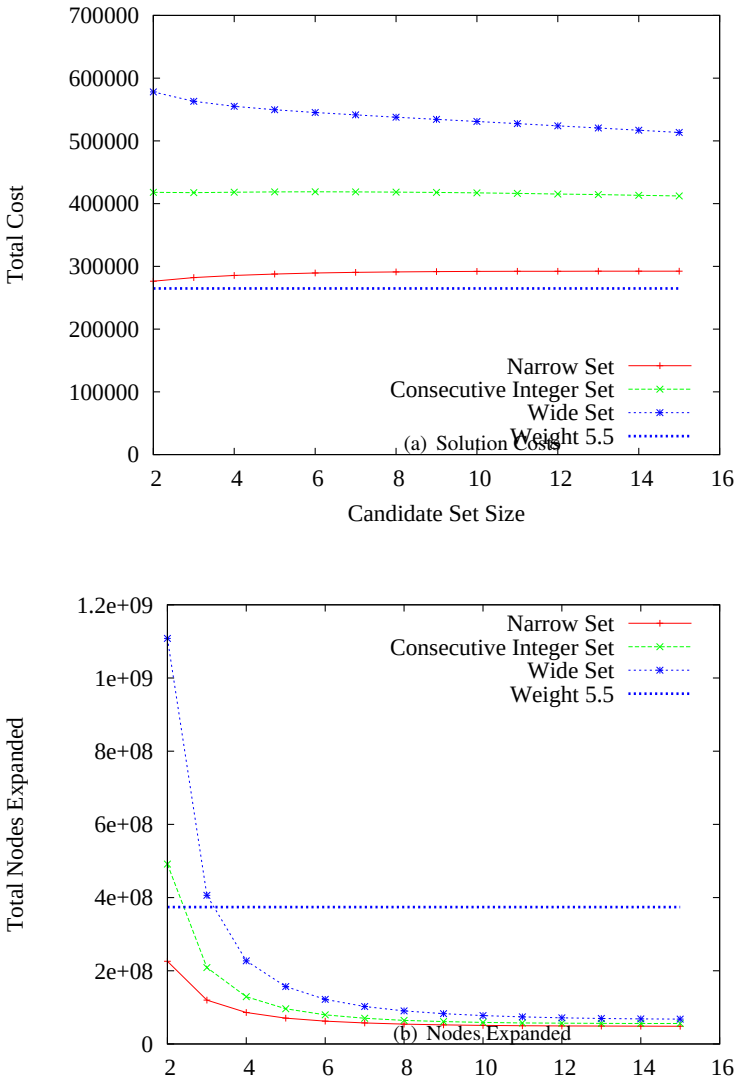


Figure 4.15: The Performance of Different Starting Configuration Sets on the 4×5 Sliding Tile Puzzle.

of dovetailing bests the single best weight as soon as the candidate set size is at least of size 3. When dovetailing is performed over all 15 configurations, dovetailing improves upon the single best weight of 9 by a factor of 1.5. In this case, the solution quality found with dovetailing is very similar to that found by the weight of 9. Also note that in this puzzle, all of the worst candidate sets include the weight of 2 which suggests that this weight is not contributing much to the configuration portfolio.

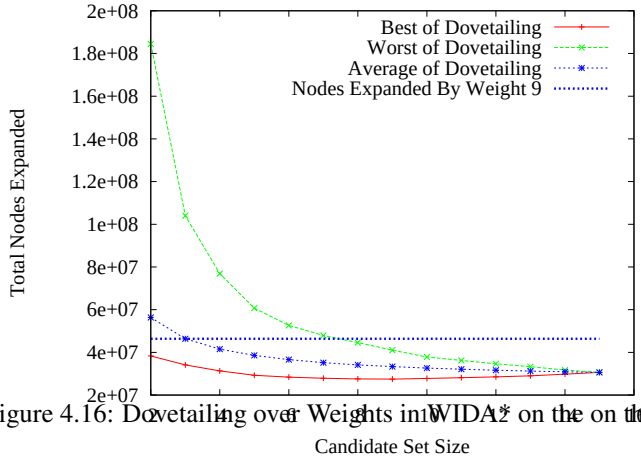
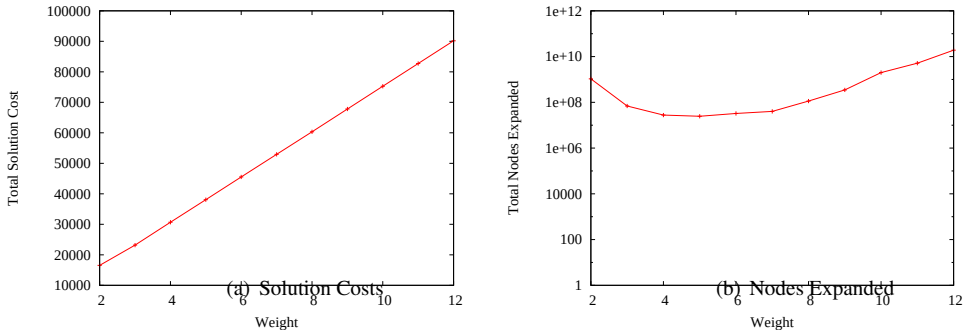


Figure 4.16: Dovetailing over Weights in WIDA* on the 14 Pancake Puzzle.

The performance of WIDA* on the 16 pancake puzzle is shown in Figure 4.17. Note, the figure only includes data for integer weights from 2 to 12 due to that fact that it took too long to calculate all the necessary data for larger weights. Among the weights shown, the weight of 5 was the most effective.



(a) Solution Costs (b) Nodes Expanded

Figure 4.17: WIDA* on 1,000 16 Pancake Puzzles.

Dovetailing exhibits similar strength when applied to the 16 pancake puzzle. Figure 4.18 shows the results of the dovetailing simulations on the 16 pancake puzzle. Figure 4.19 shows the performance of parallel dovetailing as a function of solution quality. When dovetailing over the 11 configurations, each with a unique integer weight from 2 to 12, the average performance of dovetail-

ing surpasses the weight of 5 when the candidate set size is at least of size 5. The number of nodes expanded by dovetailing over all 11 configurations is 1.9 times lower than the number expanded by the weight of 5 alone. Dovetailing in this way also results in a solution quality similar to that found by a weight of 7. Notice that again, the worst candidate sets tend to be those with a poor solution quality because they contain mostly high weights.

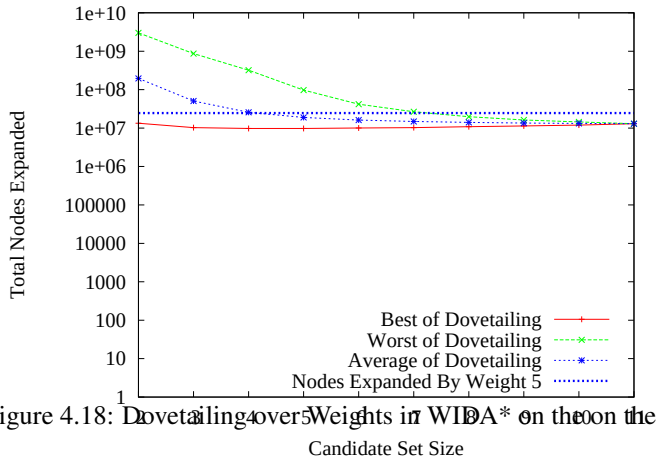


Figure 4.18: Dovetailing over Weights in WIDA* on the 16 Pancake Puzzle.

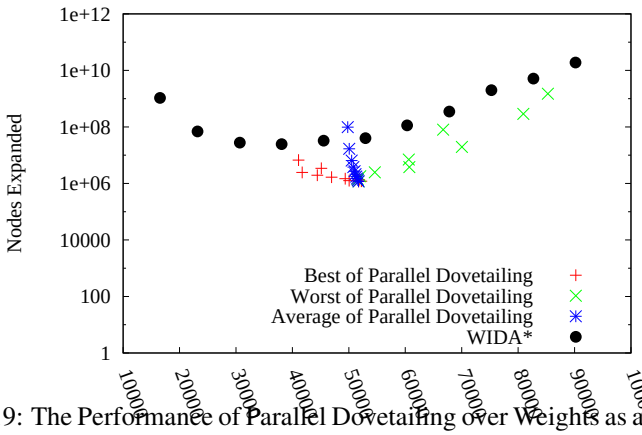


Figure 4.19: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WIDA* on the 16 Pancake Puzzle.

While Figure 4.19 indicates that parallel dovetailing is achieving super-linear speedups with the 11 configurations mentioned above, dovetailing was also simulated over all 15 integer weights from 2 to 16 on the 16 pancake puzzle as described in Section 4.1. Dovetailing in this way outperformed the weight of 5 by a factor of 1.8 and had an average solution quality similar to that found by the weight of 6. This performance indicates that parallel dovetailing will exhibit super-linear speedups

and will therefore be a useful way of parallelizing WIDA* in this domain even when the candidate set size is allowed to grow to 15. This is similarly true on the 14 pancake puzzle.

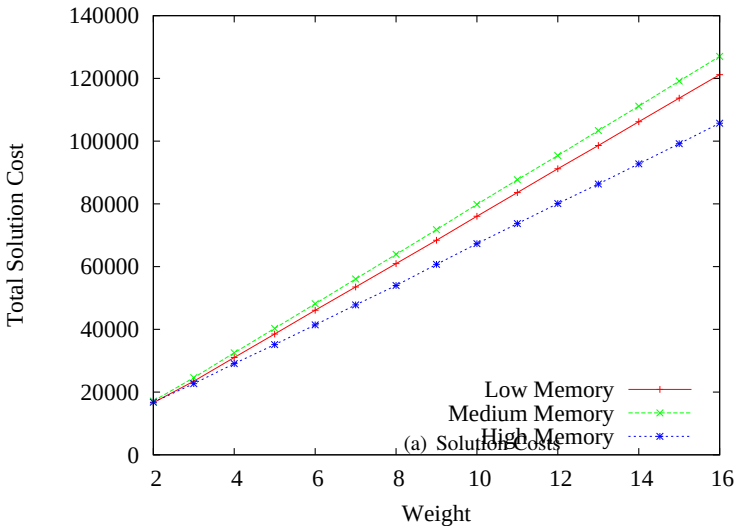
As the use of abstraction introduces the opportunity to test with many different heuristic functions, an interesting question emerges: how is the effectiveness of dovetailing related to the accuracy of the heuristic? To address this issue, we consider solving the 16 pancake puzzle with 3 different heuristics. The first function is given by the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9, 10 >$ pattern databases. The second heuristic function considered is the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 5, 6, 7, 8, 9, 10 >$ pattern databases. The final heuristic function considered will be the maximization over the $< 0, 1, 2, 3, 4, 5 >$, $< 5, 6, 7, 8, 9, 10 >$, and $< 10, 11, 12, 13, 14, 15 >$ pattern databases.

These heuristic functions have been ordered in ascending order of memory used and hence will be referred to as the *low memory*, *medium memory*, and *high memory* heuristic functions, respectively. Typically, an increase in memory used in a pattern database will increase the accuracy of the corresponding heuristic function. This is certainly true in this case as the lower memory abstractions are subsets of the higher quality ones. To demonstrate the difference in the accuracy of the functions, consider Figure 4.20 which compares the performance of WIDA* when using these three different heuristics with all of the integer weights from 2 to 16. In all but one case, higher memory heuristics outperform the lower memory ones. Note that the y-axis in Figure 4.20(b) is in a logarithmic scale. Also notice that in general, increasing the accuracy also increases the solution quality.

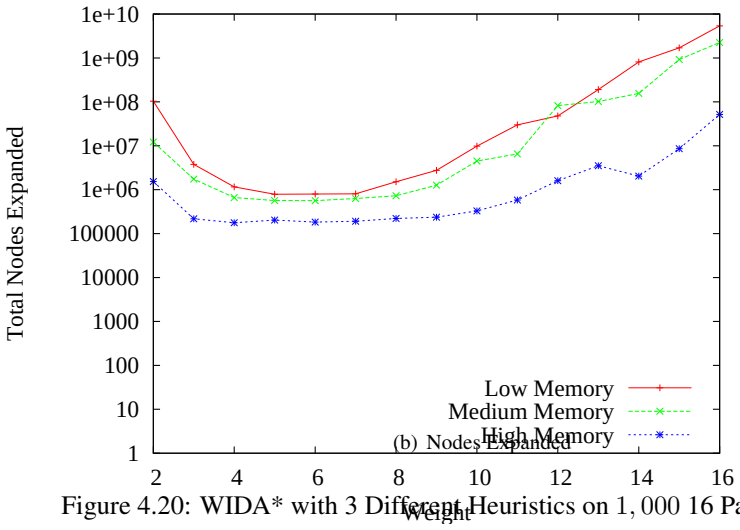
Now consider 3 different starting configuration sets, the high memory, medium memory, and low memory sets, in which all configurations use the corresponding heuristic function. Each set will consist of 15 configurations, each with a unique integer weight in the range from 2 to 16.

When comparing the performance of the different sets, the results are as expected: for any candidate set size k, the average performance of the high memory set outperforms the average performance of the medium memory set. Similarly, the average performance of the medium memory set outperforms the average performance of the low memory set. However, if instead of comparing the average performance, we consider how dovetailing over each configuration compares to the configuration in that set that has the best performance alone, we see a different relationship. In Figure 4.21, for each candidate set we have plotted the ratio of the average number of nodes expanded during the dovetailing simulations of all $\binom{15}{k}$ candidate sets to the number of nodes expanded by the single best configuration in each corresponding set. The configurations that had the fewest number of total nodes expanded in the low memory, medium memory, and high memory sets correspond to the weights 5, 5, and 4, respectively. Note, if a point is greater than 1 this means that dovetailing required more nodes on average than did the single best weight.

While dovetailing over all 3 sets results in poor performance when compared with the single best configuration in each set, it is the high memory set which exhibits the worst relative behaviour. While the other two sets show very similar performance by this metric, for most of the candidate set



(a) Solution Costs



(b) Nodes Expanded

Figure 4.20: WIDA* with 3 Different Heuristics on 1,000 16 Pancake Puzzles.

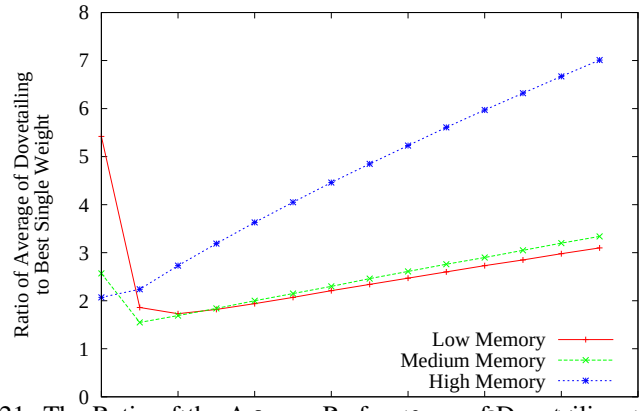


Figure 4.21: The Ratio of the Average Performance of Dovetailing over Weights in WIDA* to the Single Best Weight with 3 Different Heuristics on the 16 Pancake Puzzle.

sizes the medium memory set exhibits slightly poorer performance than does the low memory set.

Recall that the performance of dovetailing with the heuristic given by the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ pattern databases was examined earlier in Figure 4.18. This heuristic requires even less memory and is less accurate than those in Figure 4.21. In the case of this very poor heuristic case, the performance of dovetailing outperformed the single best configuration. As such, the trend that emerges is that as the accuracy of the heuristic decreases, dovetailing becomes a more effective procedure.

This relationship is caused by the fact that as the heuristic increases, fewer mistakes will be made in the traversal of the search trees. The searches will therefore become less diverse, and there will be more duplication of work between configurations. As one of the main impacts of dovetailing is to minimize the effect of any single configuration being led astray, dovetailing will therefore offer less and less as the heuristic function becomes more accurate.

