University of Alberta

BOOTSTRAP LEARNING OF HEURISTIC FUNCTIONS

by

Shahab Jabbari Arfaee

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

Examining Committee

Robert C. Holte, Computing Science

Sandra Zilles, Computer Science, University of Regina

Sean Gouglas, Department of History & Classics

Jonathan Schaeffer, Computing Science

Abstract

We investigate the use of machine learning to create effective heuristics for single-agent search. Our method aims to generate a sequence of heuristics from a given weak heuristic h_0 and a set of unlabeled training instances using a bootstrapping procedure. The training instances that can be solved using h_0 provide training examples for a learning algorithm that produces a heuristic h_1 that is expected to be stronger than h_0. If h_0 is so weak that it cannot solve any of the given instances we use random walks backward from the goal state to create a sequence of successively more difficult training instances starting with ones that are guaranteed to be solvable by h_0. The bootstrap process is then repeated using h_i instead of h_{i-1} until a sufficiently strong heuristic is produced. We test this method on the 15- and 24-sliding tile puzzles, the 17- , 24- , and 35-pancake puzzles, Rubik's Cube, and the 15- and 20-blocks world. In every case our method produces heuristics that allow IDA* to solve randomly generated problem instances quickly with solutions very close to optimal.

The total time for the bootstrap process to create strong heuristics for large problems is several days. To make the process efficient when only a single test instance needs to be solved, we look for a balance in the time spent on learning better heuristics and the time needed to solve the test instance using the current set of learned heuristics. We alternate between the execution of two threads, namely the learning thread (to learn better heuristics) and the solving thread (to solve the test instance). The solving thread is split up into sub-threads. The first solving sub-thread aims at solving the instance using the initial heuristic. When a new heuristic is learned in the learning thread, an additional solving sub-thread is started which uses the new heuristic to try to solve the instance. The total time by which we evaluate this process is the sum of the times used by both threads up to the point when the instance is solved in one sub-thread. The experimental results of this method on large search spaces demonstrate that the single instance of large problems are solved substantially faster than the total time needed for the bootstrap process while the solutions obtained are still very close to optimal.

Acknowledgements

First and foremost, I would like to thank my advisors, Robert C. Holte and Sandra Zilles, for their great support and valuable guidance in this project. Rob and Sandra have been a tremendous help with developing new ideas on this research.

I thank Neil Burch for providing the code for Rubik's Cube and Richard Valenzano for sharing his implementation of BULB. I would also like to thank Mehdi Samadi for providing helpful clarification about his work.

Last but not the least, my greatest appreciation goes to my family for their endless support and care.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Consider the problem of traveling from an initial location to a destination location within a city. Heuristic search is a common approach to find a path from the initial location to the destination location. The above problem can be converted to a graph search problem in which each node in the graph is a location within the city. Each edge in the graph then corresponds to a direct path between two locations, *i.e.,* when no other location exists in the middle of the path between the two locations. Heuristic search finds a solution to the problem by searching the graph and finding a path in the graph that leads from the initial location to the destination location.

Heuristic search uses some information, called the heuristic function (heuristic, for short), in addition to the search graph itself, to speed up finding a path between the initial location and the destination location. Although in some cases, finding a path would be sufficient, finding a path with the lowest cost, which is called an optimal solution path, is often more desirable. Any solution path with a cost larger than the cost of an optimal path is called a suboptimal solution path. The heuristic function at each location estimates the cost of reaching the destination from that location. If, for each location in the graph, the value of the heuristic function is a lower bound on the cost of the lowest cost path from the location to the destination location, the heuristic function is called admissible. Heuristic search algorithms exist that guarantee finding an optimal solution path when an admissible heuristic function is given.

In addition to the cost of the path, the time that it takes for the algorithm to find the path is also very important. Modern heuristic search systems require "good" heuristics to ensure that the time needed to find the solution path is acceptable. It can happen that a near-optimal path is preferred to an optimal one as the time needed to find the optimal path can be very long (*e.g.,* on the order of days) whereas suboptimal paths can be found quite quickly. In this thesis, we study a general technique to create heuristics that enable solutions that are close to optimal to be found quickly.

1.1 Approach to the Problem

1.1.1 Bootstrapping

Current methods to automatically create admissible heuristics are unable to create strong heuristics for large problems; therefore, even the best heuristics created by them are too weak to solve arbitrarily instances of large problems in a reasonable time. For example, the best such heuristic for the 24-puzzle[1] needs about two days on average to solve an arbitrary instance of the problem optimally [42]. The problem solving can become much faster if an inadmissible heuristic is used, *i.e.,* when optimal solutions are not required.

One possible approach to creating inadmissible heuristics is to apply machine learning to a set of problem instances whose distance-to-goal is known (the training set) to create a heuristic function that estimates distance-to-goal for an arbitrary problem instance. The learned heuristic might be inadmissible as it may overestimate the distance-to-goal for a new problem instance. This idea has been applied with great success to small search spaces (*e.g.,* the 15-puzzle) [13, 55], but could not be directly applied to larger spaces, *e.g.*, the 24-puzzle, because of the infeasibility of creating a sufficiently large training set containing a sufficiently broad range of distances to goal.

In this thesis, we directly learn heuristics for large search problems by an iterative procedure to improve an initial (weak) heuristic function. We call our approach "bootstrap learning of heuristic functions" (bootstrapping, for short). We experimentally show that bootstrapping succeeds, without modification or manual intervention, to create effective heuristics on both small (*e.g.*, the 15-puzzle) and large problems (*e.g.*, the 24-puzzle).

Initially, this procedure requires an initial heuristic function h_0 and a set of states we call the bootstrap instances. Unlike previous machine learning approaches to create heuristics, there are no solutions given for any instances, and h_0 is not assumed to be strong enough to solve any of the given instances. A standard heuristic search algorithm is run with h_0 in an attempt to solve the bootstrap instances within a given time limit. The set of solved bootstrap instances, together with their solution lengths (not necessarily optimal), is fed to a learning algorithm to create a new heuristic function h_1 that is intended to be better than h_0. After that, the previously unsolved bootstrap instances are used in the same way, using h_1 as the heuristic instead of h_0. This procedure is repeated until all but a handful of the bootstrap instances have been solved or until a succession of iterations fails to solve a large enough number of bootstrap instances that were not solved on the previous iterations.

The initial heuristic, the bootstrap instances, and the features used for learning are user-supplied, but they do not have to be carefully crafted. If the initial heuristic h_0 is too weak to solve even a few of the given bootstrap instances within the given time limit we enhance h_0 by a random walk method that generates bootstrap instances at the "right" level of difficulty (easy enough to be solvable with h_0 but hard enough to yield useful training data for improving h_0). These instances are generated

[1]The puzzles used as testbeds in this thesis are described in Section 2.1.1.

using random walks backward from the goal.

Using bootstrapping to create heuristics for large search spaces, *e.g.,* the 24-puzzle, raises a new issue. The time to create "good" heuristics can be so large that it can only be justified if the heuristic is going to be used to solve many problem instances. For example, it takes about 18 days for bootstrapping to create heuristics for the 24-puzzle that enable solving arbitrary instances of the problem almost optimally in a few seconds. When only a single instance of the 24-puzzle needs to be solved, using the admissible heuristic is preferred to bootstrapping as (i) it is faster in total for the admissible heuristic to solve the instance, and (ii) the solution generated is optimal. Therefore, an approach different than bootstrapping is needed when only a single instance is to be solved.

1.1.2 Interleaving Bootstrapping and Problem Solving

In order to address the problem of solving only a single instance efficiently, we present a variation of bootstrapping that provides a balance between the time spent on learning better heuristics and the time needed to solve the test instance using the current set of available heuristics. This variation interleaves learning new heuristics and solving the problem instance in an attempt to minimize the sum of the learning and solving times. We call this approach "interleaving bootstrapping and problem solving" (interleaving, for short). Interleaving works as follows.

Two threads are created; one, which we call the learning thread, for learning better heuristics from the training instances created by random walks backwards from the goal state, and the other, which we call the solving thread, for trying to solve the test instance. These two threads were alternatively run. The time allocated to each thread is a parameter of interleaving and is set manually.

The solving thread itself is comprised of "solving sub-threads". The first solving sub-thread aims at solving the instance using the initial heuristic. Whenever a new heuristic is learned in the learning thread, a new solving sub-thread is started. This new sub-thread uses the new heuristic to try to solve the test instance. The process stops when the instance is solved in one of the solving sub-threads. The total time by which we evaluate this process is the sum of the times used by both threads (including all the sub-threads) up to the point when the instance is solved in one sub-thread.

1.2 Contributions of this Research

Some parts of Chapter 3 of this thesis are published in the Third Annual Symposium on Combinatorial Search (SoCS 2010) [36]. The major contributions of this research are as follows.

1. We introduced an incremental bootstrapping process to learn heuristic functions for search problems. This work substantially extends previous methods in three regards.

   (i) It does not require the distance-to-goal for any of the given training states to be given.

   (ii) It does not require a strong initial heuristic. Whenever the initial heuristic is so weak that bootstrapping cannot start, bootstrapping is augmented with a random walk method for generating successively more difficult problem instances.

2. We provided experimental evidence that bootstrapping succeeds in producing effective heuristics to solve randomly generated problem instances quickly with solutions that are very close to optimal. On all domains our method systematically outperforms Weighted IDA* (the IDA* equivalent of Weighted A* [48]) and BULB [20]. We further compared our method with the state-of-the-art optimal methods to show that when relatively little degradation from the optimal solution is allowed, our method can solve problems much faster than optimal methods. For example, our method solves an average 24-puzzle instance in less than 10 seconds with solutions that are within 10% of the optimal cost, whereas the optimal methods require 2 days, on average, to solve 24-puzzle instances.

3. We introduced a variation of bootstrapping aimed at minimizing the sum of the learning and solving time. This variation involves interleaving the process of learning heuristics and solving the test instance using the current set of learned heuristics. We experimentally showed that interleaving is very effective in decreasing the learning time of bootstrapping for large search spaces. For example, on the 24-puzzle, its total time to solve each random instance is less than 16 minutes on average while the cost of solutions are, on average, within 7% of optimal.

1.3 Outline

This thesis proceeds as follows. Basic background on heuristic search and the machine learning concepts used throughout this thesis is provided in Chapter 2.

Chapter 3 describes our bootstrap learning method to create strong heuristics from given (weak) ones and our random walk technique to create a sequence of successively more difficult instances to prime Bootstrap. It also includes the experimental results of our method on the 15- and 24-sliding tile puzzles, the 17- , 24- , and 35-pancake puzzles, Rubik's Cube, and the 15- and 20-blocks world.

Chapter 4 presents a variation of the ideas discussed in Chapter 3 that aims to decrease the total time of learning new heuristics and solving a single target instance by interleaving the learning and solving processes. The experimental results and a brief review of notable planning techniques to solve a single instance of a problem are also presented in this chapter.

Chapter 5 reviews other research related to learning heuristic functions to guide search and methods that use random walks to generate successively more difficult training instances. Chapter 6 concludes the thesis. It provides a summary of the findings and limitations of our work. It also discusses some possible future directions of this research.

Chapter 2

Essential Background

This chapter consists of a brief overview of the heuristic search and machine learning concepts used throughout this thesis. In Section 2.1, first the concept of heuristic search is introduced. Next, the test domains used in our experiments are introduced. Then heuristics and a general way of creating heuristics using abstraction are presented. Finally, the heuristic search algorithms used in this thesis and the concept of a weighted heuristic are discussed. In Section 2.2, two learning algorithms, neural networks and linear regression, are described.

2.1 Heuristic Search

Heuristic search is a technique to solve an instance of a search problem. The search space is a graph G that consists of a finite set of nodes V and a set of edges E. Each node represents a state in the search space while each edge (a, b) indicates that state b can be directly reached from state a by a legal move. State b is called a successor of state a. Each edge in the graph G is associated with a cost. We assume all the edge costs are non-negative. In all the test domains used in this thesis (see Section 2.1.1), the edge costs are always 1. The cost of a path between two nodes a and b is the sum of the edge costs in the path between a and b (the length of the path when all the edge costs are 1).

A problem instance is represented by an initial state s and a goal state g. A solution consists of a sequence of edges that leads from s to g. An optimal path between s and g is a path (which is not necessarily unique) with the minimum cost. The cost of an optimal path is called the optimal solution cost of the problem instance. Any solution with a larger cost than the optimal solution cost is called a suboptimal solution. Heuristic search finds the solution to each problem instance by traversing the graph G and finding a sequence of edges that leads from the start state s to the goal state g. The order in which a heuristic search algorithm traverses a graph G is determined by a heuristic function. The heuristic function for each state n estimates the cost of reaching the goal state g from state n.

In the rest of this section, we first briefly introduce the test domains for our experiments. Then we present the concept of heuristic functions and review a general approach to create heuristics using abstraction. Finally, we introduce the search algorithms that were used throughout this thesis and the concept of weighted heuristic.

2.1.1 Domains

We used the following domains for our experiments.

(i) *Sliding-tile puzzle (n^2–1-puzzle)* [61]: The puzzle is an $n \times n$ grid containing of $n^2 - 1$ numbered tiles (numbers are ranging from 1 to n^2–1) and an empty tile called the blank. Each state is represented as a vector of length n^2, in which component k of the vector names what is located in the k^{th} puzzle position (either a number representing a tile or a symbol representing the blank). Every operator swaps the blank with a tile adjacent to it. The objective is to rearrange the tiles from an initial configuration (the start state) to a goal configuration (the goal state) corresponding to $(\text{blank}, 1, \ldots, n^2 - 1)$. Figure 2.1 shows a sample start state and the goal state for the 15-puzzle ($n = 4$).

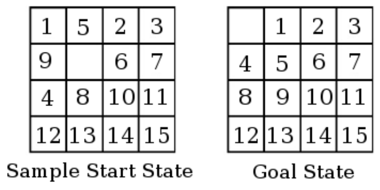


Figure 2.1: A sample start state and the goal state for the 15-puzzle.

The number of reachable states in a n^2–1-puzzle is $\frac{n^2!}{2}$. In our experiments we used $n = 4$ (the 15-puzzle) and $n = 5$ (the 24-puzzle) which respectively contain more than 10^{13} and 10^{25} reachable states. The largest version of the puzzle that has been solved optimally by abstraction-based heuristic search methods is the 24-puzzle [42].

(ii) *n-pancake puzzle* [11] : Each state in the n-pancake puzzle is represented as a permutation of the numbers from 1 to n. It can also be considered as a stack of n pancakes in which the value of the component k of the permutation represents the size of the k^{th} pancake in the stack. In the goal state, the numbers in the permutation will be in an ascending order, *i.e.,* the pancakes are stacked in ascending order of their size (from top to bottom). The goal state for the 6-pancake puzzle is shown in Figure 2.2.[1]

Each state has n–1 successors with the l^{th} successor formed by reversing the order of the first l+1 elements of the permutation (the top l+1 pancakes in the stack). It is possible to reach any state from any other state, therefore, the n-pancake puzzle has $n!$ reachable states. In our experiments we used $n = 17$, $n = 24$, and $n = 35$ which respectively contain more than

[1]This figure is taken from Helmert [28] and modified.

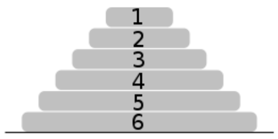


Figure 2.2: The goal state for the 6-pancake puzzle.

10^{14}, 10^{23} and 10^{40} reachable states. The largest version of the puzzle that has been solved optimally by general-purpose abstraction-based methods is $n = 19$ [30].

(iii) *Rubik's Cube* [41]: The version of the puzzle that we used is the standard 3x3x3 cube made up of 20 movable 1x1x1 "cubies" with coloured stickers on each exposed face. The movable cubies can be divided into 8 corner and 12 edge cubies. Each move rotates a face of the cube 90, 180, or 270 degrees clockwise. In the goal state, all the stickers on each face are of the same color. Figure 2.3 shows the goal state for Rubik's Cube.[2] More than 4×10^{19} reachable states exist in this puzzle [41]. The best general purpose admissible heuristics for Rubik's Cube need more than one day, one average, to solve each random instance of the problem optimally [41].

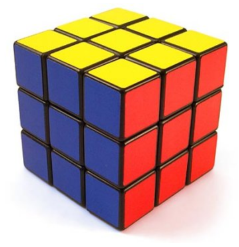


Figure 2.3: The goal state for Rubik's Cube.

(iv) *n-blocks world* [60]: Each state in the n-blocks world is represented as a pair. The first component is a permutation of $\{1, \ldots, n\}$ while the second component is an (n–1)-digit binary number. The permutation represents a sequence of blocks, given by concatenating the sequences of blocks in all the stacks of blocks in the state, starting with the stack that contains block 1, followed in order of the smallest block number contained in a stack. The n–1 digits in the binary number encode, for every two adjacent numbers in the permutation, whether or not they belong to different stacks. Figure 2.4 shows a sample start state of the 5-blocks world. This state is represented by the pair $\{$(1,3,5,2,4), (0100)$\}$ using our representation. The goal state that we chose has all the blocks in one stack in order of the smallest block number at the bottom to the largest block number on the top. For example, the goal state for 5-blocks world is represented by the pair $\{$(1,2,3,4,5), (0000)$\}$ and it is shown in Figure 2.4.

[2]The figure is taken from `http://www.math.cornell.edu/~mec/Winter2009/Lipa/Puzzles/pics/rubiks-cube.jpg` and modified.

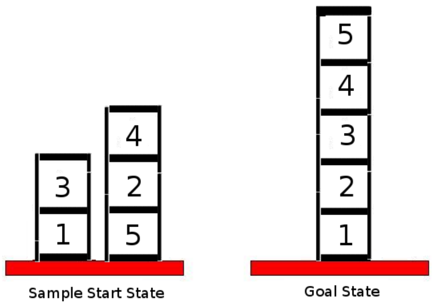


Figure 2.4: A random start state and the goal state for the 5-blocks world.

In each state, only the blocks that are at the top of a stack are allowed to move. For example, in the start state shown in Figure 2.4, only blocks 3 and 4 are allowed to move. Each action moves one block from the top of a stack to the top of another stack or to the table. For example, block 3 can either move to the top of block 4 or to the table. The objective is to change an initial state into the goal state by moving one block at a time. We used $n = 15$ and $n = 20$ in our experiments; the number of reachable states in these domains is more than 10^{13} and 10^{20} respectively [60].

The above representation is called the "handless" blocksworld. In Section 4.3.5, we implemented another version of the blocksworld that uses a hand. The hand is used to pick up and put down the blocks. For example, if block 3 needs to be moved to the top of block 4 in the start state shown in Figure 2.4, it first needs to be picked up by the hand and then the hand needs to put it down on top of block 4. The hand can only pick up a block when it is empty, *i.e.,* when the hand does not hold any other block.

2.1.2 Heuristics

A heuristic function $h(s)$ is a function that estimates the distance from state s to the goal state g. A heuristic h is admissible if for every state s the value of $h(s)$ does not overestimate the cost of the optimal path from state s to the goal state g. The heuristic function h is said to be consistent if for any adjacent nodes a and b in graph G, $h(a)$–$h(b)$ never overestimates the cost of reaching b from state a. When the heuristic value of the goal state ($h(g)$) is zero, a consistent heuristic function will be admissible by definition.

Manhattan distance (MD) is an example of an admissible and consistent heuristic for the sliding-tile puzzle. It is the sum of the vertical and horizontal distances of each non-blank tile from its location at the goal state. For the start state of the 15-puzzle shown in Figure 2.1, tiles 1, 4, 5, 8, and 9 are not in their goal locations; therefore the value of the MD heuristic for this state is 6 (1+1+1+1+2 for tiles 1, 4, 5, 8, and 9). MD is admissible because each non-blank tile should at least be moved from its current location to its position in the goal state and each legal move only moves one non-blank tile one step vertically or horizontally; therefore, MD is less than or equal to the length of any solution path to the goal. It is consistent because each move will change the value of the heuristic by one and the distance between two adjacent nodes in the search space graph is always one.

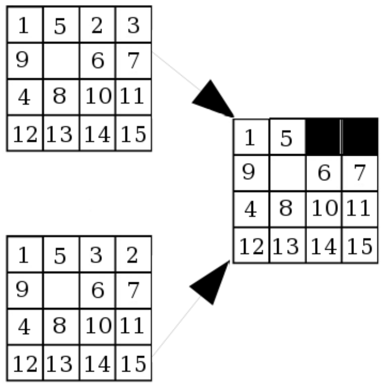


Figure 2.5: Mapping of two states in the 15-puzzle to an abstract state.

A general approach to create heuristics is abstraction. An abstraction is a mapping that maps the nodes in the graph of search space S to nodes in the graph of an abstract search space $\phi(S)$. For example, consider an abstraction of the 15-puzzle that is made by considering tiles 2 and 3 to be equivalent. Figure 2.5 shows that the two states of the 15-puzzle on the left will be mapped to the abstract state on the right if such an abstraction is used for the 15-puzzle. In this mapping the edges are inherited, *i.e.,* if an edge exists between two nodes a and b in the graph of search space S, an edge will exist between $\phi(a)$ and $\phi(b)$ or a and b will be both mapped to the same abstract state. Consider two sets of states A and B in the search space that are respectively mapped to two abstract states $\phi(A)$ and $\phi(B)$. The cost of the shortest cost path between $\phi(A)$ and $\phi(B)$ in the abstract space is never more than the cost of any path between any state $a \in A$ and $b \in B$ in the original search space. Therefore, the distance between the abstract start state $\phi(s)$ and the abstract goal state $\phi(g)$ is guaranteed to be an admissible heuristic [3, 9, 24, 34] for the search space S.

Pattern databases (PDBs) [9] are the most common technique to create heuristics using abstraction. A PDB is a precomputed table of the optimal cost of all the states in the abstract search space from the abstract goal state $\phi(g)$. For each abstract state, the cost of the shortest cost path between the abstract state and the abstract goal state $\phi(g)$ will be stored in the table as the entry for the abstract state. This entry can be used as the heuristic value for the states in the search space that are mapped to this abstract state. When all the edge costs are one, a breadth first search[3] is run to make the table. The breadth first search runs backwards from $\phi(g)$ until all the abstract states are visited.

The memory required to store the table and the time to build the entire pattern database depend

[3]Breadth first search starts with expanding the start state and adding all its successors to a First-In-First-Out queue. Then, the state on the front of the queue is removed and all it successors are added to end of the queue. This process is repeated until the goal state is found.

on the granularity of abstraction. A fine-grained abstraction requires more memory and needs more time to build the PDB but leads to a more accurate heuristic. For example, Holte *et al.* [34] reported that it takes about 3 hours to build a 7-8 additive pattern database [42] for the 15-puzzle. The memory needed to store the PDBs is about 577 megabytes.

To build the 7-8 additive PDB for the 15-puzzle, the non-blank tiles are divided into two groups such that each non-blank tile only belongs to one group. The first group contains tiles 1 to 7 (7 tiles) while the other group contains tiles 8 to 15 (8 tiles). Then a PDB is built for each group; each PDB stores the minimum number of moves required to move the tiles in the group to their locations in the goal state while not counting the moves that involve a tile from the other group. This pair of PDBs is called additive because the value of these two PDBs can be added to create an admissible heuristic for each state. If the moves of the tiles that are not in the group are also counted, the resulting PDB is called non-additive. An admissible heuristic can be created from more than one non-additive PDBs by computing their maximum value for each state.

Our experiments showed that making three non-additive 6-tile PDBs[4] for the same problem takes less than 6 minutes and all the PDBs together need about 173 megabytes of memory. However, solving an instance of the 15-puzzle using a standard search algorithm, *e.g.,* IDA* [39], with the 7-8 additive PDB is on average more than 10,000 times faster than solving the same instance using the maximum of three 6-tile non-additive PDBs. The extensive memory requirement of PDBs limits their application.

Several enhancements have been proposed to extend the range of the problems to which abstraction is applicable. Korf and Felner [42] studied additive PDBs for problems that can be divided into a set of disjoint subproblems. Here, disjoint means that each move in the search space only changes one subproblem. Yang *et al.* [66] generalized this concept of additivity to additive abstractions that can be applied to any search space. Furthermore, compression schemes [2, 12, 56] have been used to decrease the memory requirements of PDBs.

Although these enhancements have improved the performance of the PDBs, it is easy to imagine problems so large that even the best PDB heuristic created (considering the available memory) by these systems will be too weak to enable arbitrary instances to be solved reasonably quickly. For example, the biggest abstraction that can be held in memory for the 24-puzzle (a 6-6-6-6 additive PDB [42]) solves a random instance of the 24-puzzle optimally in two days on average. Therefore, it seems that the PDB heuristic is not very helpful in large domains and one cannot imagine PDBs and their enhancements to be applicable to even larger domains when optimal solutions need to be computed (*e.g.,* the 35-pancake puzzle which is about 10^{15} times larger than the 24-puzzle).

[4]A 6-tile PDB for the 15-puzzle is built by considering 9 of the non-blank tiles to be equivalent in the abstract search space, *e.g.,* when tiles 7 to 15 are considered to be equivalent in the abstract search space. The pattern database built based on this abstraction only considers the location of tiles 1 to 6 and the blank tile to compute the heuristic value of each state of the 15-puzzle.

2.1.3 Heuristic Search Algorithms

A heuristic search algorithm is an algorithm that traverses the graph of the search space using the guidance of the heuristic function to make decisions on which path to follow during the search. A heuristic search algorithm is complete if it is guaranteed to find a solution if one exists.

The performance of search algorithms is measured using time complexity and solution quality. The time complexity of a heuristic search algorithm is the time that the algorithm needs to find a solution. For each heuristic search algorithm, this time is generally proportional to the number of nodes that were generated during the search. For example, if a heuristic search algorithm generates x and $2x$ nodes to solve two problem instances, we expect the time to solve the first problem instance to be about half of the time needed to solve the second problem instance. The solution quality of the algorithm compares the cost of the solution found by the algorithm to the optimal solution cost. The suboptimality of a solution is defined as the difference between the cost of the solution found by the algorithm and the optimal solution cost divided by the optimal solution cost. For example, a suboptimality of 7% means the solution generated was 7% longer than the optimal one. We do not consider the space complexity, the amount of memory that an algorithm requires, as a measure of performance in this thesis as the algorithms that we use in this thesis are bounded memory algorithms.

In the rest of this section, different heuristic search algorithms such as A*, IDA*, RBFS, beam search and BULB are introduced. We use the term *nodes generated* for nodes that were visited by the search algorithm. A node is called *expanded* whenever all its successors are generated.

A*

Best first search algorithms [48] explore the search space graph using an evaluation function f. Best first search ensures that the nodes in the search graph will be expanded in increasing order of their f-value. For each state n in the search graph, $f(n) = g(n) + h(n)$ where $h(n)$ is the heuristic value of state n and $g(n)$ is the cost of reaching state n from the start state. The most famous best first search algorithm is A* [48].

A* uses two lists: OPEN and CLOSED. OPEN stores the nodes that have been generated but not expanded yet. The node that has the minimum f-value will be always selected for expansion from OPEN. CLOSED stores the nodes that have been expanded during the search, *i.e.,* their successors were added to OPEN. Whenever a node is generated, both OPEN and CLOSED will be checked to make sure that the generated node is not a duplicate of a previously generated node. If a node that is in either OPEN or CLOSED is reached during the search with a lower f-value it will be either removed from CLOSED and added to OPEN (if it was in CLOSED) or its f-value will be updated in OPEN (if it was already in OPEN). A* returns a solution when the goal is selected for expansion. If the heuristic h is admissible, A* is guaranteed to find the optimal solution.

A* has high memory requirements as it keeps all the visited states during the search in memory.

Korf suggested algorithms called IDA* [39] and RBFS [40] that have linear memory requirements in terms of the depth of the solution. These algorithms remove the intensive memory consumption of A* by ignoring duplicate detection at the cost of increasing the number of nodes generated.

IDA*

Iterative Deepening A* (IDA*) [39] performs a series of depth first searches bounded by a cost. The cost bound for the first iteration is the heuristic value of the start state. In each iteration of IDA*, all the nodes with an f-value of at most the cost bound will be expanded in a depth first order, *i.e.,* going deeper and deeper in the graph by expanding the first successor of each node. Nodes whose f-value exceed the current bound will be pruned and the cost bound for the next iteration will be set to the minimum f-value of the pruned nodes. The algorithm finishes when a goal is reached whose f-value does not exceed the current cost bound.

IDA* reduces the memory requirements of A* to linear with respect to the maximum search depth. However, it can expand the same node numerous times in total as the search starts anew from the start state at each iteration (with a new cost bound). It has been proved that IDA* will expand the same number of nodes as A* asymptotically if the search graph is a tree [39]. IDA* given an admissible heuristic is guaranteed to find the optimal solution.

RBFS

Recursive Best-First Search (RBFS) [40] is similar to IDA* in terms of memory requirements, but unlike IDA*, it expands nodes in the same order as A*. RBFS performs depth first searches. Instead of expanding the children in a specific order (as is done by IDA*), RBFS selects the child with the lowest f-value for expansion. It has a cost bound similar to IDA* and also keeps track of the f-value of an alternative path from the ancestor of the current node. The search backtracks to the alternative path when the f-value of the current path exceeds the f-value of the alternative path. During backtracking, the f-value of each node along the path will be updated by the lowest f-value of its children. The algorithm finishes when a goal is reached whose f-value does not exceed the current cost bound. Similar to A* and IDA*, RBFS finds an optimal solution given an admissible heuristic.

Beam Search

Beam search is a variant of breadth/best first search that reduces the memory requirements of these algorithms. Beam search prunes the non-promising nodes, the ones with the highest heuristic values, during the search to reduce its memory requirements. The number of promising nodes, those that have the lowest heuristic values, that will be kept in memory is called the "beam width" (B) and is the parameter of the beam search. The set of promising nodes that are kept in memory during each iteration is called the "beam".

The algorithm begins with expanding the start state and sorting the successors of the start state based on their heuristic value. The best B successors, *i.e.,* those with lowest heuristic values, will be added to the current beam. If fewer than B successors were generated, all of them will be added to the current beam. For the next step, all the nodes in the current beam are expanded and the B successors with the lowest heuristic values will be added to the next beam. The search succeeds when a goal is found or fails when the beam becomes empty.

Beam search is not guaranteed to find an optimal solution as it does not consider all parts of the state space. Furthermore, by pruning some parts of the state space (possibly all the paths to the goal state) beam search is not a complete search algorithm either.

