University of Alberta

CLUSTER-AND-CONQUER: A PARADIGM FOR SOLVING STATE-SPACE PROBLEMS

by

Levi Henrique Santana de Lelis

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

Doctor of Philosophy

Department of Computing Science

Fall 2013
Edmonton, Alberta

Abstract

Many important problems can be cast as state-space problems. In this dissertation we study a general paradigm for solving state-space problems which we name Cluster-and-Conquer (C&C). Algorithms that follow the C&C paradigm use the concept of equivalent states to reduce the number of states explored to solve the problem. The concept of equivalent states is captured by a *type system*, which is a partition of the states in the state space. C&C algorithms assume that states of the same type are equivalent to one another, thus only one state of each type must be explored. Although our type systems only approximate the ideal partition of states into equivalent states, the solutions found by most of our C&C algorithms are guaranteed to converge to optimal solutions.

The C&C paradigm is general and can be used to solve different problems. In this dissertation we advance and develop C&C algorithms for solving different state-space problems. Namely,

- we advance Conditional Distribution Prediction (CDP) and Stratified Sampling (SS), two existing algorithms for predicting the search tree size of Iterative-Deepening A* (IDA*). Among other contributions, (1) we introduce different heuristic-based type systems, and (2) we apply the concept of active sampling to increase the accuracy of the predictions. Our versions of CDP and SS represent the current state of the art for predicting the IDA* search tree size;

- we present Two-Step Stratified Sampling (TSS), an algorithm for predicting the search tree size of Depth-First Branch and Bound (DFBnB). TSS is the only known algorithm able to produce accurate predictions of the DFBnB search tree size in a reasonable amount of time;

- we present Solution Cost Predictor (SCP) and Bidirectional Stratified Sampling (BiSS), the first algorithms designed for predicting the optimal solution cost of state-space problems. We show that one can reduce from days to minutes the time required for learning heuristic functions by using BiSS to label the training set;

- we present Stratified Tree Search (STS), a search algorithm for finding suboptimal solutions of state-space problems when the only heuristics available are too weak to usefully guide existing methods.

Acknowledgements

I would like to express my deepest gratitude to my PhD advisors Rob Holte and Sandra Zilles. Rob and Sandra's enthusiasm, patience, guidance, and high standards were of paramount importance for the success of this research. I also would like to thank Jonathan Schaeffer, Joerg Sander, Martin Mueller and Mike Bowling from the Department of Computing Science at the University of Alberta.

During my PhD studies I had the chance to collaborate with researchers from outside the University of Alberta. In particular I would like to thank Roni Stern, Ariel Felner, Rina Dechter and Lars Otten. The special thanks goes to Roni, with whom I spent countless hours discussing ideas, planning experiments, writing papers, analyzing results, creating conjectures, and proving theorems.

Among other friends, I am thankful to AmirAli Sharifi, Shahab Jabbari Arfaee, Shahin Jabbari Arfaee, Mike Delp, Marc Bellemare, Sheehan Khan, Rick Valenzano, Jacqueline Smith, Hootan Nakhost, Fan Xie, Chris Rayner, Cailu Zhao, Richard Zhao, Jagoda Walny, Chenlei Zhang, Alejandro Hernandez Herdocia, Diego Serrano Suarez, Bernardo Avila Pires and Guni Sharon. I also wish to thank the Rocca Brazilian Jiu Jitsu team, specially Diego Lapetina and Tuca Alvares.

I am also very grateful to my friends in Brazil. Above all, I wish to thank Claudeci G. Varejão Junior and Luana C. Lima da Fonseca for shooting a video with messages from my dearest friends before I moved to Canada. I fancy they had not realized at the time they shot the video how important those messages would be to help me endure the distance from them. I also would like to thank my former advisor Antonio de Padua Braga—now I have a winter coat which underwent two PhDs!

For their support and love, I thank the greatest people of all time: my parents and my siblings.

Finally, for their financial support I am grateful to Alberta Innovates - Technology Futures, AICML, NSERC, and the Department of Computing Science at the University of Alberta. This work was also supported by the Laboratory for Computational Discovery at the University of Regina.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many important real-world problems can be cast as state-space search problems. For instance, state-space search algorithms have been applied to a number of interesting applications: chemical compounds discovery [27], number partitioning [39], bin packing [41], sequence alignment [40], automating layouts of sewers [9], network routing [61], and domain-independent planning [6], among others. In this dissertation we study a family of algorithms that uses a paradigm for solving state-space problems we name Cluster-and-Conquer (C&C).