4.4 Dovetailing and WRBFS

As WRBFS is also a linear-space algorithm like WIDA*, dovetailing can be performed with this algorithm without any significant concerns regarding restrictions on available memory. In this section we will show that while dovetailing over weights is not as effective as it is for WIDA*, it can still enhance the performance of WRBFS on larger puzzles.

4.4.1 Dovetailing over Weights with WRBFS on the Sliding Tile Puzzle

The performance of WRBFS on the 4×4 sliding tile puzzle was demonstated previously in Figure 2.14. For the dovetailing simulations, the starting configuration set was selected as the 15 configurations that differ only in that each is assigned a different integer weight in the range from 2 to 16.

The results of these simulations are shown in Figure 4.22.

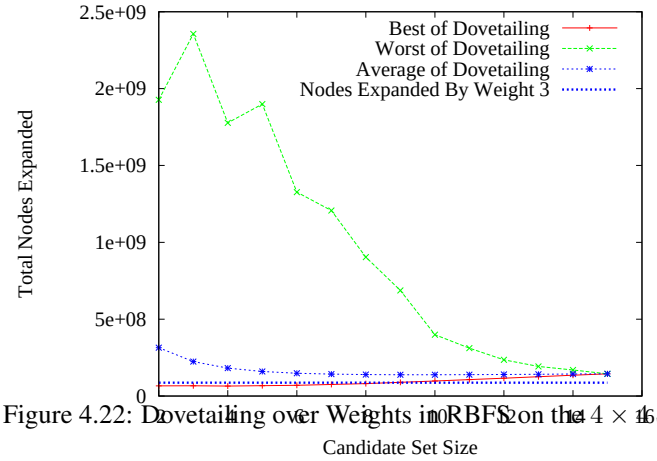
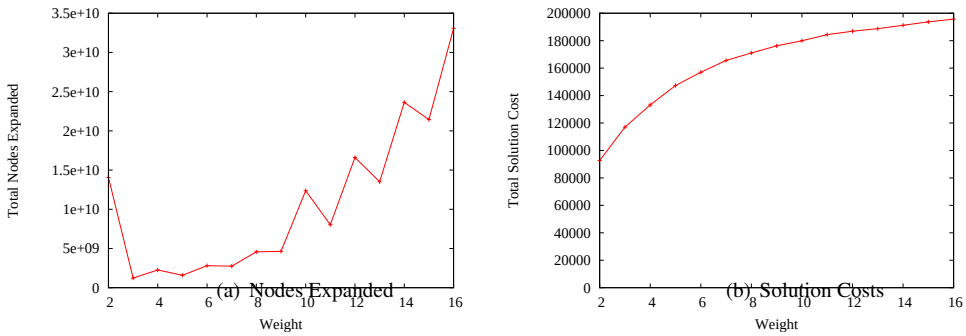


Figure 4.22: Dovetailing over Weights in RBFS on the 4×4 Sliding Tile Puzzle.

While the performance of the worst candidate set is quite poor, the average performance is similar to that of the best weight of 3. Dovetailing over all 15 configurations requires 1.7 times as many node expansions as does the weight of 3 alone. The solution quality found by this candidate set is similar to that found by the weight of 4.5.

While the performance of dovetailing over all 15 configurations does not match that of the single best weight, it is less than 2 times worse. However, recall that no tuning was performed in the construction of this candidate set. As tuning would be needed to determine that the weight of 3 is indeed the best weight, this additional cost could be considered as an acceptable overhead.

The performance of dovetailing over WRBFS configurations appears even more promising when it is applied to the larger 4×5 puzzle. Figure 4.23 shows the performance of WRBFS with 15 different configurations, each with a different integer weights in the ranfe of 2 to 16. The best weight is again the weight of 3.



(a) Nodes Expanded (b) Solution Costs

Figure 4.23: WRBFS on $1,000$ 4×5 Sliding Tile Puzzles.

The dovetailing simulations were performed with the starting configuration set consisting of those 15 tested configurations. The simulation results are shown in Figure 4.24. While the worst candidate set still shows poor performance when compared to the single best weight of 3, the average performance is better than this single weight once the candidate set size is at least 6. When the candidate set is of size 15, dovetailing requires 2.5 times fewer node expansions than the weight of 3.

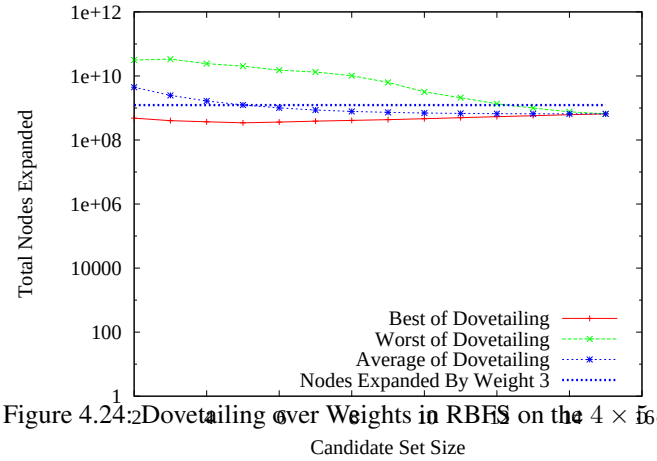


Figure 4.24: Dovetailing over Weights in RBFS on the 4×4 Sliding Tile Puzzle.

Figure 4.25 shows the performance of parallel dovetailing as a function of solution quality. The figure indicates that the average solution quality found by dovetailing, particularly for the larger candidate set sizes, is better than that found by most of the individual weights alone. For example, the solution quality found by the candidate set containing all configurations in the starting configuration set is between that found by the weights of 5 and 6 alone. Similar to the behaviour of dovetailing with WIDA*, the worst candidate sets are those containing mostly high weights — hence the poor solution quality. Again, these weights have generally poor behaviour alone, and are often unable to overcome the deficiencies of one another as well as other candidate sets.

These results indicate that parallel dovetailing offers a simple, yet effective way to trivially parallelize WRBFS. In the case of the 4×4 puzzle, the speedup from using 15 processors is by a factor of 9.0. For the larger puzzle, we see a super-linear speedup of 37 when using only 15 processors. Overall, parallel dovetailing appears to be an effective form of parallelization of the algorithm. Note, to the best of our knowledge, there have been no other attempts to parallelize either the RBFS or WRBFS algorithm.

While we have not tested WRBFS on the larger puzzles, the above results do suggest that the effectiveness of dovetailing scales with the size of the puzzle. This behaviour is again related to the fact that dovetailing helps to avoid mistakes made early in the search due to inaccurate heuristic values. The cost of making such mistakes increases with the domain size. As it was in WIDA*,

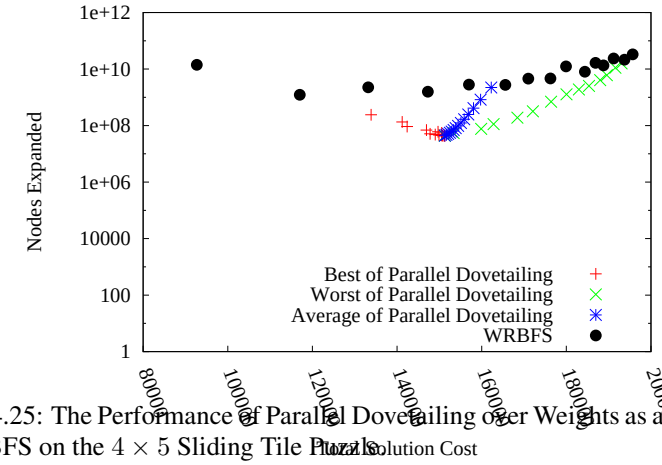


Figure 4.25: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WRBFS on the 4×5 Sliding Tile Puzzle

dovetailing offers a simple way of minimizing this effect.

4.4.2 Dovetailing over Weights with WRBFS on the Pancake Puzzle

In this section, we perform the same experiments on the 14 and 16 pancake puzzles as were performed with the WA* and WIDA* algorithms. Recall that the heuristic functions used are given by the $< 0, 1, 2, 3, 4, 5, 6 >$, and the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ pattern databases, respectively.

The behaviour of WRBFS on the 14 pancake puzzle with the aforementioned heuristic was shown previously in Figure 2.15. Recall that for any weights of size at least 10, all searches were identical. This again highlights the issue of candidate set selection, which will be considered later.

For the dovetailing experiments, the starting configuration set chosen consists of the 9 configurations each with a different integer weight in the range of 2 and 10 inclusive. The results of the simulations are shown in Figure 4.26. As with WA* on this puzzle, dovetailing actually degrades the performance when compared to the single best weight of 10. For example, when dovetailing over all 9 configurations, 5.5 times more nodes need to be expanded during problem-solving. While the average solution quality is closer to that found by the weight of 3, dovetailing still requires 2.5 times more node expansions than this weight alone.

In the case of the 16 pancake puzzle, the behaviour of WRBFS is shown in Figure 4.27(a). Due to the length of time for problem-solving, we only present the statistics for the integer weights from 2 to 12. Notice that like WA*, RBFS does not improve its performance as the weight increases, as both algorithms did in the 14 pancake puzzle. Instead, the performance hits its peak performance at the weight of 5.

Doevtailing also offers more benefits to WRBFS on the 16 pancake puzzle than it does on the 14

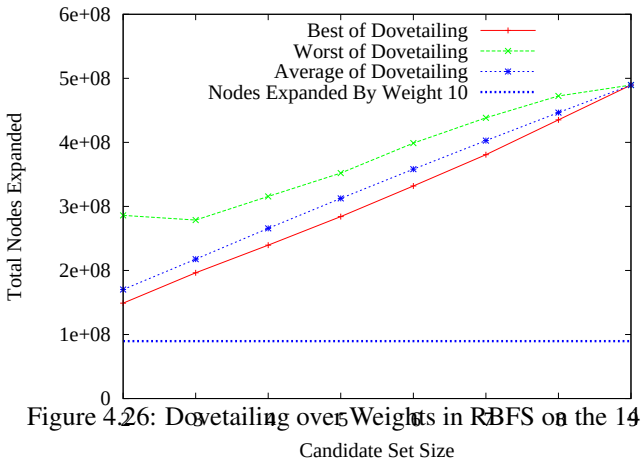
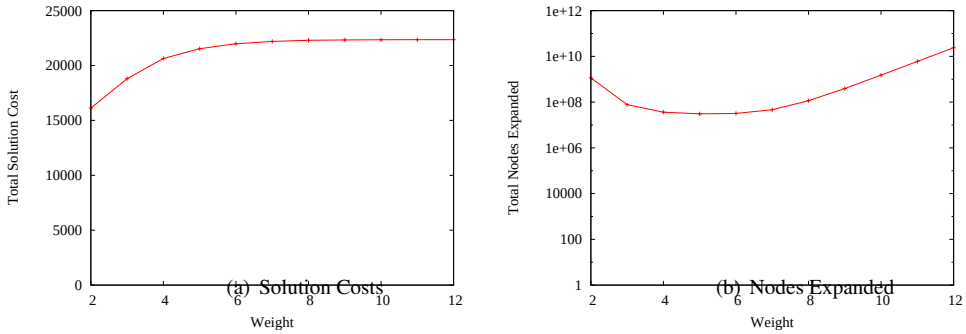


Figure 4.26: Dovetailing over Weights in RBFS on the 14 Pancake Puzzle.



(a) Solution Costs (b) Nodes Expanded

Figure 4.27: WRBFS on 1000 16 Pancake Tile Puzzles.

pancake puzzle. The starting configuration set consists of the 11 configurations shown individually in Figure 4.27. The results of these experiments are shown in Figure 4.28. Figure 4.29 also shows the performance of parallel dovetailing in this domain as a function of the solution quality.

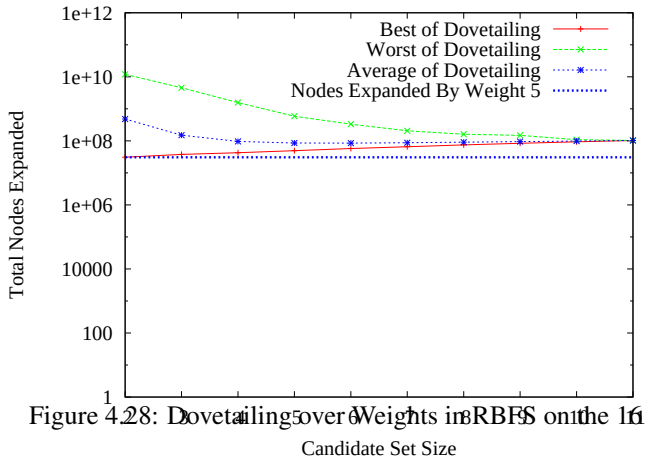


Figure 4.28: Dovetailing over Weights in RBFS on the 16 Pancake Puzzle.

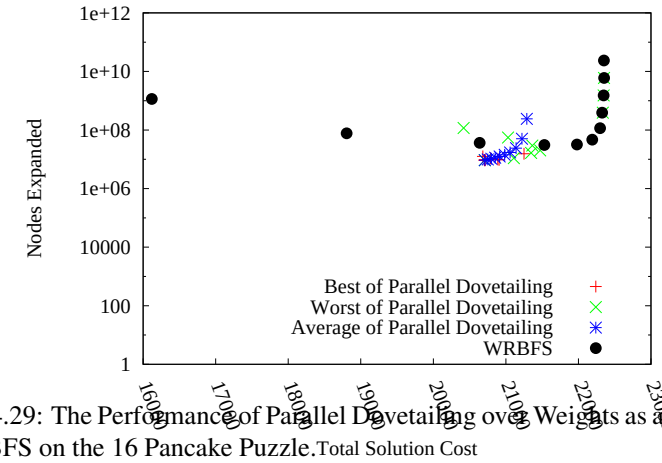


Figure 4.29: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in WRBFS on the 16 Pancake Puzzle.

In this puzzle, the results of dovetailing are much closer to the single best weight of 5. When using all 11 configurations, dovetailing expands 3.3 times as many nodes as does the weight of 5. The solution quality found by this candidate set is similar to that found by the weight of 4, which expands 2.8 times fewer nodes than does dovetailing. Figure 4.29 also demonstrates that even the solution quality found by the weakest candidate sets generally outperform the solution quality found by the weights of 5 and higher alone.

However, if dovetailing is performed over all 15 integer weights in the range of 2 to 16 inclusive,

dovetailing appears as a much more attractive enhancement. In this case, dovetailing only requires 1.7 times as many node expansions as the weight of 5. Again, this cost can be considered an acceptable overhead since no pre-computation was required. These results also indicate that parallel dovetailing becomes an effective method for parallelizing WRBFS on larger puzzles as the procedure yields a speedup of 8.8 when using 15 processors in the 16 pancake puzzle. In the 14 pancake puzzle, the speedup is only by a factor of 1.6 times when using 9 processors.

It should also be noted that the effectiveness of dovetailing also degrades as the heuristic function increases in accuracy. For example, when using 15 configurations on the 16 pancake puzzle, each with an integer weight in the range from 2 to 16 and the heuristic function given by the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9, 10 >$ pattern databases, dovetailing expands 9.3 times as many nodes as does the single best weight alone.

4.5 Dovetailing over Beam Widths in BULB

Recall that one of the parameters to the BULB algorithm is the memory limit L, which is the maximum number of states in memory at any time. In the case of dovetailing, this limit can be used to control how much memory is allocated to each configuration.