Limited Discrepancy Search

Harvey and Ginsberg [23] suggested a search algorithm based on the intuition that the failure of a heuristic search algorithm (in the beam search family) might be because of a small number of wrong decisions during the search. Their Limited Discrepancy Search (LDS) is a backtracking search designed to work with binary trees[5] of a finite depth. Figure 2.6 shows the first three iterations of LDS on a binary tree[6] of depth three. In the binary tree, for each state, the heuristic value of the left child is always lower than the heuristic value of the right child. In the first iteration of the search, for each state, the child with the lower heuristic value is selected for expansion. Therefore, only the leftmost path in the tree is visited in the first iteration.

A discrepancy happens when the heuristic is not followed at some point to make a decision, *e.g.,* when a right child is selected for expansion. LDS has a limit on the number of discrepancies allowed for each iteration. This limit is set to zero for the first iteration. When the search fails to find a goal at the end of each iteration, the number of allowed discrepancies increases by 1. Therefore, in the second iteration new parts of the tree will be explored that differ from the first iteration by at most one discrepancy. The order in which the discrepancies are allowed is first at the top of the search tree and then further down the tree. This is based on an intuitive justification that heuristic functions are generally least accurate near the start state. The numbers at the leaf nodes in the figure show the order in which the leaf nodes are visited during each iteration.

BULB

To make beam search complete, Furcy and Koenig [20] enhanced it with backtracking. Their algorithm, which is called BULB (Beam search Using Limited discrepancy Backtracking), has two parameters: a limit on the number of states that can be stored in memory in total (M) and the beam width parameter (B). BULB is complete if a solution of length $\frac{M}{B}$ exists for the problem. For example, with a beam width of $50,000$ and a memory limit of 5 million nodes, BULB will not be able to solve all the instances of the 24-puzzle using the MD heuristic (the optimal solution for these

[5]LDS can be extended to work on non-binary trees.
[6]This figure is from David Furcy's PhD thesis [19].

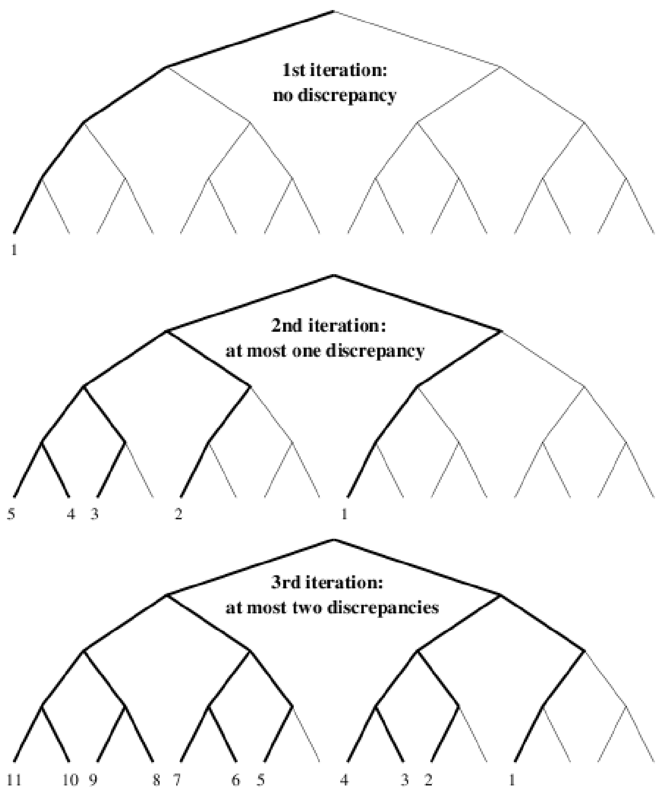


Figure 2.6: Limited Discrepancy Search on a binary tree.

problems instances can be more than 100 while $\frac{M}{B}$ is 100). BULB does not guarantee any bounds on the quality of the solutions returned by the search. Increasing the beam width generally seems to improve the quality of solutions. Increasing the beam width to infinity makes BULB the same as breadth first search.

The main concept in BULB is Limited Discrepancy backtracking [23]. Suppose that more than B successors are generated at some stage during the search when the current beam was expanded. The successors will be sorted based on their heuristic values into a number of slices denoted as $1, 2, \cdots, K$. Each slice i ($1 \leq i \leq K-1$) will contain B successors while slice K can contain less than B successors (depending on the number of nodes that were generated by expanding the previous beam). A discrepancy occurs whenever instead of using the B successors with the lowest heuristic values at the next level (slice 1), another slice is used. Figure 2.7 shows parts of a search graph for BULB. The slices that are expanded by BULB are colored in gray and only their successors are shown in the search graph. Figure 2.7 shows that BULB is using one discrepancy as slice 2 is expanded instead of slice 1 in depth 2.

BULB has a limit on the number of discrepancies that can happen during the search. This limit is set to zero for the first iteration. Backtracking happens whenever the memory limit M is reached

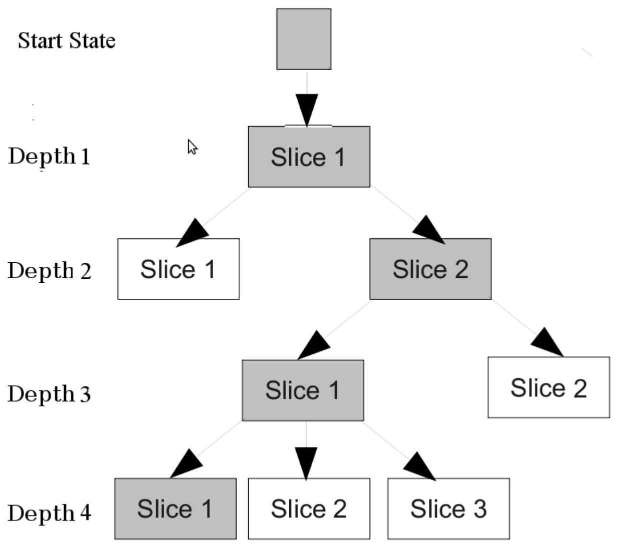


Figure 2.7: An example of BULB.

before finding a solution. The search backtracks to the beam whose successors were generated using a discrepancy. For example if the memory limit M is reached for the search shown in Figure 2.7, then the search backtracks to depth 1. Then a new slice that again causes a discrepancy will be selected for expansion (*e.g.,* if slice 2 was expanded last time, now slice 3 will be selected for expansion). If no such slice exists, the successors will be constructed without a discrepancy (slice 1 will be expanded). For example, as no such slice 3 exists in depth 2 of Figure 2.7, slice 1 will be expanded (no discrepancy). If backtracking reaches the root of the graph and the first beam was constructed without a discrepancy, then the discrepancy limit will be incremented and a new iteration of BULB starts. The search stops when the goal state is reached.

2.1.4 Weighted Heuristics

A weighted heuristic is constructed by multiplying the heuristic function h by a constant factor $W \in \mathbb{R} \geq 0$ which is called the weight [48]. We use the term "Weighted" before the name of each algorithm to show that the algorithm uses a weighted heuristic. For example, Weighted IDA* refers to using a weighted heuristic with IDA*.

When $W \geq 1$ and h is admissible, the resulting weighted heuristic function will likely decrease the amount of search. Ira Pohl [48] proved that Weighted A* generates solutions that are at most W times longer than the optimal ones. A similar bound can be proved for Weighted IDA* (W-IDA*) and Weighted RBFS (W-RBFS).

2.2 Learning Algorithm

A learning algorithm is an algorithm that learns the patterns underlying the training data and generalizes from the given training examples to make predictions on in new cases. Neural networks and linear regression, which will be discussed here, belong to the supervised learning class in which each training example is a pair. The first element of the pair is a vector of feature values representing the input object while the second element is the desired output value.

One way of speeding up search, which is our goal in this thesis, is to learn a heuristic from a set of training data. For example, in our system each training example consists of some user-defined features of a state as the input and the solution length of the state (not necessarily optimal) as the output. A learning algorithm is used to predict the distance from any state to the goal, the heuristic, in new cases. In the next section, we briefly introduce neural networks and linear regression as they are frequently used in the heuristic search and planning for learning heuristic/control knowledge.

2.2.1 Neural Networks

A Neural Network (NN) [53] is a multi-layer structure in which each layer is composed of units that are connected to the units of the next layer. Figure 2.8 shows a simple neural network[7] with one output, three hidden, and two input units. Each arrow in the figure indicates a weight. Smaller arrows in the figure indicate the weights associated to bias nodes. Bias can be considered as a unit with an input value that is always equal to one. The output value of each unit is computed in two steps. First the weighted sum of the inputs of the previous layer (including the bias units) is computed. Then this value will be used as an input to a non-linear function (*e.g.,* a sigmoid function). The output of the non-linear function will be the output of the unit. The output of each layer in the network forms the input to the next layer.

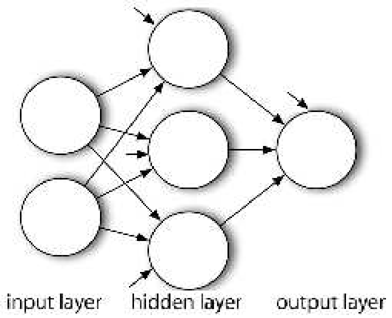


Figure 2.8: A simple neural network.

Backpropagation [54] is used to train the neural network, *i.e.,* to learn the weights. The back-propagation algorithm works in two phases. For each training example, it first computes the output of the neural network using the current weights and calculates the difference between this value

[7]This figure is taken from `http://www.heatonresearch.com/node/704` and modified.

and the desired output given for this training example. The difference is called the error. Then it propagates the error backwards to update the weights. Mean square error is a commonly used error function which is defined as:

$$MSE = \sum_{t \in T} \frac{(F(t) - F'(t))^2}{|T|} \tag{2.1}$$

in which T is the set of training examples and $|T|$ is the number of training examples. For each training example $t \in T$, $F(t)$ and $F'(t)$ respectively show the desired output for t and the output of the network for training example t. The backpropagation stops after a finite number of iterations or when the error function falls below a fixed threshold.

2.2.2 Linear Regression

Linear regression models the relationship between an output variable y and a set of input variables X using a linear function. After developing such a model, linear regression can be used to predict the value of y when a new input is given without its accompanying output value.

The model to predict the output of a new input variable can be represented as:

$$y' = \sum_{i} w_i x_i + c \tag{2.2}$$

in which x_i is the value of the i^{th} input variable, w_i is a weight, c is a constant, and the output y' is the weighted linear combinations of inputs. The weights are learned by minimizing the sum of the squared prediction error $(y - y')^2$ for each training example in which y is the given output for the training example and y' is the output predicted by the linear regression.

One possible technique to learn the weights for linear regression is to use gradient descent. Gradient descent starts with assigning random small numbers to each w_i. Then for each training example t, the prediction error is computed and will be used to update the weights. This process will be repeated until a termination condition (*e.g.,* the error falling below a threshold) is met.

2.3 Summary

In this chapter, a brief overview of basic concepts of heuristic search and the learning algorithms used in heuristic search literature for learning a heuristic function were presented. First, the basic ideas of heuristic search such as the concept of heuristic, weighted heuristic, and heuristic search algorithms were introduced. Then, neural network and linear regression as examples of machine learning algorithms used to learn heuristics were introduced.

Chapter 3

Bootstrap Learning of Heuristics

In this chapter, we first describe the algorithmic approach of our method for learning heuristic functions for heuristic search and planning domains. Our method consists of two procedures. The first one, called the Bootstrap procedure, incrementally updates a given initial (weak) heuristic. In addition to the initial heuristic, Bootstrap uses a set of unlabeled training instances, which we call the bootstrap instances. Bootstrap requires the initial heuristic to be strong enough to solve several of the bootstrap instances in the given time limit. If it is not, a set of easier instances will be automatically generated by the second procedure, called the RandomWalk procedure. These easier instances are needed to improve the initial heuristic until the easiest bootstrap instances can be solved.

Then, we present experimental results of the combination of the Bootstrap and RandomWalk processes on several test domains. The experimental results show that our method produces a heuristic that allows IDA* to solve randomly generated problem instances quickly while generating solutions very close to optimal.

Some parts of this chapter have been published in the Third Annual Symposium on Combinatorial Search (SoCS 2010) [36].

3.1 Method

In this section, we present a more detailed description of our method along with examples of applying it to search problems.

3.1.1 The Bootstrap Algorithm

Algorithm 1 shows our Bootstrap procedure. Its inputs consist of a heuristic function h_{in} and a set Ins of states to be used as bootstrap instances. We do not assume that h_{in} is sufficiently strong that any of the given bootstrap instances can be solved using it. Bootstrap needs a state space and a fixed goal state g. It also uses global variables to represent the current time limit ($t_{\max}$), the upper bound on the time limit (t_{∞}), and a fixed threshold ($ins_{\min}$).

The Bootstrap procedure proceeds in two stages. In the first stage, for every instance i in Ins,

Algorithm 1: The Bootstrap procedure.

```
procedure Bootstrap(h_in, Ins): h_out
uses global variables $t_{\max}$, $t_\infty$, $ins_{\min}$, g
create an empty training set $TS$
while (size(Ins) $\geq$ $ins_{\min}$) && ($t_{\max} \leq t_\infty$) do
    for each instance $i \in$ Ins do
        if Heuristic Search($i$, g, h_in, $t_{\max}$) succeeds then
            for each state $s$ on $i$'s solution path do
                Add (feature vector($s$), distance($s$,g)) to $TS$
            end
        end
    end
    if (#(new bootstrap instances solved) > $ins_{\min}$) then
        h_in := learn a heuristic from $TS$
        remove the solved instances from Ins
        clear $TS$
    else
        $t_{\max}$ := 2 $\times$ $t_{\max}$
    end
end
return h_in
```

a heuristic search algorithm is run with start state i and h$_{in}$ as its heuristic (line 6). Every search is cut off after a limited period of time (t$_{\max}$). If i is solved within that time then some user-defined features of i, together with its solution length, are added to the training set. In addition, features and solution lengths for all the states on the solution path for i are added to the training set (lines 7 and 8). This increases the size of the training set at no additional cost and balances the training set to contain instances with long and short solutions.

The second stage examines the collected training data. If "enough" bootstrap instances have been solved then the heuristic h$_{in}$ is updated by a learning algorithm (line 13). Otherwise the time limit is increased without changing h$_{in}$ (line 17). Either way, as long as the current time limit (t$_{\max}$) does not exceed a fixed upper bound (t$_\infty$), the Bootstrap procedure is repeated on the remaining bootstrap instances with the current heuristic h$_{in}$. "Enough" bootstrap instances here means a number of instances above a fixed threshold ins$_{\min}$ (line 12). The procedure terminates if t$_{\max}$ exceeds t$_\infty$ or if the remaining set of bootstrap instances is too small (smaller than ins$_{\min}$).

To illustrate the behaviour of the Bootstrap procedure, Table 3.1 shows each iteration of the Bootstrap procedure on the 15-puzzle when Ins contains 5000 randomly generated solvable instances, ins$_{\min}$ is 75, and t$_{\max}$ is 1 second. The definition of the initial heuristic h_0 and the features used for learning are described in Section 3.3.1. The **Iteration**, **Number Solved**, and **Remaining Unsolved** columns respectively show the current iteration of the Bootstrap procedure, the number of bootstrap instances solved in this iteration, and the total number of bootstrap instances that remained unsolved (out of the 5000 user-provided bootstrap instances) after this iteration. The **Solution Cost** and **Nodes Generated** columns show the solution cost and number of nodes generated averaged over the solved instances. The **Optimal Cost** column shows the optimal solution cost for the solved instances while the **Suboptimality** column indicates the suboptimality of the solutions found.

The first row of Table 3.1 shows the result of the initial iteration, which uses h_0 as the heuristic. All 5000 instances in Ins were attempted but IDA* with h_0 was only able to solve 857 of them in the time limit of 1 second. All these 857 instances were solved optimally since h_0 was admissible. The states along these 857 solution paths, together with their distances to the goal, form the training set to which a learning algorithm is applied to create a new heuristic, h_1.

Iter-ation	Number Solved	Remaining Unsolved	Solution Cost	Optimal Cost	Sub-optimality	Nodes Generated
0	857	4143	45.81	45.81	0.0%	611,656
1	3508	635	54.13	53.57	1.0%	215,976
2	553	82	59.07	57.76	2.3%	330,887
3	77	5	60.32	58.32	3.4%	324,002

Table 3.1: Bootstrap iterations for the 15-puzzle.

An attempt is then made using h_1 to solve each of the 4143 instances that were not solved using h_0. The next row (iteration 1) shows that 3508 of these were solved in the time limit, but not all were solved optimally. On average the solutions found were 1% longer (0.56 moves) than optimal. All the states along all these solution paths were used to learn a new heuristic h_2, which was then used in an attempt to solve each of the 635 instances that were not solved on the first two iterations. The next row (iteration 2) shows that 553 of these were solved, with a greater average suboptimality than on the previous iteration. The heuristic, h_3, learned from these solutions solved 77 of the 82 instances not solved to this point, and those solution paths provide the training data to create a new heuristic, h_4. The Bootstrap process ends at this point because there are fewer than ins_{min} unsolved instances (5 unsolved instances), and h_4 is returned as the final heuristic.

The suboptimality of the solutions found by Bootstrap increases in successive iterations because on each iteration the learning algorithm might be given solution lengths larger than optimal, biasing the learned heuristic to even greater overestimation. The number of nodes generated in each iteration shows that Bootstrap is improving the heuristic as (1) the number of nodes generated in iterations 1-3 is lower than the number of nodes generated by the initial heuristic on iteration 0 and (2) the instances solved in iterations 1-3 are harder than those solved in iteration 0. Note that the speedup obtained by Bootstrap is at the cost of generating suboptimal solutions.

It can happen that Bootstrap fails to solve enough instances in one iteration. When this happens, t_{max} increases without changing the heuristic. Table 3.2 shows iterations of the Bootstrap procedure on Rubik's Cube when Ins contains 5000 randomly generated solvable instances, ins_{min} is 75, t_{max} is 1 second, and t_{∞} is 512 seconds. The time limit used for each iteration of the Bootstrap procedure is shown in column **Time Limit**. The definition of h_0 and the features used for learning are given in

Section 3.3.3.

Iter-ation	Number Solved	Remaining Unsolved	Solution Cost	Nodes Generated	Time Limit
0	2564	2436	6.58	33,772	1
1	355	2081	13.81	3,118,882	16
2	126	1955	15.63	11,180,904	32
3	82	1873	16.61	27,364,222	64
4	166	1707	17.81	59,888,730	128
5	149	1558	18.79	54,415,700	128
6	162	1396	19.75	59,818,531	128
7	163	1233	20.58	60,467,465	128
8	76	1157	20.60	70,262,338	128
9	255	902	21.11	141,268,160	256
10	85	817	21.48	139,496,014	256
11	135	682	21.53	266,919,602	512
12	218	464	21.82	255,906,281	512
13	206	258	22.01	301,639,498	512
14	192	66	22.58	262,832,032	512

Table 3.2: Bootstrap iterations for Rubik's Cube Using 5000 bootstrap instances.

The first row of Table 3.2 shows that h_0 is strong enough to solve 2564 of the user-provided instances within the time limit of 1 second in the initial iteration. A new heuristic, h_1 is created using the data collected from the initial iteration. h_1 is too weak to solve a sufficient number of the unsolved bootstrap instances; therefore, the time limit was multiplied by two until h_1 solved more than ins_{min} instances. Here, the time limit was increased four times (from 1 to 16) until Bootstrap solved enough instances from the 2436 unsolved bootstrap instances. The process finished after iteration 14 because less than ins_{min} unsolved instances (66 unsolved instances) remain.

The Bootstrap process may also terminate when t_{max} exceeds t_{∞}. Table 3.3 shows iterations of the Bootstrap procedure on Rubik's Cube when Ins contains only 500 randomly generated solvable instances (as before ins_{min} is 75, t_{max} is 1 second, and t_{∞} is 512 seconds).

Iter-ation	Number Solved	Remaining Unsolved	Solution Cost	Nodes Generated	Time Limit
0	256	244	6.53	33,141	1
1	86	158	14.64	82,229,055	512

Table 3.3: Bootstrap iterations for Rubik's Cube using 500 bootstrap instances.

The process terminated after iteration 1 because the last heuristic created by Bootstrap only solved 34 of the 158 unsolved bootstrap instances using a time limit of t_{∞}. The last heuristic created by Bootstrap is returned as the final heuristic.

3.1.2 The RandomWalk Algorithm

The initial heuristic h_{in} can be so weak that the heuristic search algorithm in Bootstrap is unable to solve enough bootstrap instances in Ins, using h_{in}, to get a sufficiently large set of training data. Therefore, a procedure is needed to generate training instances that are (i) easier to solve than the bootstrap instances the user provided but (ii) harder to solve than instances solvable by simple breadth-first search in acceptable time (to guarantee a high enough quality of training data). We used random walks of a suitably chosen length backward from the goal to generate these instances.

Algorithm 2 describes the RandomWalk procedure. It first tests whether the initial heuristic is strong enough to solve a sufficient number of the user-provided bootstrap instances (line 4). If so, the Bootstrap procedure can be started immediately (line 9). Otherwise, we perform random walks backward from the goal up to depth "length" and collect the final states as training instances (RWIns).

These training instances are used as input to the Bootstrap procedure (line 6) to create a stronger heuristic. This process is repeated with increasingly longer random walks (line 7) until it produces a heuristic that is strong enough for bootstrapping to begin on the user-given instances or fails to produce a heuristic with which sufficiently many instances in RWIns can be solved within time limit t_∞.

Algorithm 2: The RandomWalk procedure.

```
1 procedure RandomWalk ($h_{in}$, Ins, lengthIncrement): $h_{out}$
2 uses global variables $t_{\max}$, $t_\infty$, $ins_{\min}$, g
3 length := lengthIncrement
4 while ($h_{in}$ is too weak to start the Bootstrap process on Ins) && ($t_{\max} \leq t_\infty$) do
5     RWIns := generate instances by applying "length" many random moves backward
      from the goal
6     $h_{in}$ := Bootstrap($h_{in}$, RWIns)
7     length := length + lengthIncrement
8 end
9 return Bootstrap($h_{in}$, Ins)
```

The choice of "lengthIncrement" is an important consideration. If it is too large, the instances generated may be too difficult for the current heuristic to solve and the process may fail as the time limit ($t_{\max}$) needs to increase to fill the gap between the difficulty of generated instances and the weakness of the current heuristic. If it is too small, a considerable amount of time will be wasted (i) applying the Bootstrap process to instances that do not substantially improve the current heuristic and (ii) testing whether this new heuristic is strong enough to solve a sufficient number of user-provided bootstrap instances.

In our experiments, the lengthIncrement was set as follows.

1. Run a breadth-first search backwards from the goal state with a time limit given by the initial value of $t_{\max}$. Let S be the set of states thus visited.

2. Repeat 5000 times: do a random walk backwards from the goal until a state not in S is reached. Set lengthIncrement to be the floor of the average length of these 5000 random walks. Note that we always disallow the inverse of the previous move while doing the random walks.

For example, to compute the lengthIncrement for the 24-pancake puzzle, the breadth-first search backwards from the goal visited states with a distance of at most 4 from the goal state during the $\mathrm{t}_{max} = 1$sec allocated to the search. Repeating the random walk from the goal for 5000 times, the lengthIncrement was computed to be 5 for this domain.

Table 3.4 illustrates the RandomWalk procedure on the 20-blocks world when Ins contains 5000 randomly generated solvable instances, $\mathrm{ins}_{\mathrm{min}}$ is 75, $\mathrm{t}_{\mathrm{max}}$ is 1 second, and 200 random walk instances are generated (RWIns) for each distinct random walk length. The value of lengthIncrement is set at 20 by our method. Random walks are necessary in this domain because the initial heuristic h_0 is too weak to solve a sufficient number ($\mathrm{ins}_{\mathrm{min}}$) of the bootstrap instances within the time limit of $\mathrm{t}_{\mathrm{max}}$. The definition of h_0 and the features used for learning are given in Section 3.3.4.

The **RW length** and **Time Limit** columns respectively show the value of "length" and the time limit ($\mathrm{t}_{\mathrm{max}}$) for each row. The **Number Solved** and **Solution Cost** columns are the same as in Table 3.1. The first row shows the result of the initial iteration. 200 instances have been generated by random walks of length 20, and passed to the Bootstrap procedure along with h_0. The average solution cost of these instances is 8.97. IDA* using h_0 as the heuristic was able to solve 197 of these 200 instances within the time limit so there is just one iteration of the Bootstrap process, which returns a new heuristic, h_1. This heuristic is then used in an attempt to solve the bootstrap instances. It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

Row	RW length	Number Solved	Solution Cost	Time Limit
1	20	197	8.97	1
2	40	145	11.83	1
3	60	115	14.20	1
4	60	79	16.23	2
5	80	99	16.31	2
6	80	95	19.66	4
7	100	174	20.59	4
8	120	139	23.9	4

Table 3.4: RandomWalk procedure applied to the 20-blocks world.

The random walk length is increased by 20 (the value of lengthIncrement) and a set of 200 random walk instances is generated by random walks of length 40 and passed to the Bootstrap procedure along with h_1. Here, 145 of them are solved in a single Bootstrap iteration and the Bootstrap procedure returns a new heuristic, h_2, since there are now fewer than $\mathrm{Ins}_{\mathrm{min}}$ unsolved RandomWalk instances. This heuristic is used to attempt to solve the bootstrap instances. It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

The random walk length is increased by 20 and 200 random walk instances are generated by random walks of length 60 and passed to the Bootstrap procedure along with h_2. The Bootstrap process (Row 3) is only able to solve 115 of these instances using h_2. A new heuristic, h_3, is learned from these but is not passed back to the RandomWalk procedure because there are still more than ins_{min} unsolved RandomWalk instances. A second iteration of the Bootstrap procedure attempts to solve them with its new heuristic, h_3, but fails to solve a sufficient number (ins_{min}) and therefore doubles the time limit and attempts them again with h_3. Row 4 shows that this iteration of the Bootstrap procedure succeeds in solving 79 of them with the new time limit, and from these it learns a new heuristic, h_4. Since there are now fewer than ins_{min} unsolved RandomWalk instances, the Bootstrap procedure returns h_4 to the RandomWalk process. This heuristic is used in an attempt to solve the bootstrap instances. It is too weak to solve a sufficient number of them in the time limit so another iteration of the RandomWalk process is needed.

As the table shows, in total 6 iterations of the loop in the RandomWalk process were executed (6 distinct values of the RandomWalk length) and for each of these iterations either one or two Bootstrap iterations were required to find a heuristic that could solve the random walk instances. The time limit had to be increased 2 times. The RandomWalk process ended because the heuristic, h_8, corresponding to the final row of the table was able to solve a sufficient number of the bootstrap instances that the bootstrap procedure could finally be started on the set of bootstrap instances with h_8 as its initial heuristic.

The rest of this chapter describes the experimental results of our method on various heuristic search and planning domains.

3.2 Experimental Settings

In this section, we first describe the settings used for our experiments. In the next section, we present experimental results on four different heuristic search and planning domains.

3.2.1 Settings

The following settings were used for our experiments.

Hardware. All the experiments ran on a 2.6 GHz computer with 32GB of RAM. Although we had a large amount of memory available, we only used a small portion of it in each experiment to store the pattern databases.

Search Algorithm. IDA*, Weighted IDA* (W-IDA*) and BULB were the search algorithms used. We used standard techniques to prune duplicates. For example, in Rubik's Cube we disallowed twisting the same face twice in a row. Furthermore, we arbitrarily forced an order for each pair of opposite faces and disallow moving the two opposite faces consecutively in the opposite order. Therefore, for Rubik's Cube the branching factor was reduced from 18 to about 13.35 [41].

Learning algorithm. A neural network (NN) with one output neuron representing distance-to-goal and three hidden units was trained using backpropagation [54]. We used the neural network toolbox of *Matlab* in our implementation. The weights were initialized randomly to real numbers between -1 and 1. Mean squared error (MSE) was used as the error function and training was ended after 500 epochs or when $MSE < 0.005$. Each training example fed to the network consisted of features of a state and the solution length obtained for that state during bootstrapping.

Features. Machine learning applications need "good" features to succeed. In our experiments we did not carefully engineer the features used, exploit special properties of the domain, or alter our features on the basis of early experimental outcomes or as we scaled up the problem. The input features for the NN are described separately below for each domain.

Most of the features are weak heuristics created using abstraction for the respective problems.[1] The issue of automatically creating good features is outside the scope of this thesis. In the context of planning, Yoon *et al.* [67] studied this issue for learning search control knowledge.

Heuristics. The initial heuristic h_0 for each domain was defined as the maximum of the heuristics used as features for the NN. After each iteration of our method, the new heuristic was defined as the maximum over the output of the NN and the initial heuristic.

Bootstrap instances. Ins always consisted of either 500 or 5000 solvable instances. For the sliding-tile puzzle, the pancake puzzle, and the blocks world, bootstrap instances are generated uniformly at random. We used random solvable permutations for the sliding-tile puzzle and the pancake puzzle and the random state generator described by Slaney and Thiébaux [60] for the blocks world to create bootstrap instances. For Rubik's Cube, the bootstrap instances were generated by random walks of various lengths between 1 and 25.

Numeric parameters. In all experiments, $\text{ins}_{\min} = 75$, $\text{t}_{\max} = 1\text{sec}$, $\text{t}_{\infty} = 512\text{sec}$, and the size of the set RWIns was 200.

We never attempted fine-tuning any of the numeric parameters used in our experiments or altered them on the basis of early experimental outcomes. The next section presents the experimental results of Bootstrap on four different domains.

3.3 Experimental Results

The tables below summarize the results on our test domains, which are based on a set of test instances generated independently of the bootstrap instances. In the tables, the column **h (Algorithm)** denotes the heuristic used; the search algorithm, if different from IDA*, is given in parentheses. The symbol #k indicates that the same heuristic is used in this row as in row k. The **Cost** and **Nodes** columns show the average solution cost and average number of nodes generated to solve the test instances. The **Subopt** column indicates the suboptimality of the solutions found. For example, Subopt=7%

[1]The features used for Rubik's Cube are not weak heuristics. We chose them to be able to compare our results to BULB [20].

means the solutions generated were 7% longer than optimal on average.

The **Solving Time** column shows the average search time (in seconds) to solve the test instances. Unless specifically stated, no time limit was imposed when systems were solving the test instances. Each entry of the **Learning Time** column shows the total time used by our method, since the process has begun, to learn the heuristic corresponding to that row. Each "m", "h", and "days" represent the units of time (minutes, hours, and days respectively).