1.1 The Cluster-and-Conquer Paradigm

In the state-space search problem illustrated in Figure 1.1 the agent, in the top-right corner, needs to plan a shortest path from its initial location to the location marked with the letter g (bottom-left corner); the shaded area represents an obstacle the agent cannot traverse. From its initial location the agent has two options: it can either go to location b (state agent-is-at-b) or to location c (state agent-is-at-c). The agent will find solution paths of the same length either by going through b or c. Thus, from the perspective of finding any optimal path, the path from b to g is equivalent to the path from c to g. If we know that the states agent-is-at-b and agent-is-at-c are equivalent, when looking for a shortest path we can discard one of the two possibilities and only consider the other for planning.

Algorithms that use the C&C paradigm for solving state-space problems use the concept of equivalent states to reduce the number of states that have to be explored to solve a given problem, as illustrated in the path-planning problem above. The concept of equivalent states is captured by C&C algorithms through a *type system*, which is a partition of the states in the state space. In the example above, the states agent-is-at-b and agent-is-at-c are equivalent and ideally would have the same type. C&C algorithms assume that states of the same type are equivalent to one another, thus only one state

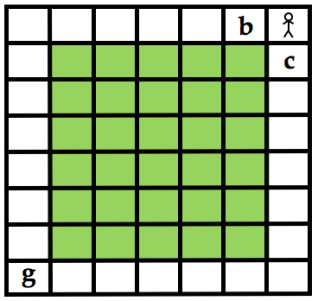

Figure 1.1: The agent in the top-right corner needs to plan a path to the location marked with a letter g; the shaded area represents an obstacle the agent cannot traverse.

of each type must be explored.

The type systems we use are only approximations of the ideal partition of states into equivalent states. That is, in our type systems, states of the same type are not necessarily equivalent to each other. For example, in the path-planning problem shown above we would not necessarily group into the same type only states that have solution paths of the same length. Although our type systems only approximate the exact partition of the state space into equivalent states, if time and memory allow, most of the C&C algorithms presented in this dissertation are guaranteed to eventually find the exact solution for the state-space problems they were designed to solve.

Algorithms using the C&C paradigm tend to be efficient in the sense that they only explore one state of each type while solving a given state-space problem. As we are going to see in the upcoming chapters, we have effective ways of controlling the size of the type systems we use, which allow the user of C&C algorithms to control to some extent the algorithm's running time and memory needs.

1.1.1 The Generality of the Cluster-and-Conquer Paradigm

The C&C paradigm is general and can be used to solve different problems. For example, in this dissertation, in addition to the path-planning problem illustrated above, we also use the C&C paradigm to design algorithms for estimating the size of the tree expanded by search algorithms. In this case, instead of assuming that states of the same type lead to solutions of the same length, we assume that states of the same type root subtrees of the same size, and again, only one state per type must be explored.

Overview of a General C&C Algorithm

A C&C algorithm $\mathcal{A}$ designed to solve a particular state-space problem $\mathcal{P}$ is divided into two parts: the **cluster** part and the **conquer** part.

- The cluster part is given by a type system, which is a partition of the states in $\mathcal{P}$'s state space. The type system tries to approximate the partition of $\mathcal{P}$'s state space into equivalent states, i.e., states of the same type lead to solutions of the same length if $\mathcal{P}$ is a path-planning problem, or states of the same type root subtrees of the same size if $\mathcal{P}$ is a tree size prediction problem.

- The conquer part is the design of $\mathcal{A}$ for using the type system to find a solution for $\mathcal{P}$. This is done by exploring one state of each type encountered during search.

  - The conquer part could require steps other than the one in which $\mathcal{A}$ searches the state space by exploring one state of each type. For example, as we are going to see in Chapter 3, when predicting the search tree size a C&C algorithm produces different predictions for each type encountered during search. In an extra step the C&C algorithm merges all predictions in a single prediction of the size of the search tree.

The main contributions of this dissertation are in the conquer part of the C&C algorithmic design.

1.2 Contributions

The C&C paradigm is general and can be used to solve a variety of different problems. In this dissertation we advance existing C&C algorithms and develop novel C&C algorithms for solving different problems related to state-space search. To be specific, we work on the following four problems: (i) we advance existing C&C algorithms for predicting the search tree size of Iterative-Deepening A* (IDA*) [43]; (ii) we develop C&C algorithms for predicting the search tree size of Depth-First Branch and Bound (DFBnB) [4]; (iii) we develop the first algorithms for predicting the optimal solution cost of state-space search problems; and (iv) we develop a novel C&C search algorithm for finding suboptimal solutions of state-space search problems.