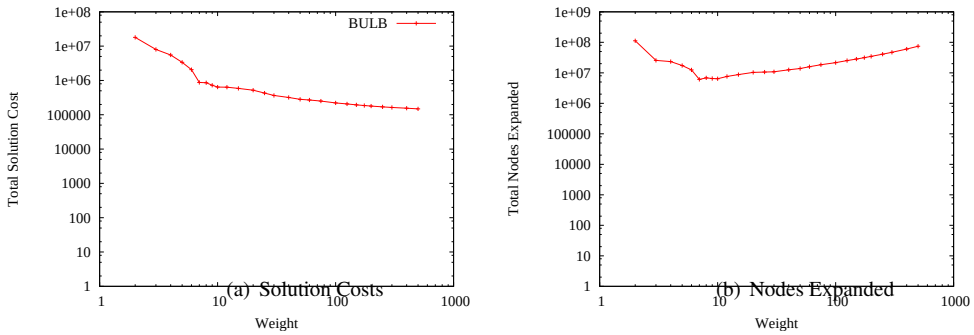
In all the following experiments, we will take a similar approach as is taken in Section 4.2. All configurations will be assigned the same limit L and dovetailing over k such configurations will be compared to the single best configuration in the set (which also has the limit L). While the memory requirements of dovetailing will actually be kL, we are considering experiments in this way with an eye toward parallel machines with distributed memory. Like WA*, even under these favourable conditions, dovetailing over BULB configurations will still yield poor performance.

In all dovetailing simulations in this section, the starting configuration set will consist of 15 identical configurations that only differ in the beam width. The beam widths used will be those in the set $BW = \{3, 5, 7, 10, 15, 25, 30, 40, 50, 60, 75, 100, 150, 200, 300\}$. These widths have been chosen so as to offer a diversity in the solution quality and search efficiency.

4.5.1 Dovetailing over Weights with BULB on the Sliding Tile Puzzle

In this section, we will consider the performance of dovetailing over BULB configurations on the 5×5 and the 6×6 sliding tile puzzles. Figure 4.30 shows the performance of the algorithm when using BULB with different beam widths on $1,000$ 5×5 sliding tile problems. The memory limit used in these experiments is $100,000$ states. Note, for all beam widths from 6 to 175, all $1,000$ problems could be solved without any backtracking, and both axes in the figure are in a logarithmic scale.

The results of the dovetailing simulations are shown in Figure 4.31. The starting configuration set used is given by the beam width set BW mentioned above. The figure demonstrates that dovetailing over beam widths is ineffective in the 5×5 domain. For all candidate set sizes, even the best



(a) Solution Costs (b) Nodes Expanded

Figure 4.30: BULB on $1,000$ 5×5 Sliding Tile Puzzles.

set expanded more nodes than did the beam width of 10 alone. Dovetailing over all 15 configurations requires 9.4 times as many node expansions as the beam width of 10 alone. Even when compared to the beam width of 25, with which dovetailing has the most similar solution quality, dovetailing still expands 5.8 times as many nodes.

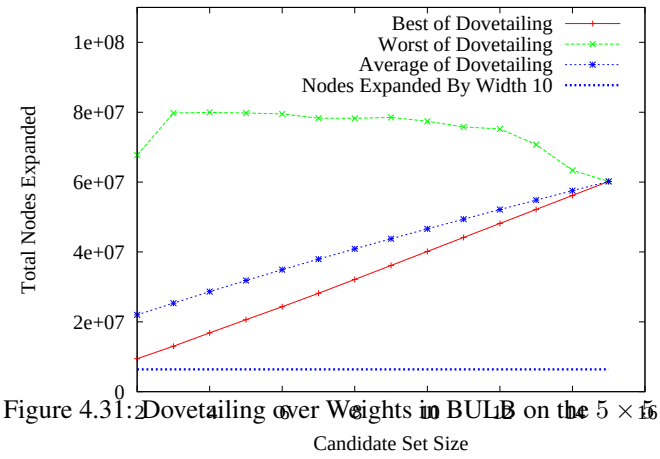
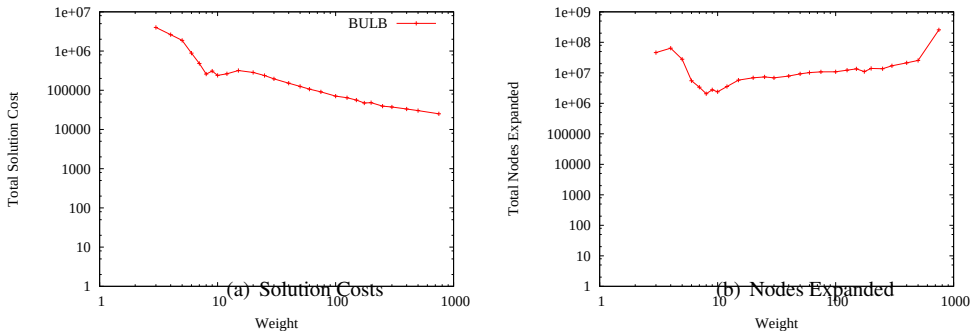


Figure 4.31: Dovetailing over Weights in BULB on the 5×5 Sliding Tile Puzzle.

The performance of dovetailing on the 6×6 puzzle is not much better. First, consider the performance of the individual configurations, each with a unique beam width and the memory limit of $200,000$. Figure 4.32 shows the performance of such configurations on 100 6×6 sliding tile puzzles. Note, only the beam widths of 7, 8, 9, and 10 were able to solve all problems without any backtracking and both axes have a logarithmic scale. The beam width in the set BW with the best performance is that of the weight 10.

The results of the dovetailing simulations over the configuration set given by widths in BW are shown in Figures 4.33, which demonstrates that dovetailing is ineffective even in this larger domain when compared to the single best configuration of beam width 10 alone. For all candidate



(a) Solution Costs (b) Nodes Expanded

Figure 4.32: BULB on $1,000$ 6×6 Sliding Tile Puzzles.

set sizes, even the best set again performs worse than the beam width of 10. Dovetailing over all 15 configurations also takes 8.4 times more node expansions than the beam width of 10 alone, and 2.0 times more node expansions than the beam width of 60 alone, with which the set has the most similar average solution quality.

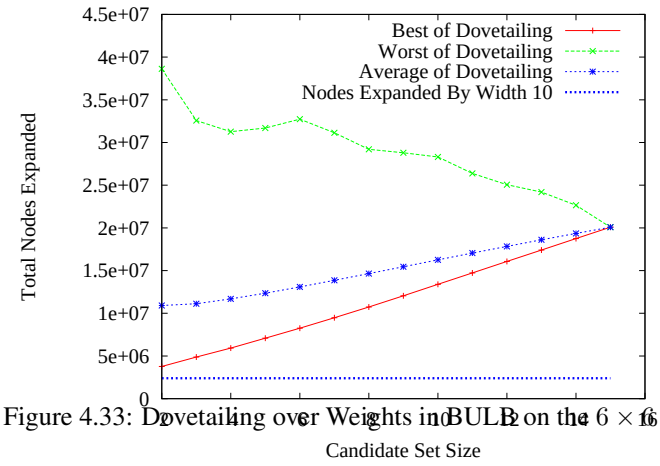


Figure 4.33: Dovetailing over Weights in BULB on the 6×6 Sliding Tile Puzzle.

The above results demonstrate that dovetailing over beam widths is not suitable for use with the BULB algorithm in this domain. Figure 4.34 (which has a logarithmic scale for both axes) shows the performance of parallel dovetailing as a function of solution quality. This figure shows that parallel dovetailing is a poor form of parallelization of the BULB algorithm, as the speedup gained from the use of 15 processors are only 1.6 and 1.8 on the 5×5 and 6×6 puzzles, respectively. While we are unaware of any other attempts to parallelize BULB, parallel dovetailing appears to be a poor use of resources in this domain. We will consider why dovetailing is ineffective when used with BULB at the end of the next section.

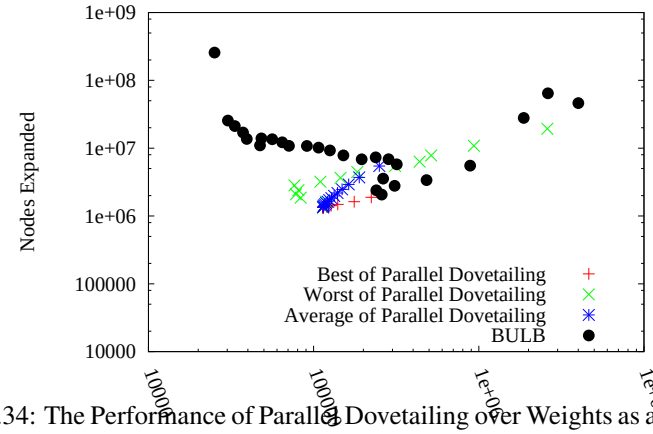


Figure 4.34: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in BULB on the 6×6 Sliding Tile Puzzle.

4.5.2 Dovetailing over Weights with BULB on the Pancake Puzzle

Dovetailing over BULB configurations that differ only by beam width is similarly ineffective in the pancake puzzle. The results of the dovetailing simulations for the 14 pancake puzzle domain when using the $< 0, 1, 2, 3, 4, 5, 6 >$ pattern database heuristic are shown in Figure 4.35. These results are for the problem set of size $1,000$. The memory limit used in these experiments was set at $50,000$ states. Recall that Figure 2.18 depicted the performance of many BULB configurations on this problem set.

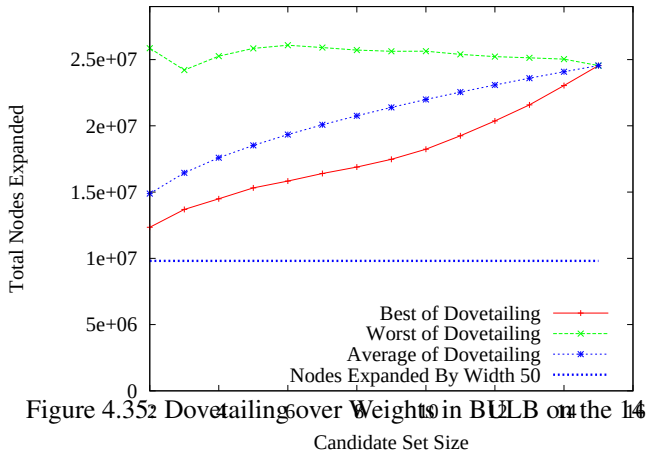
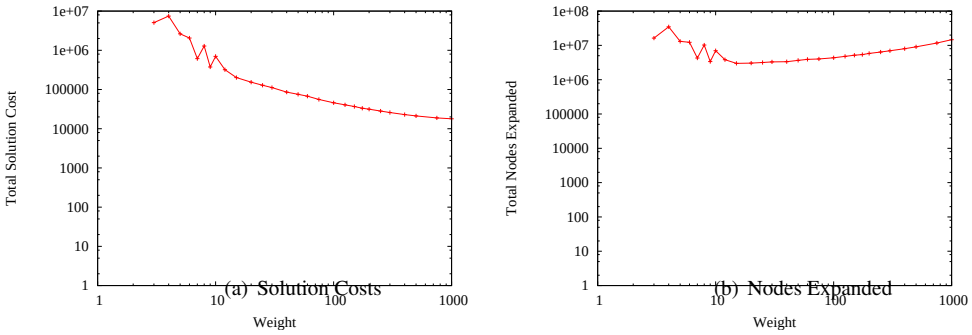


Figure 4.35: Dovetailing over Weights in BULB on the 14 Pancake Puzzle.

While dovetailing is somewhat more useful in this domain than on either of the sliding tile puzzles, the procedure still remainds mostly ineffective. When dovetailing over all 15 configurations, there are 2.5 times more nodes expanded than are expanded by the beam width of 50 which has

the best performance of all configurations. Dovetailing also expands 2.1 times more nodes than the beam width of 100 with which it has the most similar average solution length.

The performance of dovetailing on the 16 pancake puzzle with the heuristic function given by the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ pattern database heuristics is even worse. Figure 4.36 shows the performance of a number of configurations with the memor limit of $50,000$ states, on a problem set of size $1,000$. Note that only beam widths of 2, 3, and 4 required any backtracking.



(a) Solution Costs (b) Nodes Expanded

Figure 4.36: BULB on $1,000$ 16 Pancake Puzzles.

Figure 4.37 shows the results of the dovetailing simulations using the configuration set given by BW. In this case, dovetailing results in much poorer performance than the best width of 15 alone. When dovetailing over all 15 configurations, 5.3 more node expansions are needed to solve all problems than are required by the beam width of 15 alone. The solution quality of such dovetailing is similar to that found by the beam width of 60 which expands 4.1 times fewer nodes than the dovetailing procedure.

As was the case in the sliding tile puzzle, parallel dovetailing exhibits poor performance in this domain. To see this, consider Figure 4.38 which shows the performance of parallel dovetailing as a function of solution quality on the 16 pancake puzzle. While parallel dovetailing is somewhat effective in the 14 pancake puzzle, for which it results in a speedup of 6 when using 15 processors, the performance is quite poor on the 16 pancake puzzle for which the speedup is only by a factor of 2.8 when using the same number of processors.

These results suggest that dovetailing is poorly suited for BULB configurations in both the pancake puzzle and the sliding tile puzzle, largely because the beam search algorithm already addresses some of the issues that dovetailing tackles. Beam search naturally considers multiple alternative paths at the same time through the parallel expansion of all nodes on the beam. As for dovetailing, the consideration of multiple paths means that if any single path is led astray, there are others that will hopefully compensate. Since beam search inherently has this property, dovetailing can offer little to the algorithm. As a result, the overhead of running multiple instances of the same algorithm in parallel dominates the search time and causes the poor performance seen above.

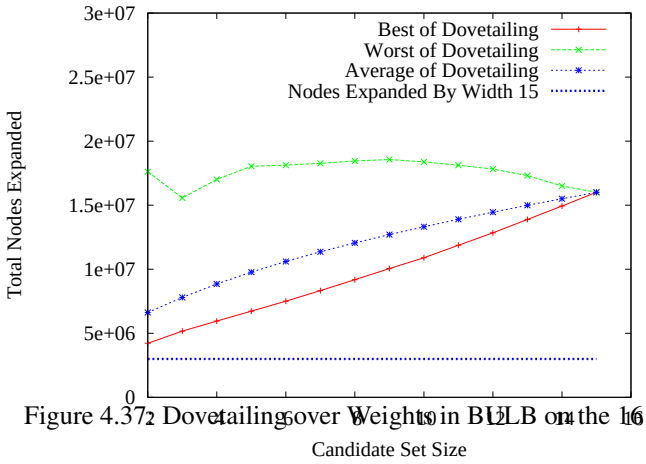


Figure 4.37: Dovetailing over Weights in BULB on the 16 Pancake Puzzle.

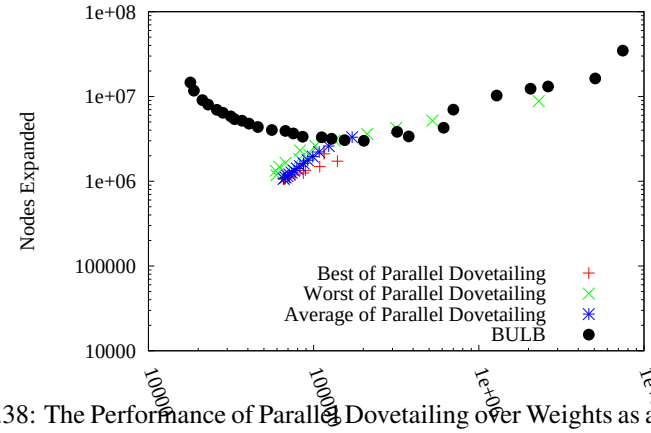


Figure 4.38: The Performance of Parallel Dovetailing over Weights as a Function of Solution Quality in BULB on the 16 Pancake Puzzle.

4.6 Chapter Summary

In this chapter, we considered the performance of dovetailing and parallel dovetailing over the main parameter spaces of WA*, WIDA*, WRBFS, and BULB. In the case of WA* and BULB, dovetailing caused a degradation in the search speed for all domains tested. In WA*, this behaviour is related to the fact that aggressively proceeding down any branch of the search tree is a strong policy in these domains. As the heuristic functions will cause the algorithm to make mistakes during expansion, the fact that the algorithm holds all generated states in memory means that it is capable of quickly changing to an alternate candidate path if the current line of play is lead astray. The ineffectiveness of dovetailing with the BULB algorithm is caused for similar reasons. BULB already simultaneously considers alternate paths through the state space thereby minimizing the impact if any single path is guided poorly. As dovetailing solves a similar problem, it adds little to BULB at the cost of running many instances in parallel.