In the tables, each row shows the data for a system that we implemented or found in the literature. The run-times taken from the literature are marked with an asterisk to indicate they may not be strictly comparable to ours. Hyphenated row numbers "row-y" indicate Bootstrap results after iteration y.

All weighted IDA* and BULB results (excluding the results for Rubik's Cube) are based on our own implementations. The parameters (W=weight) and (B=beam width) reported for W-IDA* and BULB are those for which the results are most similar to Bootstrap (i) in terms of suboptimality and (ii) in terms of the number of generated nodes (see boldface numbers in the tables).

The last Bootstrap iteration shown in the tables represents the last successful iteration of the Bootstrap process. If there were fewer than $\text{ins}_{\min}$ unsolved bootstrap instances remaining after that iteration (15-puzzle, 17- , 24- , and 35-pancake puzzle, Rubik's Cube using 5000 bootstrap instances, 15-blocks world, and 20-blocks world using 5000 bootstrap instances), the Bootstrap process terminated as soon as that iteration was done and the "Bootstrap Completion Time", which measures the entire time required by the Bootstrap process, is equal to the "Learning Time" reported for the final iteration. However, if there were $\text{ins}_{\min}$ or more unsolved bootstrap instances remaining after the last iteration shown in a table (24-puzzle, Rubik's Cube with 500 bootstrap instances, and 20-blocks world with 500 bootstrap instances), another Bootstrap iteration would have been attempted on those instances but Bootstrap terminated it without creating a new heuristic because $\text{t}_{\max}$ exceeded t_{∞}. In such a case the "Bootstrap Completion Time" includes the time taken by the final, unsuccessful iteration. For example, the "Learning Time" for Row 1-3 of Table 3.7 shows that it takes 1 day and 11 hours to learn the final heuristic for the 24-puzzle using 500 bootstrap instances, but the "Bootstrap Completion Time" is reported as 2 days and 2 hours. The difference (15 hours) is the time required by an iteration after the one reported in Row 1-3, which failed to solve $\text{ins}_{\min}$ new instances within the time limit of t_{∞}.

3.3.1 Sliding-Tile Puzzle

15-puzzle

For the 15-puzzle the input features for the NN were Manhattan distance (MD), number of out-of-place tiles, position of the blank, number of tiles not in the correct row, number of tiles not in the correct column, and five heuristics, each of which is the maximum of two 4-tile pattern databases (PDBs). The total memory used for this experiment was about 6 megabytes. The time to build the pattern databases and generate bootstrap instances, which we call the pre-processing time, was a few seconds.

The results in Tables 3.5 and 3.6 are averages over the standard 1000 15-puzzle test instances [42], which have an average optimal cost of 52.52, except for the last row in Table 3.6, which is the average over 700 random instances with an average optimal solution length of 52.62 [13].

Table 3.5 shows the results for bootstrapping on the 15-puzzle. The initial heuristic (h_0) was sufficient to begin the Bootstrap process directly, so no RandomWalk iterations were necessary. Row 1 of Table 3.6 shows the results when h_0 is used by itself as the final heuristic. It is included to emphasize the speedup produced by Bootstrap.

Rows 1-0 to 1-2 of Table 3.5 show the results for the heuristic created on each iteration of the Bootstrap method when it is given 500 bootstrap instances. The next four rows (2-0 to 2-3) are analogous, but when 5000 bootstrap instances are given. In both cases, there is a very clear trend: search becomes faster in each successive iteration (see the Nodes and Solving Time columns) but suboptimality becomes worse. In either case, bootstrapping produces very substantial speedup over search using h_0. For instance, using 500 bootstrap instances produces a heuristic in 11 minutes that makes search more than 4000 times faster than with h_0 while producing solutions that are only 4.5% (2.4 moves) longer than optimal.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 11 minutes						
1-0	0 (first)	53.09	1.1%	422,554	0.424	8m
1-1	1	53.94	2.7%	76,928	0.075	10m
1-2	2 (final)	55.12	4.5%	32,425	0.041	11m
5000 bootstrap instances Bootstrap Completion Time = 1 hour 52 minutes						
2-0	0 (first)	53.15	1.2%	388,728	0.379	1h 20m
2-1	1	53.91	2.6%	88,235	0.087	1h 48m
2-2	2	55.34	5.4%	21,800	0.022	1h 51m
2-3	3 (final)	**56.55**	**7.7%**	**10,104**	0.010	1h 52m

Table 3.5: 15-puzzle, Bootstrap.

There are two key differences between using 500 and 5000 bootstrap instances.

1. The most important difference is the total time of the Bootstrap process. Because after every iteration an attempt is made to solve every bootstrap instance, having 10 times as many bootstrap instances makes the process roughly 10 times slower. For example, in the 15-puzzle, using 5000 bootstrap instances is more than 9 times slower than using 500 bootstrap instances.

2. The larger bootstrap set contains a larger number of more difficult problems, and those drive the Bootstrap process through additional iterations (in this case one additional iteration), producing, in the end, faster search but worse suboptimality than when fewer bootstrap instances are used.

There are rich sets of tradeoffs inherent in the Bootstrap approach. For example, the time-suboptimality tradeoff suggests that the search effort decreases at the cost of increasing the solution length.

Another tradeoff is between the learning time and solving time: a heuristic that takes more time to be learned solves problems faster on average. For example, the heuristic in row 2-3 of Table 3.5 solves problems about 38 times faster than the heuristic learned in row 2-0. It takes 1 hour and 52 minutes to learn the former heuristic while the latter takes 1 hour and 20 minutes. Therefore, computing the final heuristic of Bootstrap (row 2-3) will be efficient, in terms of sum of learning and solving times, only when more than five thousand instances of 15-puzzle are solved.

In this chapter, we do not address the issue of how to choose among these options, we assume that a certain number of bootstrap instances are given and that the heuristic produced by the final Bootstrap iteration is the system's final output. In Chapter 4, we present one approach to exploit the relationship between learning time and solving time when there is only one problem instance to solve.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of Initial Heuristic					
1	h_0	52.52	0%	132,712,521	116.478
Results of W-IDA*					
2	h_0 (W-IDA*,W=1.6)	**57.20**	**8.9%**	310,104	0.194
3	h_0 (W-IDA*,W=2.3)	74.53	41.9%	**10,734**	0.007
Results of BULB					
4	h_0 (BULB,B=250)	**56.57**	**7.7%**	26,013	0.017
5	h_0 (BULB,B=90)	59.17	12.7%	**10,168**	0.006
Results of h_{sum}					
6	h_{sum}	163.27	211.0%	1,538	0.001
Results from previous papers					
7	Add 7-8	52.52	0%	136,289	0.063*
8	#7 + reflected lookup	52.52	0%	36,710	0.027*
9	#8 + dual lookup	52.52	0%	18,601	0.022*
10	NN using tile positions +#8+MD (RBFS)	54.26	3.3%	2,241	0.001*
11	PE-ANN version of #10 (RBFS)	52.61	0.2%	16,654	0.014*
12	NN "A"	54.45	3.5%	24,711	7.380*

Table 3.6: 15-puzzle, other methods.

Table 3.6 shows the results of other systems applied to the same test instances (except for the last row). Rows 2 through 5 are when our initial heuristic (h_0) is used with W-IDA* and BULB. Both algorithms are dominated by Bootstrap, *i.e.*, if W and B are set so that W-IDA* and BULB compare to Bootstrap in either one of the values Subopt or Nodes, then the heuristic obtained in the final Bootstrap iteration (Table 3.5, Row 2-3) is superior in the other value.

Row 6 shows the results with the heuristic h_{sum}, which is defined as the sum of the heuristic values among the NN's input features (h_0 is the maximum of these values). The issue of which subset of heuristic features should be used in h_{sum} is outside the scope of this thesis and in all domains we simply add all of the heuristic features. Although h_{sum} can, in general, be much greater than the actual distance to goal, h_{sum} might be quite an accurate heuristic when a moderate number of weak heuristics are used for NN features, as is the case in our experiments. By comparing its performance with Bootstrap's we can see the return on investment for learning how to combine the different heuristics as opposed to just giving them all equal weight as h_{sum} does. As the results show, h_{sum} performs very poorly in terms of Subopt but is superior to all the results reported in Table 3.6 in terms of Nodes and Solving Time.

Rows 7-9 show state-of-the-art results for optimally solving the 15-puzzle. Rows 7 and 8 refer to results from Korf and Felner [42], where the maximum of sum of two disjoint 7-8 PDBs, and their reflections across the main diagonal are the heuristics. Row 9 is from Felner *et al.* [15], where the heuristic is as in Row 8, augmented with dual lookup[2] (in both the regular and the reflected PDB). Bootstrap with 5000 bootstrap instances (Table 3.5, Row 2-3) outperforms all of these systems in terms of Nodes and Solving Time.

The last three rows of Table 3.6 show state-of-the-art results for the one-step heuristic learning systems. The idea behind the one-step learner of a heuristic is: a learning system is trained on the set of states whose optimal distance-to-goal is known. Then, the learning system will be used to create a heuristic function that estimates distance-to-goal for an arbitrary state. Section 5.2.2 describes one-step learners of heuristics in Rows 10-12 of Table 3.6 in detail.

Rows 10 and 11 are taken from Samadi *et al.* [55]. Row 10 uses the tile positions as features for the NN along with the heuristic from Row 8 and Manhattan Distance. Row 11 is the same as Row 10 but using a modified error function during the neural net training to penalize overestimation. This drives suboptimality almost to 0, at the cost of substantially increasing the search time.

Row 12 shows the results from Ernandes and Gori [13] for the setting in their paper that generated the fewest nodes ("A"). These are averages over 700 random instances with an average optimal solution length of 52.62, not over the 1,000 test instances used by all other systems in this section. The NN input features are the positions of the tiles and the initial heuristic (MD augmented with a set of correction techniques, *e.g.,* linear conflicts[3]). This system performs better than Bootstrap in terms of suboptimality but worse in terms of nodes generated, most likely because it aims to optimize solution quality rather than nodes generated.

[2]In permutations state spaces, *e.g.,* the 15-puzzle, for each state s, a state s^d exists that is the same distance from the goal as s. This state s^d is called the dual of state s. A regular PDB lookup for state s returns the value stored in the PDB for the abstract state $\phi(s)$. One the other hand, a dual PDB lookup for state s returns the stored value for the abstract state $\phi(s^d)$ [15].

[3]Linear conflict [42] refers to the cases when two tiles are in their goal row/column but are in a reversed order relative to their goal locations. For example, in the bottom left 15-puzzle state shown in Figure 2.5, tiles 2 and 3 are in their goal row but in a reversed order relative to their goal locations. To move these tiles to their goal locations, one of them needs to be moved down to allow the other tile to move to its correct location and then it needs to be moved back up to be in its correct location. MD simply ignores these moves as it does not consider the interaction between the tiles. Therefore, two moves can be added to MD in the case of a linear conflict and the heuristic will still remain admissible.

Bootstrap with 5000 bootstrap instances (Table 3.5, Row 2-3) outperforms all of these systems in terms of Nodes except for Row 10, which also outperforms Bootstrap in terms of suboptimality.

The superior performance of the system in Row 10 is simply the result of having vastly stronger heuristics as input features to the NN. The use of stronger heuristics helps whenever the maximum of the heuristics used in the feature vector has a higher value than the output of the NN. To prove the above hypothesis, we reran our Bootstrap system with 5000 bootstrap instances, exactly as described above, but with the following features replacing the weak PDB features used above: the value of the 7- and 8-tile additive PDBs individually, for both the given state and its reflection, and the maximum of the sum of 7- and 8-tile PDB values for the state and its reflection. With these input features, and the corresponding h_0, Bootstrap solved all 5000 bootstrap instances on its first iteration, and the heuristic it produced, when used with RBFS, solved the 1000 test instances with an average suboptimality of 0.5% while generating only 9,402 nodes on average[4]. We thus see that Bootstrap, when given a heuristic that is strong enough that it can solve all bootstrap instances in the given time limit, is equivalent to the one-step systems previously reported. But, as was seen in Table 3.5, its iterative RandomWalk and Bootstrap processes take it beyond the capabilities of those systems by enabling it to perform very well even when the initial heuristic is not strong enough to solve the bootstrap instances.

24-puzzle

Table 3.7 shows our results on the 50 standard 24-puzzle test instances [42], which have an average optimal cost of 100.78. The input features for the NN are the same as for the 15-puzzle. Note that here 4-tile PDBs, though requiring more memory, are much weaker than for the 15-puzzle. The amount of required memory for the 24-puzzle was about 50 megabytes while the pre-processing phase took about two minutes.

The initial heuristic is sufficiently weak that eight RandomWalk iterations were necessary before bootstrapping itself could begin (ten iterations were required when there were only 500 bootstrap instances). The first four rows (1-0 to 1-3) show the results when 500 bootstrap instances are given. The next eleven rows are analogous (2-0 to 2-10), but when 5000 bootstrap instances are given. In both cases, the trend is very similar to the 15-puzzle: search becomes faster in each successive iteration (see the Nodes and Solving Time columns) but suboptimality becomes worse.

Table 3.8 shows the results of other systems on the same test instances. Row 1 reports on W-IDA* using our initial heuristic (h_0) multiplied by a weight (W) chosen so that Subopt is roughly equal to the Subopt value achieved by the final Bootstrap heuristic (Table 3.7, Row 2-10). In Row 2, W is chosen so that Nodes is roughly equal to the Nodes value achieved by the final Bootstrap heuristic (Table 3.7, Row 2-10). The results of similar settings for BULB's beam width (B) when h_0

[4]It seems that in the 15-puzzle, RBFS performs slightly better in terms of solution quality and worse in terms of nodes generated compared to IDA* with the inconsistent and inadmissible heuristic learned from the bootstrap instances. IDA* using the same heuristic as in Row 9 generated 8,776 nodes on average over the same set of test problems while the solutions were 0.8% longer than the optimal ones.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 2 days and 2 hours						
1-0	0 (first)	105.94	5.1%	2,195,190,123	4,987.10	3h
1-1	1	106.50	5.7%	954,325,546	2,134.78	8h
1-2	2	106.77	5.9%	663,058,673	1,204.32	15h
1-3	3 (final)	106.92	6.1%	164,589,698	273.51	1day 11h
5000 bootstrap instances Bootstrap Completion Time = 17 days and 17 hours						
2-0	0 (first)	106.26	5.4%	1,316,197,887	2,439.72	1day 04h
2-1	1	106.42	5.6%	390,589,747	723.54	1day 22h
2-2	2	106.46	5.6%	215,726,120	406.01	2days 13h
2-3	3	106.70	5.9%	198,324,168	370.66	3days 15h
2-4	4	106.78	6.0%	115,721,236	214.59	5days 10h
2-5	5	107.30	6.5%	77,653,775	144.56	7days 15h
2-6	6	107.70	6.9%	31,631,208	60.34	11days 02h
2-7	7	107.84	7.9%	29,956,637	55.85	12days 21h
2-8	8	109.62	8.8%	12,547,576	23.51	14days 05h
2-9	9	110.26	9.4%	5,236,088	9.84	15days 17h
2-10	10 (final)	**110.46**	**9.6%**	**5,221,203**	9.83	17days 06h

Table 3.7: 24-puzzle, Bootstrap.

is used are shown in Rows 3 and 4. Bootstrap (Table 3.7, Row 2-10) dominates in all cases, in the sense that if W and B are set so that W-IDA* and BULB compare to Bootstrap in either one of the values Subopt or Nodes, then the heuristic obtained in the final Bootstrap iteration (Table 3.7, Row 2-10) is superior in the other value.

Row 5 shows the results with the heuristic h_{sum}. As the results show, h_{sum} performs very poorly in terms of both Subopt and Nodes.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of W-IDA*					
1	h_0 (W-IDA*,W=1.5)	111.3	**10.4%**	14,183,039,982	12,896.9
2	h_0 (W-IDA*,W=2.6)	181.69	80.3%	**5,849,910**	4.3
Results of BULB					
3	h_0 (BULB,B=18000)	111.35	**10.5%**	6,896,038	22.3
4	h_0 (BULB,B=16000)	112.87	12.0%	**5,286,874**	18.2
Results of h_{sum}					
5	h_{sum}	295.94	193.7%	49,810,188	46.0
Results from previous papers					
6	Add 6-6-6-6	100.78	0%	360,892,479,670	47 hours*
7	#6 (DIDA*)	100.78	0%	75,201,250,618	10 hours*
8	#6, Add 8-8-8	100.78	0%	65,135,068,005	?
9	#6, W=1.4 (RBFS)	110.30	9.4%	1,400,431	1.0*
10	PE-ANN, Add 11-11-2 (RBFS)	101.41	0.7%	118,465,980	111.0*
11	#10, W=1.2 (RBFS)	104.48	3.7%	582,466	0.7*

Table 3.8: 24-puzzle, other methods.

Rows 6 to 8 show the results of state-of-the-art heuristic search methods for finding optimal solutions. Row 6 shows the results using the maximum of sum of four disjoint 6-tile PDBs and their reflections across the main diagonal as a heuristic [42]. Row 7 shows the results for DIDA*[5] [69] using the same heuristic. In Row 8 the heuristic used is the maximum of the heuristic from Row 6 and a partially created disjoint 8-tile PDB [14] (Solving Time was not reported). The very large time required by these systems shows that the 24-puzzle represents the limit for finding optimal solutions with today's abstraction methods and memory sizes. Row 9 [55] illustrates the benefits of allowing some amount of suboptimality, Here RBFS is used with the heuristic from Row 6 multiplied by 1.4. The number of nodes generated has plummeted. Although this result is better, in terms of Nodes and Solving Time, than Bootstrap (Table 3.7, Row 2-10), it hinges upon having a very strong heuristic since we have just noted that W-IDA* with our initial heuristic is badly outperformed by Bootstrap.

Rows 10 and 11 show the PE-ANN results [55]. Note that this is not a direct application of heuristic learning to the 24-puzzle because it was infeasible to generate an adequate training set for a one-step method. Samadi *et al.* [55] manually divided the problem into subproblems, and learned a heuristic for each subproblem. In particular, they divided the 24-puzzle into two disjoint 11-tile subproblems, learned a heuristic for each subproblem and added the learned heuristics to get a heuristic for the 24-puzzle. The main problem with this approach is that the crucial choices of which subproblems to use and how to combine them were made manually. Row 10 shows that our method is superior to PE-ANN used in this way by a factor of more than 20 in terms of nodes generated, although it is inferior in terms of suboptimality. Row 11 shows that if PE-ANN's learned heuristic is suitably weighted it can outperform Bootstrap in both Nodes and Subopt.

3.3.2 Pancake Puzzle

17-pancake puzzle

For the 17-pancake puzzle the input features for the NN were six 5-token PDBs[6], a binary value indicating whether the middle token is out of place, and the number of the largest out-of-place token. The total memory used for this domain was about 4.5 megabytes while the pre-processing took a few seconds. All the results in Tables 3.9 and 3.10 are averages over the 1000 test instances used by Yang *et al.* [66], which have an average optimal solution length of 15.77. Optimal solution lengths were computed using the highly accurate, hand-crafted "break" heuristic[7].

Consider a state in the 5-pancake puzzle such as $(4, 1, 5, 2, 3)$[8]. To compute the break heuristic,

[5]Dual IDA* (DIDA*) [69] is a search algorithm in which at each state s the search decides to search from either s or s^d to the goal state. For example, if the heuristic value of s^d is larger than the heuristic value of s, the search algorithm can decide to continue its search from s^d instead of s.

[6]To make a 5-token PDB for the 17-pancake puzzle, an abstraction of the 17-pancake puzzle is made by considering 12 numbers in the permutation to be equivalent. For example, if numbers 6 to 17 are considered to be equivalent in the abstract search space, then the pattern database that was built for this abstraction only considers 5 numbers in the permutation and is a called a 5-token PDB.

[7]The heuristic is from Tomas Rokicki. See `http://tomas.rokicki.com/pancake/` for more details.

[8]In this example, pancakes of size 4 and 3 are respectively at the top and the bottom of the stack of pancakes. This example is taken from Rokicki, see `http://tomas.rokicki.com/pancake/` for more detail.

an extra number equal to the number of pancakes plus one will be added at the end of the state. Therefore, the state becomes $(4, 1, 5, 2, 3, 6)$. A "break" is defined as a point in which the difference between two consecutive numbers in the sequence is not one. The total number of breaks in a state is the value of the break heuristic for that state. For example, the above state has 4 four breaks; therefore, the heuristic value for this state is 4. There is no break in the goal state (the heuristic value of the goal state is zero). Each move can reduce the number of breaks by at most one. Therefore, the break heuristic is both admissible and consistent.

Table 3.9 shows the results for bootstrapping on the 17-pancake puzzle. The initial heuristic (h_0) was too weak for us to evaluate it on the test instances in a reasonable amount of time. The first two rows (1-0 and 1-1) show the results for the heuristic created on each iteration of the Bootstrap method using 500 bootstrap instances. The next three rows (2-0 to 2-2) are analogous, but when 5000 bootstrap instances are given. In both cases, we see the same trends as in the 15-puzzle: (a) search becomes faster in each successive iteration but suboptimality becomes worse; and (b) having more bootstrap instances is slower and results in extra Bootstrap iterations. Note that a suboptimality of 7% here means the solutions generated are only 1.1 moves longer than optimal.

When there were 5000 bootstrap instances h_0 was able to solve enough bootstrap instances to begin the Bootstrap process directly, but when there were only 500 two iterations of the RandomWalk process were needed before bootstrapping on the bootstrap instances could begin. Therefore, the heuristic that Bootstrap using 500 instances uses for its iteration 0 is stronger compared to the heuristic that Bootstrap using 5000 instances uses. This results in the total time of the process using 5000 bootstrap instances being less than 6 times more than the total time when 500 bootstrap instances were used.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 26 minutes						
1-0	0 (first)	16.26	3.1%	253,964	0.112	22m
1-1	1 (final)	16.50	4.6%	75,093	0.034	26m
5000 bootstrap instances Bootstrap Completion Time = 2 hours and 23 minutes						
2-0	0 (first)	16.15	2.4%	15,232,606	6.700	1h 24m
2-1	1	16.52	4.8%	76,346	0.034	2h 15m
2-2	2 (final)	**16.89**	**7.1%**	**30,341**	0.014	2h 23m
5000 bootstrap instances + duality Bootstrap Completion Time = 2 hours and 4 minutes						
3-0	0 (first)	16.10	2.1%	14,631,867	30.790	1h 21m
3-1	1	16.43	4.2%	57,119	0.111	1h 57m
3-2	2 (final)	**16.88**	**7.0%**	**7,555**	0.009	2h 04m

Table 3.9: 17-pancake puzzle, Bootstrap.

The final three rows in Table 3.9 examine the effect of infusing into the bootstrapping process the domain-specific knowledge that for every pancake puzzle state s there exists another state s^d, called the dual of s, that is the same distance from the goal as s [69]. To exploit this knowledge, when training the NN for every training example (s,$cost$) we added an additional training example (s^d,$cost$). When calculating the heuristic value for s we took the maximum of the values produced by the NN when it was applied to s and to s^d and $h_0(s)$ and $h_0(s^d)$. The initial heuristic here was given by the maximum over the heuristic values occurring in the feature vectors for s and s^d. A comparison of Rows 3-2 and 2-2 shows that the additional knowledge substantially reduced search time without affecting suboptimality.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of break heuristic					
1	break	15.77	0%	6,731	0.002
Results of W-IDA*					
2	h_0 (W-IDA*,W=2)	**16.90**	**7.1%**	20,949,730	9.800
3	h_0 (W-IDA*,W=8)	47.55	201.5%	**8,650**	0.004
Results of BULB					
4	h_0 (BULB,B=5000)	**16.96**	**7.5%**	955,015	0.715
5	h_0 (BULB,B=10)	40.31	155.5%	**11,005**	0.008
Results of h_{sum}					
6	h_{sum}	35.06	122.3%	776	0.001
Results from previous papers					
7	Add 3-7-7	15.77	0%	1,061,383	0.383*
8	#7 + dual lookup	15.77	0%	52,237	0.036*
9	#8 (DIDA*)	15.77	0%	37,155	0.026*
10	50 lookup + dual lookup	15.77	0%	673,340	49.30*
11	#10 (DIDA*)	15.77	0%	411,830	60.39*

Table 3.10: 17-pancake puzzle, other methods.

Table 3.10 shows the results of other systems. Row 1 shows the results for the break heuristic. Break is a hand-crafted heuristic; it is only included for completeness and it is not comparable with our method which uses general techniques to create heuristics. Rows 2 to 5 are when our initial heuristic (h_0) and duality is used with W-IDA* and BULB. Both algorithms are dominated by Bootstrap (Table 3.9, Row 3-2).

Row 6 shows the results of using the sum of the heuristics in the feature vector as the heuristic. Duality is also used here. Therefore, the maximum value of the sum of the heuristics for s and s^d is taken. Similar to the 15-puzzle, h_{sum} performs very poorly in terms of Subopt but is superior to all the results reported in Table 3.10 in terms of Nodes and Solving Time.

Rows 7 to 9 are for the state-of-the-art optimal heuristic search methods that have been applied to the 17-pancake puzzle. Rows 7 and 8 [66] use a set of three additive PDBs, without and with dual lookups. Row 9 uses dual lookups and dual search [69]. In terms of Nodes and Solving Time, Bootstrap outperforms these systems even without exploiting duality (Table 3.9, Row 2-2).

Rows 10 and 11 show the best results, in terms of Nodes, from Helmert and Röger [30] when a different abstraction, called the relative-order abstraction[9], is used to make the pattern database.

[9]The relative-order abstraction only considers the relative order of a set of pancakes in the abstract state. For example,

Row 10 shows the results when the maximum of 50 regular and dual PDB lookups is used as the heuristic value of each state. Here, each of the 50 PDBs is built by considering the relative order of a random set of 5 pancakes. Row 11 shows the results with the same heuristic as row 10 when DIDA* [69] is used as the search algorithm. Bootstrap, even when it uses 500 bootstrap instances and does not exploit duality (Table 3.9, Row 1-2), outperforms these system in terms of both Nodes and Solving Time.

24-pancake puzzle

The experimental setup for the 24-pancake puzzle is identical to that for the 17-pancake puzzle. A 5-token PDB is a much weaker heuristic (although it takes more memory) when there are 24 pancakes. The pre-processing took about 2 minutes and the memory required for this domain was about 31 megabytes.

1000 randomly generated instances, with an average optimal solution cost of 22.75, were used for testing. The initial heuristic is so weak that four RandomWalk iterations were necessary before bootstrapping itself could begin. Table 3.11 is analogous to Table 3.9, with 500 bootstrap instances, 5000 bootstrap instances, and 5000 bootstrap instances with duality exploited. All the trends seen in previous domains are evident here.

No previous system that automatically creates heuristics has been applied to this problem domain because of its size. Table 3.12 includes results only for the break heuristic, W-IDA*, BULB and h_{sum}. Row 1 shows the results of the break heuristic. The results for the next three lines are when those algorithms use duality, so the appropriate comparison is with Row 3-6 of Table 3.11. Neither W-IDA* nor BULB was able to achieve a "Nodes" value similar to Bootstrap, so Rows 2 and 4 of the table just show the minimum number of nodes these two algorithms generated (we tried 15 values[10] for W between 1.1 and 10, and 15 values[11] for B between 2 and 20,000.). As can be seen, W-IDA* and BULB produce a very high degree of suboptimality when generating the fewest nodes and are therefore not competitive with Bootstrap.

Looking for settings for which W-IDA* or BULB can compete with Bootstrap in terms of suboptimality was not successful. Allowing 10 times more time than IDA* with Bootstrap's final heuristic needed on each test instance, W-IDA* did not complete any instances at all. The best results of BULB in terms of suboptimality is shown in Row 3. It shows that BULB even with a greater suboptimality than Bootstrap's final iteration (Row 3-6 of Table 3.11) will generate more than 65 times more nodes than Bootstrap.

The result of h_{sum} is shown in Row 5. Again the maximum of h_{sum} in s and s^d is taken as the heuristic. Although h_{sum} performs very poorly in terms of both Nodes and Subopt compared

if a set of three pancakes in an n-pancake puzzle is considered to make a PDB (all other pancakes are considered to be equivalent in the abstract space), the PDB will only have 6 entries (regardless of the size of the problem), each entry contains the heuristic value for a relative order of three pancakes.

[10]We tried $W \in \{1.1, 1.2, 1.5, 1.8, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

[11]We tried $B \in \{2, 3, 5, 7, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 1 hours and 19 minutes						
1-0	0 (first)	24.59	8.1%	9,502,753	9.70	42m
1-2	2	24.86	9.3%	3,189,610	3.20	1h 02m
1-4	4 (final)	25.09	10.3%	1,856,645	1.89	1h 19m
5000 bootstrap instances Bootstrap Completion Time = 14 hours and 43 minutes						
2-0	0 (first)	32.69	6.9%	24,488,908	32.68	6h 27m
2-2	2	24.48	7.5%	10,372,652	14.31	8h 08m
2-4	4	24.65	8.3%	4,389,271	5.65	10h 04m
2-6	6	24.89	9.4%	2,443,556	3.49	12h 01m
2-8	8	25.08	10.2%	1,547,765	2.19	12h 59m
2-10	10	25.14	10.5%	1,369,474	2.07	13h 10m
2-12	12	25.29	11.2%	1,285,021	1.82	13h 58m
2-14	14	25.41	11.7%	1,086,280	1.55	14h 26m
2-16	16 (final)	25.50	12.1%	770,999	0.80	14h 43m
5000 bootstrap instances + dual lookup Bootstrap Completion Time = 9 hours						
3-0	0 (first)	24.72	8.2%	3,345,657	7.08	6h 52m
3-1	1	24.92	9.5%	746,831	1.57	7h 42m
3-2	2	25.23	10.9%	445,420	0.93	8h 16m
3-3	3	25.38	11.6%	323,678	0.53	8h 40m
3-4	4	25.50	12.1%	226,475	0.37	8h 50m
3-5	5	25.74	13.1%	168,890	0.28	8h 57m
3-6	6 (final)	**25.96**	**14.1%**	**92,098**	0.16	9h 00m

Table 3.11: 24-pancake puzzle, Bootstrap.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of break heuristic					
1	$break$	22.75	0%	46,442	0.03
Results of W-IDA*					
2	h_0 (W-IDA*,W=8)	52.34	130.1%	2,128,702	1.88
Results of BULB					
3	h_0 (BULB,B=20,000)	31.45	38.2%	6,050,234	43.2
4	h_0 (BULB,B=10)	643.08	2726.7%	267,017	1.10
Results of h_{sum}					
5	h_{sum}	39.74	74.7%	193,927	0.14

Table 3.12: 24-pancake puzzle, other methods.

to the Bootstrap, its solution quality has improved in comparison to the 17-pancake puzzle. This is because the heuristics in the feature vector became weaker and the chance for overestimating the heuristic value of each state using h_{sum} has decreased.