On the other hand, dovetailing was generally successful when applied to WRBFS as it was usually solved all problems nearly as quickly as the single best configuration alone if not better, and it did so without any offline tuning. Dovetailing was also shown to significantly improve the performance of WIDA*. As these algorithms only consider a single line of play at any time, the cost of making a mistake due to poor heuristic guidance is magnified. In the case of WIDA*, this effect may lead the algorithm to blindly search large areas of the state space as discussed in Section 2.9.3. On the other hand, poor heuristic guidance causes WRBFS to perform a lot of backtracking. Dovetailing helps to mitigate these effects by simultaneously considering multiple paths and thereby decreasing the chance that all paths have been lead in a bad direction.

Parallel dovetailing over the main parameter spaces al all algorithms considered has also been shown to offer at least some speedup in all domains tested. In particular, this procedure offers super-linear speedups in all WIDA* experiments and in the WRBFS sliding tile experiments. Parallel dovetailing also offers near-linear speedups when applied with WRBFS in the pancake puzzle domain. While the results are less impressive when dovetailing is applied with WA* and BULB, we have shown that parallel dovetailing over WA* configurations does compare favourably with state-of-the-art systems in the sliding tile puzzle domain. As such, parallel dovetailing represents a simple yet effective algorithm for multicore machines.

Chapter 5

Dovetailing Over Operator Orderings

Another design choice that can have a large impact on search speed is the operator order. In this chapter, we will consider dovetailing over candidate sets in which the configurations differ in the static operator ordering used during the search. All experiments in this chapter will be performed through the use of dovetailing simulations as outlined in Section 4.1.

To test the effectiveness of dovetailing over operator orderings, we will consider one puzzle size from each of the sliding tile and pancake domains. Each of WA*, WIDA*, and WRBFS will be tested with a number of different starting configuration sets denoted S_w. In any such S_w, all configurations will have the same weight value of w but differ in operator ordering. In the figures, the starting configuration sets will be denoted by the corresponding weight value. The BULB algorithm will be tested similarly, except each starting configuration will contain configurations that all have the same beam width. Also note that due to the large number of parameters being tested, we will only consider problem sets of size 100 in this chapter.

In the case of the sliding tile puzzle, recall that regardless of the puzzle size, there are exactly 4 operators. As such, there are $4! = 24$ operator orderings. The starting configuration sets used for all dovetailing experiments will therefore consist of 24 configurations, each with a constant weight or beam width, but a different operator ordering.

In the case of the pancake puzzle, the number of operators will depend on the number of pancakes. Specifically, for the N pancake puzzle, there are $N!$ operators. As it is infeasible to consider all possible operator orderings, we will only consider $N - 1$. The first two operator orderings are $\{2, 3, 4, ..., N - 1, N\}$ and $\{N, N - 1, ..., 4, 3, 2\}$. The $N - 3$ remaining operator orderings where constructed as follows: each ordering had its first element assigned as a unique member of the set $\{3, 4, 5, ..., N - 2, N - 1\}$, and the remaining operators were ordered randomly. This construction ensures that all $N - 1$ operator orderings in the starting configuration set have a different first operator.

Recall that when performing the dovetailing simulations, the ordering of configurations in a

candidate set will have a small impact on the search quality. We will again ignore this factor and instead use a single strategy for ordering all sets. For the ordering of configurations in the sliding tile puzzle candidate sets, each operator order was assigned a unique positive integer. The configurations are then ordered in ascending value of this integer. In the pancake puzzle candidate sets, the configurations are ordered in ascending value of the first operator.

Note that in several of the figures in this chapter, we will compare the average performance of dovetailing against the average performance of all configurations in the starting configuration. This is unlike the figures in Chapter 4 in which the performance of dovetailing is compared against the configuration in the starting configuration set with the best performance. The evaluation of dovetailing over operator orderings is done in this way because it is common for researchers to select an arbitrary operator ordering with little or no experimentation. The reasoning behind this decision is given below.

Recall that the figures in Chapter 2 showing performance of WA*, WIDA*, WRBFS, and BULB on the 4×4 sliding tile puzzle and 14 pancake puzzles suggest that the average performance over a number of problems will change relatively smoothly with the value of parameters. As such, when looking for strong average performance it is usually possible to quickly find a configuration with high average performance by performing a binary search in the parameter space. Such a binary search can only be performed in parameter spaces that are either real or integer valued. In the case of operator order, we usually do not have a natural ordering of configurations over which performance is expected to be relatively smooth. Combined with the fact that there is usually little *a priori* information to guide in the selection of an operator ordering, it is common to simply select any ordering and stick to it. Due to such random selection, it is therefore more accurate to consider the average performance over all configurations than just the single best configuration.

Also note that the figures will show one of two ratios. The first is the ratio of the average number of nodes expanded by all $\min(\binom{n}{k}, 10,000)$ candidate sets of size k (where the starting configuration set is of size n) divided by the average number of nodes expanded by the n configurations in the starting configuration set. This ratio will be used in cases where dovetailing performs worse than selecting a configuration at random. All data points greater than 1 will indicate that dovetailing is performing worse than the average configuration.

One such figure is 5.1 which summarizes the results of the WA* dovetailing simulations in the 4×5 sliding tile puzzle. Each line corresponds to a different starting configuration set. In this case, dovetailing does not work well for most of the candidate set sizes, so the value shown for any size k is the ratio of the the average performance of dovetailing over candidate sets of size k to the average of the batch results for each of the 24 operator orderings. For example, consider the line for the weight 6 starting configuration set. The average number of nodes expanded over $10,000$ candidate sets of size 11 is $3,273,214$. The average number of nodes expanded by the 24 weight 6 configurations alone is $2,221,011$. Therefore, the value shown for the candidate set size of 11 on

the weight 6 line is $3,273,214/2,221,011 = 1.47$. This means that dovetailing over candidate sets of size 11 is on average expanding 1.47 times more nodes than the average configuration.

In cases where dovetailing outperforms the average configuration, the inverse relation will be depicted. Specifically, this relation is the number of nodes expanded by the average configuration to the average expanded by dovetailing. In these figures, all data points greater than 1 correspond to instances in which dovetailing is outperforming the average configuration. One such figure is 5.3 which shows how many times fewer nodes dovetailing is expanding when compared to the average WIDA* configuration.

We use these two different ratios to make it clear how many times worse or better dovetailing is doing in comparison to the average configuration. The ratio shown will be indicated in the figure.

5.1 Dovetailing over Operator Orderings in WA*

In the case of WA*, operator ordering will affect in which order elements are added to the $OPEN$ list. Since the nodes in the $OPEN$ list are sorted, the ties between nodes that are generated by the same parent and have an equal f-cost and g-cost will be broken differently based on the order in which these elements are added to the list. This will cause the order in which nodes are selected for expansion to change.

This difference can cause a significant change in the number of nodes expanded. For example, we tried using the 24 different WA* configurations to solve 100 4×5 problem instances. All 24 configurations had the weight of 3 and a memory limit of $1,000,000$ states, but each has a different operator ordering. When performing such experiments, all configurations solved between 97 and 100 problems with only 7 of the 24 configurations solving all problems. Of those that solved all of the problem instances, the ordering that required the least amount of work expanded $6,657,411$ nodes while the ordering that required the most amount of work expanded $8,766,492$ nodes. Note, unless otherwise mentioned, we will only consider dovetailing with starting configuration sets in which all configurations successfully solved all problems in the test set.

Figure 5.1 summarizes the set of experiments regarding WA* dovetailing simulations on the 4×5 sliding tile puzzle. The figure shows that on some small candidate set sizes and larger weights, the average performance of dovetailing actually outperforms the average performance of the individual configurations. However, as the candidate set size increases, the overhead of running multiple instances of WA* simultaneously dominates the run time. Regarding parallel dovetailing, when using 24 processors where each is assigned a different configuration in the starting configuration set with a weight of 10, the improvements in search time are by factors of 8.8 and 6.6 over the average time needed by a single-processor with any of the 24 operator orderings and the most efficient ordering, respectively. The performance of parallel dovetailing over operator orderings will be compared to wPBNF in Section 5.1.1.

Figure 5.2 shows the results of the equivalent set of 16 pancake puzzle experiments. The heuristic

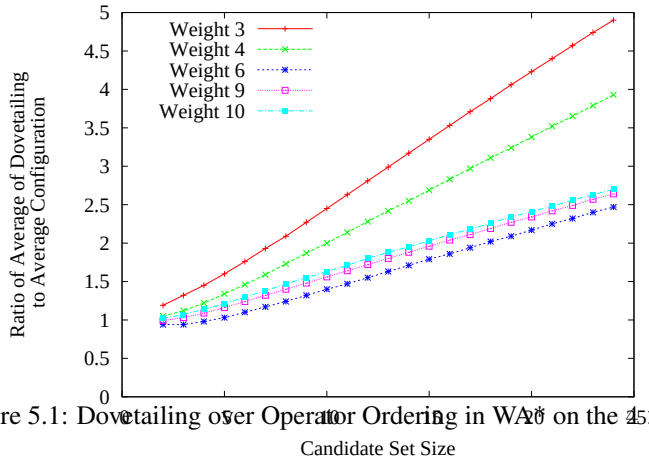


Figure 5.1: Dovetailing over Operator Ordering in WA* on the 15×5 Sliding Tile Puzzle.

function used for these experiments is given by the maximization over the $< 0, 1, 2, 3, 4, 5 >$ and $< 6, 7, 8, 9 >$ pattern databases. As in Figure 5.1, only values less than 1 indicate that the average performance of dovetailing is better than the average performance of individual configurations.

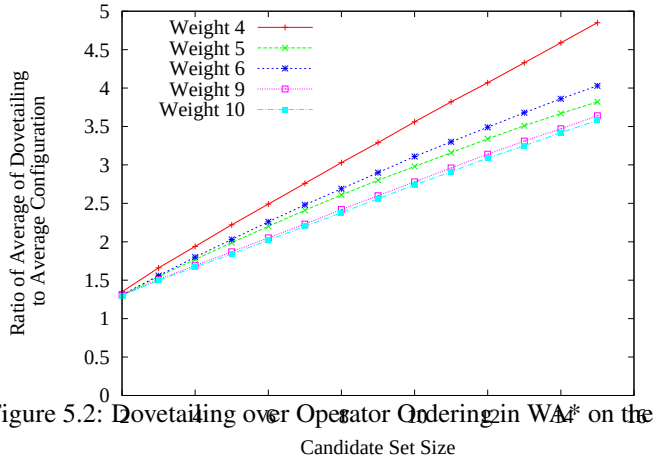


Figure 5.2: Dovetailing over Operator Ordering in WA* on the 16 Pancake Puzzle.

Dovetailing over operator ordering with WA* again appears ineffective. Similarly, parallel dovetailing only offers modest speedups. For example, when using 15 processors where each is assigned a different configuration in the weight 10 starting configuration set, the improvement in search time is by a factor of 4.1 over the average time needed by individual configurations. When parallel dovetailing over all configurations with the weight of 10 is compared against the single best configuration with the weight of 9, the speedup is by a factor of 3.6.

One of the trends evident in both Figures 5.1 and 5.2 is that dovetailing is usually more effective on the starting configuration sets with larger weights than on those with smaller weights. This effect

occurs because while there is a solution below every branch in the search tree in these domains, the amount of work needed to find that solution will differ between branches. In WA*, larger weights cause the algorithm to increasingly commit to a single line of play. The difference in tie-breaking caused by different operator orderings can change which line of play is being committed to. Dovetailing over operator orderings helps overcome issues that arise when WA* commits to lines of play that require a lot of work in order to find a solution.

Also of note, increasing the candidate set size has shown to improve the average solution quality of the solutions found. For example, consider the weight 3 starting configuration used in the 4×5 sliding tile puzzle. The average total cost of all solutions found with each of the 24 configurations in this set when run individually is $11,843$. The average total solution cost found when dovetailing over candidate sets of size 2 is $11,869$. However, the solution quality found when dovetailing over all 24 configurations is $11,765$. This behaviour is even more pronounced for larger weights. For the weight 10, the average total solution cost found when dovetailing over candidate sets of size 2 with the weight of 10 is $17,755$, which is actually better than any individual configuration in this set. When dovetailing over all 24 configurations, the total solution cost is $16,219$ which represents a 12% improvement over the average total solution cost of the configurations alone. Similar behaviour is also seen in the pancake puzzle domain.

Unfortunately, the amount of diversity introduced through the use of different operator orderings is still not enough to make dovetailing a useful enhancement to WA*. As changing operator orderings will only change the order in which nodes with the same parent are added to the $OPEN$ list, most of decisions made by different configurations will be very similar. As such, before dovetailing will successfully enhance WA*, it will be necessary to find another aspect of WA* algorithm design that introduces much more diversity.

5.1.1 Comparing Parallel Dovetailing over Operator Orderings with WA* to wPBNF

In this section, the dovetailing experiments will be performed in the same manner as was described in Section 4.2.2. As was done above, several different starting configuration sets will be considered, each of which will contain configurations with a different associated weight. The configurations within the same starting configuration set will again differ only in the operator ordering used.

For the smaller weights in the shared memory case, there were several occurrences of multiple configurations in the same candidate set being unable to solve the same problem. This only occurred when using higher numbers of processors since each thread is assigned a smaller fraction of the total memory. To handle this problem, the following strategy was taken: if none of the configurations in a candidate set can solve a problem p, all but one of the processors are stopped. The remaining processor is then allowed to use all the memory available while the other processors remain idle. Further, we assume the worst case possible: the configuration that continues searching is the config-

	Speedup With Different Numbers of Processors											
					Parallel Dovetailing							
	wPBNF				Shared Memory				Distributed Memory			
Weight	2	4	5	8	2	4	5	8	2	4	5	8
1.4	**1.12**	**1.65**	**1.92**	**2.62**	1.00	0.99	0.98	0.98	1.00	1.01	1.02	1.02
1.7	0.76	**1.37**	**1.50**	**1.49**	**0.95**	1.04	1.06	1.07	0.95	1.04	1.06	1.08
2.0	0.62	1.10	**1.34**	**1.46**	**1.00**	**1.15**	1.19	1.23	**1.00**	**1.15**	1.19	1.24
3.0	0.62	0.90	0.84	0.78	**1.06**	**1.65**	**1.83**	**2.11**	**1.06**	**1.65**	**1.83**	**2.11**
5.0	0.60	0.76	0.72	0.64	**1.25**	**1.75**	**1.9**	**2.18**	**1.25**	**1.75**	**1.9**	**2.18**

Table 5.1: The Speedup of wPBNF and the Average Speedup of Parallel Dovetailing over Operator Orderings on 43 4×4 Sliding Tile Puzzle Problems.

uration in the candidate set that requires the highest number of node expansions in order to solve the problem. While this approach gives a pessimistic evaluation of shared memory parallel dovetailing, we will show it will have minimal impact on the results.

Table 5.1 shows the speedup of both wPBNF and parallel dovetailing on the 43 easiest 4×4 sliding tile puzzle problems. For wPBNF, the table shows the speedup achieved from the weight w wPBNF search when compared to a serial WA* search with weight w.

In the case of parallel dovetailing, the weight w row indicates that the starting configuration used contains only configurations with a weight of w. The value shown in the weight w row and k processors column is the average speedup seen by parallel dovetailing over k processors with the weight w starting configuration set, when compared with the single configuration in that starting configuration set with the best performance alone. Note, we compare with the best configuration in this case since the wPBNF papers may have done the same.

For each weight and each number of processors in the table, we also show the highest speedup in bold. Also note that the algorithms only require less time than the serial version of WA* for entries larger than 1.

First, let us compare the shared and distributed memory versions of parallel dovetailing. The performance of the two memory architectures are only different for low weighted starting configuration sets and high numbers of processors. This is because these are the only situations in which all the configurations in a candidate set run out of memory before any single configuration has found a solution. In these cases, a distributed memory architecture will outperform the shared memory which has to default to the worst case procedure outlined above. However, the table shows that the effect of this worst case procedure is minimal, and only occurs where the speedup gained through parallel dovetailing is small.

The table shows that for low weights, wPBNF outperforms parallel dovetailing over operator orderings. For these weights, there is little diversity in the performance of the operator orderings. On the other hand, wPBNF handles these cases well as the processors are working on mutually exclusive sets of the state space during expansion.

As mentioned in Section 4.2.2, wPBNF shows poor performance in this domain for larger weights. In contrast to this result, the best speedup achieved with parallel dovetailing is with the largest weight considered. Recall that due to the behaviour of WA*, this is also the weight that requires the least amount of search effort in the serial case. Therefore, parallel dovetailing is able to enhance even the best performance of WA* while wPBNF cannot.

5.2 Dovetailing over Operator Orderings in WIDA*

As described in Section 2.9.3, WIDA* is often forced to examine large portions of a search tree without any heuristic guidance or pruning. In these areas of the search space, the order in which nodes at any branch of the search tree are considered is completely determined by the operator ordering. If the the portion of the search tree contains a goal, some operator orderings may find it quickly while others will require a large number of node expansions before doing so. As such, proper operator ordering selection for WIDA* on a problem-by-problem basis will potentially yield large gains in the search efficiency.

In Figure 5.3, experiments are shown regarding dovetailing over operator orderings with WIDA*. The figure is similar to Figure 5.1, except the inverse relation is depicted. Here, values greater than 1 imply that dovetailing required fewer nodes expanded on average when compared to using a single ordering.

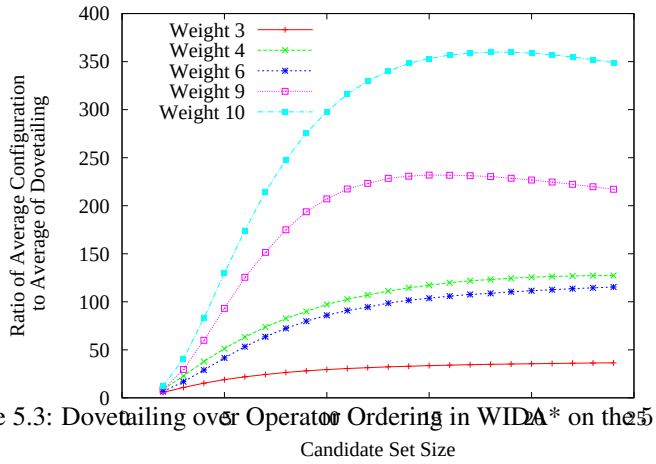


Figure 5.3: Dovetailing over Operator Orderings in WIDA* on the 5×5 Sliding Tile Puzzle.

The figure indicates that dovetailing over configurations that differ only in operator orderings offers large speed increases when compared with the average WIDA* instance. In Table 5.2, it is shown that the improvement over even the operator ordering with the best performance for each particular weight is also large. Notice that the minimum in terms of nodes expanded by dovetailing over all 24 operator orderings occurs at the weight of 8. This is in contrast to the fact that when

Weight	Nodes Expanded By the Best Order	Nodes Expanded By Dovetailing over 24 Orders	Factor of Improvement
3	273,782,770	49,318,913	5.6
4	246,230,595	16,968,531	14.5
5	411,505,075	11,092,577	37.1
6	441,818,307	10,280,071	43.0
7	487,919,186	10,216,976	47.8
8	325,019,519	9,646,247	33.7
9	541,136,094	11,210,600	48.3
10	1,866,590,738	13,096,426	142.5

Table 5.2: Dovetailing over Operator Ordering in WIDA* Compared to the Best Single Order on the 5×5 Sliding Tile Puzzle.

using a single configuration, the minimum number of nodes expanded occurs with the weight of 4.

Dovetailing over operator orderings also extends the range of weight values over which WIDA* is an efficient search algorithm. For example, Table 5.2 shows that the number of nodes expanded by even the best configuration in the starting configuration sets with weights of 9 and 10 is quite high in comparison to the performance of the weight of 4. However, the performance of dovetailing over all 24 orders is actually similar (and quite low) on all three of these starting configuration sets.

Both Figure 5.3 and Table 5.2 also indicate that the improvement gained through the use of dovetailing over operator ordering increases with the value of the weight. This is because of the behaviour described in Section 2.9.3 in which increasing the weight causes WIDA* to search a larger subtree without heuristic guidance. Running multiple operator orderings simultaneously through dovetailing decreases the chance of this happening.

In Figure 5.4, we show the performance of dovetailing over operator orderings on the 16 pancake puzzle. The figure shows that dovetailing is outperforming the average configuration on all candidate set sizes, for all weights starting configuration sets considered. Note, the y-axis is shown in logarithmic scale.

In this puzzle, dovetailing does not improve WIDA* on the low weights as much as it does on the 5×5 sliding tile puzzle. However, the improvements seen with high configuration sets and the larger weights is more dramatic. This is again due to the fact that larger weights are more susceptible to making poor decisions while traversing the search tree. The improvement is similarly large when we compare the performance of dovetailing over all 15 orders to the best configuration in each starting configuration set. This trend can be seen in Table 5.3.

The above results indicate that parallel dovetailing over operator orderings is a very effective parallelization of WIDA*. The speedups gained through such parallel dovetailing can be calculated for each of the starting configuration sets considered above by simply multiplying the factor of improvement in Tables 5.2 and 5.3 by the number of processors used: 24 in the case of the sliding tile puzzle, and 15 in the case of the pancake puzzle. The speedup is super-linear for all starting configurations considered.

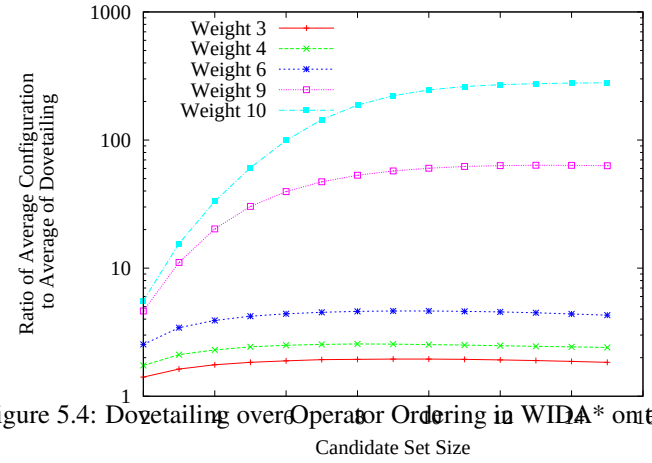


Figure 5.4: Dovetailing over Operator Ordering in WIDA* on the 16 Pancake Puzzle.

Weight	Nodes Expanded By the Best Order	Nodes Expanded By Dovetailing over 24 Orders	Factor of Improvement
3	4,234,363	3,835,205	1.1
4	1,671,646	1,187,152	1.4
5	1,733,165	668,793	2.6
6	2,120,445	911,044	2.3
7	2,505,688	1,003,828	2.5
8	5,881,005	1,323,972	4.4
9	19,229,160	1,249,807	15.4
10	80,527,176	1,033,192	77.9

Table 5.3: Dovetailing over Operator Ordering in WIDA* Compared to the Best Single Order on the 16 Pancake Puzzle.

In many cases, the speedup from parallel dovetailing is very large. For example, consider the weight 8 starting configuration used in the 5×5 sliding tile puzzle. As sequential dovetailing over 24 operator orderings involves the expansion of $9,646,247$ nodes, the time required by parallel dovetailing will be approximately that needed for $9,646,247/24 = 401,927$ node expansions. This means that parallel dovetailing with 24 processors will actually take 612 times less search time than the single best configuration in any starting configuration set considered — namely, the best configuration in the weight 4 set.

While dovetailing over operator orderings significantly improves the search time of WIDA*, the effect on solution quality is mostly negligible. This is also true when considering individual configurations. For example, in the weight 10 starting configuraton set used in the sliding tile puzzle experiments, the difference between the fastest and slowest configurations is by a factor of 5.1. However, the difference in the average solution quality found by these configurations is 0.2%. The difference between the average total solution quality found by dovetailing over candidate set sizes of 2 when compared to the candidate set sizes of 24 is only 0.03%. Both of these candidate set sizes also find solutions that are almost identical to the average over the configuration sets. The results are similar in the pancake puzzle domain.

5.3 Dovetailing over Operator Orderings in WRBFS

Recall that when a node n is expanded for the first time by WRBFS, the children of n are sorted by their f-cost. The initial order of the children (before sorting) will effect how the sorting algorithm breaks ties between nodes with an equal f-cost. This initial order is given by the operator ordering. Therefore, different operator orderings can change the order in which nodes are expanded for the first time.

When a node is re-expanded, the value assigned to any child c is the maximum of $f(c)$ and the minimum threshold of all f-costs seen thus far. Often, this results in several children being assigned the same f-cost. As described above, the operator ordering will affect the order in which nodes are expanded in the presence of ties. This increase in ties will thereby increase the impact of the operator ordering. As such, operator ordering will also affect the order in which nodes are re-expanded. In this section, we will show that dovetailing will often effectively enhance WRBFS due to the diversity introduced to the search tree traversal in these two ways.

In Figure 5.5, the results of the simulations regarding dovetailing over operator orderings in WRBFS are shown. Note that like the figures in Section 5.2, a value greater than 1 for any candidate set size k indicates that dovetailing over k configurations on average requires fewer node expansions than the average configuration.

Several interesting trends emerge from Figure 5.5. First, notice that much like the performance of WIDA* on the 5×5 puzzle, the relative improvement is greater for the larger weights than it is for the smaller weights. The reasoning behind such behaviour is similar to that given for WA*

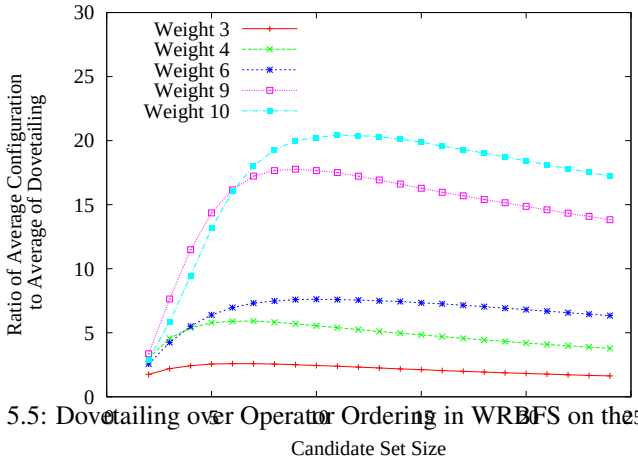


Figure 5.5: Dovetailing over Operator Ordering in WRBFS on the 4×5 Sliding Tile Puzzle.

Weight	Nodes Expanded By the Best Order	Nodes Expanded By Dovetailing over 24 Orders	Factor of Improvement
3	82,013,967	94,798,145	0.9
4	110,396,467	87,437,546	1.3
5	73,823,540	22,595,097	3.3
6	114,610,812	49,159,763	2.33
7	89,791,503	19,555,388	4.6
8	183,524,940	37,268,972	4.9
9	165,478,833	22,314,006	7.4
10	373,128,797	64,443,033	5.8

Table 5.4: Dovetailing over Operator Ordering in WRBFS Compared to the Best Single Order on the 4×5 Sliding Tile Puzzle.

which exhibited similar trends. Also note that while using all 24 operator orderings does work well on all weights, the best average performance is found with smaller numbers of configurations. The candidate set size with the best average performance for a weight w starting configuration set, also increases with the weight.

When the comparisons are made between dovetailing and the single best configuration, the behaviour is slightly different. This relationship is shown in Table 5.4. Here we see similar behaviour to that observed in Figure 4.23, in that when comparing consecutive integer weights, the odd weights always outperform the even weights. The performance of dovetailing over the different starting configuration sets shows a similar trend. Currently, we do not have a reason for why odd weights outperform even weights in this domain and leave such an investigation for future work.

While the results shown above suggest that dovetailing over configurations that only differ in operator ordering is an effective enhancement to WRBFS, the results on the 14 pancake puzzle are mixed. Figure 5.6 shows these results. The y-axis in this figure shows the same relation that it does in Figure 5.5.

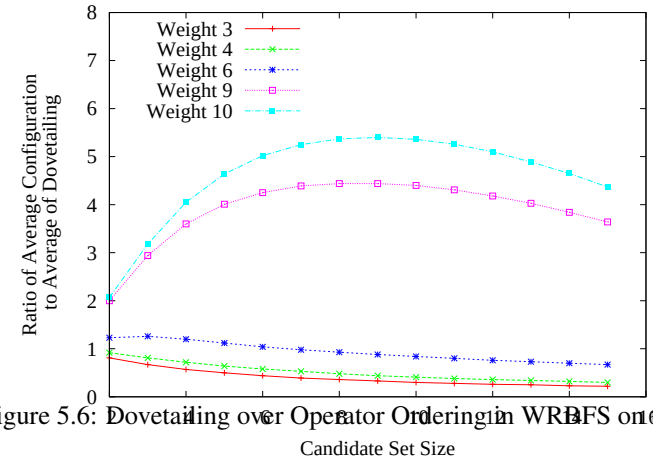


Figure 5.6: Dovetailing over Operator Ordering in WRBFS on the 16 Pancake Puzzle.

Weight	Nodes Expanded By the Best Order	Nodes Expanded By Dovetailing over 24 Orders	Factor of Improvement
3	6,929,572	41,818,769	0.17
4	2,762,023	12,378,956	0.22
5	2,165,567	7,113,658	0.30
6	2,263,928	5,238,205	0.43
7	3,113,635	4,654,909	0.67
8	6,628,702	10,101,465	0.66
9	19,446,612	10,950,786	1.8
10	75,287,230	35,951,542	2.1

Table 5.5: Dovetailing over Operator Ordering in WRBFS Compared to the Best Single Order on the 16 Pancake Puzzle.

The performance of dovetailing is highly dependent on the weight in this domain. Dovetailing degrades the performance of small weights yet significantly improves the performance of larger weights. A similar trend is seen when comparing the performance of dovetailing to the performance of the configuration in each starting configuration set that has the best performance. This comparison is shown in Table 5.5. Clearly, dovetailing is doing well to minimize the effect of the mistakes made by the high weights. Unfortunately, with the lower weights, the search trees are evidently too similar.

When considering the parallel variant of dovetailing over operator orderings, the results again depend on the domain. For every starting configuration except the weight 3 set, the speedups are super-linear in the 4×5 sliding tile puzzle when dovetailing over all 24 configurations. This is not the case in the pancake puzzle in which the speedups depend on the starting configuration set. For the small weighted starting configuration set, dovetailing over operator ordering is an ineffective form of parallelization. However, super-linear speedups are seen with the higher weights.

Much like WA*, dovetailing over increasing numbers of operator orderings in WRBFS also improves the quality of the solutions found. The behaviour is similarly magnified for larger weights.

Due to the similarity of the observed behaviour of WRBFS with WA*, we will only consider the weight 10 starting configuration set used in the sliding tile puzzle experiments. The average total cost of the 24 configurations in the set is $18,112$ when the configurations are used individually. When dovetailing over candidate sets of size 2, the average total cost is $17,743$. The total cost found when dovetailing over the candidate set containing all 24 configurations is $16,817$. This is an improvement of almost 8%.

5.4 Dovetailing For Optimal Problem Solving

For weight values of at least 3, Sections 5.2 and 5.3 demonstrate that dovetailing over operator ordering can often help to improve upon the standard WIDA* and WRBFS algorithms. These results suggested the following question: can dovetailing (or parallel dovetailing) over operator orderings be used with a weight of 1 to speed up optimal search?

In particular, we are interested in the performance of dovetailing over operator orderings with RBFS and IDA* since the linear-space nature of these algorithms allows them to optimally solve problems in larger domains than an algorithm like A*. Below, we will examine the performance of these linear-space algorithms in two test domains: the 4×4 sliding tile puzzle domain and the 14 pancake puzzle. The heuristic functions will be given by the Manhattan distance and $< 0, 1, 2, 3, 4, 5, 6 >$ pattern database, respectively. For the 4×4 sliding tile puzzle, the test set used is the Korf test set. For the 14 pancake puzzle, we will use a test set containing 100 problems.