35-pancake puzzle

The input features for the NN are very similar to those for the 17- and 24-pancake puzzle. Here, we added an extra 5-token PDB as the number of tokens in the puzzle has increased. The pre-processing time to build the pattern databases was about 18 minutes while the memory used to hold the pattern databases was about 272 megabytes. 50 randomly generated instances, with an average optimal solution cost of 33.6, were used for testing. The initial heuristic is so weak that 7 RandomWalk iterations were necessary before bootstrapping itself could begin (9 iterations were required when there were only 500 bootstrap instances). Table 3.13 has rows for selected iterations with 500 bootstrap instances, 5000 bootstrap instances, and 5000 bootstrap instances with duality exploited.

The trends are almost the same as for the 17- and 24-pancake puzzle: (a) search becomes faster in each successive iteration but suboptimality becomes worse; (b) having more bootstrap instances is slower and results in extra Bootstrap iterations, which produces faster search but greater suboptimality; and (c) duality substantially reduces the search effort. However, it seems that the number of nodes generated does not always decrease in each successive iteration (for example, see rows 3-2 and 3-4 in Table 3.13). Although bootstrapping is expected to speed up the search in each successive iteration, here we only test the resulting heuristic on 50 test instances. Therefore, such anomalies might simply be the result of using too few test instances.

Table 3.14 compares a few iterations of Bootstrap when 50 and 1000 test instances are used. When using 50 test instances, some numbers do not follow the trend (see boldface numbers in the Table 3.14); however, these anomalies disappear when the number of test instances is increased to 1000. Similar anomalies exist in Tables 3.18, 3.20, and 3.16 because only 200, 50, and 10 test instances were used respectively.

No previous system that automatically creates heuristics has been applied to this problem domain because of its size, so Table 3.15 includes results only for the break heuristic, W-IDA*, BULB, and h_{sum}. Row 1 shows the results of the break heuristic.

W-IDA* was not able to compete with Bootstrap (Row 3-23 of Table 3.13) in terms of nodes generated, so Row 2 of the table just shows the minimum number of nodes that W-IDA* generated. We tried 15 values[12] for W between 1.1 and 10. As can be seen, W-IDA* produces a very high degree of suboptimality when generating the fewest nodes. Looking for settings for which W-IDA* can compete with Bootstrap in terms of suboptimality was not successful. Allowing 10 times more time than IDA* with Bootstrap's final heuristic (Row 3-23 of Table 3.13) needed on each test instance, W-IDA* did not complete any instances at all. Therefore, similar to the 17- and 24-pancake puzzle W-IDA* is not competitive with Bootstrap.

For BULB, we tried 15 values[13] for B (between 2 and 20,000). BULB is not competitive with Bootstrap in terms of either Nodes or Subopt. The best results of BULB are shown in rows 3 and 4.

Row 5 shows the result of h_{sum} using duality. In comparison to the smaller pancake problems, the results of h_{sum} has improved in terms of solution quality but it is still inferior to Bootstrap in terms of both Nodes and Subopt.

[12]We tried $W \in \{1.1, 1.2, 1.5, 1.8, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

[13]We tried $B \in \{2, 3, 5, 7, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 1 days and 2 hours						
1-0	0 (first)	37.12	10.5%	178,891,711	217	7h
1-1	1	37.02	10.2%	181,324,430	219	9h
1-2	2	37.42	11.4%	169,194,509	202	11h
1-3	3	37.35	11.2%	191,333,354	228	16h
1-4	4 (final)	37.78	12.4%	131,571,637	158	1day 02h
5000 bootstrap instances Bootstrap Completion Time = 8 days and 11 hours						
2-0	0 (first)	36.66	9.1%	2,766,675,135	4,168	1day 17h
2-2	2	36.86	9.7%	1,591,749,582	1,923	2days 07h
2-4	4	37.02	10.2%	586,345,3534	687	2days 20h
2-6	6	37.25	10.8%	295,187,243	345	3days 08h
2-8	8	37.47	11.5%	134,075,802	157	3days 18h
2-10	10	37.70	12.2%	65,290,479	102	4days 04h
2-12	12	37.74	12.3%	47,998,040	76	4days 20h
2-14	14	37.72	12.3%	45,571,411	71	5days 15h
2-16	16	37.76	12.3%	39,128,839	45	5days 23h
2-18	18	38.02	13.2%	38,126,208	43	6days 05h
2-20	20	37.92	12.9%	39,440,284	44	6days 16h
2-22	22	38.14	13.5%	36,423,262	52	7days 00h
2-24	24	38.36	14.2%	25,034,580	42	7days 10h
2-26	26	38.60	14.9%	26,089,593	43	7days 23h
2-28	28	38.65	15.0%	13,156,609	21	8days 07h
2-30	30 (final)	38.77	15.4%	14,506,413	21	8days 11h
5000 bootstrap instances + duality Bootstrap Completion Time = 3 days and 17 hours						
3-0	0 (first)	37.48	11.5%	147,325,035	339	1day 02h
3-2	2	37.82	12.6%	35,062,330	82	1day 12h
3-4	4	38.10	13.4%	35,167,589	82	1day 21h
3-6	6	38.18	13.6%	17,851,201	42	2days 05h
3-8	8	38.28	13.9%	15,465,958	35	2days 12h
3-10	10	38.66	15.1%	11,978,938	28	2days 18h
3-12	12	38.8	15.5%	11,047,491	25	2days 23h
3-14	14	39.04	16.2%	6,592,431	15	3days 03h
3-16	16	38.92	15.8%	6,763,642	11	3days 06h
3-18	18	39.10	16.4%	5,540,040	9	3days 10h
3-20	20	39.06	16.3%	7,340,341	12	3days 13h
3-22	22	38.98	16.0%	3,880,717	6	3days 16h
3-23	23 (final)	**39.50**	**17.6%**	**2,655,945**	4	3days 17h

Table 3.13: 35-pancake puzzle, Bootstrap.

3.3.3 Rubik's Cube

For Rubik's Cube, the input features for the NN were the three PDBs used by Korf [41], namely, one PDB for the eight corner cubies and two PDBs each for six edge cubies. We used 333 megabytes of memory for the PDBs[14] and the pre-processing took about 16 minutes.

[14]This is more than the amount of memory that Korf reported using the same PDBs [41]. We store each entry of the pattern database in 8 bits ignoring the fact that each entry is less than 12 and can be stored in 4 bits.

row	iteration	Cost	Subopt	Nodes	Solving Time
50 test instances					
3-2	2	37.82	12.6%	35,062,330	82
3-3	3	38.10	13.4%	**42,459,691**	98
3-4	4	38.10	13.4%	35,167,589	82
3-5	5	38.06	13.3%	17,372,344	38
3-6	6	38.18	13.6%	**17,851,201**	42
3-7	7	38.34	14.1%	**18,767,205**	44
3-8	8	38.28	13.9%	15,465,958	35
3-9	9	38.26	13.9%	**20,630,541**	48
1000 test instances					
3-2	2	37.94	12.6%	51,576,124	120
3-3	3	38.03	12.8%	43,326,193	101
3-4	4	38.11	13.1%	34,164,746	80
3-5	5	38.21	13.5%	26,357,552	60
3-6	6	38.30	13.6%	26,030,494	60
3-7	7	38.34	13.8%	19,790,206	46
3-8	8	38.43	14.0%	15,745,643	37
3-9	9	38.53	14.3%	14,709,532	35

Table 3.14: 35-pancake puzzle, Bootstrap on different number of test instances.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of break heuristic					
1	$break$	33.6	0%	670,337	1
Results of W-IDA*					
2	h_0 (W-IDA*,W=9)	71.1	111.6%	1,478,146,011	1,606
Results of BULB					
3	h_0 (BULB,B=20,000)	81.6	142.85%	202,149,804	1,750
4	h_0 (BULB,B=20)	3547.3	10457.3%	53,610,580	170
Results of h_{sum}					
5	h_{sum}	54.32	61.7%	5,449,933	5

Table 3.15: 35-pancake puzzle, other methods.

The 10 standard Rubik's Cube instances [41] were used for testing. The average optimal solution cost for these instances is 17.5. The initial heuristic was sufficient to begin the Bootstrap process directly, so no RandomWalk iterations were necessary. Row 6 of Table 3.17 shows the results when the initial heuristic is used by itself as the final heuristic [41].

Table 3.16 shows all the iterations of the Bootstrap process when 500 and 5000 bootstrap instances are given. In both cases, bootstrapping produces very substantial speedup over search using h_0. For instance, using 500 bootstrap instances produces a heuristic that makes search more than 43 times faster than with h_0 while producing solutions that are only 4% (0.7 moves) longer than optimal. The trends across Bootstrap iterations are almost the same as those observed in all previous experiments. Similar to the 35-pancake puzzle, the number of nodes generated does not always decrease between two consecutive iterations of Bootstrap (*e.g.,* see rows 2-2 and 2-3 of Table 3.16) as the results are obtained on only a few (10) test instances.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 2 days 19 hours						
1-0	0 (first)	18.0	2.9%	67,264,270,264	78,998	05m
1-1	1 (final)	18.2	4.0%	8,243,780,391	10,348	2days
5000 bootstrap instances Bootstrap Completion Time = 31 days and 15 hours						
2-0	0 (first)	18.1	3.4%	69,527,536,555	86,125	43m
2-1	1	18.3	4.6%	7,452,425,544	10,477	10h
2-2	2	18.5	5.7%	3,314,096,404	3,976	1day 06h
2-3	3	19.2	9.7%	3,722,365,147	4,444	2days 16h
2-4	4	19.7	12.6%	974,287,428	1,119	5days 08h
2-5	5	20.4	16.6%	748,608,645	848	7days 05h
2-6	6	21.4	22.3%	599,503,676	823	9days 09h
2-7	7	21.0	20.0%	614,676,983	842	11days 07h
2-8	8	21.3	21.7%	465,772,443	626	13days 04h
2-9	9	21.6	23.4%	552,259,662	624	16days 14h
2-10	10	21.5	22.9%	518,980,590	577	19days 10h
2-11	11	21.8	24.6%	624,542,989	686	23days 20h
2-12	12	21.8	24.6%	422,066,562	464	27days 06h
2-13	13	22.3	27.4%	251,228,458	280	30days 02h
2-14	14 (final)	**22.6**	**29.1%**	**192,012,863**	208	31days 15h

Table 3.16: Rubik's Cube, Bootstrap.

The results of other systems are shown in Table 3.17. Rows 1 and 2 are when the initial heuristic (h_0) is used with W-IDA* on the same set of test instances. Row 4 shows the results with h_{sum}. Similar to the previous domains, Bootstrap (Table 3.16, Row 2-14) dominates in all cases.

For BULB, we present the results reported by Furcy and Koenig [20]. These results are obtained using h_0 but on a different set of test instances. They created 50 solvable instances by applying random walks of length 500 backwards from the goal state. The optimal solutions for these instances are not known; to estimate the suboptimality of the solutions found we assume their average optimal solution is the same as for our test instances, 17.5. All the results related to BULB are marked with an asterisk which means that they are not strictly comparable with the numbers reported for our method. Row 3 reports on BULB when B is set so that its result is close to Bootstrap (Table 3.16, Row 2-14) in terms of both suboptimality and nodes generated. The results show that BULB and Bootstrap perform equally well in Rubik's Cube.

Rows 5 to 7 show the results of state-of-the-art heuristic search methods for finding optimal solutions for the testing instances. Row 5 shows the results using the initial heuristic (h_0) [41]. Row 6 shows the results from Zahavi *et al.* [70] when dual lookups [15] for both 6-edge PDBs were added to the heuristic of Row 5. In Row 7 [70], the edge PDBs used in Row 6 are increased from 6 edges to 7 edges and dual lookup is used. Bootstrap (in both cases of 500 and 5000 bootstrap instances) outperforms all of these optimal systems in terms of Nodes and Solving Time.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of W-IDA*					
1	h_0 (W-IDA*,W=1.9)	**22.8**	**30.3%**	5,653,954,001	6,632
2	h_0 (W-IDA*,W=3.3)	31.8	81.7%	**217,463,103**	245
Results of BULB					
3	h_0 (BULB,B=50,000)	22.7*	30%*	189,876,775*	431*
Results of h_{sum}					
4	h_{sum}	27.2	55.4%	246,235,226	256
Results from previous papers					
5	h_0	17.5	0%	360,892,479,670	102,362*
6	#5 with dual lookup	17.5	0%	253,863,153,493	91,295*
7	max{8,7,7} with dual lookup	17.5	0%	54,979,821,557	44,201*

Table 3.17: Rubik's Cube, other methods.

3.3.4 Blocks World

15-blocks world

For the 15-blocks world we used 200 random test instances in which the goal state has all the blocks in one stack. Their average optimal solution length is 22.73. We used 9 input features for the NN: seven 2-block PDBs, the number of out-of-place blocks, and the number of stacks of blocks. The abstractions used for the PDBs is so coarse-grained that we only used less than 2 kilobytes of memory for this experiment. Optimal solutions were computed using the hand-crafted blocks world solver PERFECT [60].

Table 3.18 shows the Bootstrap results. The initial heuristic is so weak that three RandomWalk iterations were needed before bootstrapping. The trends are, again, (a) search is sped up in each iteration but suboptimality increases; and (b) having more bootstrap instances is slower and requires more Bootstrap iterations. A suboptimality of 7% here means the solutions generated are about 1.6 moves longer than optimal.

Table 3.19 shows the results of BULB using our initial heuristic, which is again dominated by Bootstrap (Table 3.18, Row 2-9).

An attempt to compare with W-IDA* failed due to the poor performance of W-IDA* with time limits 10 times larger than the solving time using Bootstrap's final heuristic for each test instance. Varying the weights[15] between 1.1 and 10, W-IDA* never solved more than about 70% of the instances (W=8 was best). The average number of nodes that each setting of W-IDA* generated with this time limit was always more than Bootstrap. Each setting of W-IDA* still needs to generate more nodes to find a solution for those instances that were not solved during this time limit; therefore, W-IDA* generates more nodes than Bootstrap. A similar explanation can be made to show that the setting of W-IDA* that is closest to Bootstrap in terms of suboptimality, is inferior in terms of nodes generated. Therefore, Bootstrap (Table 3.18, Row 2-9) outperforms W-IDA* in terms of both Nodes and Solving Time.

[15]We tried $W \in \{1.1, 1.2, 1.5, 1.8, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 1 hour and 46 minutes						
1-0	0 (first)	23.06	1.5%	1,157,510,765	2,656.99	52m
1-1	1	23.38	2.8%	554,160,659	2,149.07	1h 22m
1-2	2	23.80	4.7%	21,289,247	69.37	1h 44m
1-3	3 (final)	24.31	6.9%	3,651,438	15.87	1h 46m
5000 bootstrap instances Total Time to create the final heuristic = 8 hours						
2-0	0 (first)	23.09	1.6%	2,253,260,711	5,081.57	4h 46m
2-1	1	23.34	2.7%	280,752,780	1,247.28	5h 23m
2-2	2	23.48	3.3%	132,234,387	587.87	6h 03m
2-3	3	23.62	3.9%	44,616,679	101.19	6h 40m
2-4	4	23.66	4.1%	11,973,435	26.58	7h 14m
2-5	5	24.12	6.1%	2,777,423	6.13	7h 30m
2-6	6	24.25	6.7%	3,468,436	7.65	7h 40m
2-7	7	24.29	6.9%	1,252,535	2.76	7h 51m
2-8	8	24.35	7.1%	534,123	1.45	7h 57m
2-9	9 (final)	**24.40**	**7.3%**	**155,813**	0.35	8h 00m

Table 3.18: 15-blocks world (1-stack goal), Bootstrap.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of BULB					
1	h_0 (BULB,B=4000)	**24.40**	**7.3%**	972,380	2.09
2	h_0 (BULB,B=500)	28.28	24.4%	**177,187**	0.46

Table 3.19: 15-blocks world, other methods.

An attempt to compare the results to h$_{sum}$ failed because the heuristics used in the feature vector were so weak that even the sum of these values is still a weak heuristic for this domain. Using a time limit of 10 minutes for IDA* using h$_{sum}$, only 73 out of 200 instances were solved while the average solution length for the solved instances was about 5 times longer than the average optimal solution length of those instances.

We compare our solution quality (Table 3.18, Row 2-9) to three hand-crafted suboptimal solvers for the blocks world, US [21] then, GN1 [22], and GN2 [22]. These planners do not use search algorithms, so we cannot compare in terms of the number of nodes generated.

US [21] (unstack-stack) is the simplest strategy for the blocks world that was introduced by Gupta and Nau. It moves all the misplaced blocks to the table (unstack) and then builds the goal state by applying constructive moves (stack). A constructive move is a move that places a misplaced block in its correct position. GN1 [22] first applies all possible constructive moves. When no constructive move exists, it moves a random misplaced clear block[16] to the table. GN1 repeats this process until all blocks are placed in their correct positions. GN2 [22] is similar to GN1 except that in case of no available constructive move, it moves a clear block to the table that leads to a state in

[16]A block is clear when it is the topmost block on the stack (*i.e.,* no block exists on top of a clear block).

which constructive moves become available.

With an average solution length of 24.4, Bootstrap performed almost as well as GN1 (23.88) and GN2 (22.83), and slightly better than US (25.33).

20-blocks world

The experimental setup for the 20-blocks world was identical to that for 15 blocks, but here a 2-block PDB is a much weaker heuristic than for 15 blocks. The total amount of memory required for this experiment was less than 3 kilobytes while the pre-processing took a few seconds.

We used 50 random test instances with an average optimal solution length of 30.92. The initial heuristic is so weak that six RandomWalk iterations were necessary before bootstrapping (eight iterations for 500 bootstrap instances). The trends across Bootstrap iterations are those observed in the 15-blocks world.

Table 3.20 shows the Bootstrap iterations for the 20-blocks world. The heuristics used in the feature vector were so weak that solving the test instances using the early heuristics produced by Bootstrap was infeasible; therefore, rows 1-0 and 2-0 to 2-2 are not shown in the table. The completion time of Bootstrap using 500 bootstrap instances is much longer than the total time to learn the final heuristic of Bootstrap using 500 instances (rows 1-3 of Table 3.20) as “enough” instances were not solved in the last iteration of the Bootstrap and the process terminated due to $t_{\max}$ exceeding t_{∞}.

row	iteration	Cost	Subopt	Nodes	Solving Time	Learning Time
500 bootstrap instances Bootstrap Completion Time = 2 days						
1-1	1	31.38	1.5%	13,456,726,519	55,213	11h
1-2	2	31.54	2.0%	8,886,906,652	35,692	1day 02h
1-3	3 (final)	32.02	3.6%	615,908,785	2,763	1day 10h
5000 bootstrap instances Bootstrap Completion Time = 11 days and 1 hour						
2-3	3	31.56	2.1%	12,771,331,089	52,430	3days 06h
2-4	4	31.86	3.0%	8,885,364,397	35,636	4days 04h
2-5	5	31.94	3.3%	941,847,444	3,828	5days 21h
2-6	6	32.10	3.8%	660,532,208	2,734	7days 03h
2-7	7	32.14	3.9%	789,515,580	3,240	8days 05h
2-8	8	32.50	5.2%	191,696,476	791	9days 05h
2-9	9	32.84	6.2%	22,413,312	93	9days 22h
2-10	10	33.28	7.6%	11,347,282	47	10days 18h
2-11	11	33.50	8.3%	17,443,378	72	10days 10h
2-12	12	33.56	8.5%	7,530,329	31	10days 20h
2-13	13 (final)	33.78	**9.2%**	**5,523,983**	23	11days 01h

Table 3.20: 20-blocks world (1-stack goal), Bootstrap.

Bootstrap with an average solution length of 33.78 (Table 3.20, Row 2-13) is again somewhat inferior to GN1 (32.54) and GN2 (30.94), and slightly better than US (34.58).

The results of other systems on the same test instances are shown in Table 3.21. Similar to the

15-blocks world, W-IDA* with time limits 10 times larger than the solving time using Bootstrap's final heuristic for each test instance failed to solve more than half the test instances (W was varied[17] between 1.1 and 10). In the best case (W=9) W-IDA* solved 24 of the test instances. In addition, h_{sum} failed to solve any instance given a time limit of one day per instance. Therefore, Table 3.21 only shows the results for BULB.

BULB's results when B is set so that BULB is approximately equal to Bootstrap (Table 3.20, Row 2-13) in terms of Nodes is shown in Row 2. For suboptimality, BULB could not compete with Bootstrap; we tried 15 values[18] for B between 2 and 20,000. The best suboptimality achieved by BULB is shown in Row 1. It shows that even with much greater suboptimality, BULB is inferior to Bootstrap in terms of Nodes and Solving Time.

row	h (Algorithm)	Cost	Subopt	Nodes	Solving Time
Results of BULB					
1	h_0 (BULB,B=20,000)	40.14	29.8%	278,209,980	2,482
2	h_0 (BULB,B=2,400)	64.8	109.6%	**5,809,791**	32

Table 3.21: 20-blocks world, other methods.

3.4 Summary

This chapter described our incremental method that aims to generate strong heuristics from a given initial (weak) heuristic h_0 and a set of states called the bootstrap instances. A heuristic search algorithm with h_0 is run to solve the bootstrap instances within a given time limit. For each solved bootstrap instance, a set of features and the solution length for that instance is fed to a learning system to create a new heuristic function h_1 which is intended to be stronger than h_0 in terms of the average speed of solving the problems. After that, the previously unsolved bootstrap instances are used in the same way, using h_1 as the heuristic instead of h_0. This procedure is repeated until a sufficiently strong heuristic is created.

The initial heuristic h_0 can be so weak that it cannot solve even a few of the bootstrap instances within the given time limit. We then enhance h_0 by a random walk method that aims to improve the initial heuristic to a point that the new heuristic becomes able to solve enough bootstrap instances. The RandomWalk method generates training instances that are (i) hard enough to yield useful training data to improve h_0 and (ii) easy enough to be solvable by h_0.

We provided experimental evidence that machine learning can help to create strong heuristics from given (weak) ones. The key approach is a bootstrap learning procedure for improving heuristics incrementally, with successive iterations solving gradually harder problems from a user-given set of instances. We also demonstrated that random walks can be effectively used to prime the bootstrapping process if a heuristic is too weak to solve enough of the user-given instances.

[17]We tried $W \in \{1.1, 1.2, 1.5, 1.8, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.
[18]We tried $B \in \{2, 3, 5, 7, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$.

Throughout all tested domains, we observed a huge reduction in the number of nodes IDA* generates with the learned heuristics. The total time needed for learning the heuristic was rather large for large problem domains. That makes bootstrapping useful only when a large number of test problem instances need to be solved. In the next chapter, a variation of Bootstrap will be presented that decreases this total time substantially so that Bootstrap becomes practical (in terms of total time) even for solving a single test instance.

Chapter 4

Solving Single Instances

4.1 Introduction

The experimental results in Chapter 3 showed that the combination of our RandomWalk and Bootstrap methods (bootstrapping) can help to speed up search dramatically with relatively little degradation in solution quality. However, the completion time for bootstrapping becomes rather large for large search spaces (*e.g.,* the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world) so that it becomes ineffective to use bootstrapping for situations in which the user is interested in solving a single instance (or a few instances) of a search problem. For example, many planning problems require just a single instance to be solved; a task for which our bootstrapping approach may seem ineffective because of the large total time required.

In this chapter, we present a method that interleaves learning better heuristics and solving the test instance using the current heuristics so that the sum of the learning and solving times is made as small as can be. Here, we first describe our method. Then, we present experimental results of our method showing that it solves single instances of problems many orders of magnitude faster than the total time needed for bootstrapping while the solutions obtained are still very close to the optimal ones. Finally, we first briefly summarize related work on solving a single instance of a heuristic search/planning problem.

4.2 Method

The experimental results with our bootstrap learning method in Chapter 3 showed that most of the bootstrapping time was spent on the instances that were not solvable in each iteration of the process using the heuristic of that iteration. On the other hand, the RandomWalk process was not very time-consuming as most of the random walk instances were at the "right" level of difficulty. Therefore, one possible approach to decrease the total time of Bootstrap is to use the RandomWalk process to iteratively improve the initial heuristic until the heuristic becomes able to solve the single instance of the problem.

In other words, given a single instance of a problem, which we call ins*, as the test instance, we can run our RandomWalk process to learn heuristics to solve this instance. If the initial heuristic is good enough to solve the instance in time t_{max} then the system returns the solution and terminates. Otherwise, the system would invoke the RandomWalk process until it could solve the instance in time t_{max}. The system increases the length of the random walk to create harder instances and increases t_{max} when fewer than ins_{min} instances were solved in the current iteration. The total time of this system would be the total time for solving the instance.

This system (in terms of total time) is faster compared to the case of using many bootstrap instances. In other words, unlike the bootstrap instances (provided by the user), the RandomWalk process produces instances that are most of the time at the "right" level of difficulty; therefore, less time would be spent on the instances that are not solvable within the time limit of t_{max} in each iteration.

Figure 4.1 shows the results of such a system after 25 iterations of the RandomWalk process (about 80 minutes of total time) when each of the 50 random instances of the 24-puzzle used by Korf and Felner [42] is used as the single test instance. In other words, the method just described was repeated 50 times, each time using one of the 50 test instances as ins^*. The x-axis shows the total time of the process while the y-axis shows the number of instances that were solved in that time or less. For example, a point with the x-value of 20 and y-value of 19 in Figure 4.1 shows that if this system is given 20 minutes of total time for each instance, 19 out of 50 test instances will be solved.

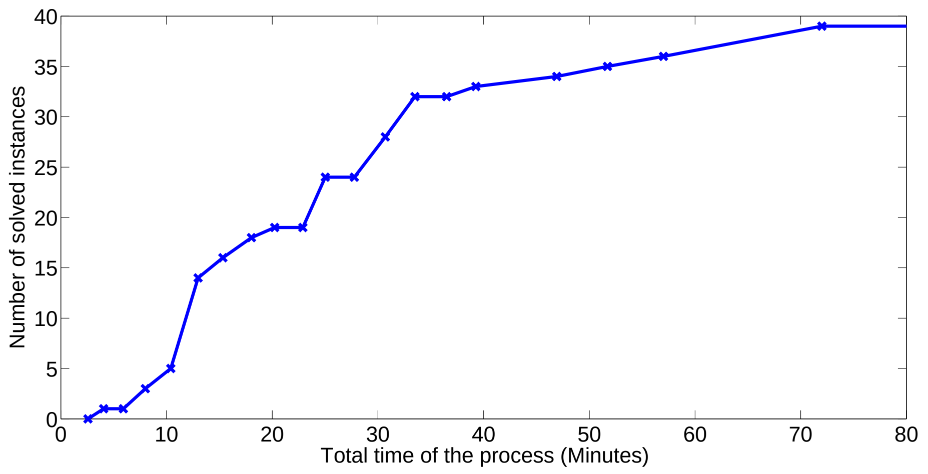


Figure 4.1: Number of solved test instances versus the total time of the process on the 24-puzzle.

Figure 4.1 shows that after 80 minutes of total time, only 39 out of 50 test instances were solved. This is the result of using the time limit t_{max} and forcing ins^* to be solvable within that time limit. When dealing with a large set of bootstrap instances, t_{max} is less critical, because we only need to solve a small fraction of the bootstrap instances in each iteration. In contrast, in the case of a solving a single instance of the problem without any bootstrap instances the total time will strongly depend on the parameter t_{max}. If t_{max} is too low, we might need a great many iterations of the RandomWalk process. If $t_{\max}$ is too high, we force the solver, when using too weak a heuristic, to spend the full amount of $t_{\max}$ in vain while it would be advantageous to invest more in learning. For example, in the above example the time limit is so low (4 seconds after 25 iterations of the RandomWalk process) that only 39 of the 50 test instances were solved even after great many iterations of the RandomWalk process.

Avoiding $t_{\max}$ completely by fixing a training time and then trying to solve the test instance with the last heuristic learned after the fixed amount of training is one other possibility. For example, if we stop the RandomWalk process in the same experiment after only 10 minutes, the 50 test instances of the 24-puzzle will need 278 seconds of solving time on average (the hardest test instance needs about 86 minutes of solving time). Note that this solving time is in addition to the 10 minutes spent on learning better heuristics. On the other hand, if the process stops after 20 minutes, the test instances need 38 seconds of solving time on average (the hardest test instance needs about 3 minutes). Unfortunately, here the training time is the critical parameter that cannot be set without prior knowledge.

These observations motivate us to interleave solving and learning the heuristic. We alternatively invest some time in two threads, namely in (i) learning a better heuristic (the learning thread), and in (ii) trying to solve the test instance ins* (the solving thread). The solving thread is comprised of sub-threads which themselves are executed in an interleaved fashion. The first "solving sub-thread" aims at solving ins* using the initial heuristic. As soon as a new heuristic is learned in the learning thread, we start an additional solving sub-thread, which uses the new heuristic to try to solve ins*. No thread is ever stopped completely until ins* is solved in one of the solving sub-threads. The total time by which we evaluate this process is the sum of the times used by both the learning and solving threads (including all the solving sub-threads) up to the point when ins* is solved in one sub-thread.

Algorithm 3: The Interleaving procedure.

```
1  procedure Interleaving(ins*, h_in, $t_s$, $t_l$): solution
2  Create a list, Solvers, containing just one solving sub-thread using h_in.
3  solved := false
4  while (!solved) do
5      solved := InterleavedSolving(Solvers,$t_s$)
6      $h$ := continue(RandomWalk,$t_l$)
7      if ($h$ is a new heuristic) then
8          add a solving sub-thread using $h$ to the beginning of Solvers
9      end
10 end
11 return solution from the interleaved solving processes
```

Algorithm 3 shows our interleaving approach. The algorithm uses two threads, one for learning new heuristics and the other for trying to solve the test instance. The learning thread calls the RandomWalk procedure (line 6). This RandomWalk procedure does not use any bootstrap instances at all and is suspended after a time limit of t_l. Therefore, "continue(RandomWalk,t_l)" in line 6 of the algorithm means that the RandomWalk process runs for t_l seconds. Then, it is suspended when the time limit t_l is reached.