Table 5.6 shows the results from the dovetailing over operator ordering simulations. The set of operator orderings considered are the same as those considered with the weighted algorithms. For every candidate set size k, the table shows the ratio of the average number of nodes expanded when dovetailing over k configurations to the average number of nodes expanded by each of the configurations. This value is shown for each algorithm on both domains. If the value is greater than 1, then the average dovetailing simulation required more node expansions than did the average configuration alone. Note, the average speedup found by parallel dovetailing can be found by dividing the candidate set size by the factor of nodes expanded by single-processor dovetailing. For example, the average speedup found in the sliding tile puzzle domain when using parallel dovetailing over 4 IDA* instances, each on a separate processor, will be $4/2.8 = 1.4$.

The table shows that in all situations tested, the average configuration outperforms the average performance of dovetailing. Similarly, parallel dovetailing over operator orderings is clearly not an effective form of parallelization for these algorithms. For example, the speedup achieved on the 4×4 puzzle when using 24 processors is only by a factor of 1.6 for both algorithms. The speedup achieved on the 14 pancake puzzle when using 13 processors is also 2.0 for both algorithms.

Below, we will analyze why dovetailing is ineffective in the case of optimal problem-solving with IDA* and suggest that a similar argument can be made for RBFS. Some of this work is similar to the analysis done by Powley and Korf in their evaluation of the effectiveness of Parallel Window

	Ratio of the Nodes Expanded by Dovetailing To the Nodes Expanded by the Average Configuration			
	4×4 Sliding Tile Puzzle		14 Pancake Puzzle	
Candidate Set Size	Dovetailing With IDA*	Dovetailing With RBFS	Dovetailing With IDA*	Dovetailing With RBFS
2	1.6	1.7	1.6	1.5
3	2.2	2.3	2.1	2.0
4	2.8	2.8	2.6	2.4
5	3.3	3.4	3.1	2.9
6	3.9	4.0	3.6	3.3
13	8.0	8.1	6.6	6.4
24	15.0	14.7	NA	NA

Table 5.6: Dovetailing over Operator Ordering for Optimal Problem-Solving.

Search [39].

First, notice that the total amount of work performed during all iterations except the last one will be identical. During these non-final iterations, the search will involve raising the threshold to the cost of the optimal solution path. Changing the operator ordering only changes the order in which potential solution paths in the search space are considered. Any difference in search time between operator orderings occurs because the final iteration is stopped as soon as a solution is found.

Before continuing, we need to introduce the notion of the *iteration branching factor*. The iteration branching factor b_i is the ratio of the number of nodes expanded when searching to a cost threshold t to the number of nodes expanded in the previous iteration for a large t. This value is an estimate of how the size of iterations grow. With this definition, it should be clear that on any problem p that requires d iterations to solve, the average number of nodes expanded on some non-final jth iteration is cb_i^{j-1}, where c is the average number of nodes expanded during the first iteration. The total number of nodes expanded in the first $d-1$ iterations, denoted ψ, is given by the following:

$$\psi = c + cb_i + cb_i^2 + ...cb_i^{d-1} \quad (5.1)$$

$$= cb_i^{d-1}(b_i^{2-d} + b_i^{3-d} + ...b_i^1 + 1) \quad (5.2)$$

Now let $x = 1/b_i$ and substitute in 5.2. The result is the following: $\psi = cb_i^{d-1}(1+x+x^2+...+x^{2-d})$. If we assume $b_i > 1$, $x < 1$. Therefore, ψ can be approximated as $\psi = cb_i^{d-1}(1+x+x^2+...)$ since $1+x+x^2+...$ converges to $1/(1-x)$ for $x < 1$. Substituting b_i back into the equation yields $\psi = cb_i^{d-1}(b_i/(b_i-1)) = cb_i^d/(b_i-1)$. Note, this value will remain constant over all configurations that have the same value of b_i.

The total number of nodes in the search tree of the final iteration is cb_i^d. This means that the ratio between the total number of nodes expanded in the final iteration to ψ is $cb_i^d/(cb_i^d/(b_i-1)) = (b_i-1)$. However, only a fraction of the nodes in the final iteration will be expanded since search is

stopped once a solution is found. Let a_θ denote the average proportion of the last iteration expanded by a configuration θ. Using the above ratio, this means that the average number of nodes expanded in the final iteration of a search using θ can be expressed as $a_\theta(b_i - 1)\psi$.

Now let us compare the expected number of nodes expanded by configuration θ against dovetailing. The expected number of nodes expanded by θ will be $T_\theta = \psi + a_\theta(b-1)\psi = (1 + a_\theta(b-1))\psi$. Let α_Θ be expected proportion of the final iteration that is expanded by the configuration in the candidate set Θ that required the minimum number of node expansions on p. When dovetailing over Θ where $|\Theta| = k$, the expected number of nodes expanded will be $T_\Theta = k\psi + k\alpha_\Theta(b_i - 1)\psi$. The expected ratio of T_Θ to T_θ is then given by $k(1 + \alpha_\Theta(b_i - 1))/(1 + a_\theta(b_i - 1))$. Dovetailing over Θ will only outperform some configuration θ if this ratio is less than 1. Notice that as b_i increases to ∞ the ratio reduces to $\alpha_\Theta k/a_\theta$.

The poor performance exhibited by dovetailing in the 4×4 sliding tile puzzle and the 14 pancake puzzle implies that the average of this ratio is greater than 1 in these domains with the corresponding heuristics. Below, we will empirically estimate the values of a_θ, α_Θ, and b_i so as to evaluate the effectiveness of this model. To perform these calculations, two test sets were constructed from the easiest 50 problems from each of these domains. Each problem was solved using IDA* such that the search continued even when a solution was found. The ratio of the number of nodes in the last iteration to the total number of nodes in all previous iterations was then used to estimate $b_i - 1$.

Having expanded the entire final iteration on each problem, it is simple to calculate the proportion of nodes in the final iteration expanded by each configuration using the data collected for the dovetailing simulations previously considered in this section. The average minimum proportion over all configurations of the final iteration examined is also calculated for each problem. Averaging these values over all 50 problems gives us a_θ and α_Θ.

In the 4×4 sliding tile puzzle, the candidate set Θ contains all 24 operator ordering configurations. The value for b_i found is 6.2. The value of a_θ, where θ is the average configuration is 0.27. The value of α_Θ is 0.06. This predicts that the ratio of T_Θ to T_θ is 13.2. This compares well to the actual average ratio of the number of nodes expanded by dovetailing to the average number expanded by individual configurations over all 50 problems, which was found to be 13.1. Note, the actual average ratio over all 100 problems is 13.3.

In the 14 pancake puzzle, the value for b_i is also larger than the brute-force branching factor of 12 (13 operators are applicable per state, but one returns the state to the parent and can be immediately pruned). For these experiments, the candidate set contains all 13 configurations considered above. The value for b_i found over the 50 problems is 21.5. For the average configuration, a_θ was calculated as 0.32. The value of α_Θ over all 13 configurations is 0.04. This predicts that the ratio of T_Θ to T_θ is 3.1 which can be compared to the actual average ratio of 4.0 over the 50 problems. The actual average ratio over all 100 problems is 4.4.

Note that this analysis is not immediately applicable to dovetailing over configurations all with

the same weight w, where $w > 1$. This is because the iterations do not grow as uniformly as they do in optimal search. Typically, WIDA* will perform a small number of very small iterations (in the case of the 4×4 puzzle all non-final iterations expanded fewer than 25 nodes), with the final iteration dominating search time. We leave such modelling as future work.

5.5 Dovetailing over Operator Orderings in BULB

In this section, we will consider applying dovetailing over operator orderings to the BULB algorithm. Recall that in beam search, all the nodes in the deepest beam are expanded and sorted. In practice, this list of successors L is built by iteratively expanding nodes in the deepest beam. These nodes are then appended to L. It is only once all nodes in the deepest beam have been expanded that L is sorted.

The order in which nodes are appended to L is given by the operator ordering. The operator ordering will therefore affect the way ties are broken by the sorting algorithm. This can change in which beam slice a node ends up in after the sort. For example, consider a BULB search with a beam width of 4. Consider the successors of some beam b. If the number of successors with the minimum h-cost is 6, only 4 of these nodes will be in the first slice. Let n be one of these 6 successors. Some operator orderings will result in n being in the first slice, and others will have n being in the second slice. As the slice that a node is in will change the order in which it is expanded, operator ordering can significantly affect the traversal of the search tree.

Figures 5.7 and 5.8 show the results of the dovetailing simulations for different BULB starting configurations in the 6×6 sliding tile puzzle and 16 pancake puzzle domains, respectively. Each starting configuration corresponds to a different beam width. In the figures, dovetailing is only outperforming the average configuration if the ratio is less than 1.

Notice that in both domains, dovetailing is generally a more effective enhancement with the smaller beam widths. Smaller beam widths result in a greedier search of the space. If a greedier search is mislead, it requires extra effort when trying to correct itself. Dovetailing minimizes this problem by having multiple instances running at the same time in the hopes that not all instances will be mislead. Smaller beam widths also have an increased chance of splitting nodes with an equal f-cost among multiple slices. Therefore, the amount of diversity between configurations is much greater for small beam widths than it is for larger beam widths.

The only starting configuration set which did not follow this trend is the beam width 7 starting configuration set in the 6×6 sliding tile puzzle. In this set, only one of the configurations ever needed to backtrack, and it only needed to do so on a single problem. This is unlike the rest of the starting configuration sets considered in which backtracking was needed by all configurations. Without any backtracking, all configurations in the beam width 7 set end up solving the problems in a similar amount of time. As such, dovetailing yields poorer performance in this starting configuration set than it does in starting configuration sets with a larger beam width.

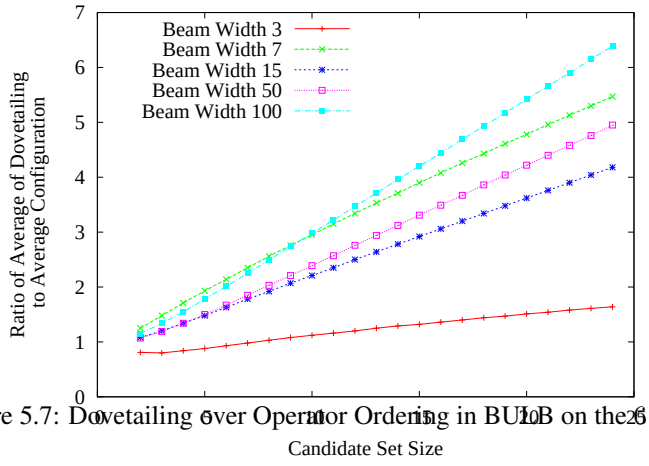


Figure 5.7: Dovetailing over Operator Orderings in BULB on the 6×6 Sliding Tile Puzzle.

Overall, the performance of dovetailing and parallel dovetailing offers little to BULB in both domains except on the smallest of beam widths. In the case of parallel dovetailing, either super-linear or near-linear speedups are seen for the starting configuration with the beam width of 3. However, as the beam width increases, the speedups seen decrease rapidly. For example, with the beam width 100 starting configuration set, the speedup is only by a factor of 3.8 when using 24 processors in the 6×6 sliding tile puzzle, and 2.1 when using 15 processors in the 16 pancake puzzle.

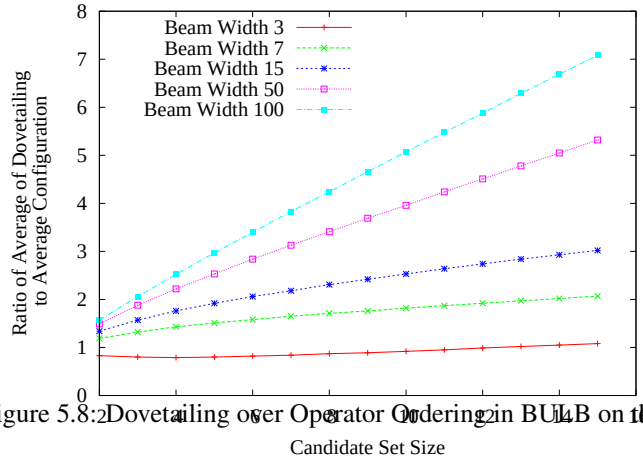


Figure 5.8: Dovetailing over Operator Orderings in BULB on the 16 Pancake Puzzle.

As was the case in WA* and WRBFS, dovetailing over an increasing number of operator orderings also improves the solution quality when the procedure is applied to BULB. This effect is the most pronounced on the low beam widths. On the starting configuration set with beam width 3, the average total cost in the 6×6 sliding tile puzzle is $3,132,084$ and $913,421$ for candidate set sizes

of 2 and 24 respectively. These correspond to an improvement in solution quality over the average solution quality of the configurations alone by factors of 1.13 and 3.88, respectively.

The solution quality also improves with the number of candidate sets by a significant amount when using large beam widths. For example, with the beam width 100 starting configuration, dovetailing over all 24 configurations achieved an improvement of 2.4 in the solution quality over the average configuration alone.

5.6 Chapter Summary

In this chapter, dovetailing over configurations that differ only in the static operator ordering used was considered for the WA*, WIDA*, WRBFS, and BULB algorithms. When using a starting configuration set in which all configurations have a weight w, dovetailing was shown to improve the solution quality of the WA* and WRBFS algorithms when compared to the average solution quality found when searching with the configurations alone. In the case of WA*, this significantly increased the search time. A similar result was seen in the BULB.

Dovetailing over operator orderings was also considered as a parallelization of WA* and was compared to the wPBNF algorithm in the sliding tile puzzle domain. wPBNF has been shown to perform well with low weights, although it was unable to speedup the search with larger weights. In contrast, the amount of speedup gained through parallel dovetailing actually increased with the weight value and outperformed wPBNF in these situations.

In the case of WRBFS, the performance of dovetailing over operator orderings was dependent on the domain considered. In the sliding tile puzzle, the algorithm saw significant speedups with dovetailing. In the pancake puzzle, dovetailing significantly effectively enhanced weighted starting configuration sets but significantly increased the search time for low weighted sets.

WIDA* was the only algorithm which showed improvement in both domains when dovetailing over operator orderings was applied to it. This improvement increased as the weight became larger. The only weights on which dovetailing was less effective was on very small weights. In the extreme case of this behaviour (*ie.* with a weight of 1), dovetailing over operator ordering was shown to be an ineffective procedure when used for optimal problem-solving with the IDA* algorithm. The same was also shown to be true for RBFS.

Chapter 6

Conclusion

Recall that the initial problem we considered was that of proper configuration selection for single-agent search algorithms. One of the motivations behind such an investigation is that while batch tuning does well to find a single configuration that has a good average performance over problems in a domain, this configuration will often perform poorly on a number of individual problems. In such cases, configuration selection should be performed on a problem-by-problem basis.

Unfortunately, a number of issues arise when developing such as system, as outlined in Section 3.3.1. As an alternative, we considered the use of dovetailing as a enhancement to the WA*, WIDA*, WRBFS, and BULB single-agent search algorithms. Dovetailing involves simultaneously searching with a configuration portfolio as opposed to a single configuration. Dovetailing allows configurations to overcome the weaknesses of one another, as opposed to the traditional use of a batch tuning which can result in poor performance due to the over-reliance on a single configuration.

	Dovetailing Speedup over the Single Best Configuration On the Largest Puzzle Problem Tested			
	Using 15 Configurations in the Sliding Tile Puzzle		Using 15 Configurations in the Pancake Puzzle	
Algorithm	Sequential	Parallel	Sequential	Parallel
WA*	0.21	3.2	0.10	1.6
WIDA*	121	1826	1.8	27
WRBFS	2.5	37	0.59	8.9
BULB	0.12	1.8	0.19	2.9

Table 6.1: A Summary of Speedup Results for Dovetailing Over the Main Parameter Space of Algorithms.