The solving thread uses the current list of available solving sub-threads (Solvers) to solve the test instance (by the call to the interleaved solving procedure in line 5). The solving thread is suspended after a time limit of t_s. When the algorithm starts, Solvers only contains one solving sub-thread that uses the initial heuristic. Whenever RandomWalk finishes creating a new heuristic, a solving sub-thread using that heuristic is created and added at the beginning of Solvers (line 8). The process stops when a solution for the test instance is found.

The ratio of t_s:t_l determines the time allocated to solving (t_s) and the time allocated to learning (t_l). Determining the best ratio t_s:t_l is beyond the scope of this thesis; in the current system it is set manually. In Section 4.3, we run experiments with ratios of 1:1, 1:2, 1:5, and 1:10. Generally, when the initial heuristic is very weak, the solving sub-thread that uses this heuristic needs a lot of time to find a solution for the test instance. As we mainly use weak initial heuristics in our experiments to solve large search problems, we did not use ratios favouring the solving thread.

For the allocation of solving time among the various solving sub-threads, we considered two strategies, which we call “uniform” and “exponential”. Other strategies, such as Röger and Helmert’s alternation technique [52], are certainly possible.

Uniform This strategy allocates the same amount of time to all the available solving sub-threads every time line 5 in Algorithm 3 is executed. Note that this results in sub-threads that were started earlier—which presumably are using weaker heuristics—getting allocated more time, in total, than sub-threads using stronger heuristics. Pseudocode for this strategy is given in Algorithm 4.

This uniform strategy to allocate the same time budget to all available solving sub-threads borrows from Valenzano *et al.* [64]. Their work considers simultaneous searches with different parameter settings (*e.g.,* Weighted A* searches with different weight parameters). Their results show that on a set of testing instances of a standard heuristic search domain (*e.g.,* the sliding-tile puzzle), simultaneous searches speed up the problem solving over the best single parameter setting on the same set of test instances. Although our uniform strategy to allocate time to solving sub-threads is similar to the concept of simultaneous searches with different parameter settings, we are looking at a different problem as (i) not all the solving sub-threads in our method are available from the beginning and (ii) there is also a learning time overhead associated with each new solving sub-thread.

Exponential Because heuristics created later in the learning process are expected to be stronger than those created at early stages, the more recently created heuristics may be more likely to quickly solve the test instance, and it therefore seems reasonable to invest more time in solving sub-threads using the heuristics learned in later iterations. The “exponential” strategy is to halve the time allocated to the solving sub-threads using previous heuristics when a new heuristic has been created. Thus the solving sub-thread for the new heuristic gets half the total time available for solving (t_s)

Algorithm 4: Uniform allocation for solving sub-threads.

```
procedure InterleavedSolving (uniform) (Solvers,time):status
$t := \frac{\text{time}}{|\text{Solvers}|}$
for each solver sub-thread $S \in Solvers$ do
    if continue($S$, $t$) succeeds then
        return true
    end
end
return false
```

on each round until another heuristic is created. The reason not to suspend solving sub-threads with weak heuristics completely is that there is still a chance that they are closer to finding a solution than the solving sub-thread using the most recently created heuristic. This may be (i) because more time has been invested in the sub-threads using weaker heuristics already or (ii) because a weaker heuristic may occasionally still behave better on one particular test instance than an overall stronger heuristic.

Algorithm 5: Exponential allocation for solving sub-threads.

```
procedure InterleavedSolving (exponential) (Solvers,time):status
$t$ := time
for $i$ = 1 to $|Solvers|$ do
    $S$ := $i^{th}$ sub-thread in Solvers
    if $i \neq |Solvers|$ then
        $t := t/2$
    end
    if continue($S$, $t$) succeeds then
        return true
    end
end
return false
```

The pseudocode for this time allocation strategy is shown in Algorithm 5. The time invested in the solving sub-thread using the best available heuristic is twice as large as that invested in the sub-thread using the second best heuristic, which again is a factor of two larger than the time for the next "weaker" sub-thread, and so on. The weakest two sub-threads will always be allocated the same amount of time, so that the total time spent on the sub-threads sums up to the time allocated to the solving thread overall.

This strategy for allocating the total solving time into time budgets for the currently available heuristic solvers borrows from the hyperbolic dove-tailing approach to interleaved search introduced by Kirkpatrick [38]. Kirkpatrick proved his approach to be average-case optimal and worst-case optimal for a certain variation of the so-called cow path problem, which was first studied by Baeza-Yates, Culberson, and Rawlins [1].

Kirkpatrick [38] studied a multi-list traversal problem in which the length of each list is unknown. Traversing each list is associated with a cost equal to the length traversed in the list and there is no cost associated with re-traversing the previously traversed part of each list. The objective is to traverse one list to the end while minimizing the total traversal cost.

Our problem of how to allocate time to solving sub-threads in order to make the total time of solving a test instance as small as possible is similar to the multi-list traversal problem in the sense that each solving sub-thread in our strategy can be considered as a list in the multi-list traversal problem. Traversing a list to the end would then be the same as finding a solution by a solving sub-thread. The main difference between these two problems is that not all the solving sub-threads in our problem are available from the beginning. In fact only the first solving sub-thread (the one that uses the initial heuristic) is available from the beginning. Whenever a new heuristic is learned, a new solving sub-thread will be added. Furthermore, each new solving sub-thread has the inherent cost of learning the heuristic that is not considered in Kirkpatrick's study. Therefore, the problem studied by Kirkpatrick does not model the search problem we are facing. Hence we do not have any formal guarantees on the efficiency of our exponential allocation method.

4.3 Experimental Results

This section discusses the experimental results of both versions of our interleaving approach on single instances of the largest search spaces used in Chapter 3, namely the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world. We further run experiments on the blocksworld instances used in the AIPS planning competition[1]. The experimental settings for each domain, *i.e.,* the initial heuristic, the features, and the neural network settings, are the same as those used in Chapter 3. Furthermore, the test instances and the numeric parameters, *i.e.,* $\text{ins}_{\min}$, $\text{t}_{\max}$, t_{∞}, and the size of the set RWIns, are the same as those used in Chapter 3.

We use a fixed ratio of t_s:t_l of time for training and solving processes. We report the results with ratios of 1:1, 1:2, 1:5, and 1:10 *i.e.,* t_s was set to 1 second while t_l varied between 1, 2, 5, and 10 seconds.

The tables and figures below summarize the results on our test domains. In the tables, the **allocation** (either uniform or exponential) and **ratio (**t_s**:**t_l**)** columns show the experimental setting used for that line of the table. The **min**, **max**, **mean**, **med**, and **std** columns respectively show the minimum, maximum, mean, median, and standard deviation of the solving time on the test instances. The **Subopt** column indicates the average suboptimality of our solutions.

The figures show the total time to solve each instance using our interleaving approach with different settings for the ratio. The instances on the x-axis are always sorted with respect to the setting that resulted in the best performance in terms of average total time. Furthermore, a logarithmic scale is always used for the y-axis (total solving time).

[1]See `http://www.cs.toronto.edu/aips2000/` for more details about the instances.

Most of the figures are consisted of two parts. The top part shows the results when the y-axis starts from zero and the bottom part shows the results when the y-axis starts from a time in which the minimum solving time by all the settings shown in the figure is achieved. The bottom part of the figure is only included so that the difference between difference strategies can be better visualized.

4.3.1 24-puzzle

We used each of the 50 standard test instances of the 24-puzzle [42] as the single instance to be solved. These instances have an average optimal solution cost of 100.78.

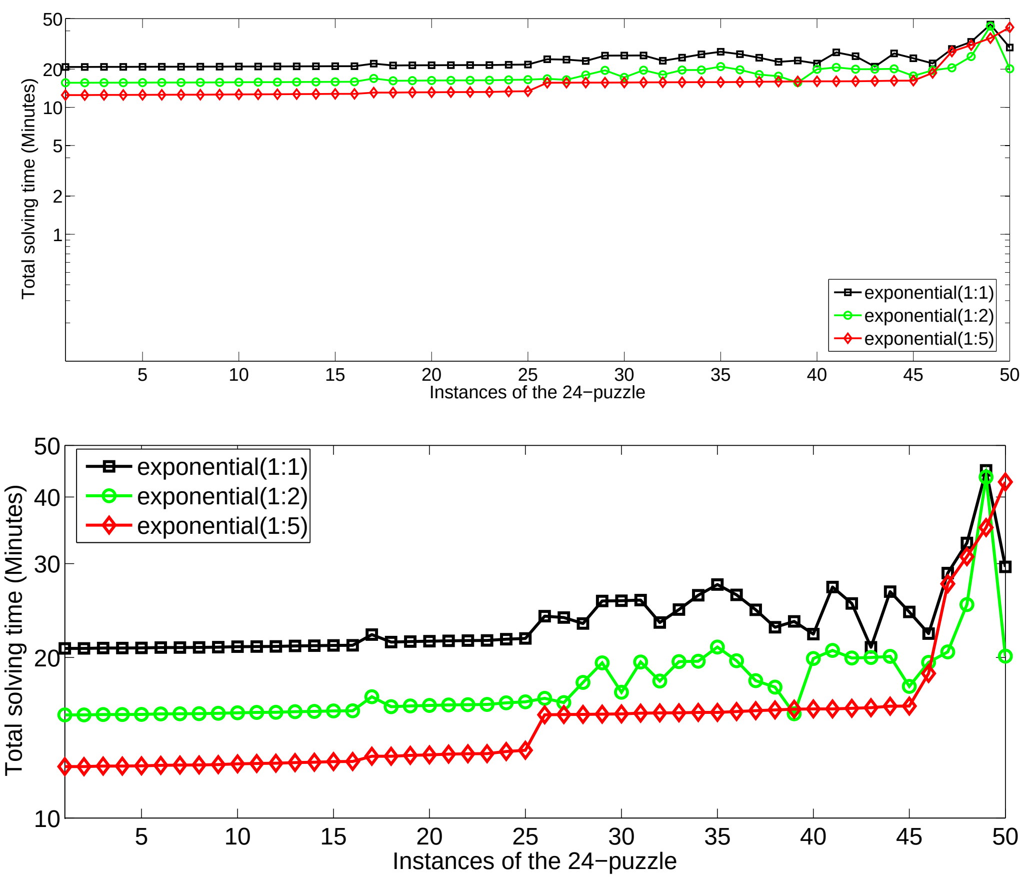


Figure 4.2: Total solving time for each instance of the 24-puzzle - exponential allocation.

Figure 4.2 shows the total time to solve each instance using the exponential interleaving approach with ratios 1:1, 1:2, 1:5. The instances on the x-axis are sorted with respect to the time that the approach with ratio of 1:5 needs to solve them. Using a ratio of 1:5, almost all the test instances (47 out of 50) are solved in less than 30 minutes (16 minutes and 30 seconds on average) while the solutions were, on average, only 6.9% (7 moves) longer than optimal.

Note that the exponential interleaving that uses a ratio of 1:5 (“exponential (1:5)” in Figure 4.2) corresponds to a step function. Each interval in that function shows the instances that were solved

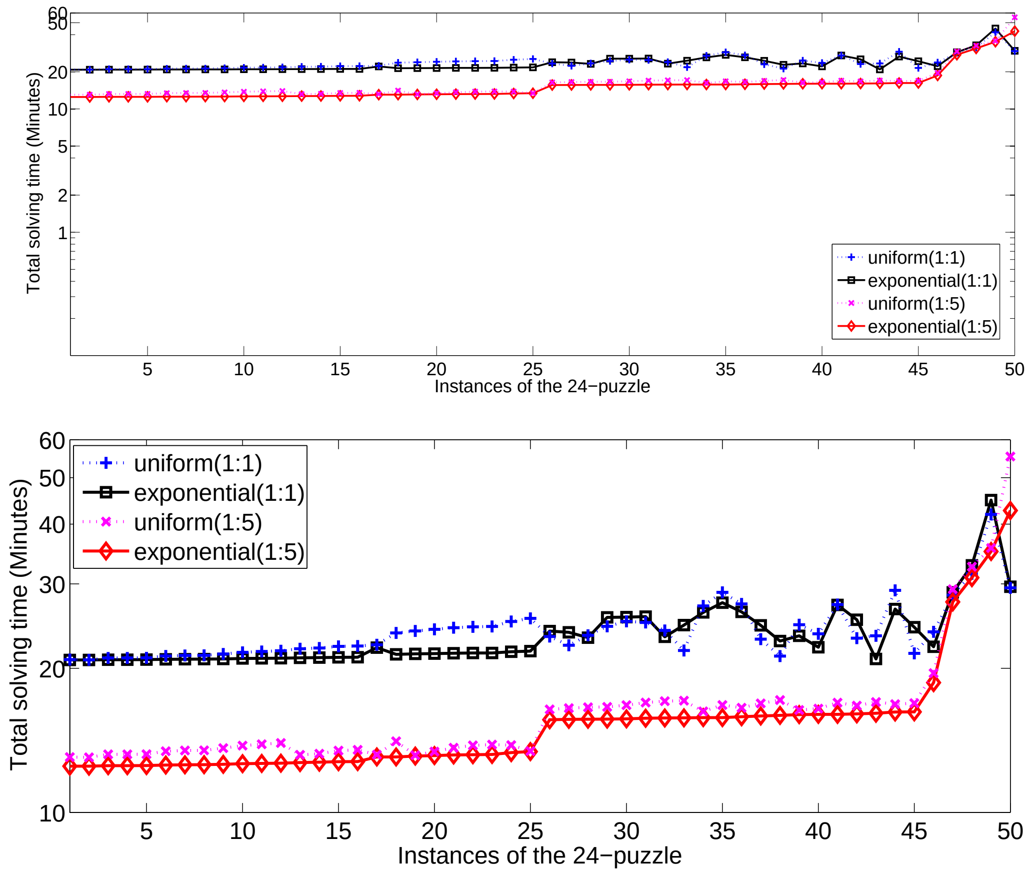


Figure 4.3: Total solving time for each instance of the 24-puzzle - exponential and uniform allocations.

using the same heuristic. For example, the first interval (for instances 1-16) contains all the instances that were solved using the heuristic learned from tenth iteration of the RandomWalk process.

Figure 4.2 shows that increasing the time allocated to the learning process relative to the solving process, speeds up the process and reduces the total solving time for almost all the instances. It is because the heuristics created in the earlier iterations of the RandomWalk process are so weak that the solving sub-threads that use them need a lot of time to solve the problem. Therefore, it is reasonable to allocate less time to these solvers and focus on learning better heuristics.

Figure 4.3 compares the solving time of two time-allocation strategies for ratios of 1:1 and 1:5. Note that the results for the ratios of 1:2 and 1:10 are not included in Figure 4.3 just to make the figure easier to read. Figure 4.3 shows that for a fixed ratio, the exponential strategy for allocating time among the solvers outperforms the uniform strategy for almost all the instances.

Table 4.1 shows the results for both time-allocation strategies for the solvers (uniform and exponential) for all four t_s:t_l ratios. Note that when the ratio is set to 1:20 (for both time-allocation strategies), the total time of the process increased compared to the case of using a ratio of 1:10. Therefore, the results for the ratio of 1:20 are not shown in the table. The trends observed from

Table 4.1 are:

1. The average suboptimality increases when the time allocated to the learning process increases relative to the solving process. This is simply because the solver that uses a heuristic from later iterations of the RandomWalk process will be able to solve the instance in a shorter total time. The more iterations of the RandomWalk process, the more suboptimal the solutions.

2. The mean and median values decrease and the range (max-min) increases as more time is allocated to the learning process. It turns out that, on average, the actual solver that first solves the test instance requires only a couple of seconds of solving time. Considering the total times reported in Table 4.1, most of the solving is spent on unsuccessful trials using other solvers. Increasing the learning time makes the system produce stronger heuristics faster. This in turn decreases the total solving time for most instances (mean and median decreases). However, the hardest instances, *i.e.*, those with the largest solving times, appear to require about the same amount of solving time as in a setting with less learning time.

3. The exponential strategy for allocating time among the solvers outperforms the uniform strategy for all t_s:t_l ratios.

row	ratio (t_s:t_l)	allocation	min	max	mean	med	std	Subopt
1	1:1	exponential	20m 48s	44m 54s	23m 36s	21m 54s	4m 07s	6.4%
2	1:1	uniform	24m 06s	41m 54s	24m 10s	23m 24s	3m 36s	6.4%
3	1:2	exponential	15m 36s	43m 36s	18m 03s	16m 30s	4m 15s	6.7%
4	1:2	uniform	15m 36s	44m 57s	18m 34s	17m 03s	3m 28s	6.8%
5	1:5	exponential	12m 30s	42m 42s	15m 50s	14m 30s	5m 53s	6.9%
6	1:5	uniform	13m 02s	55m 22s	16m 53s	15m 10s	7m 14s	6.9%
7	1:10	exponential	11m 36s	48m 48s	15m 59s	14m 36s	6m 42s	6.9%
8	1:10	uniform	11m 31s	53m 46s	15m 48s	14m 14s	6m 55m	6.9%

Table 4.1: Statistics on solving a single instance of the 24-puzzle.

The average total time spent on a test instance (including the learning time) is substantially lower than the total time spent by our Bootstrap process using a large set of bootstrap instances but no interleaving. According to Table 3.7, the latter requires more than 2 days when using 500 bootstrap instances and about 18 days when using 5000 bootstrap instances. The first heuristics created by Bootstrap for the 24-puzzle (rows 1-0 and 2-0 of Table 3.7 for using 500 and 5000 bootstrap instances) need about 80 and 40 minutes of solving time on average, respectively, to solve each instance of the 24-puzzle. These solving times are comparable to the total times reported for the interleaving (see Table 4.1); however, the standard Bootstrap method has an additional overhead of 3 and 27 hours (see the "Learning Time" Column of rows 1-0 and 2-0 of Table 3.7) to create those two heuristics. Therefore, our method to solve a single test instance is substantially faster than the normal Bootstrap method.

4.3.2 35-pancake Puzzle

Here, each bootstrap instance is a random instance of the 35-pancake puzzle. We used 50 of them, same as those instances uses in Section 3.3.2. These instances have an average optimal solution length of 33.6. Figure 4.4 shows the total time to solve each instance using the exponential strategy for allocating time among the solvers when the ratio t_s:t_l is set at 1:1, 1:2, and 1:5.

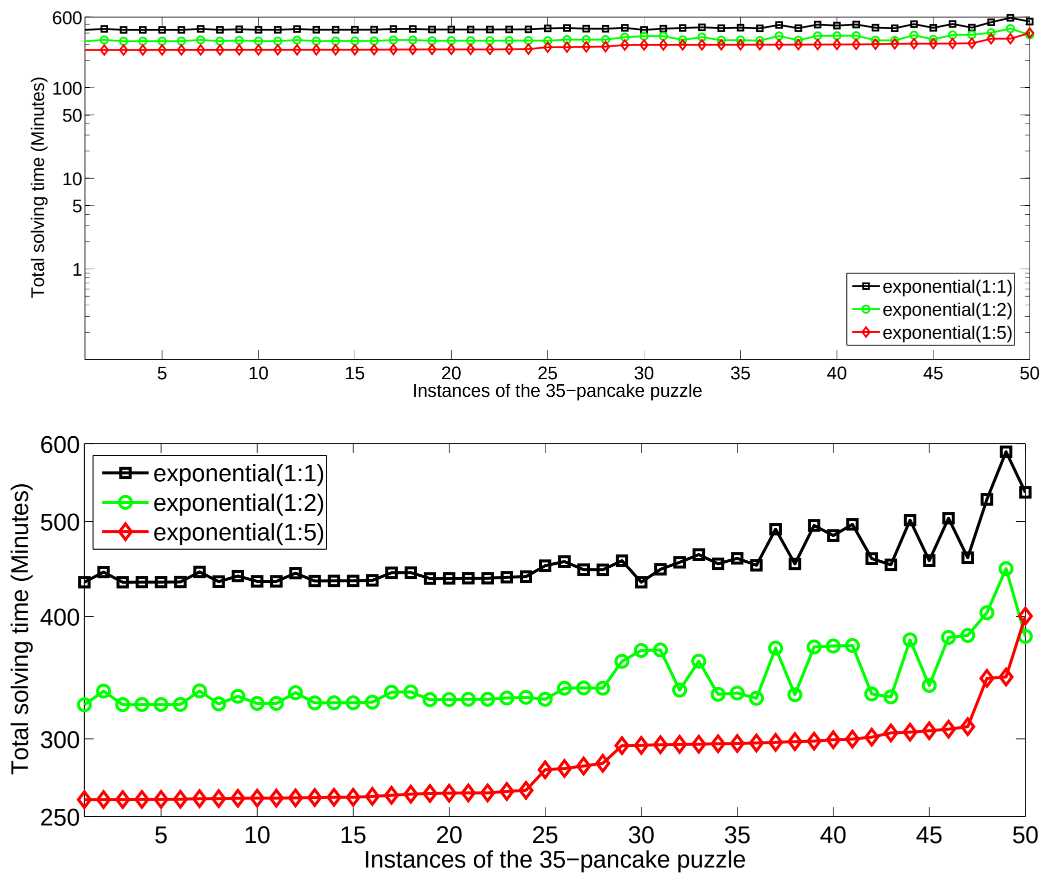


Figure 4.4: Total solving time for each instance of the 35-pancake puzzle - exponential allocation.

The best solving time averaged on these 50 instances was achieved using the ratio of 1:10. It solved the instances on average in 4 hours and 28 minutes while the solutions were 8.4% (2.9 moves) longer than optimal (see Table 4.2). Similar to the 24-puzzle, increasing the time allocated to the learning process relative to the solving process, speeds up the process while suboptimality slightly increases (see Figure 4.4 and Table 4.2). Note that the total time of the process increases when the ratio is set to 1:20 for both time-allocation strategies (not shown in the figures or tables).

Figure 4.5 compares our two different time-allocation strategies for two fixed ratios of 1:1 and 1:5. It again shows that for a fixed ratio, the exponential strategy outperforms the uniform strategy in terms of total time for almost all the instances.

Table 4.2 provides detailed statistics on each of these allocation strategies and ratio settings. The

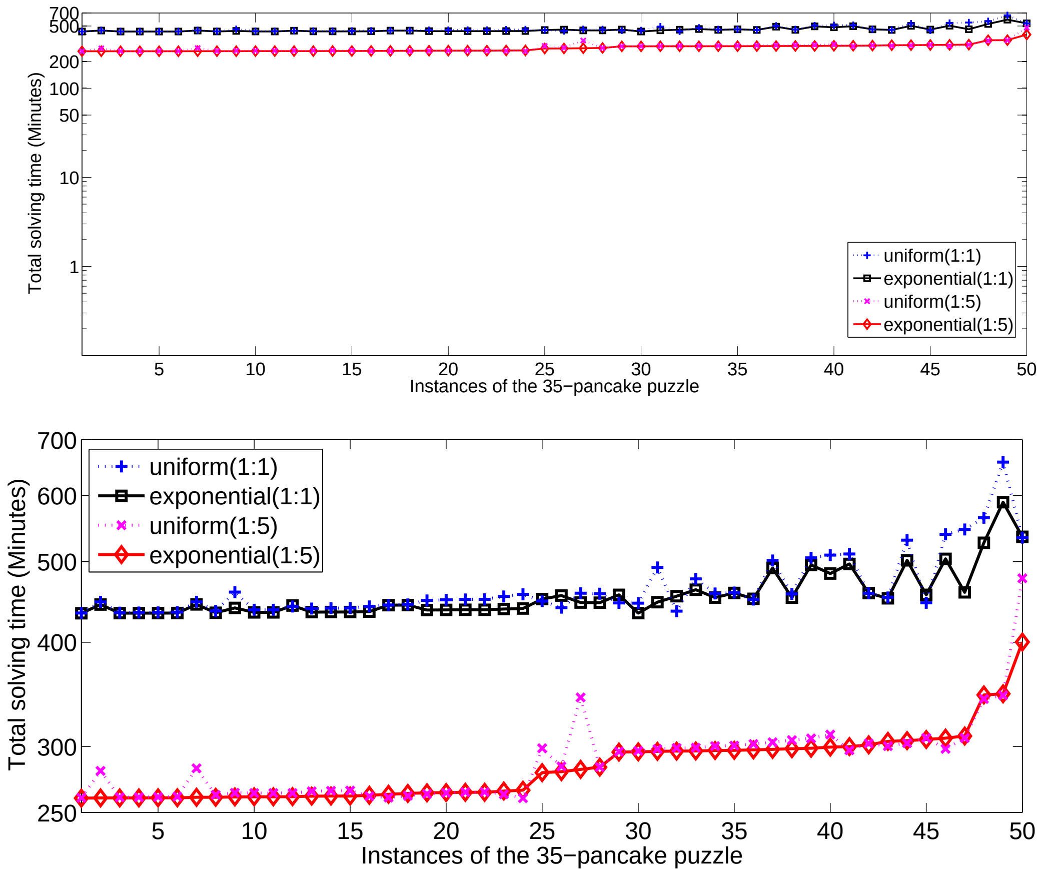


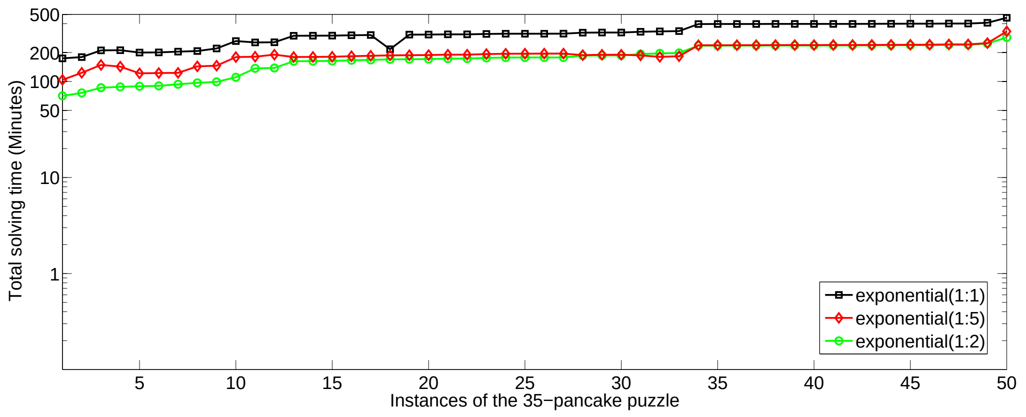
Figure 4.5: Total solving time for each instance of the 35-pancake puzzle - exponential and uniform strategies.

trends observed in this experiment were the same as those observed for the 24-puzzle. However, the two strategies for allocating time among the solvers seem to perform almost equally well here. The exponential strategy apparently has an advantage on the hardest instances in the test set, as the entries in the “max” column suggest.

row	ratio (t_s:t_l)	allocation	min	max	mean	med	std	Subopt
1	1:1	exponential	7h 13m	9h 48m	7h 36m	7h 28m	30m	7.9%
2	1:1	uniform	7h 14m	10h 59m	7h 45m	7h 30m	42m	8.0%
3	1:2	exponential	5h 24m	7h 28m	5h 45m	5h 32m	24m	8.0%
4	1:2	uniform	5h 24hm	7h 05m	5h 46m	5h 33m	24m	8.0%
5	1:5	exponential	4h 19m	6h 42m	4h 45m	4h 39m	28m	8.2%
6	1:5	uniform	4h 19m	7h 58m	4h 48m	4h 42m	36m	8.3%
7	1:10	exponential	3h 58m	7h 19m	4h 28m	4h 31m	43m	8.4%
8	1:10	uniform	4h 14m	7h 50m	4h 51m	4h 45m	37m	8.4%

Table 4.2: Statistics on solving a single instance of the 35-pancake puzzle.

The results in Section 3.3.2 suggested that using domain knowledge (*e.g.,* duality) will improve the results. Here, we repeated the previous experiment by adding the dual heuristic lookup. Figure 4.6 shows the total time to solve each instance using the exponential strategy for allocating time among the solvers with three different ratios of t_s:t_l (1:1, 1:2, and 1:5). The average solving time for the 50 test instances using the ratio of 1:2 is under 3 hours while the solutions are on average 11.9% (4 moves) longer comparing to the optimal ones. Unlike the previous experiments, changing the ratio from 1:2 to 1:5 increased the total solving time for almost all the instances. Here, the time to create each heuristic is rather short and the early heuristics created by the process are fairly accurate. Therefore, allocating more time to the solving process compared to the learning process, *e.g.,* using a ratio of 1:2 compared to a ratio of 1:5, leads to solving the problem instances faster for almost all the instances.



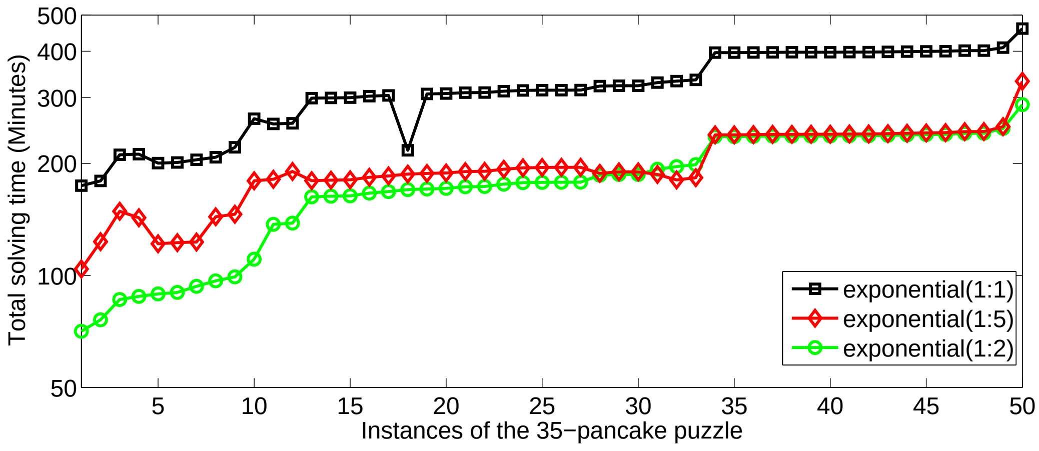


Figure 4.6: Total solving time for each instance of the 35-pancake puzzle using duality - exponential strategy.