Table 6.1 shows the speedup found when dovetailing in the largest puzzle tested with each algorithm in each domain. This table is concerned with dovetailing and parallel dovetailing over the main algorithm parameter spaces. The speedups shown are for the candidate sets that contain all the configurations in the starting configuration set used with that algorithm. The numbers shown represent the speedup gained from performing dovetailing when compared to the single best con-

figuration in the candidate set. The parallel values assume that each configuration is assigned to a separate processor.

Table 6.2 shows an analogous summarization of results for dovetailing over operator orderings. In this case, the comparison is between sequential dovetailing and the average performance over all configurations in the candidate set. All configurations in the candidate sets are identical except that each uses a different operator ordering. The setting of the main parameter in the candidate set is shown beside the algorithm name (*ie.* all configurations in the candidate set used with WA* have a weight of 6).

	Dovetailing Speedup over the Average Configuration On the Largest Puzzle Problem Tested			
	Using 24 Operator Orderings in the Sliding Tile Puzzle		Using 15 Operator Orderings in the Pancake Puzzle	
Algorithm	Sequential	Parallel	Sequential	Parallel
WA* with weight 6	0.40	9.6	0.25	3.8
WIDA* with weight 6	223	5352	4.3	65
WRBFS with weight 6	6.1	146	0.67	10
BULB with beam width 15	0.24	5.8	0.33	5.0

Table 6.2: A Summary of Speedup Results for Dovetailing Over Operator Orderings.

While these tables show that dovetailing offers significant speedups in many situations (or at least similar behaviour without any offline tuning), it does not actually solve the problem of proper configuration selection. Instead, the task of selecting a single effective configuration is replaced with the task of selecting an effective candidate set of configurations. Where offline time is available, batch tuning over possible candidate sets is one solution. The use of a more sophisticated automatic configuration tuner, such as the aforementioned Iterative Local Search system described in Section 3.3, is also a possibility.

Where offline time is not available — such as is the case with general systems that must be able to handle many different domains — previous approaches cannot be used. In such situations, we assert that the use of a single configuration is more error-prone approach than is the selection of a candidate set. This is because the use of multiple configurations helps overcome the deficiencies of any single configuration at a linear cost in the candidate set size.

When using only a single configuration, there is no such backup. Even if a strong single configuration is found, it cannot be expected to do well on all problems in all domains. Such a configuration can always be placed in a candidate set so as to help prevent poor performance on certain problems found to be difficult by that configuration.

As a future direction of research, we consider the problem of automatically constructing good candidate sets for a specific domain. In the remainder of this chapter, we will detail some preliminary research into several aspects of this topic. First, we will address the issue of having multiple

configurations in a candidate set which duplicate search effort. We will then describe some preliminary ideas towards understanding the distribution of work performed over a set of configurations, using a distribution to estimate the ideal size k for a candidate set, and finding the best candidate set of size k from a larger starting configuration set. Finally, we will finish with some closing remarks.

6.1 Removing Duplicates From Candidate Sets

In Sections 2.8.2 and 2.10.1, it was shown that on the 14 pancake puzzle, all weights larger than 10 yield identical searches when used with both WA* and WRBFS. If the automatic construction of candidate sets is not performed carefully, dovetailing can perform unneccessary work due to presence of essentially duplicate configurations in the candidate set.

To address this issue, we suggest the use of the following procedure, which we will explain by example. Consider having a candidate set Θ for some WRBFS search, where $|\Theta| = k$. On the first problem in the problem set, all configurations will begin working on only a single search tree instead of k trees as is done normally. Each configuration will maintain its own bounds at every level of the tree. However, as soon as some configuration θ disagrees with the rest of the $k - 1$ configurations in terms of which node should be expanded next, θ will break away from the others. This break will involve having the search tree being copied. Work on the two subtrees — that being worked on by θ and that be worked on by Θ/θ — will then continue in a dovetailing fashion.

This process can be generalized as follows. Instead of having only a single configuration split away, θ will be separated into subsets $\theta_1, ..., \theta_j$ such that there is no intersection between these configuration sets, $\theta_1 \cup ... \cup \theta_j = \theta$, and each of these subsets selects a unique node to expand. At this point in the search, the search tree is copied j times and each of the subsets is assigned an independent search tree. This process of splitting off into separate search trees continues until a solution is found. All configurations will then be contained in a set of groups $g_1, ..., g_l$, where for all i, the configurations in g_i performed identically. For any future searches, θ can be pruned of all configurations except one in each of g_i.

This procedure allows for the removal of essentially duplicate configurations from the candidate set. It should be clear that this procedure requires no more work or memory than dovetailing itself.

The main disadvantage of using this procedure is that it significantly complicates the implementation of the algorithms. Further domains need to be considered so as to determine if in practice, many design choices will induce configurations that are different in parameter values but the same in execution. For example, it is all but impossible for different beam widths to induce the same search. Similarly, different operator orderings are expected to induce different searches even in algorithms that involve sorting, as long as each node in the search tree does not have a unique f-cost. As such, the presence of "essentially duplicate" configurations may be rare enough such that it is not worth complicating dovetailing with the use of this procedure.

6.2 Analyzing the Distribution of Work Performed by a Starting Configuration Set

One of the properties of a candidate set that determines whether dovetailing will be effective is the distribution of search effort required by the configurations in the set. For example, if the search effort needed by the configurations in some candidate set Θ is similar on many problems, using only a single configuration in Θ will outperform dovetailing due to the overhead of simultaneously running multiple configurations.

In the work of Gomes *et al.*, the distributions of search effort over sets of cofigurations for constraint satisfaction problems were shown to have very long tails [20]. The variability in run-time is even evident when only considering configurations that differ only in the random seed. These distributions were shown to belong to the class of Pareto-Lévy of distributions which have an infinite mean and variance.

One of the ways of identifying these distributions is to consider a log-log plot of the fraction of configurations that required at least x time to solve some problem p. This is because Pareto-Lévy distributions will show approximate linear behaviour in such plots. Note, the fraction of configurations that required at least x times to solve p is the same as the cumulative distribution of work needed on problem p subtracted from 1.

We are currently investigating single-agent search problems so as to determine what design choices show a similar behaviour. Figure 6.1 is a log-log plot of the cumulative distribution subtracted from 1 for 3 different search procedures on the hardest problem in the Korf test set. The line marked "Without Dovetailing" is the cumulative distribution of work for a set of WIDA* configurations subtracted from 1. The starting configuration set is of size 480. All configurations in this set are identical, except each has been given a unique weight from the set $\{1, 1.05, 1.1, ..., 24.90, 24.95, 25\}$.

Notice that the relationship is not linear, particularly over the space in which the first 95% of the problems are solved. The last 5% of the configurations, which are generally the smallest of weights, somewhat skew this interpretation. If we remove these configurations from the set, we conclude that the distribution does not belong to the class of Pareto-Lévy distributions. However, despite this finding, the tail of the distribution can still be said to be long.

The work of Gomes *et al.* also demonstrated that the use of random restarts dramatically improved the performance of the CSP solver by decreasing the length of the distribution tails. The connection between dovetailing and random restarts has already been discussed in Section 3.5, and the figure shows that the impact of dovetailing is similar to that reported for restarts. In the figure, we show the cumulative distribution of work done subtracted from 1 when dovetailing over candidate sets of size 5 and 20. These were found through dovetailing simulations. The candidate set used consisted of the 480 configurations described above. For each candidate set size, we have considered $10,000$ randomly selected candidate sets.

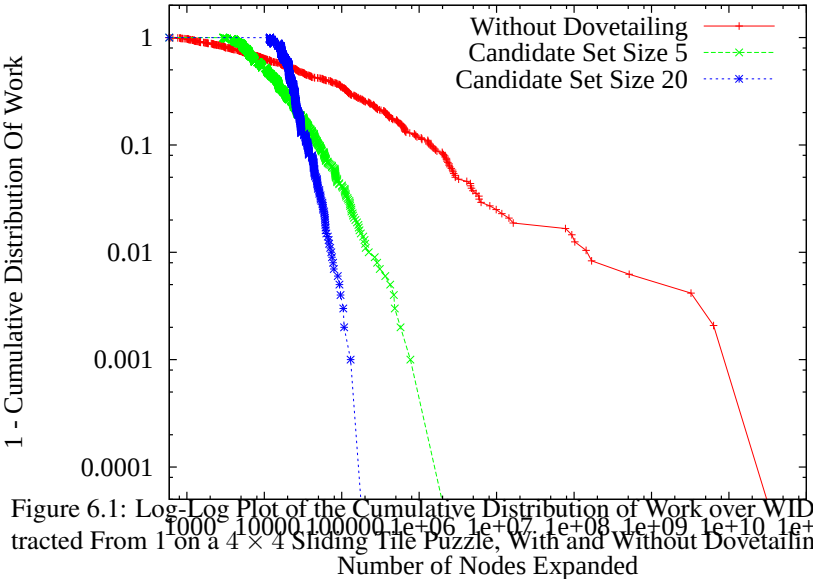


Figure 6.1: Log-Log Plot of the Cumulative Distribution of Work over WIDA* Configurations Subtracted From 1 on a 4×4 Sliding Tile Puzzle, With and Without Dovetailing.

The introduction of dovetailing severely decreases the length of the tails of the distribution. The larger the candidate set size, the smaller the tail becomes. The cost of dovetailing is also on display in the figure. Notice that the minimum number of nodes needed by dovetailing over the candidate sets of size 20 is larger than a significant proportion of the 480 configurations. However, the behaviour of the worst case is improved substantially.

As the distribution of search effort does not appear to fall in the class of Pareto-Lévy distributions, we have been working in collaboration with Karen Buro [9] on determining what distributions it is similar to. Preliminary results on a number of 4×4 sliding tile puzzle problems suggest that the log of the distribution of work is most similar to a gamma or beta distribution. However, these results require more analysis.

If the distribution of search effort for a domain is understood, this knowledge could potentially be used to inform candidate set selection. In the next section, we consider some early ideas into using the distribution to help determine an appropriate size for candidate sets.

6.3 Finding Effective Candidate Set Sizes

Consider the problem of finding an effective candidate set from a larger starting configuration set. One approach which we are currently investigating involves first determining a candidate set size that is expected to yield good performance. Once a candidate set size has been decided upon, it will then be necessary to find an effective set of configurations of that size. An early approach to that problem will be considered in Section 6.4.

The issue of candidate set size selection involves balancing the coverage of the candidate set against the cost of dovetailing. If the size is too small, the probability that the candidate set contains a configuration that has good performance on every problem in a problem set will decrease. However, as the size increases, so does the overhead of running a number of configurations simultaneously.

In order to estimate an effective candidate set size, we can use the distribution of the number of nodes expanded by configurations in a starting configuration set Ω on a problem p. This distribution will be given by the random variable X. So as to simplify the mathematics, Ω will be assumed to be infinite.

Let us now consider candidate sets of size k that consist of random configurations from Ω, selected with replacement. Let $\Phi(k)$ denote the expected amount of work needed by dovetailing over any such random candidate set of size k. This value will be given by the expected minimum of the number of nodes expanded by k configurations in Ω, multiplied by k.

Let $F_Y(b)$ denote the cumulative density function of Y, formally defined as $F_Y(b) = P(Y \leq b)$. f_Y will be used to denote the probability density function of Y, formally described as the derivate of $F_Y(b)$ with respect to b. Finally, let $X_1, ..., X_k$ denote the distributions over each of the k randomly selected configurations, and let Ψ denote the distribution of the minimum over these k random variables. This means that $F_\Psi(x) = 1 - P(\cap_{i=1}^{k} X_i > x)$. The following algebraic expressions then follow:

$$\begin{aligned} F_\Psi(x) &= 1 - P(\cap_{i=1}^{k} X_i > x) && (6.1)\\ &= 1 - P(X_1 > x)P(X_2 > x)...P(X_k > x) && (6.2)\\ &= 1 - P(X > x)^k && (6.3)\\ &= 1 - [1 - F_X(x)]^k && (6.4) \end{aligned}$$

Line 6.2 follows since the configurations were selected at random and therefore the associated random distributions are independent. Line 6.3 follows from the fact that the configurations are all taken from the same starting configuration set and so all of the X_i's share the same distribution, namely X. Line 6.4 is true by the definition of the cumulative distribution function.

With this expression, it is now possible to calculate the probability density function $f_\Psi(x)$. By simply differentiating $F_\Psi(x)$ with respect to x, we find that $f_\Psi(x) = k f_X(x)[1 - F_X(x)]^{k-1}$.

Where $E(Y)$ denotes the expected value of Y, it should be clear that $\Phi(k) = kE(\Psi)$. As the expected value of a distribution Y is given by $E(Y) = \int_{-\infty}^{\infty} y F_Y(y) dy$, we find the following:

$$\Phi(k) = k^2 \int_0^\infty x f_X(x)[1 - F_X(x)]^{k-1}.$$

By simply evaluating this formula for different values of k, it will be possible to estimate the candidate set size which leads to the minimum expected amount of work, denoted k^*. k^* can then be used as the candidate set size for all other problems in the same domain, based on the assumption that different problems in the same domain share similar distributions.

Unfortunately, closed-form solutions can only be found for certain distributions. For example, similar calculations have resulted in exact solutions of $E(X^k_{min})$ where X is the exponential or geometric distribution [10]. In order to use this formula in practice, it will be necessary to evaluate the expression numerically.

When considering WIDA* configurations, the value of $E(\Psi)$ is strongly related to the parameter of α_Θ described in Section 5.4. Whereas α_Θ is the expected proportion of the work needed in the final iteration, $E(\Psi)$ considers the entire search time of the procedure. This consideration of the distribution of work is also a more general model that is not necessarily constrained to the case of optimal search.

The effectiveness of this model in estimating a good value for k^* will depend on how well the underlying assumptions apply. One of these assumptions is in regards to how much the performance of the configurations will correlate between problems. When dovetailing over WA* configurations that differ in the weight walue, the model will fail since higher weights tend to do better on all problems. When dovetailing over WIDA* configurations that differ in operator ordering, there is little correlation between problems. As such, the model is expected to work better in this case.

The other assumption involves the similarity of the work distributions between problems. Our preliminary results on the 4×4 puzzle suggest that in that domain, the assumption holds over WIDA* configurations that differ in weight value. However, more investigation is needed into this claim, particularly as it applies to other domains and other algorithms.

6.4 Finding Effective Candidate Sets of Size k

Once a good size for candidate sets has been found, a set of configurations of that size still needs to be selected. In certain situations, randomly selecting k configurations from the starting configuration set will be an effective policy. One such case would be where the algorithm relies heavily on stochastic behaviour and the configurations only differ in the random number generator. However, generally there is expected to be some stronger relationship between configurations, and diversity in the candidate set should be an aim. For example, consider configurations that only differ in the static operator ordering. It would be expected that in most cases, candidate set configurations should avoid having the same operator as the first in the ordering so as to avoid having multiple configurations expanding nodes in a similar order. As such, selecting configurations at random is not expected to be a general solution.

In this section, we will outline one idea for candidate set selection of size k, that begins with an arbitrary set of configurations from a starting configuration set Ω, and improves this set based on information gathered through the solving of problems. This idea will be based on the consideration of candidate selection as a modified N-armed bandit problem.

First, we will define the traditional version of the problem. We follow the definition given by Sutton and Barto [44]. The N-armed bandit problem can be described as follows: an agent is

faced with N slot machines, each of which will have a different distribution of rewards. These distributions are unknown to the agent who must successively select a slot machine to play. The agent's task is to maximize its payoff over time. To do so effectively, the agent must build a model of the payoffs of each machine and play the machines which offer the highest expected outcome most often. However, the agent must also avoid over-playing these machines due to the fact that the stochasticity of the machines can lead to errors in the models.

In the case of selecting a candidate set of size k, we can consider the arms to each correspond to a configuration in Ω. However, instead of selecting only a single arm to play, the agent will simultaneously play k arms. A play will be made for each problem instance that is to be solved, and the arms selected will correspond to the candidate set to use for that problem. That candidate set will then be used for problem-solving.