Figure 4.7 compares the two different allocation strategies for fixed ratios of 1:1 and 1:2 when dual lookups are used. Similar to the 24-puzzle experiment, the exponential allocation outperforms the uniform allocation in almost all the test instances.

Table 4.3 summarizes the statistics for different allocation strategies and ratios used for the

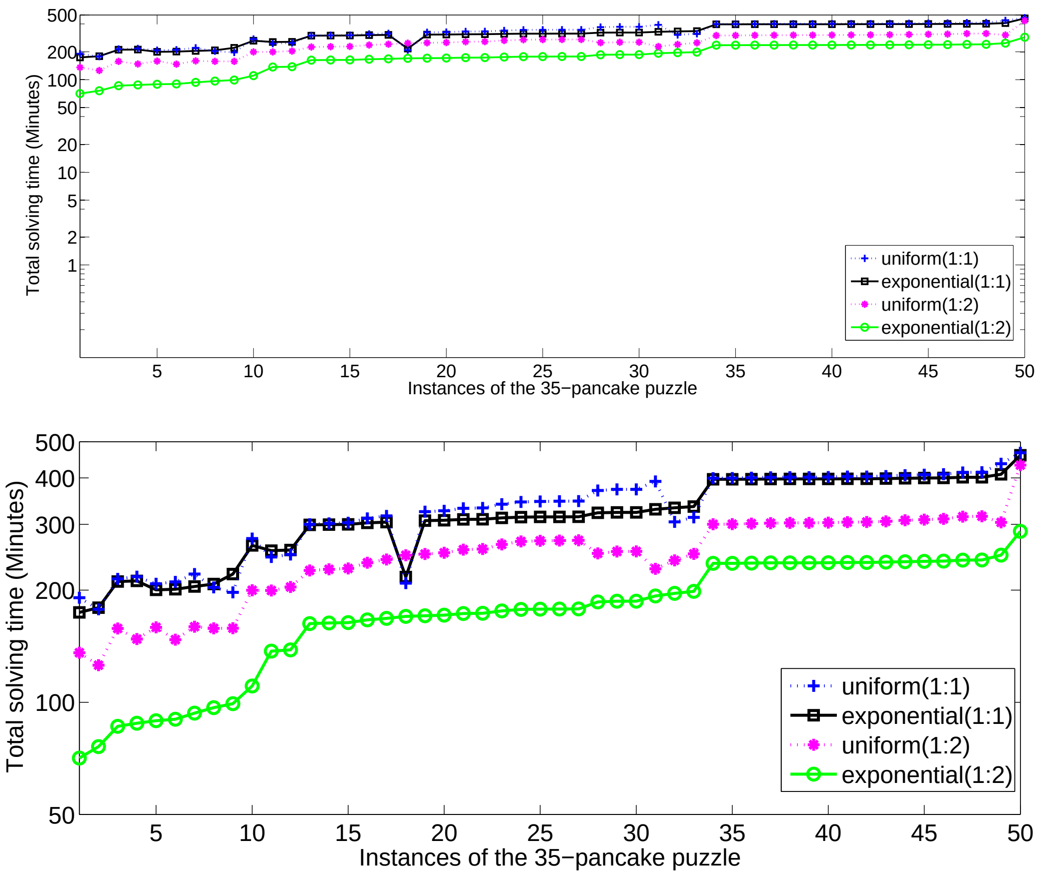


Figure 4.7: Total solving time for each instance of the 35-pancake puzzle using duality - exponential and uniform strategies.

35-pancake puzzle when duality is added. Note that although the total time changes substantially between different settings, the suboptimality remains almost constant. These results show that using duality substantially decreases the total time for all the instances at a small cost of degradation in solution quality.

Similar to the 24-puzzle, the average solving time shows a substantial improvement over the total time needed by the standard Bootstrap (see Table 3.13) while the solutions generated here are even closer to optimal.

4.3.3 Rubik's Cube

As in Section 3.3.3, each of the 10 instances of Rubik's Cube used by Korf [41] is used as the test instance. These instances have an average optimal solution length of 17.5. Figures 4.8 and 4.9 and Table 4.4 show the results of our interleaving approach for different ratios of t_s:t_l. Although these results show speedup over the total time of the Bootstrap (compare results in Table 4.4 to the total time of Bootstrap in Table 3.16), the time that the interleaving approach needs to solve a single instance of Rubik's Cube is still very long.

row	ratio (t_s:t_l)	allocation	min	max	mean	med	std	Subopt
1	1:1	exponential	2h 54m	7h 40m	5h 18m	5h 14m	1h 14m	12.0%
2	1:1	uniform	1h 58m	7h 47m	5h 30m	5h 42m	1h 19m	12.0%
3	1:2	exponential	1h 11m	4h 48m	3h 00m	2h 58m	56m	11.9%
4	1:2	uniform	2h 06m	7h 12m	4h 10m	4h 14m	1h 01m	11.8%
5	1:5	exponential	1h 44m	5h 32m	3h 17m	3h 10m	44m	11.7%
6	1:5	uniform	07m	7h 24m	3h 24m	3h 14m	1h 00m	11.8%
7	1:10	exponential	1h 48m	6h 30m	3h 06m	2h 52m	47m	11.8%
8	1:10	uniform	1h 48m	7h 48m	3h 33m	3h 30m	58m	11.8%

Table 4.3: Statistics on solving a single instance of the 35-pancake puzzle using duality.

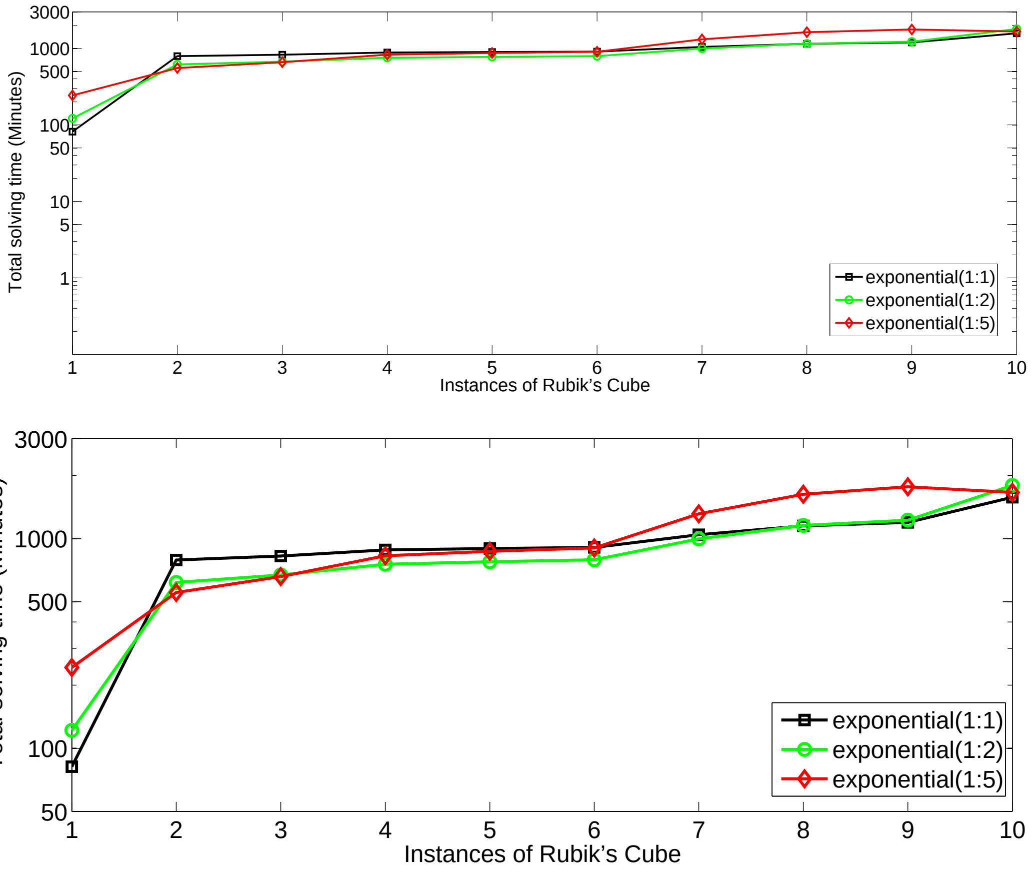
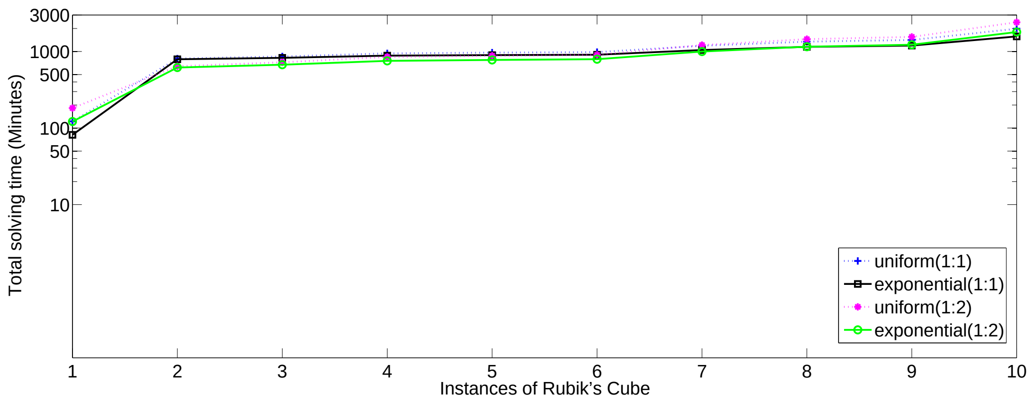


Figure 4.8: Total solving time for each instance of Rubik's Cube - exponential strategy.

Table 4.4 summarizes the results of solving a single instance of Rubik's Cube using our interleaving approach. The trends in these results are a bit different from the previous experiments. For example, for all different settings of allocation and ratio the solutions were only 5.1% (0.9 moves) longer than the optimal ones.

After the first two heuristics were learned from the random walk instances (the $lengthIncrement$ was set at 5 for this domain), the time to learn a new heuristic from random walk instances becomes



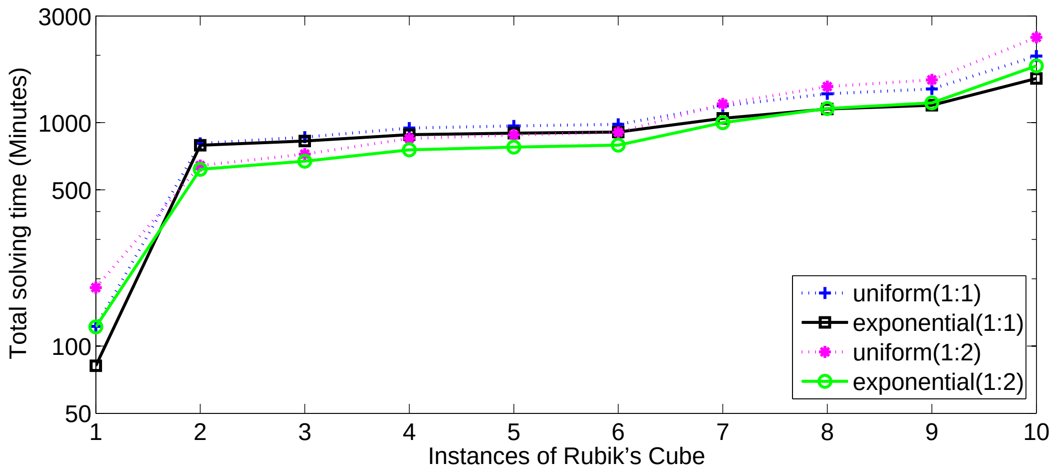


Figure 4.9: Total solving time for each instance of Rubik's Cube - exponential and uniform strategies.

so long that it is more efficient to spend time on solving the instance rather than learning a new heuristic. For example, it takes more than 6 hours to learn the heuristic for instances created by random walks of length 15 from the goal[2]. Therefore, it seems more reasonable just to use the heuristic learned from instances generated by random walks of length 10 to solve the instance. This long learning time causes all the instances to be solved with the same heuristic regardless of the settings used; therefore, the suboptimality is the same for all the settings.

4.3.4 20-blocks World

As in Section 3.3.4, each test instance is a random state created using the random state generator described by Slaney and Thiébaux [60]. For this experiment, 50 of them were used with an average optimal solution length of 30.92. Figures 4.10 and 4.11 and Table 4.5 show the experimental results of our interleaving approach with different settings for allocation and ratio on these 50 instances of the 20-blocks world. Figure 4.10 shows the distribution of the solving time for the 50 instances

[2]It took about two hours to solve 81 of the 200 random walk instances as the time limit increased from 1 to 64. Then, it took another 4 hours to solve 80 of the remaining 119 random walk instances.

row	ratio (t_s:t_l)	allocation	min	max	mean	med	std	Subopt
1	1:1	exponential	1h 22m	26h 16m	15h 36m	15h 2m	6h 20m	5.1%
2	1:1	uniform	2h 02m	33h 04m	17h 42m	16h 15m	8h 02m	5.1%
3	1:2	exponential	2h 02m	29h 54m	14h 51m	13h 04m	7h 23m	5.1%
4	1:2	uniform	3h 02m	40h 08m	18h 01m	14h 53m	10h 13m	5.1%
5	1:5	exponential	4h 03m	29h 29m	17h 23m	14h 45m	8h 42m	5.1%
6	1:5	uniform	6h 05m	48h 30m	20h 44m	17h 27m	13h 28m	5.1%
7	1:10	exponential	57m	47h 34m	21h 40m	20h 09m	13h 05m	5.1%
8	1:10	uniform	5h 35m	53h 46m	26h 50m	25h 02m	14h 57m	5.1%

Table 4.4: Statistics on solving a single instance of Rubik's Cube.

used for the experiment when the exponential interleaving approach is used with different ratios. Similar to the 24-puzzle experiment, increasing the time allocated to the learning process relative to the solving process decreases the total time for almost all the instances.

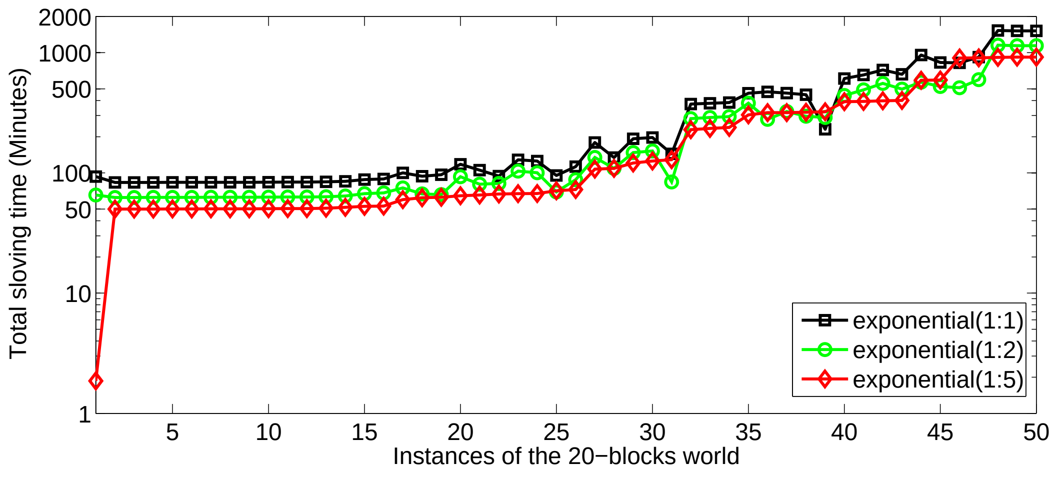


Figure 4.10: Total solving time for each instance of the 20-blocks world - exponential strategy.

Figure 4.11 shows that for a fixed ratio, exponential allocation of resources to the solving subthreads results in shorter solving time for almost all the instances compared to a uniform allocation. However, the two approaches are very similar in this domain.

Table 4.5 shows that in all different settings the solutions generated were only 1.2% (0.36 move) longer than the optimal ones. In addition, at least 37 of the 50 instances were solved optimally for each setting. In this experiment, our initial heuristic is so weak that it takes a few iterations of RandomWalk until the heuristic becomes strong enough that the solver using it can solve the instance in a reasonable amount of time. After this point, for a few iterations, the new heuristics learned solves the instance faster without changing the solution quality. This is the reason why we observed a constant suboptimality of 1.2% when the ratio is set to 1:1, 1:2, 1:5, and 1:10. As usual though, increasing the time allocated to the learning process relative to the time allocated to the

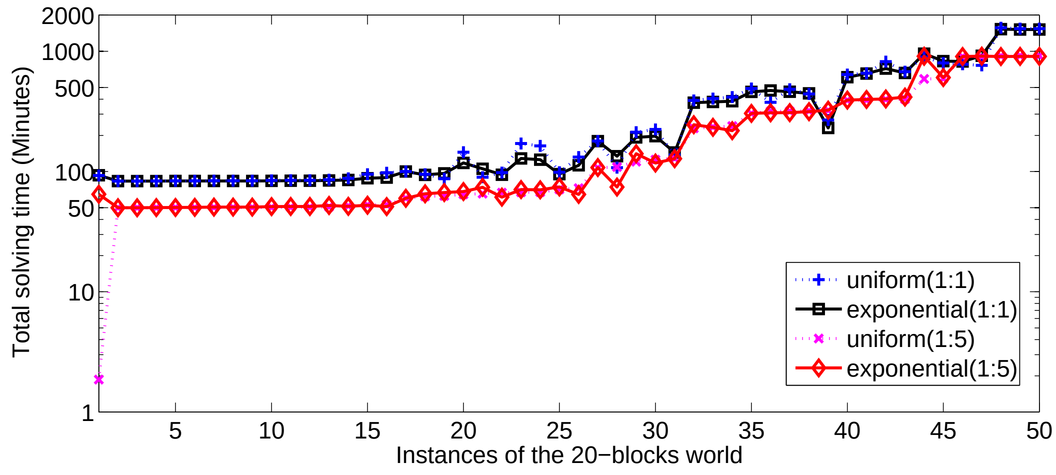


Figure 4.11: Total solving time for each instance of the 20-blocks world - exponential and uniform strategies.

solving process, reduces the average total time of solving the instances of 20-blocks world.

The speedup compared to the initial Bootstrap method (which needed 2 days when using 500 bootstrap instances and 11 days when using 5000 bootstrap instances) is again remarkable. In addition, the solution lengths are much closer to optimal than before (cf. Table 3.20 for Bootstrap results on the 20-blocks world).

row	ratio (t_s:t_l)	allocation	min	max	mean	med	std	Subopt
1	1:1	exponential	1h 24m	25h 30m	5h 42m	2h 06m	6h 34m	1.2%
2	1:1	uniform	1h 23m	26h 10m	5h 48m	2h 21m	6h 34m	1.2%
3	1:2	exponential	1h 02m	19h 12m	4h 10m	1h 36m	4h 45m	1.2%
4	1:2	uniform	1h 2hm	19h 30m	4h 12m	1h 38m	4h 47m	1.2%
5	1:5	exponential	02m	15h 18m	3h 52m	1h 12m	4h 30m	1.2%
6	1:5	uniform	50m	15h 12m	4h 00m	1h 14m	4h 41m	1.2%
7	1:10	exponential	46m	15h 01m	3h 46m	1h 41m	4h 23m	1.3%
8	1:10	uniform	46m	15h 25m	3h 50m	1h 41m	4h 32m	1.3%

Table 4.5: Statistics on solving a single instance of the 20-blocks world.

4.3.5 Blocksworld Instances of AIPS Planning Competition

We further tested our technique on the 35 instances of blocksworld domains of varying size that were used in the Track 1 of the AIPS planning competition in 2000. The features and the initial heuristic used for these problems are the same as those used for the 15- and 20-blocks world (see Section 3.3.4). Table 4.6 shows the results on the 20 instances of the set. The first column names the instances, where x-y refers to the y^{th} instance that consists of x blocks. The other columns show the total time (in seconds) and suboptimality achieved by the exponential interleaving with different ratios. In this table, "Time" entries with a check mark indicate that the total time was below 0.1 seconds; “Subopt” entries with a check mark indicate the corresponding instance was solved optimally. The 15 instances with the fewest blocks (between 4 and 8) are not shown; all were solved optimally by our system in less than one-tenth of a second total time. Table 4.6 shows that our interleaving method using ratios of 1:1, 1:2, and 1:5 is capable of solving all the instances in less than 30 minutes (the time limit that is used to solve an instance in the AIPS planning competition) while the solutions are very close to the optimal ones.

instance	Optimal	ratio (1:1)		ratio (1:2)		ratio (1:5)	
		Time	Subopt	Time	Subopt	Time	Subopt
9-0	30	✓	✓	✓	✓	✓	✓
9-1	28	✓	✓	✓	✓	✓	✓
9-2	26	✓	✓	✓	✓	✓	✓
10-0	34	22	✓	33	✓	65	✓
10-1	32	✓	✓	✓	✓	2	✓
10-2	34	13	✓	19	✓	38	✓
11-0	32	73	✓	58	✓	47	✓
11-1	30	58	✓	42	✓	46	✓
11-2	32	2	✓	3	✓	6	✓
12-0	34	12	✓	18	✓	37	✓
12-1	34	3	✓	4	✓	9	✓
13-0	42	1451	4.8%	1102	4.8%	914	4.8%
13-1	44	170	✓	1024	✓	861	✓
14-0	38	23	✓	51	✓	67	✓
14-1	36	62	✓	73	✓	176	✓
15-0	40	313	5%	310	5%	242	5%
15-1	52	627	✓	475	✓	393	✓
16-1	54	1347	✓	1271	✓	1105	✓
16-2	52	1001	✓	751	✓	603	✓
17-0	48	331	✓	258	✓	230	✓

Table 4.6: Blocksworld, results for exponential interleaving.

Yoon, Fern, and Givan [67] learned a heuristic and a policy to solve a subset of these blocksworld problem instances. They used the first 15 instances of the set, the ones not shown in Table 4.6, as training instances to learn the heuristic/policy. Their method is discussed in detail in Section 5.2.4.

When the learned policy is used to rank the states in a greedy best first search, their system solved all the 20 test instances in the time limit of 30 minutes. The solutions generated were on average 17% longer than optimal while the time to solve the instances was about 100 seconds, on average. When greedy best first search was used with the maximum of the learned heuristic and FF’s heuristic [31], their system solved all the test instances in the time limit of 30 minutes. The solutions generated were 120% longer than optimal and the system spent about 613 seconds on average to solve the instances. Our exponential interleaving with a ratio of 1:5 solved all the 20 instances on average in 242 seconds while the solutions generated were 0.5% longer than optimal.

We also compared the results of our interleaving method to some of the domain-independent planners[3] discussed in Section 4.5.2.[4]

FF [31] solved 29 of the 35 instances in the time limit of 30 minutes. The solutions generated for the solved instances were, on average, 2% (0.48 move) longer than optimal while it took less than 6 seconds on average for FF to solve the instances that it could solve in under 30 minutes. Our exponential interleaving approach using a ratio of 1:5 solved the same 29 instances optimally. It took about 25 seconds, on average, for the interleaving approach to solve these instances.

Fast Downward [27], when preferred operators are used, solved all the 35 instances in the time limit of 30 minutes. It took Fast Downward, on average, about a second to solve the instances while the solutions were, on average, 146% longer than optimal. Our interleaving approach using a ratio of 1:5 solved the same instances in about 138 seconds, on average, while the solutions generated were, on average, 0.4% longer than optimal.

Optimal planners have also been applied to these blocksworld problem instances. The admissible landmark heuristic introduced by Karpas and Domshlak [37] along with their variation of A* (see Section 4.5.2) solved 20 of the 35 test instances optimally within a time limit of 30 minutes. The average solving time on these 20 instances was 66 seconds. Our exponential interleaving approach when the ratio was set at 1:5 solved the same set of instances optimally in less than a second, on average.

4.4 Using One Solving Sub-thread

This section investigates the idea of not splitting up the solving thread into a set of sub-threads when the current set of heuristics has more than one element. In other words, we investigate what would happen if we allocate all the solving time to the most recently learned heuristic. In particular, whenever a new heuristic is created, we abort the current solver and start a new search using the latest heuristic created. For example, when the first heuristic is created by the learning process, the search with h_0 in the solving thread is stopped and restarted using the new heuristic.

The advantage of this strategy is that all the solving time is always spent on the heuristic that is expected to be strongest. On the other hand, this strategy denies the fact that the other searches, which have been aborted, might have been very close to finding a solution to the test instance.

We implemented this strategy on the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world. All other settings, *e.g.,* the features and test instances, are the same as those used in Section 4.3. Table 4.7 shows the results when ratios 1:1, 1:2, 1:5, and 1:10 are used. The definition of each column in the table is the same as for Tables 4.1–4.5.

These results are mostly very close to the results obtained when exponential time-allocation

[3]All these planners are domain-independent and they are not provided with any features/heuristics of the state space. Our system uses a hard-coded description of the domain and our weak features are provided by the user. Therefore, our results are not strictly comparable to these systems.

[4]Here, we took the results from previous papers. These results are obtained using a different computational setting than ours and are not strictly comparable to ours.

row	ratio (t_s:t_l)	min	max	mean	med	std	Subopt
24-puzzle							
1	1:1	20m 24m	44m 15m	23m 10s	21m 36s	3m 54s	6.4%
2	1:2	15m 36s	43m 01s	17m 48s	16m 12s	4m 11s	6.7%
3	1:5	12m 29s	34m 48s	15m 14s	12m 59s	4m 47s	6.6%
4	1:10	11m 26s	70m 54s	16m 06s	14m 18s	10m 8s	6.8%
35-pancake puzzle							
5	1:1	7h 13m	9h 36m	7h 32m	7h 20m	29m	7.9%
6	1:2	5h 24m	7h 12m	5h 42m	5h 30m	24m	8.0%
7	1:5	4h 20m	6h 44m	4h 42m	4h 40m	4h 28m	8.2%
8	1:10	3h 49m	7h 3m	4h 28m	4h 21m	43m	8.5%
35-pancake puzzle using duality							
9	1:1	2h 48m	7h 36m	5h 12m	5h 06m	1h 16m	12.0%
10	1:2	2h 08m	6h 09m	3h 54m	3h 48m	58m	11.9%
11	1:5	1h 42m	5h 24m	3h 12m	3h 4m	44m	11.7%
12	1:10	1h 48m	6h 26m	3h 07m	2h 48m	46m	12.0%
Rubik's Cube							
13	1:1	41m	19h 27m	13h 29	13h 50m	4h 54m	5.1%
14	1:2	31m	14h 35m	10h 06m	10h 22m	3h 41m	5.1%
15	1:5	25m	11h 40m	8h 05m	8h 18m	2h 57m	5.1%
16	1:10	22m	35h 48m	9h 55m	7h 36m	9h 24m	5.1%
20-blocks world							
17	1:1	45m	24h 39m	4h 51m	1h 16m	6h 18m	1.2%
18	1:2	34m	18h 29m	3h 40m	57m	4h 45m	1.2%
19	1:5	28m	14h 47m	3h 28m	46m	4h 30m	1.2%
20	1:10	25m	14h 28m	3h 23m	1h 18m	4h 22m	1.2%

Table 4.7: Statistics on solving a single instance of test problems.

strategy is used. The exception is Rubik's Cube; here, the average time to solve single instances of the problem is substantially smaller than when exponential strategy is used. As mentioned earlier, the time to create efficient heuristics for Rubik's Cube is so long that spending all the solving time for the latest heuristic is a better strategy than splitting the solving time to the set of current heuristics.

Similar to the results obtained in Section 4.3, changing the ratio from 1:5 to 1:10, *i.e.,* increasing the time allocated to the learning process in comparison to the time allocated to the solving process, mostly increases the total time of the process. Furthermore, using ratios 1:15 and 1:20, not included in the table, increased the total time on average for Rubik's Cube and the 35-pancake puzzle.

4.5 Related Work

This section briefly reviews related work that can be applied to solve a single instance of a search problem. In the heuristic search section (Section 4.5.1), we mainly discuss the possibility of using the available heuristic search methods for the single instance case. In the planning section (Section 4.5.2), we introduce planning systems that performed well in the AIPS planning competition[5].

4.5.1 Related Work in Heuristic Search

Different heuristic search approaches can be considered for quickly solving a single instance of a search problem. One approach is to find a good heuristic for the problem and solve the instance using the heuristic[6]. As discussed in Chapter 2, pattern databases are a general technique to create a heuristic by using abstraction. The quality of the heuristic depends on the granularity of the abstraction. A finer-grained abstraction will lead to a better heuristic compared to a less fine-grained abstraction, at the cost of increasing (i) the time needed to compute the heuristic and (ii) the memory required to store the heuristic value for the abstracted states.

In addition to the problems discussed in Chapter 2, finding the best granularity of abstraction is not trivial when the objective is to reduce the total time to build the pattern database and solve the instance. For example, the 7-8 additive pattern database [42] for the 15-puzzle needs about 3 hours to be computed [34]. On the other hand, we built three non-additive 6-tile pattern databases for the same problem in less than 6 minutes. Although using the 7-8 additive PDB makes the search with IDA* thousands of times faster compared to the three 6-tile non-additive PDBs, the time to build the 7-8 PDB is so large that using the 7-8 additive PDB is only efficient when more than 10,000 new instances of the 15-puzzle need to be solved.

Considering that only a small percentage of the entries in the pattern database will be used during the search for a single instance of the problem, Holte, Grajkoswki, and Tanner [34] and Larsen *et al.* [44] suggested hierarchical heuristic search algorithms that only compute the entries of the pattern database that are needed to solve the given problem instance.

In order to compute the heuristic value of each state, hierarchical heuristic search is used to find the exact distance between an abstraction of the state and an abstraction of the goal state. This process will be repeated. The search in each level is guided by the distances computed for an even more abstract level. Holte *et al.* [34] showed that for many single-agent search domains (*e.g.,* the 15-puzzle) this approach will improve the time to solve the given instance compared to making the entire high-performance pattern database (*e.g.,* the 7-8 additive PDB for the 15-puzzle). Larsen *et al.* [44] improved these results by using combination of forward and backward searches to avoid re-expanding the abstract states.

An alternative approach would be to use heuristic search algorithms that can deal with large state spaces. These algorithms find suboptimal solutions in order to speed up problem solving over algorithms that find the optimal solutions. Beam search is an important family of such algorithms while BULB [20] is the most promising one to consider as previous experimental results reported

[5]see `http://planning.cis.strath.ac.uk/competition/` for more details.