In order to apply bandit algorithms, a procedure for the assignment of rewards is necessary. Currently, we use the following strategy: where $\Theta \subseteq \Omega$ is selected as the candidate set to use on problem p and $\theta \in \Theta$ is the configuration which solves p during dovetailing, θ is assigned a reward of 1. All other configurations in Θ are assigned a reward of 0 since they failed to solve the problem.

We can then use a modified version of the UCB1 algorithm [2]. UCB1 was initially designed for the traditional N-armed bandit problems. It starts by selecting each of the bandit arms once. In order to make the $t+1$st play, where $t \geq N$, the algorithm requires the calculation of the *UCB1-value*, denoted $V(j)$, of each arm j. $V(j)$ is defined as follows:

$$V(j) = \bar{x_j} + C\sqrt{\ln t / t_j}$$

where $\bar{x_j}$ is the average reward for arm j seen thus far; C is a positive, real-valued algorithm parameter; and t_j is the number of times that arm j has been selected thus far. The $t+1$st play is then made as the arm with the highest UCB1-value. Note, in the original formulation of the algorithm, $C = \sqrt{2}$. The extension to other constants was introduced by Kocsis and Szepesvári [29].

For candidate set selection, we use a modified version of the UCB1 algorithm called *UCB1 dovetailing*. When selecting a candidate set of size k, the set is first filled with configurations that have yet to be used in any candidate set thus far. For any such configuration θ, $t_\theta = 0$. If only k' such configurations remain (where $k' < k$), then the candidate set is filled with the $k - k'$ configurations with the highest UCB1-values. This candidate set is then used to solve some problem p in the problem set. When problem solving completes, the rewards are assigned, and the value of t_θ is incremented for all configurations in the candidate set. The process then repeats when a candidate set is to be selected for the next problem. Notice that this approach to candidate set selection has no information regarding the problem to solve. Instead, decisions are based on statistics collected from previous problem-solving instances.

The execution of UCB1 dovetailing is deterministic. However, there are several design choices that can affect the performance of the algorithm. The most obvious of these is the value of C.

This C parameter allows for the designer to tune how much the algorithm balances exploitation and exploration. The algorithm is said to make an exploitive selection when a configuration is added to the candidate set that has shown to have a high average reward thus far. As mentioned before, this model of the value of a configuration can be flawed. As such, other configurations that appear less desirable should be selected from time to time so as to avoid exploiting the wrong configuration. Increasing C will increase how likely the algorithm makes such exploratory selections.

Another aspect of the algorithm that affects its performance is the way in which configurations are selected for the candidate set in the presence of ties. This includes selecting between two configurations that have yet to ever be used in a candidate set. In our experiments, ties are broken by a static ordering of the starting configuration set $\theta_1, \theta_2, ..., \theta_N$. Configurations with a smaller index in this ordering are always preferred in the presence of ties.

To see how the static ordering of configurations can significantly impact the performance of the algorithm, consider a starting configuration set $\{\theta_1, \theta_2, \theta_3, \theta_4\}$. Assume $k = 2$ and that configurations θ_1 and θ_2 perform poorly on all problems in the problem set, while θ_3 and θ_4 perform well on all problems. If the initial ordering is given by the ascending order of index, then the candidate set $\{\theta_1, \theta_2\}$ will be tested on the first problem, and $\{\theta_3, \theta_4\}$ will be tested on the second problem. Assume that θ_1 and θ_3 solve problems 1 and 2 during dovetailing, respectively. The average rewards of these configurations after solving the first two problems will be 1, and the average rewards found by θ_2 and θ_4 will be 0.

If the ordering is given by $\{\theta_1, \theta_3, \theta_2, \theta_4\}$, the candidate sets used on problems 1 and 2 will be $\{\theta_1, \theta_3\}$ and $\{\theta_2, \theta_4\}$ respectively. After solving these problems, the relative ordering of the configurations by reward value is much more accurate than it is after solving the 2 problems with the above ordering. While the algorithm will converge to a correct model of each configuration eventually — even with the first ordering — this may take a lot of time. As we are considering problem sets with a relatively small finite size, the impact of having the second ordering exploit the correct configurations more often, particularly early on, may be quite large.

Figure 6.2 shows a preliminary test with this algorithm. The starting configuration set consists of 15 WIDA* configurations that differ only in the value of the weight. Each configuration has a unique integer weight in the range from 2 to 16, inclusive. The problem set consists of 1000 5×5 sliding tile puzzle problems. For each candidate set size, $10,000$ different random orderings of the starting configuration set are considered.

For two values of C, the figure shows the average performance over the $10,000$ orderings for each of the candidate set sizes. For comparison, we also show the average performance of regular dovetailing over the same starting configuration set.

The figure shows that with both values of C, the average performance of UCB1 outperforms the average performance of regular dovetailing on all candidate set sizes. The same is true when C was set as 0.1, $\sqrt{2}$, and 10. However, the improvement is generally small except on the smaller candidate

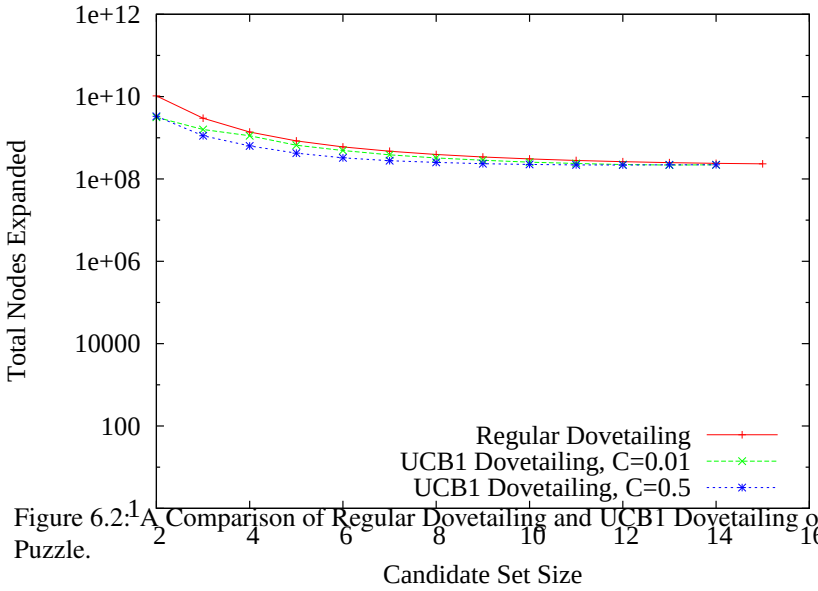


Figure 6.2: A Comparison of Regular Dovetailing and UCB1 Dovetailing on the 5×5 Sliding Tile Puzzle.

set sizes. For example, consider the candidate set size of 2. With this set size, 3 of the 5 values of C tested improved the average performance by at least a factor of 3, 1 of the 5 values improved the performance by almost a factor of 2; and the final value, 10, showed almost identical performance to regular dovetailing.

Our investigation of UCB1 dovetailing remains in its early stages. A more in-depth analysis as to where UCB1 dovetailing succeeds and fails is still needed.

6.5 Contributions and Closing Remarks

In this thesis, we have considered the problem of configuration selection for suboptimal single-agent search algorithms. In this end, we considered the performance of WA*, WIDA*, WRBFS, and BULB in two domains: the sliding tile puzzle and the pancake puzzle. It was shown that while offline tuning can capably find a configuration that has good average performance, this configuration will often exhibit poor performance on some number of individual problems. To deal with this issue, we considered the use of dovetailing which simultaneously runs multiple instances of the same algorithm, each with a different configuration, by interleaving execution.

Chapters 4 and 5 were concerned with the evaluation of the performance of dovetailing as an enhancement to suboptimal single-agent search algorithms. The design choice spaces over which dovetailing was considered were those of operator orderings and the main parameters in the aforementioned algorithms. The results of single-processor dovetailing with WA* and BULB were negative: dovetailing caused the speed of both of these algorithms to degrade in all domains considered.

However, when parallel dovetailing, at least some speedup is seen in almost all tests. For example, it has also been demonstrated that parallel dovetailing outperforms the state-of-the-art parallelization of WA* in the sliding tile puzzle domain for higher weight values. This suggests that while dovetailing is not an effective addition to these algorithms, parallel dovetailing remains an intriguing approach to parallel single-agent search.

When dovetailing was used with WRBFS, the results are mixed. In the larger sliding tile puzzles, dovetailing improves upon the use of any individual configuration alone. Parallel dovetailing also showed super-linear speedups in this domain. In the pancake puzzle, dovetailing over configurations that differ in the weight value performed similarly to the single best configuration found with batch tuning. However, dovetailing did not require any offline computation in order to do so. When considering the use of configurations in this domain that only differ in operator ordering, dovetailing was only effective for the higher weight values. Parallel dovetailing was also shown to be an effective parallelization of the WRBFS algorithm in most of the situations tested.

The algorithm which benefited the most from the use of dovetailing is WIDA*. Dovetailing significantly improved the performance of WIDA* in almost all tests performed. It was not unusual to see dovetailing improve upon even the single configuration with the best average performance by several orders of magnitude. The results suggest that WIDA* should never be used without dovetailing except where the configurations use weight values that are near 1.

While we have shown that dovetailing can be an effective enhancement for suboptimal search algorithms and that the procedure helps to deal with the issue of configuration selection, this work also suggests a number of areas for future work. Among these is an investigation into automatic configurations selection. While we have included some preliminary results on this topic earlier in this chapter, much work remains to be done in this area.

Another important step is to consider other design choices of the examined algorithms so as to determine additional methods of introducing diversity among configurations. For example, we are currently experimenting with dovetailing over incomplete versions of BULB, where each configuration only performs a single iteration of a BULB search. This is similar to the approach taken by the Parallel Window Search algorithm described in Section 3.5. BULB has shown to be a very effective algorithm in practice, and so by introducing such diversity it may be possible to further extend the size of problems it can solve. Additionally, dovetailing over configurations that differ in the heuristic function being used, or even completely different algorithms, remains an interesting area of research.

Finally, dovetailing should also be tested with the above algorithms in more domains. In the domains already considered, there is a solution below any branch in the search tree. In many domains, this is not true, and instead there are dead-ends in the tree. Dovetailing is expected to perform well in these domains as the use of multiple configurations is expected to mitigate the issues that occur when any configuration becomes stuck in a dead-end. However, this remains to be shown.

Bibliography

[1] *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*. AAAI Press, 2007.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[3] Amitava Bagchi and Ambuj Mahanti. Search Algorithms Under Different Kinds of Heuristics-A Comparative Study. *J. ACM*, 30(1):1–21, 1983.

[4] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

[5] Vadim Bulitko, Mitja Lustrek, Jonathan Schaeffer, Yngvi Björnsson, and Sverrir Sigmundarson. Dynamic control in real-time heuristic search. *J. Artif. Intell. Res. (JAIR)*, 32:419–452, 2008.

[6] Ethan Burns, Seth Lemons, Wheeler Ruml, and Rong Zhou. Parallel Best-First Search: Optimal and Suboptimal Solutions. In *Proceedings of the International Symposium on Combinatorial Search (SoCS-09)*, 2009.

[7] Ethan Burns, Seth Lemons, Wheeler Ruml, and Rong Zhou. Suboptimal and Anytime Heuristic Search on Multi-Core Machines. In *ICAPS*, 2009.

[8] Ethan Burns, Seth Lemons, Rong Zhou, and Wheeler Ruml. Best-First Heuristic Search for Multi-Core Machines. In *IJCAI*, 2009.

[9] Karen Buro, 2009. Private communication.

[10] Gianfranco Ciardo, Lawrence M. Leemis, and David Nicol. On the Minimum of Independent Geometrically Distributed Random Variables. Technical Report TR-94-12, 1994.

[11] Diane J. Cook and R. Craig Varnell. Maximizing the Benefits of Parallel Search Using Machine Learning. In *AAAI*, pages 559–564, 1997.

[12] Joseph C. Culberson and Jonathan Schaeffer. Searching with Pattern Databases. In Gordon I. McCalla, editor, *Canadian Conference on AI*, volume 1081 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 1996.

[13] Henry W. Davis, Anna Bramanti-Gregor, and Jin Wang. The Advantages of Using Depth and Breadth Components in Heuristic Search. In *ISMIS*, pages 19–28, 1988.

[14] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[15] Matthew P. Evett, James A. Hendler, Ambuj Mahanti, and Dana S. Nau. PRA*: Massively Parallel Heuristic Search. *J. Parallel Distrib. Comput.*, 25(2):133–143, 1995.

[16] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K-Best-First Search. *Ann. Math. Artif. Intell.*, 39(1-2):19–39, 2003.

[17] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual Lookups in Pattern Databases. In *IJCAI*, pages 103–108, 2005.

[18] David Furcy and Sven Koenig. Limited Discrepancy Beam Search. In *IJCAI*, pages 125–131, 2005.

[19] David Furcy and Sven Koenig. Scaling up WA* with Commitment and Diversity. In *IJCAI*, pages 1521–1522, 2005.

[20] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.

[21] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *IJCAI (1)*, pages 607–615, 1995.

[22] Malte Helmert. The Fast Downward Planning System. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

[23] Malte Helmert and Gabriele Röger. How Good is Almost Perfect? In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 944–949. AAAI Press, 2008.

[24] Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple Pattern Databases. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 122–131. AAAI, 2004.

[25] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry A. Kautz, Bart Selman, and David Maxwell Chickering. A Bayesian Approach to Tackling Hard Computational Problems. In Jack S. Breese and Daphne Koller, editors, *UAI*, pages 235–244. Morgan Kaufmann, 2001.

[26] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic Algorithm Configuration Based on Local Search. In *AAAI* [1], pages 1152–1157.

[27] Y Kitamura, M Yokoo, T Miyaji, and S Tatsumi. Multi-state commitment search. In *Tools for Artificial Intelligence*, pages 431–439, 1998.

[28] Kevin Knight. Are Many Reactive Agents Better Than a Few Deliberative Ones? In *IJCAI*, pages 432–437, 1993.

[29] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[30] Richard E. Korf. Iterative-Deepening-A*: An Optimal Admissible Tree Search. In *IJCAI*, pages 1034–1036, 1985.

[31] Richard E. Korf. Linear-Space Best-First Search. *Artif. Intell.*, 62(1):41–78, 1993.

[32] Greg Lee and Vadim Bulitko. GAMM: genetic algorithms with meta-models for vision. In Hans-Georg Beyer and Una-May O'Reilly, editors, *GECCO*, pages 2029–2036. ACM, 2005.

[33] Matthew McNaughton, Paul Lu, Jonathan Schaeffer, and Duane Szafron. Memory-Efficient A* Heuristics for Multiple Sequence Alignment. In *AAAI/IAAI*, pages 737–743, 2002.

[34] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM, 2001.

[35] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1982.

[36] Donald J. Patterson and Henry A. Kautz. Auto-Walksat: A Self-Tuning Implementation of Walksat. *Electronic Notes in Discrete Mathematics*, 9:360–368, 2001.

[37] Ira Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artif. Intell.*, 1(3):193–204, 1970.

[38] Ira Pohl. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *IJCAI*, pages 12–17, 1973.

[39] Curt Powley and Richard E. Korf. Single-Agent Parallel Window Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 13(5):466–477, 1991.

[40] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

[41] Stuart J. Russell and Eric Wefald. Principles of metareasoning. *Artif. Intell.*, 49(1-3):361–395, 1991.

[42] Anthony Stentz. The Focussed D* Algorithm for Real-Time Replanning. In *IJCAI*, pages 1652–1659, 1995.

[43] Bryan Stout. Smart moves: Intelligent Pathfinding. *Game Developer Magazine*, October:28–35, 1995.

[44] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[45] Rong Zhou and Eric A. Hansen. Beam-Stack Search: Integrating Backtracking with Beam Search. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *ICAPS*, pages 90–98. AAAI, 2005.

[46] Rong Zhou and Eric A. Hansen. Parallel Structured Duplicate Detection. In *AAAI* [1], pages 1217–.