[6]Note that here we only focus on techniques that can be generally used for all the problem domains. Therefore, hand-crafted heuristics similar to the break heuristic for the pancake puzzle (that can solve single instance of large pancake puzzles very fast) are not considered.

by Furcy and Koenig [20] showed that it is very successful in finding solutions for large state spaces using weak heuristics (*e.g.,* the 48-puzzle using MD).

The problem with BULB is that it is not trivial to choose the beam width parameter in order to find solutions that are very close to the optimal ones. One solution would be to use an anytime version of BULB [20] that can either (i) continue searching the state space with the same parameters when a goal is found and hope that BULB finds solutions of shorter length or (ii) increase the beam width whenever it finds a solution with the current beam width.

In conclusion, our experiments with BULB and the results reported by Furcy and Koenig [20] show that although BULB can find solutions that are fairly close to the optimal ones very fast (*e.g.,* BULB can find solutions that are 15-20% longer than the optimal solution for the 24-puzzle in a few seconds), it is impractical to use it for cases in which even closer to optimal solutions are needed.

4.5.2 Related Work in Planning

The problem of solving a single instance of a planning problem has been widely studied in the planning literature. Here, we briefly introduce notable domain independent heuristic search planners that proved to be efficient in solving a single instance of the planning problem. In Section 4.3.5, we compared the results of our interleaving approach to some of these planners on the blocksworld domain.

The main ideas behind a heuristic search planner are [3]: 1) to automatically extract the heuristic from the planner's encoding of the search space (*e.g.,* the STRIPS encoding), and 2) to solve the planning problem as a heuristic search problem using the extracted heuristic and a heuristic search algorithm.

The idea of extracting a heuristic from the problem encoding was first suggested by McDermott [45]. His planner, called UNPOP, searches backwards from the goal state for a sequence of actions that reaches the initial state. The heuristic used for the search is computed from an AND/OR graph and estimates the distance of each state from the initial state. UNPOP considers the planning task as the conjunction of subgoals that all must be achieved. For example, each subgoal in the blocksworld is the correct position of a block. In the graph, AND branches are considering the conjunction of subgoals that all must be satisfied while OR branches are a sequence of actions that satisfies each subgoal. For each state, the heuristic is the sum over AND branches; each of them satisfying a subgoal of the problem. The value of each AND branch is the minimum over OR branches. Each of the OR branches is one possible plan to satisfy the subgoal. The leaf nodes in the graph are conditions that are true at the initial state. UNPOP uses hill-climbing[7] as the search method and whenever a new state is reached, a new graph will be made to compute the heuristic value of that state.

[7]Hill-climbing is a best first search algorithm. In hill-climbing, the best successor of the current state, *e.g.,* the one with the lowest heuristic value, will be selected for expansion. If more than one such successor exists, one of them is selected randomly. A local minimum occurs when the heuristic value of the current state is lower than the heuristic value of all its successors.

One general approach to extract a heuristic from the encoding is to solve a relaxed form of the problem and use the plan length of the relaxed problem as a heuristic estimate for the original problem. Bonet and Geffner's Heuristic Search Planner (HSP) [3] relaxes the planning problem by ignoring the delete list[8]. However, finding the actual cost of the relaxed problem is NP-hard [3]. Therefore, an estimate of this cost is computed. HSP computes the cost of achieving individual subgoals from the state s and combines them to form the heuristic. If the maximum of such costs is taken (h_{max}), the resulting heuristic will be admissible but will not be very informed. If the sum of these costs is taken (h_{add}), the heuristic will be more informed but (possibly) inadmissible (h_{add} is only admissible when the subgoals are independent). HSP then uses this heuristic along with a hill-climbing approach to search from the start state to the goal state. HSP performed very well in the AIPS planning competition in 1998 [3].

HSP using h_{add} is neither optimal nor complete. Therefore, in the AIPS planning competition in 2000 a variation of it (HSP2 [3]) that uses h_{add} with Weighted A* was used.

Computing the heuristic in HSP and HSP2 is very costly. For example, for the 15-puzzle neither of the planners generates more than one thousand nodes per second[9]. The reason for the low node generation rate in HSP and HSP2 is that the cost of satisfying each subgoal is computed from scratch in every new state [3].

HSPr [3] overcame this problem by searching the state space backwards (from the goal state to the initial state). The heuristic value for each state s estimates the cost of reaching the initial state from s and is computed as follows. For each atom in state s, the cost of reaching the atom from the initial state is computed. The costs for all the atoms in state s will be added together to compute the heuristic value for state s. Whenever the cost of reaching an atom from the initial state is computed, it is stored and will be used to compute the heuristic value of each state that contains that atom.

Hoffmann and Nebel [31] used ideas similar to HSP for their Fast Forward (FF) planner that led to huge success in the AIPS planning competition in 2000. FF is different from HSP in three important details [31]. First, FF finds an explicit solution for the relaxed problem and uses its length as the heuristic value for the original problem while HSP assumes that the relaxed problem can be divided into subgoals and solves each subgoal and combines the costs to compute a rough solution length for the relaxed problem. Note that the Relaxed Plan Length (RPL) heuristic that FF uses is not guaranteed to be a lower bound on the cost of the original problem, therefore, it is not admissible. Second, an enforced hill-climbing[10] strategy is used in FF as the search algorithm. Finally, FF uses a heuristic to prune the search space. For each state in the original problem, only those successors that were used to solve the relaxed problem will be considered (helpful actions).

[8]In STRIPS encoding, each action a is a triple a = (pre(a),add(a),del(a)). Pre, add, and del respectively correspond to action's precondition, the conditions that will be added and removed by applying the action.

[9]Bootstrap generates more than a million nodes per second for the 15-puzzle.

[10]Enforced hill-climbing is different from hill-climbing whenever a local minimum occurs. In the case of a local minimum, enforced hill-climbing performs a breadth first search from the current state until it finds a state that has a heuristic value lower than the heuristic value of the current state. Then, the search continues from this new state.

Helmert introduced the Fast Downward (FD) planner [27] which performed well in the AIPS planning competition in 2004. The FD planner has three steps to solve a problem. First, it translates STRIPS tasks to a representation in which the causal structure between subgoals can be better represented. Second, it extracts knowledge in the form of causal graphs from the new representation. Causal graphs represent the dependency between the state variables. The heuristic is created from the causal graphs. This heuristic is inadmissible as it ignores positive interaction between state variables. Finally, FD uses a greedy best first search algorithm along with the heuristic computed in the second step to solve the problem.

The search is enhanced with preferred operators and deferred evaluations. Preferred operators are conceptually similar to the helpful actions used in FF. They are defined as operators that are more promising during the search. FD uses two OPEN lists. One for the states that were reached by using preferred operators and the other one for other states reached during the best first search. The next state to be expanded will be alternately chosen from these OPEN lists. Unlike the helpful actions used in FF, preferred operators do not prune the state space; therefore, the search will remain complete.

Deferred evaluation means that the successors of a state will only be evaluated when they were selected for expansion. In other words, the successors will not be evaluated when they were inserted into the OPEN list (they will be added to the list using the heuristic value of their parent). As computing the heuristic for each state is very expensive, this technique can decrease the number of states evaluated (especially when combined with preferred operators) while increasing the number of states expanded [27].

Landmarks are an important concept in domain-independent planning, and were introduced by Hoffmann *et al.* [32]. According to Karpas and Domshlak [37], "a landmark is a fact that must be true at some point in every solution path". For example, consider an instance of the 3-blocks world in which blocks 1 and 2 are on the table and block 3 is stacked on top of the block 2. In the goal state, block 2 is stacked on top of block 1[11]. Therefore, block 2 must become clear at some point during any solution of the instance because only clear blocks are allowed to move based on the definition in the blocksworld domain. Therefore, $Clear(2)$ is a landmark for this problem instance. Every condition that must be true at the goal state is a landmark by definition. For example, $On(2, 1)$ which means that the block 2 should be on top of the block 1 in the goal state is a landmark for the previous example.

Hoffmann *et al.* [32] studied an algorithm to extract the landmarks from the relaxed plan graph and order them in a way that a solution can be reached by satisfying the landmarks in order. Considering the previous example, $Clear(2)$ should be ordered before $On(2, 1)$. Instead of directly finding a solution path from the start state to the goal state, a path will be found to satisfy each landmark. A planner (*e.g.,* FF planner [31]) will be used to find such a path. Although this method shows

[11]Here, the goal state is a set of states in which this condition holds. This example is taken from Richter *et al.* [51].

substantial speedup over the planner without landmarks (*e.g.,* the FF planner to directly search from the initial state to a goal state), it is not guaranteed to be optimal or complete.

Richter *et al.* [51] studied another way of finding and combining landmarks that (1) reduces the plan length and (2) decreases the chance of failure compared to Hoffmann *et al.*'s planner. Richter *et al.*'s planner, called LAMA, was the winner of the AIPS planning competition in 2008[12]. LAMA uses landmarks as a heuristic function. The heuristic value of each state s counts the number of landmarks that still need to be satisfied at state s. Note that this heuristic is inadmissible as an action may satisfy more than one landmark at one time.

Domain independent admissible heuristic functions have recently been studied in the planning community. Edelkamp [12] extended pattern databases [9] to create domain independent admissible heuristic for STRIPS planning. The abstraction technique used by Edelkamp is slightly different than the one defined by Culberson and Schaeffer [9]. To construct the pattern database, first groups of mutually exclusive atoms for the problem will be identified. A group of atoms is mutually exclusive when at each state of the search space, only one of the them is true. Then, some set of mutually exclusive groups will be selected and the atoms that do not belong to any of these selected groups will be ignored in the abstract search space.

Haslum and Geffner [24] created an admissible heuristic which is the generalization of the h_{max} heuristic of Bonet and Geffner [3]. This heuristic, which is called h^m, considers each possible subset of m subgoals that are not satisfied in the current state and must be satisfied in the goal state. The cost of the most costly subset to achieve is considered as the heuristic value for the current state. For example, consider a 3-blocks world instance in which blocks 1, 2, and 3 are on the table. The goal is to stack the blocks in one stack in which blocks 1 and 3 are respectively at the bottom and top of the stack. The value of the h^1 heuristic for this instance is 1, because each of the subgoals On(2,1) and On(3,2) can be achieved in one step. Based on the definition, h^1 is the same as h_{max} [3]. If h^2 is considered, the heuristic value of the state will be 2 because now both atoms On(2,1) and On(3,2) should be satisfied. As m increases the computational cost of computing h^m increases exponentially. Haslum and Geffner [24] used this heuristic with $m = 2$ and IDA* for their planner, which is called HSPr*. To avoid recomputing the heuristic in each state ideas similar to HSPr [3] are used. HSPr* resulted in successful results on the 8-puzzle and small blocksworld instances (at most 15-blocks world).

Landmarks have recently been considered by Helmert [29] and Karpas and Domshlak [37] to derive admissible heuristics for planning domains. Karpas and Domshlak [37] studied the idea of creating an admissible heuristic from the inadmissible landmark heuristic used in LAMA [51]. Here, each action that satisfies more than one landmark will share its cost among different landmarks that it satisfies. For example, if one action satisfies two landmarks, a cost of 0.5 will be assigned to each landmark. The admissible heuristic will then be made by adding the cost of landmarks (rather than

[12]See `http://ipc.informatik.uni-freiburg.de/` for more details.

counting the number of landmarks).

Karpas and Domshlak used this heuristic with a variation of A* to solve planning problems. In this variation, whenever a state is reached via a different path it should be re-evaluated as each path that reaches the same state can satisfy a different set of landmarks during the path; therefore, the state can have a different heuristic value. The experimental results of this variation of A* with the admissible heuristic showed that it is still not strong enough to solve most of the planning problems used in the AIPS planning competitions optimally in the time limit of 30 minutes.

Using machine learning to learn heuristics or other forms of the control knowledge has recently been widely considered in the planning community. We discuss two successful systems [16, 67] that used machine learning to solve planning problems in detail in Chapter 5.

4.6 Summary

In this chapter we investigated the question of whether a variation of our bootstrapping method can quickly solve a single instance of a given problem domain. Instead of minimizing the solving time at the expense of requiring very large learning times, as in the previous chapter, we looked for a balance in the learning and solving times so that the sum of the learning and solving times is made as small as can be.

We presented a method that involves interleaving the learning and solving processes. The only parameter for this method is the ratio of solving time to learning time. We allocated the solving time among the various solving sub-threads by strategies that we called "uniform" and "exponential". The experiments with both versions of our interleaving approach were performed on the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, the 20-blocks world and IPC2 blocksworld instances, when the ratio was set to 1:1, 1:2, 1:5, and 1:10.

Our experimental results showed that in all cases the time to solve the single instance in these domains is substantially smaller than the total time needed by our Bootstrap method to learn its final heuristic. Therefore, it is experimentally shown that even in the single instance case, bootstrapping can be very effective.

Chapter 5

Related Work

This chapter briefly surveys previous work that used machine learning to speed up search programs. We categorize the previous work based on the strategy used for speeding up problem solving. For instance, if an iterative process is used to improve an initial heuristic function, then the system is discussed in the bootstrap learning section. We also survey methods that use random walks to generate successively more difficult training instances (see Section 5.1.4).

5.1 Bootstrap Learning

Bootstrap learning (bootstrapping, for short) is a technique to iteratively improve the performance of a learning algorithm through training. The idea of bootstrapping to speedup search programs was first studied by Rendell [49, 50].

5.1.1 Penetrance Learning System

Rendell's Penetrance Learning System (PLS) starts with no heuristic and performs a breadth first search producing statistics to form an evaluation function for a unidirectional search. The evaluation function, called the penetrance, measures the probability that a given state is on the solution path to the goal. Computing this evaluation function for all the states of the problem is infeasible as it needs a breadth first search tree of all the possible instances of the problem. Instead, Rendell used a bootstrap learning technique to estimate the value of this function.

Performing the breadth first search until a certain number of nodes are expanded (e.g., 1500 nodes in 15-puzzle), PLS maps all the states visited during the search to a feature space. For example, if two features f_1 and f_2 are used, each state s visited during the search is mapped to the feature space based on the value of $f_1(s)$ and $f_2(s)$. This feature space can be represented by a rectangle each side of which corresponds to one of the features. The length of each side of the rectangle is equal to the maximum value of the feature corresponding to that side. Then, the feature space is partitioned into rectangular regions by considering the splits that can be applied in the integer values of the features. The penetrance of each region is estimated as the number of states in the region that are on the solution path to the goal divided by the total number of states in the region. An evaluation function is made by training a linear regressor over the feature space using the data collected from the regions.

The evaluation function will then be used to solve the new set of training (bootstrap) instances and this process will be repeated until the residual error of the linear combination of features converges. The final evaluation function of PLS uses the data collected from all iterations of the process.

PLS was the first system to solve all 50 random instances of the 15-puzzle suboptimally (1976). Six features were used in the feature vector; all of them considering the basic characteristics of the problem space. For example, one feature counts the number of rows/columns in which all the tiles belong to that row/column. The total number of nodes generated on 50 random instances of the 15-puzzle was less than 1000 while the solutions generated were more than 2 times longer than optimal (the average length of solutions found by PLS was 113).

The main difference between our system and PLS is in the set of bootstrap instances. PLS needs the user to provide at least a solvable bootstrap instance using the current evaluation function for each iteration; however, our system fills the gap between the difficulty of bootstrap instances using either an increment in the time limit or a set of its own instances at the right level of difficulty (random walk instances). Furthermore, the penetrance that PLS learns is not exactly a heuristic and it is only used to rank the states during a best first search.

5.1.2 Learning an Admissible Heuristic from an Inadmissible Heuristic

Bramanti-Gregor and Davis [5] used a similar bootstrapping approach to learn an admissible heuristic from a non-admissible one. First, they compute a non-decreasing function, called $MAXH(x)$, that returns the maximum value of the inadmissible heuristic, on the states whose actual distance to the goal state is at most x. Then the admissible heuristic h_M will be computed from $MAXH$. For each state s, $h_M(s)$ is the minimum value of x in which the inadmissible heuristic value of state s is less than or equal to $MAXH(x)$. In other words, $MAXH$ projects the true distance to the goal to a space in which the distance to the goal is overestimated; then h_M projects the new overestimated distance back to the true distance space.

$MAXH(x)$ cannot be computed directly as it requires the optimal solution length for all the states in the search space. Therefore, an estimate of it will be learned using an A* search on a set of training (bootstrap) instances. The process starts with setting $MAXH(x)$ to zero for all values of x. A* will be run on each bootstrap instance and whenever a node n is selected from the OPEN list, the value of $MAXH$ will be updated. This estimate will then be used to compute an estimate of h_M. This process will be repeated iteratively on a new set of bootstrap instances to update the estimate for $MAXH$ and h_M. The process is terminated manually by the user.

The goal of this system is to learn an admissible heuristic that contains the insight of the inadmissible heuristic and thus can reduce the search effort while finding (near-)optimal solutions.

The experimental results of the system are on the 8-puzzle. The inadmissible heuristic is the sum of MD and the sequence score[1]. The admissible heuristic learned from this heuristic after 8 iterations generated 15% fewer nodes than MD still while finding the optimal solutions. A similar system when manually stopped after the first iteration led to about 50% improvement in node generation while the solutions were on average 6% longer than optimal.

5.1.3 Bootstrap Learning to Solve a Single Instance of a Problem

Humphrey, Bramanti-Gregor, and Davis [35] studied another bootstrapping approach to solve a single instance of a search problem. Their algorithm, SACH, improves the heuristic from failed search attempts on the single instance. In other words, SACH iteratively learns a heuristic to solve a single instance of the problem.

SACH starts with no heuristic and expands the search tree until the number of nodes expanded using A* reaches a constant bound ($10,000$ nodes expanded for the 15-puzzle). SACH then uses a user defined feature vector and linear regression to learn a better heuristic (features used for the 15-puzzle are tiles-out-of-place, MD, and reversal count[2]) based on the data collected from states visited during the A* search.

Repeating this procedure, SACH improves the heuristic gradually. Failing to solve the instance after a maximum number of attempts (20 attempts for the 15-puzzle), SACH multiplies the heuristic by a weight and repeats the process. The process stops whenever a solution to the instance is found or when the process has been repeated for a maximum number of weight increases (20 for the 15-puzzle). Here failure in each attempt, will lead to a constant increment of the weight (0.1 for the 15-puzzle).

SACH successfully solved the 15 hardest of the 100 standard test instances of the 15-puzzle [39], the ones with the longest optimal solutions. The total number of nodes generated over all iterations of SACH was $724,032$ which is more than 2000 times less than the number of nodes that IDA* using MD will generate [39]. The generated solutions are 2% longer on average than optimal.

SACH can be extended so that the heuristic is learned from failed attempts on multiple instances of the problem while the goal is to solve all those instances. Bramanti-Gregor and Davis [6] reported the results of such a system on the 20 easiest of the 100 standard instances of the 15-puzzle [39], the ones with the shortest optimal solutions. The total number of nodes generated over all the iterations of this system was $160,742$ which is about 100 times less than the number of nodes that IDA* using MD generates on the same set of test instances [39]. The solutions generated are on average 1% longer than optimal.

[1]Sequence score checks the non-central tiles in turn, and adds 2 to the value of the heuristic for each tile not followed by its proper successor. A non-blank tile in the center adds one to the value of the heuristic. This can be understood better knowing that this heuristic is made for the case that in the goal state the blank is in the central position, *i.e.,* the goal state for the 8-puzzle is (1 2 3 8 blank 4 7 6 5) in which 1 is the tile in the top left position of the grid and 5 in the bottom right position of the grid.

[2]Reversal count adds one to the value of the heuristic whenever two adjacent tiles are interchanged from the goal position.

5.1.4 Generating Training Instances Using Random Walks

In all the systems discussed so far, the user provides the bootstrap instances. However, some systems like our RandomWalk method, generate their own bootstrap instances. The idea of creating instances in an increasing order of difficulty for single-agent search problems was first studied by Finkelstein and Markovitch [18].

Finkelstein and Markovitch [18] used the idea of creating instances with increasing level of difficulty using random walks backwards from the goal to learn macro-operators[3] for the sliding-tile puzzle. For example, they created instances using 100 random moves backwards from the goal for the 15-puzzle. To make harder instances, they increase the length of the random walk by 100 each time. The main difference between this approach and ours is that their initial random walk length (to produce easy instances) and the increment (to produce more difficult instances) is user specified.

Fern, Yoon, and Givan [16] used random walks of increasing length to learn a control policy for planning problems. The goal of the system is to learn an efficient policy that can solve planning problems with high success rate and low average length of solutions.

Their system starts with an initial policy and easy training instances. For example, it starts with a random policy and training instances that are created by random walks of length one. Then, it improves the policy from the training instances until the success ratio[4] of the policy exceeds some success threshold. The problems then become harder by increasing the length of the random walk. The process stops when the length of random walks exceeds a threshold or when the policy's progress stops.

Two key differences exist between this system and our method of creating instances in an increasing order of difficulty (random walk instances). First, in this system, random walks instances are generated by random walks forward from the initial state. Second, the choice of the initial random walk length and the increment are user specified.

5.2 One-step learning

In single-agent search, the basic idea behind a one-step learner of heuristics is: a learning system is trained on the set of states whose optimal distance-to-goal is known. Then the learned heuristic will be used to guide the search. Although the idea is very simple, it cannot be applied to large search spaces (*e.g.,* the 24-puzzle) as collecting sufficient training data is infeasible for such spaces.

[3]A macro-operator is a sequence of original operators for a problem. For example, the sequence {left, up, right} can be considered as a macro-operator for the sliding-tile puzzle. Whenever this macro-operator is used the blank tile moves to left, then up, and finally right.

[4]The success ratio of a policy π on a set of problem instances T is a real number between 0 and 1 that determines the ratio of the number of instances in T that can be solved using the current policy π in a time limit $t_{\max}$, to the size of T. If all the instances in T are solved then success ratio is 1; if none of them is solved, then the success ratio is 0.

5.2.1 Learning Heuristic by Partitioning the Search Space

One-step learners typically assume that the initial heuristic is sufficiently strong that arbitrary instances of the problem (the training set) can quickly be solved using it. Then learning is used to create a heuristic that is even stronger to allow instances to be solved more quickly although with some amount of suboptimality.

The simplest one-step learner of heuristics was studied by Sarkar, Ghose, and Chakrabarti [59]. They partitioned the problem space using a set of features. This partitioning is very similar to the rectangular partitioning discussed in Section 5.1.1. Each state in the problem space is assigned to one of the partitions based on the value of the features of the state.

Similar to SACH, described in Section 5.1.3, considering the optimal distance of all the states in each partition from the goal state, the minimum of these distances is an admissible heuristic value for all the states that belong to the partition. Computing this heuristic requires the optimal solution length for all the states in the search space which is impractical for larger domains (*e.g.,* the 24-puzzle). Therefore, an estimate of this value is learned for each partition by sampling. A set of training instances is created randomly and solved using A*. Then the optimal distance of each training instance from the goal is placed in the partition that the instance belongs to. The minimum number in each partition is used as the heuristic value for all the states belong to that partition. The computed heuristic may be inadmissible as the minimum value for each partition is not computed based on the all the states in that partition.

Sarkar, Chakrabarti, and Ghose [58] revised this approach in a way that the distribution of the distance to the goal of the states in each partition can be learned. Their experimental results for the 8-puzzle showed that A* using MD, sequence score, and number of out-of-place tiles as features reduces the number of nodes expanded by a factor of more than two compared to A* using MD, while preserving the optimality of the solutions.

5.2.2 One-step Learning of Heuristic Functions Using Neural Networks

Ernandes and Gori [13] studied a one-step learning algorithm to find near-optimal solutions for Manhattan space problems (*e.g.,* the 15-puzzle). They considered the probability that a heuristic h underestimates the distance to the goal for each state as a constant; then used this estimate independently for all the states in the region on the solution path to the goal to reach a probability that an A*-style search guided by the heuristic h will optimally solve an arbitrary instance of the problem.

Different neural networks were learned with a varying probability of underestimating the heuristic value of each state. Each training example fed to the neural network is a pair. The first element of the pair is a set of feature values for a state. The second element is the optimal distance-to-goal for the state. Finally, the neural networks were combined by taking the minimum output of different neural networks for each state to enforce an optimality bound on the solutions.

Ernandes and Gori [13] reported experimental results on the 15-puzzle. They used a feature vector that encodes the 16 inputs (15 numbered tiles and the blank tile, which is represented by zero) by a vector of size 16^2 bits where the bit $16 \times k + t$ is 1 if the tile k is occupied by number t, and 0 in all other cases. The training data were randomly generated instances of the 15-puzzle. A set of correction techniques added to MD was used as the initial heuristic to optimally solve the training instances. The final heuristic was the maximum of this heuristic and the heuristic learned from the training data. They used 20,000 instances for training. The average optimal solution cost of the training instances used was no more than half the average optimal solution cost of the problem (about 26.5). The generation of training data took 100 hours.

They changed the standard error function of backpropagation (MSE) to an asymmetric error function that dynamically changes during learning to stress target underestimation. Furthermore, either the target function or the gap target function[5] was learned. They also tuned many different parameters of the neural network including the number of hidden units and the learning rate. The learning phase for the 15-puzzle took 200 hours because of the extensive amount of tuning.

The heuristic was tested on 700 randomly generated instances of the 15-puzzle with an average solution length of 52.62. The best results, considering the number of nodes generated, solved problems very close to optimal (the average solution length was 54.45) while generating only 24,711 nodes on average. However, the encoding used for the feature vector (16^2 inputs for the neural network to compute the heuristic value of each state) led to a very poor performance of their system in terms of CPU time; 7.38 seconds average solving time for the 15-puzzle when IDA* is used as the search algorithm.

Samadi, Felner, and Schaeffer [55] used ideas similar to Ernandes and Gori [13] in their one-step learner of the heuristics. They developed two similar systems, called ANN and PEANN. ANN is a neural network similar to the one used in our experiments while PEANN uses a modified error function in the neural network to penalize the overestimation. For small search spaces (*e.g.,* the 15-puzzle) their systems are very similar to Ernandes and Gori's [13]. The main differences are: 1) using a much stronger initial heuristic (7-8 additive pdbs [42]), and 2) using heuristics as features in the feature vector.

For the 15-puzzle, Samadi *et al.* [55] used 10,000 training instances, created by backward random walks from the goal, that were solved optimally to train the neural network. They also tuned different parameters while training the neural network. They improved the results reported by Ernandes and Gori [13] substantially in both suboptimality and search effort by using stronger heuristics in the feature vector. Their experimental results for the 15-puzzle are available in Section 3.3.1.

As mentioned earlier, a one-step learning system cannot be applied to larger problems (*e.g.,* the 24-puzzle) as creating sufficiently many training data requires optimally solving random instances of the problem which is infeasible. Samadi *et al.* [55] manually solved this problem using abstraction.

[5]The target function estimates the distance of the state from the goal state. On the other hand, the gap target function estimates the difference between the distance of the state from the goal state and the initial heuristic (MD augmented with linear conflicts in this case) value of the state.

Instead of directly learning a heuristic for the 24-puzzle, they learned heuristics for two disjoint 11-tile abstractions of the 24-puzzle and added the value of these heuristics to the value of the remaining 2-tile abstraction of the problem. PEANN solved random instances of the 24-puzzle using RBFS generating less than 120 million nodes while the solutions generated were less than 0.3% longer than the optimal ones. Although these results are very impressive, critical choices for selecting efficient abstractions of the problem and deciding how to combine them to obtain these results are made manually. Moreover, as the approach uses abstraction, it inherits the limitations of abstraction.

5.2.3 One-step Learning of Heuristic Functions Using Linear Regression

Bramanti-Gregor and Davis [6] studied a technique that learns a linear combination of input features as a heuristic. Their approach differs from the above approaches in two details. First, instead of optimally solving the training instance for the learning system, they used the data from partial A* searches, considering that whenever A* selects a node for expansion, the path to that node is optimal if A* uses an admissible and consistent heuristic. Second, they extended the feature vector by adding transformations (*e.g.,* square or square root) and cross products of the original features as new features. The shortcoming of this approach is that for large problems (*e.g.,* the 24-puzzle even using the best available admissible and consistent heuristic for the problem [42]) the A* search will run out of memory before collecting sufficient training data of different levels of difficulty.

They used MD, number of out-of-place tiles, and reversal count along with their transformations as features. Their experimental results show that A* solved random instances of the 8-puzzle expanding 150 nodes. The solutions were on average 1% longer than the optimal ones. A* using MD expands about 625 nodes on average on the same set of test instances.

5.2.4 Learning Heuristic Functions for Planning Problems

Yoon, Fern, and Givan [67] applied similar ideas to planning domains. Their system is different from the above systems as their training data is collected from suboptimally solved instances of each domain thus biasing the learned heuristic towards inadmissibility. Furthermore, their approach is automatic and domain-independent as the features of each domain are extracted using an automatic, domain-independent technique.

The initial heuristic of the system is the relaxed plan length computed by FF [31]. This heuristic is used to solve a few instances of the problem appearing in the beginning of the set (*e.g.,* the first 15 instances of each domain) without a time limit. They assumed that such instances correspond to easy instances of the problem. Furthermore, it is assumed that the suboptimal solutions obtained by using the initial heuristic to collect the training data are reasonably good. None of these assumptions necessarily hold in general.

The data collected from the solved instances are used to learn a weight vector over the linear combination of the features. They learned the gap target function that estimates the difference between the distance of the state from the goal state and FF's heuristic value for the state. Finally, they used a greedy best first search algorithm with the learned heuristic to solve the test instances. The experimental results of this system on the blocksworld domain was described in Section 4.3.5.

5.3 Genetic Programming

Genetic Programming (GP) [43] is an evolutionary methodology related to genetic algorithms [33]. The main difference between GP and genetic algorithms is that in GP the representations that are changed using crossover and mutation are computer programs rather than bit strings. If the program represents a heuristic function, GP is similar to bootstrapping as it iteratively improves/evolves an initial heuristic.

Hauptman *et al.* [25, 26] used GP to learn heuristic functions for the Rush Hour puzzle and FreeCell. Their system uses some (weak) heuristics of the domain along with other domain specific features to represent a state. For each problem domain a set of different heuristic functions was learned; each of them is used only in parts of the search space.

The value of each heuristic and the condition regarding when to apply each heuristic are represented as trees and called value and condition trees respectively. In both trees, the features representing the state are the leaf nodes. The combine the features, the functions $\{\times, +\}$ were used in value trees and functions $\{AND, OR, \leq, \geq\}$ were used in condition trees.

The problem instances were divided into easy and hard instances based on the number of nodes that IDA* without a heuristic generates to solve them. The easy ones were used for training while the hard ones were kept for testing. The training data in this system is not of the form of set of states whose distance to goal is known. In fact, the distance to goal for each training instance is not considered at all in this system. The system uses GP to iteratively improve the combination of features, which forms the heuristic, with the objective of reducing the number of nodes generated on training instances.

The system starts with random value and condition trees and uses GP to iteratively evolve them using crossover and mutation operators. The fitness score assigned to each individual is equal to the percentage of nodes reduction when IDA* with the learned heuristic is compared to search with no heuristic on 10 random training instances. This iterative process stops after a fixed number of iterations. All the numeric parameters of the system including the population size, reproduction probability, crossover and mutation probability, and the depth of each tree were set manually.

The main difference between this system and ours is the number of heuristics used to solve each test problem. Furthermore, this system assumes that the training instances are easy enough that IDA* using the initial heuristic or search without a heuristic can solve them in a reasonable amount of time.

Their experimental results on Rush Hour showed that the heuristics learned by genetic programming reduce the search by a factor of at least 2.5 over search without a heuristic for Rush Hour.

The heuristics learned for FreeCell also speeded up search over a hybrid A*/hill-climbing search algorithm. For FreeCell, the solutions found by GP are 30% shorter than those found by the hybrid algorithm while no such comparison is reported for Rush Hour. Furthermore, no time is reported for learning the heuristic.

5.4 Online Learning

In an online learning framework the label for each instance is predicted based on the feedback received when the true label of a previously predicted instance is discovered. In the context of learning heuristics, the true label can be considered as the optimal distance of a state from the goal state while the feedback is the difference between the distance that the learner predicts and the true distance.

Fink [17] studied an online automated mechanism for learning heuristic functions. In Fink's system, the heuristic function is a vector of importance weights over a set of features. The features were created by abstraction and the weights were learned using linear regression.

Although the feedback for all the states visited during the search can be considered (as all the optimal solution lengths for all the states visited during the A* search are available), the update rule only updates the weights based on the feedback from the state in which the current heuristic deviates the most from the optimal solution length.

A* is used to solve the testing instances. The experimental results are on very easy instances of small problems (*e.g.,* when the average solution length of solved instances is 4.22 in the 8-puzzle) and the method improves the performance of A* without a heuristic (the Dijkstra algorithm) by a factor of 10 while finding optimal solutions. Although this paper theoretically analyzes the conditions under which finding the optimal solution path is guaranteed, it is difficult to generalize the algorithm to a more tractable version that is feasible for larger problem spaces.

5.5 Linear Programming

Linear programming [65] is a technique to solve an optimization problem in which the solution to the optimization problem is constrained by a set of linear equations and the objective is to minimize a linear function. For example, the equation below shows a linear program in which x is the vector of variables (to be calculated by the optimization) while A, B, and C are known. The objective function is $C^T x$ and $Ax \leq B$ is the set of constraint equations. Different techniques can be used to solve a linear program; however, the discussion of them is outside the scope of this thesis.

$$\min_x C^T x$$
$$\text{subject to } Ax \leq B$$

Petrik and Zilberstein [47] used linear programming to learn an admissible heuristic function for planning domains. The objective of this work is not to improve the state-of-the-art heuristics for each domain, but to provide a general framework for creating domain-independent admissible heuristics. The optimization problem[6] (learning an admissible heuristic) is modeled as a minimization problem in which the goal is to minimize the cost of a goal-terminating path for each state s.

The training data for this system is collected from the states on optimal solution paths. Therefore, similar to the one-step learning methods discussed in Section 5.2, this system can only be applied to search spaces in which the training instances can be solved optimally. The training instances are created by random walks of a specific length backward from the goal state (*e.g.,* random walks of length 20 for the 8-puzzle).

A set of features is used to represent each state and the features are combined linearly with a set of weights to form the heuristic. In the above equations, C and x respectively represent features and the weights (that should be learned). The optimization problem has two sets of constraints to combine the features. The first set ensures the admissibility of the heuristic function by bounding the heuristic value of each state by the minimum cost of all goal-terminating paths from that state that were observed during the training. The second set of constraints is to enforce the consistency of the heuristic, *i.e.,* for each two consecutive states visited, the difference between their heuristic value must be less than the cost of moving from the first state to the second state.

To guarantee admissibility, the set of constraints should consider all the states in the problem space, which is infeasible even for a search space the size of the 15-puzzle. Petrik and Zilberstein showed that admissibility can also be guaranteed for a sample of fixed size; however, the upper bound on the heuristic error (the difference between the heuristic value of each state and optimal distance of the state from the goal state) becomes larger.

Their experimental results are for the 8-puzzle. They showed that when Manhattan Distance of each tile including the blank is given as features, the system is able to recover the Manhattan Distance heuristic by computing the sum of features corresponding to non-blank tiles. A similar experiment with the sequence score showed that the sequence score is an inadmissible heuristic and will be converted to an admissible one when divided by four.

5.6 Learning Other Forms of Control Knowledge

This section briefly introduces research on learning other forms of control knowledge.

5.6.1 Macro-Learning

A macro(-operator) is a sequence of original operators of a problem. Macro-learning studies different strategies to learn macros for a problem. Finkelstein and Markovitch [18] suggested an

[6]In the original paper [47], Petrik and Zilberstein modeled the problem as a maximization problem to be consistent with the Reinforcement Leaning [62] literature. They defined reward for each state s as the negative of the cost of reaching s from the initial state. As rewards are negative along the path from each state to the goal state, the optimization problem maximizes the cumulative reward achieved from each state. Here, we present their work as a minimization problem, *i.e.,* with costs not rewards, to be consistent with other descriptions in this thesis.

algorithm to learn macros for many different search domains including the (N^2-1)-puzzle when $N \in \{4, 5, ..., 10\}$. Their algorithm, called MICRO-HILLARY, first produces solvable training instances using random walks. Then it runs a hill-climbing algorithm in which the heuristic value of the state is used as the evaluation function[7]. To escape from local minima[8], an iterative limited breadth first search method is applied to search the areas near the local minimum. The sequence of operators that leads from a local minimum to a non-local minimum state is stored as a macro. The algorithm stops at a quiescence point in which no new macros can be learned from the training instances.

Instead of directly applying the same procedure to solve very large problems (*e.g.,* the 99-puzzle), an iterative approach is studied that uses the macros learned from smaller problems as basic operators for the larger problem. For example, the macros learned for the 8- and 15-puzzles were used as basic operators for the 24-puzzle. This idea led MICRO-HILLARY to solve random instances of the 99-puzzle with an average solution length of about 3000 moves. Their experiments for the 99-puzzle showed that the process reached quiescence while learning macros for the 35-puzzle and most of the final macros were learned from the 8- and 15-puzzles. The main drawback of macro-learning is that the solutions generated by macro-learning are usually much longer than optimal.

Minton [46] studied a macro-learning system, called MORRIS, to solve problems similar to planning. This work focuses on selective macro-learning, *i.e.,* learning the most useful macros for the problem. MORRIS extracts two sets of macros from the solution to the training examples, scripts (S-Macros) and tricks (T-Macros). Each of the S-Macros is a sequence of operators that is common in the solution of training examples. Whenever a new training example is solved, a new set of macros will be added to the set. There exists a limit on the number of S-Macros; when the limit is reached, the least commonly used macros (considering the solution to the training examples) will be removed from the set.

The T-Macros are similar to macros used by MICRO-HILLARY [18]. These macros are used when progress is not made when the search algorithm traverses a path in the search graph, *e.g.,* when the heuristic values of the states along the path are not decreasing.

MORRIS uses a best first search algorithm to solve a set of robot world problems. It is empirically shown that MORRIS improves the performance over (1) a similar system that considers all potential macros learned during the training, and (2) a system that does not use macros at all.

Botea, Müller, and Schaeffer [4] used macro-learning for planning. Their method is consisted of three steps. First, it extracts macros from a set of training examples. Then, the extracted macros are filtered with the objective of reducing the overhead of considering a large number of macros for each state during the search. Finally, the macros and the original operators are used in a search algorithm

[7]The algorithm does not rely on an admissible heuristic; however, they assume that the heuristic function should be fairly accurate.

[8]A local minimum is a state in which the value of the heuristic is larger than the heuristic value of all the children.

to solve the test instances.

Extracting macros from a training instance works as follows. First, a graph is created from the solution to the instance in which each node in the graph is an action performed to obtain the solution and each edge represents the causal relation between the actions, *e.g.,* when performing one action satisfies the pre-conditions of another action. Second, macros of pre-defined length are extracted from this graph considering different subsets of nodes in the graph.

The learned macros are used in a heuristic search planner similar to FF [31]. The enhancement used in FF to decrease the search effort, *e.g.,* the helpful action pruning, is also applied in this macro-learner. This system is showed to perform well in a variety of domains used in the AIPS 2004 planning competition [4].

5.6.2 Reinforcement Learning

The concept of a value function in Reinforcement Learning (RL) [62] is very close to the concept of a heuristic function in heuristic search. A value function $V_\pi(s)$ for a policy π at a state s, estimates the return when policy π is followed from state s. Therefore, the value function can be considered as the reverse of the heuristic function. For example, the value function for a state near the goal state should be more than the value function for a state further from the goal state (the heuristic value for states near the goal state is smaller). Finding the optimal value function, V_π^*, in RL is similar to finding the exact distance of each state from the goal state in heuristic search.

The main difference between RL and our work is the search algorithm. In RL, the search algorithm is a greedy best first search, which is called the policy. To find a solution for each instance, the policy will be followed, *i.e.,* in each step the action that leads to a state with the highest value function will be selected. On the other hand, the search algorithm used in our work uses $f = g + h$ to find a solution.

Yoon, Fern, and Givan [67] used RL techniques to solve planning problems. They used a search algorithm that is slightly different from a greedy best first search. The search algorithm performs a greedy best first search for a limited number of steps (*e.g.,* 50 steps) and adds all the states generated during the search to a queue. If this search fails to find a solution, the best node from the queue will be selected for expansion, the one with the lowest heuristic value. A similar limited greedy best first search will be performed from this new state. This process is repeated until it finds a solution. It is empirically shown that this search algorithm outperforms the greedy best first search on domains used in the AIPS planning competition.

5.7 Evaluation Functions for Two-Player Games

Learning an evaluation function for a two-player game is very close to learning a heuristic function in single-agent search. The main difference is that the evaluation function in two-player games considers the effect of an opponent that also tries to win the game while the heuristic in single-agent search is not affected by such. Learning evaluation functions for two-player games dates back to Samuel's checkers player [57]. At each state during the game, a limited search was performed and the leaf nodes were evaluated using the current evaluation function. These evaluated leaf nodes then were used through a mini-max strategy[9] to select the best action from the current state. Then, the evaluation function was updated so that the evaluation function at the current state becomes closer to the evaluation function at the states that appeared in the limited search. The evaluation function used in Samuel's program was a weighted sum of the features of the game.

Reinforcement Learning has been successfully applied to learn an evaluation function for two-player games. Tesauro [63] applied RL to learn an evaluation function for the game of backgammon. The evaluation function in each state estimates the probability of winning the game from that state. The training data for this system is collected from by the learner playing games against itself. The learning algorithm used for this system was a neural network in which the backpropagation was used to update the weights at the end of each self-played game. The features used for the program were only representing the position of black and white pieces on the board, *e.g.,* the number of black/white pieces exists on each point on the board. This program is showed to perform as well as the best human players in the world.

Buro [7] studied learning an evaluation function for the game of Othello by combining a set of user-defined features of the game. To collect training data that contain a broad range of situations that can occur during the game, all the games that were played by earlier versions of the program were considered. Then, the training data was refined by fixing the label for the games in which one player made a big mistake. Win, lose, or draw is assigned as the label to each training example and learning algorithms such as logistic regression, linear and quadratic discriminant analysis were used. The final program works with expanding the game graph for each state and using an algorithm similar to mini-max to assign a score to each of the actions that can be made at the state. The evaluation function for all the states during this limited search is computed using the learned evaluation function.

Clune [8] studied learning evaluation functions for general game playing. His system first abstracts the game into a set of parameters, *e.g.,* payoff or stability of the payoff. Then, it automatically extracts features of the game and learns a linear combination of the features to estimate each parameter. To learn how to combine the features, a set of training game states was generated by random walks and regression was used as the learning algorithm. The evaluation function in each state is a combination of the parameters at the state by considering the game as a compound lottery. Clune's system showed great performance in the AAAI General Game Playing Competition[10].

[9]A mini-max strategy is a strategy to choose the next action in two-player games. At each state of the game, the search tree is expanded from the current state to a fixed depth. All the leaf nodes in the tree will be evaluated using the evaluation function. Assuming that the opponent always selects an action that minimizes our value, the action that maximizes our value with respect to opponent's action is returned.

[10]See `http://www.aaai.org/Conferences/AAAI/2008/aaai08generalgame.php` for more detail about the competition.

5.8 Summary

In this chapter, a brief survey of related work on learning (in)admissible heuristic for heuristic search and planning domains is presented. We discussed each related work and compared its components to our system and mentioned the advantages and disadvantages of each work. The main shortcomings of these works in comparison to ours are: (1) some works are only applicable to small search spaces [6, 13, 17, 26, 47, 59], (2) those that are applicable to larger search spaces are extensively hand-crafted [50, 55], and (3) those that do not suffer from either (1) or (2) produce solutions that are much larger than the optimal ones [18].

Chapter 6

Conclusions

6.1 Summary

The main problem considered in this thesis was investigating the use of machine learning to create effective heuristics for search algorithms such as IDA* or heuristic-search planners. Our incremental bootstrapping method aimed to generate strong heuristics from a given (weak) heuristic h_0 through a bootstrapping process. The "easy" problem instances that can be solved using h_0 provide training examples for a learning algorithm that produces a heuristic h_1 that is expected to be stronger than h_0. The Bootstrap process is then repeated using h_i instead of h_{i-1} until a sufficiently strong heuristic is created. When h_0 is so weak that it cannot solve any of the given instances, we use a random walk technique to create a sequence of successively more difficult instances starting with ones that are solvable by h_0. We tested our method on the 15- and 24-puzzle, the 17- , 24- , and 35-pancake puzzle, Rubik's Cube, and the 15- and 20-blocks world.

Our experiments demonstrated that bootstrap learning can help to speed up search dramatically with relatively little degradation in solution quality. An inherent and non-negligible expense is the time invested in learning the heuristic function. The total times (for training and solving) reported are on the order of several days for the larger problems (*e.g.,* the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, and the 20-blocks world). Such a lengthy process would be warranted if the final heuristic was going to be used to solve numerous problem instances that were distributed in the same way as the bootstrap instances, since one would expect most of the new instances would be solved as quickly with the final heuristic as the bootstrap instances were.

However, many planning problems require just a single instance to be solved—a task for which our bootstrapping approach may seem infeasible because of the large total time required. Therefore, we also investigated the question of whether a variation of bootstrapping can quickly solve a single instance of a given problem domain. Instead of minimizing solving time at the expense of requiring very large learning times, we looked for a balance in the learning and solving times so that the sum of the learning and solving times is made as small as can be. With this goal in mind, we presented a method that involves interleaving the learning and solving processes. The interleaving method sets the ratio of solving time to learning time manually. We presented experimental results on the 24-puzzle, the 35-pancake puzzle, Rubik's Cube, the 20-blocks world and IPC2 blocksworld instances, demonstrating that, even in the single instance case, bootstrapping can be very effective.

The next section discusses the main limitations of this research and some directions for future work on the ideas presented in this thesis.

6.2 Limitations and Future Work

The main limitation of our Bootstrap system is the completion time of the Bootstrap process. For example, it takes about 18 days to learn the final heuristic for the 24-puzzle. This time makes bootstrapping only useful when a large number of test instances need to be solved. In Chapter 4, we introduced a variation of our Bootstrap method to improve this total time for solving a single testing instance. We ran our system without any initial bootstrap instances (just using random walks to create training instances at successive levels of difficulty) and interleaved learning and solving until finding a solution for the testing instance. This variation of our Bootstrap method was experimentally shown to decrease the time overhead of bootstrap learning of heuristics substantially.

Instead of ignoring the bootstrap instances, we might be able to somehow sort the bootstrap instances based on their difficulty (the number of nodes that the search algorithm using the current heuristic generates for each bootstrap instance). If we knew how many nodes each bootstrap instance will generate to find a solution at each iteration, we could have only tried the current heuristic on a subset of bootstrap instances; those that generate the fewest number of nodes to find a solution and therefore can be solved within the given time limit of that iteration.

We thought about using the existing techniques that predict the number of nodes that IDA* with a specific heuristic will generate for a single instance of a problem. For example, Zahavi *et al.* [68] proposed a formula (*GKRE*) to predict the number of nodes that IDA* generates at a certain depth given a (possibly inconsistent) heuristic. Although *GKRE* showed great prediction accuracy for domains similar to those used in this thesis, it cannot be directly used in Bootstrap because

(i) The goal depth needs to be known for each bootstrap instance as *GKRE* predicts the performance of IDA* at a certain depth d ignoring that the goal may exist at a depth less than d.

(ii) *GKRE*'s prediction is very accurate when averaged over a set of instances but it can be very noisy for a single instance. To decrease the noise, *GKRE* was augmented by an IDA* search up to a depth r and then the prediction was made for all the nodes at depth r and the *GKRE*'s prediction was the average of predictions for each node at depth r. Although this led to better predictions for a single state, it is more costly as it involves (1) expanding the search graph to a depth r for each instance and (2) making predictions for some of the states at depth r.

One other limitation of our work is that for each problem, we learn a heuristic to estimate the distance of each state in the problem space from a fixed the goal state; therefore, all the learning phase should be done anew whenever the goal state changes. Bramanti-Gregor and Davis [6] suggested learning the distance between a pair of states by using the features of both states in the feature vector. The feature vector for the pair of states can be made by either (1) doubling the size of the feature vector to include the features for both states, or (2) using a feature vector to represent the difference between the value of each feature in two states.

Many directions for future research on the ideas presented in this thesis can be considered. The first direction would be applying our interleaving method for solving a single instance of the problem to more planning domains (*e.g.,* to all the problem domains used in the AIPS planning competition[1]). To be competitive with the state-of-the-art suboptimal planners [27, 31, 67] on this set of problems, we need at least two enhancements to our current settings. First, we need to create the features using automatic domain-independent techniques (*e.g.,* see Yoon *et al.* [67]) rather than expecting the user to provide the features for each domain. Second, we would like to be able to generalize from small problem domains to larger problem domains. Our experiments in Section 4.3.5 showed that the total time of our method to solve an instance of the planning problem increases with the size of the problem and solving instances of larger problems may become infeasible within the time limit given for each problem instance (30 minutes). Therefore, it would be necessary to generalize from the learned knowledge on small problem instances to solve larger problem instances. Yoon, Fern, and Givan [16, 67] have previously used this idea to learn different types of control knowledge to solve planning problems.

In our experiments, the features for each domain were not selected carefully nor based on the outcome of early experiments. Mostly the features used were weak PDBs created by abstraction. One interesting direction for feature work would be testing the effect of other set of features on the performance of Bootstrap. Our limited observations from initial experiments suggest that:

(i) Using stronger heuristics in the feature vector decreases the number of Bootstrap iterations as stronger heuristics will likely solve more bootstrap instances at each iteration. Decreasing the number of Bootstrap iterations will likely decrease the completion time of the Bootstrap process and result in fewer nodes generated and better solution quality for the heuristic generated by the last iteration of the Bootstrap (see Section 3.3.1 for an example of making the initial heuristic stronger on the 15-puzzle. A similar trend was seen on the 17-pancake puzzle.). Regarding the number of nodes generated, the use of a stronger heuristic helps whenever the maximum of the heuristics used in the feature vector has a higher value than the output of the neural network. Regarding the solution quality, fewer Bootstrap iterations means that fewer training data biased toward inadmissibility are collected. Therefore, the last heuristic of Bootstrap is expected to find closer to optimal solutions comparing to the case that weaker

[1]see `http://planning.cis.strath.ac.uk/competition/` for more details on these domains.

heuristics are used in the feature vector.

(ii) Our observations showed that (weak) heuristic features are very important, especially to guide the search near the goal state. We performed an experiment on the 15-puzzle in which we solved all the training instances optimally using a state of the art heuristic for the 15-puzzle [42], but we only used the tile positions as the features in the feature vector. The results show that the heuristic learned from these weak features (without the guidance of a weak heuristic near the goal) fails to produce a useful heuristic.

One other future work is to investigate the effect of starting the Bootstrap procedure without any initial heuristic. In our bootstrapping, to compute the heuristic value for a state s, the maximum of the value of the initial heuristic for state s and the heuristic value created by the neural network for state s is considered. Our experiments showed that computing the maximum helps when the heuristic value calculated by the neural network for a state is smaller than the initial heuristic value for that state. Our experiments further showed that the guidance of the initial heuristic is important especially near the goal state. In order to ensure this guidance when the system starts without any heuristic, perimeter search [10] can be used in which the perimeter of the search is used to provide guidance near the goal state[2]. The perimeter can be consisted of all the states visited during the time bounded breadth first search performed to compute the $LengthIncrement$ parameter (see Section 3.1.2).

Another future direction is to consider a different search algorithm for bootstrapping. As bootstrapping has a time limit for each iteration and the search stops when the time limit is reached for each bootstrap instance, one can think of using a more efficient algorithm than IDA* because IDA* can expand the same node numerous times in total. For example, A*, if implemented efficiently, can be used.

A study of the effect of using suboptimal heuristic search algorithms, *i.e.,* those like BULB that sacrifice solution quality to find a solution faster, with Bootstrap can also be considered. This study should at least consider the changes in the completion time of the Bootstrap, the solution quality and number of nodes generated by the final heuristic of Bootstrap. Future work can also be done to investigate the effect of the learning component of our system in the total success of our results. For example, one can investigate how important it is to use a non-linear learning system (neural network) and whether we can get similar results when we combine our features using a simple linear model (*e.g.,* linear regression).

The heuristic in our system is neither consistent nor admissible. One possible method to exploit inconsistency is Bidirectional Pathmax ($bpmx$) [15] to propagate the heuristic value of a state to its children and also in the reverse direction. For example, consider that the heuristic value of a state s is 1 and the heuristic value of its child is 5 and each action in the search space has a cost of 1.

[2]This idea is due to an anonymous AAAI reviewer.

Then, the heuristic value of the state s can increase to 4 because each action (here, moving from the state to its child) changes the distance to the goal state by at most 1. Therefore, using $bpmx$ with an inconsistent heuristic may result in pruning subtrees that would otherwise be expanded. Note that $bpmx$ is designed for an admissible heuristic. Therefore, using $bpmx$ with Bootstrap may result in propagating inadmissibility to more parts of the search space. We did not extensively explore $bpmx$, but preliminary experiments showed that using $bpmx$ with the final heuristic of Bootstrap slightly increases the solution length while the number of nodes generated will slightly decrease compared to Bootstrap without $bpmx$.

6.3 Contributions and Closing Remarks

The major contribution of this thesis is to provide experimental evidence that machine learning can be used to create strong heuristics from given (very weak) ones through an incremental bootstrapping process augmented by a random walk method for generating successively more difficult problem instances. Our system was tested on eight different problem domains and successfully created heuristics that enable IDA* to solve randomly generated test instances quickly and almost optimally. This work substantially extends previous one-step methods as it does not require:

(i) the given heuristic to be very strong to start the process. For example, system developed by Ernandes and Gori [13] could not be applied to the 24-puzzle because of the infeasibility of creating a large training set as solving a thousand instances of the 24-puzzle using the best existing heuristics [14, 42] would take many CPU years.

(ii) to choose the learning features carefully. For example, to learn a heuristic for the 24-puzzle, Samadi *et al.* [55] made crucial choices on how to manually divide the problem into smaller subproblems and how to combine them instead of directly learning a heuristic for the 24-puzzle.

The total time needed for our system to create these heuristics strongly depends on the number of given bootstrap instances. In fact, a very large portion of the training time is spent on trying to solve bootstrap instances that are still too difficult for the current heuristic. With the goal of decreasing this total time in mind, we presented a method that involves interleaving the learning and solving processes and experimentally showed that it will lead to a substantial speedup over the total time of our bootstrapping method while generating solutions that are even closer to the optimal ones.

Bibliography

[1] Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J.E. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1991.

[2] Marcel Ball and Robert C. Holte. The compression power of symbolic pattern databases. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, pages 2–11, 2008.

[3] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[4] Adi Botea, Martin Müller, and Jonathan Schaeffer. Learning partial-order macros from solutions. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS'05)*, pages 231–240, 2005.

[5] Anna Bramanti-Gregor and Henry W. Davis. Learning admissible heuristics while solving problems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 184–189, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[6] Anna Bramanti-Gregor and Henry W. Davis. The statistical learning of accurate heuristics. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1079–1087, 1993.

[7] Michael Buro. Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995.

[8] James Clune. Heuristic evaluation functions for general game playing. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI'07)*, pages 1134–1139. AAAI Press, 2007.

[9] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. *Advances in Artificial Intelligence (LNAI 1081)*, pages 402–416, 1996.

[10] John F. Dillenburg and Peter C. Nelson. Perimeter search. *Artificial Intelligence*, 65(1):165–178, 1994.

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

[12] Stefan Edelkamp. Symbolic pattern databases in heuristic search planning. In *Proceedings of 6th International Conference on AI Planning and Scheduling (AIPS'02)*, pages 274–283, 2002.

[13] Marco Ernandes and Marco Gori. Likely-admissible and sub-symbolic heuristics. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 613–617, 2004.

[14] Ariel Felner and Amir Adler. Solving the 24 puzzle with instance dependent pattern databases. In *Abstraction, Reformulation and Approximation, 6th International Symposium (SARA'05)*, volume 3607 of *LNCS*, pages 248–260. Springer, 2005.

[15] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual lookups in pattern databases. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 103–108, 2005.

[16] Alan Fern, Sungwook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *Proceedings of 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, pages 191–199. AAAI Press, 2004.

[17] Michael Fink. Online learning of search heuristics. *Proceedings of the 10th International Conference on Artificial Intelligence and Statistics (AISTATS'07)*, pages 114–122, 2007.

[18] Lev Finkelstein and Shaul Markovitch. A selective macro-learning algorithm and its application to the nxn sliding-tile puzzle. *Journal of Artificial Intelligence Research*, 8:223–263, 1998.

[19] David Furcy. *Speeding up the convergence of online heuristic search and scaling up offline heuristic search*. PhD thesis, Atlanta, GA, USA, 2004.

[20] David Furcy and Sven Koenig. Limited discrepancy beam search. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 125–131, 2005.

[21] Naresh Gupta and Dana S. Nau. Complexity results for blocks-world planning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 629–633, 1991.

[22] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254, 1992.

[23] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 607–613. Morgan Kaufmann, 1995.

[24] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceeding of the 5th International Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS'00)*, pages 140–149, 2000.

[25] Ami Hauptman, Achiya Elyasaf, and Moshe Sipper. Evolving hyper heuristic-based solvers for Rush Hour and FreeCell. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 149–150, 2010.

[26] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, and Assaf Karmon. GP-rush: using Genetic Programming to evolve solvers for the Rush Hour puzzle. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO'09)*, pages 955–962, New York, NY, USA, 2009. ACM.

[27] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[28] Malte Helmert. Landmark heuristics for the pancake problem. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 109–110, 2010.

[29] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, pages 162–169, 2009.

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

[31] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[32] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[33] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[34] Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *6th International Symposium on Abstraction, Reformulation and Approximation (SARA'05)*, volume 3607 of *LNAI*, pages 121–133. Springer, 2005.

[35] Timothy Humphrey, Anna Bramanti-Gregor, and Henry W. Davis. Learning while solving problems in single agent search: Preliminary results. In *Proceedings of the 4th Congress of the Italian Association for Artificial Intelligence on Topics in Artificial Intelligence (AI*IA'95)*, pages 56–66. Springer, 1995.

[36] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Bootstrap learning of heuristic functions. In *Proceedings of the 3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, pages 52–59, 2010.

[37] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1728–1733, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[38] David G. Kirkpatrick. Hyperbolic dovetailing. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA 2009)*, pages 516–527, 2009.

[39] Richard E. Korf. Iterative-Deepening-A*: An optimal admissible tree search. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence(IJCAI'85)*, pages 1034–1036, 1985.

[40] Richard E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[41] Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI'97)*, pages 700–705, 1997.

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

[43] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[44] Bradford Larsen, Ethan Burns, Wheeler Ruml, and Robert C. Holte. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010)*, pages 114–120, 2010.

[45] Drew Mcdermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109:111–159, 1999.

[46] Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 596–599, 1985.

[47] Marek Petrik and Shlomo Zilberstein. Learning heuristic functions through approximate linear programming. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, pages 248–255, 2008.

[48] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 12–17, 1973.

[49] Larry A. Rendell. A locally optimal solution of the fifteen puzzle produced by an automatic evaluation function generator. Technical Report CS-77-36, Department of Computer Science, University of Waterloo, 1977.

[50] Larry A. Rendell. A new basis for state-space learning systems and a successful implementation. *Artificial Intelligence*, 20:369–392, 1983.

[51] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 975–982, 2008.

[52] Gabriele Röger and Malte Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 246–249, 2010.

[53] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[54] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. pages 673–695, 1988.

[55] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 357–362. AAAI Press, 2008.

[56] Mehdi Samadi, Maryam Siabani, Ariel Felner, and Robert C. Holte. Compressing pattern databases with learning. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, pages 495–499, 2008.

[57] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research Development*, 3(3):210–229, 1959.

[58] Sudeshna Sarkar, P. P. Chakrabarti, and Sujoy Ghose. Learning while solving problems in best first search. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 28:535–541, 1998.

[59] Sudeshna Sarkar, Sujoy Ghose, and P. P. Chakrabarti. Learning for efficient search. *Sadhana:Academy Proceeding in Engineering Sciences*, 2:291–315, 1996.

[60] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.

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

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

[63] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.

[64] Richard Valenzano, Nathan R. Sturtevant, Jonathan Schaeffer, Karen Buro, and Akihiro Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 177–184, 2010.

[65] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 2001.

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

[67] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, 2008.

[68] Uzi Zahavi, Ariel Felner, Neil Burch, and Robert C. Holte. Predicting the performance of IDA* with conditional distributions. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 381–386. AAAI Press, 2008.

[69] Uzi Zahavi, Ariel Felner, Robert C. Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*, pages 1076–1081, 2006.

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