1.2.1 Predicting the Size of IDA* Search Trees

A frequent impediment of the application of tree searching algorithms [68] is the inability to quickly predict the running time of an algorithm on a particular problem [38]. While one problem instance might be solved quickly, another could take a prohibitive amount of time.

Korf, Reid, and Edelkamp [50] launched a line of research aimed at creating a method to predict the running time of IDA*. Their idea was that by knowing the size of the search tree one would be able to estimate the running time of IDA*. Zahavi et al. [84] generalized the work of Korf et al. through a method called Conditional Distribution Prediction (CDP). Independent of Korf et al. and

Zahavi et al., Knuth [38] created a method for predicting the search tree size of search algorithms which was later improved by Chen [11] through a method called Stratified Sampling (SS). As we are going to see in Chapter 3, both CDP and SS are C&C algorithms.

In Chapter 3 we connect and advance CDP and SS. Namely, we make the following contributions to CDP and SS.

- For the CDP algorithm we present:
  - a method for improving the accuracy of the predictions CDP produces. Our method works by carefully ignoring rare events observed during the sampling process required by CDP.
  - a method for speeding up the CDP predictions. We show that the number of CDP predictions, which is dictated by the number of types, can be precomputed and stored in a lookup table, which speeds up the prediction process.
- For the SS algorithm we show:
  - that some of the ideas developed for the CDP algorithm can also be applied to SS to improve its predictions.
  - how to use active sampling [18] in SS to improve the accuracy of its predictions.
- We also show how to use clustering through the k-means algorithm [64] for controlling the number of types in our type systems.

The empirical results we present in this dissertation show that our enhanced versions of CDP and SS represent the current state of the art for predicting the IDA* search tree size.

These contributions were published in the Proceedings of the Symposium on Combinatorial Search (2011) [54] and (2013) [55], in the Proceedings of the Conference on Artificial Intelligence, AAAI, (2012) [57] and in the Artificial Intelligence journal (2013) [58].

1.2.2 Predicting the Size of Depth-First Branch and Bound Search Trees

Another popular search algorithm frequently used to solve optimization problems is Depth-First Branch and Bound (DFBnB). The user of DFBnB also usually does not know a priori the running time of the algorithm. In Chapter 4 we make the following contributions.

- As anticipated by Knuth [38], we show that the methods developed for predicting the size of the IDA* search trees do not produce good predictions of the size of the DFBnB search trees.

- We introduce a C&C algorithm called Two-Step Stratified Sampling (TSS) for predicting the search tree size of DFBnB. TSS is guaranteed to predict the exact size of the DFBnB search tree if granted enough time and memory.

- Our empirical results on optimization queries over graphical models, such as finding the most likely explanation in Bayesian networks [69], show that TSS is the only method able to produce accurate predictions of the size of the DFBnB search trees.

The contributions presented in Chapter 4 were published in the Proceedings of the International Joint Conference on Artificial Intelligence (2013) [56].

1.2.3 Predicting the Optimal Solution Cost of State-Space Search Problems

Search algorithms are designed to find an optimal solution for state-space search problems. The optimal solution cost is found as a byproduct of the solution. However, there are applications in which one only needs to know the optimal solution cost — the solution is not needed. For example, consider the owner of a construction company that is required to quickly assess the monetary cost of a project for bidding purposes. In such a case, only the cost of executing the project is needed. The actual construction plan could be formulated later, if the bid is won.

Therefore, an important question to be answered is the following. Can one predict the optimal solution cost of a problem without finding an actual solution? Korf, Reid and Edelkamp [50] stated: "Predicting the optimal solution cost for a given problem instance (...) is an open problem". In Chapter 5 we present C&C algorithms for predicting the optimal solution cost of state-space search problems. Specifically, we make the following contributions.

- We present Solution Cost Predictor (SCP), a C&C prediction method based on CDP, able to produce fairly accurate predictions of the optimal solution cost of state-space search problems.

- We present Bidirectional Stratified Sampling (BiSS), a C&C bidirectional variation of SS that produces good predictions of the optimal solution cost of state-space search problems.

  - We show empirically that BiSS scales to state spaces much larger than can be solved optimally. In particular, we predict the average solution cost for the sliding-tile puzzles up to the 8x8 configuration, which has more than 10^{88} reachable states, and provide indirect evidence that BiSS's predictions for these huge state spaces are accurate.

  - We show how to quickly learn strong heuristics from predictions produced by BiSS. We show that it is possible to reduce the time required for learning strong heuristic functions

from days to minutes by using BiSS's predictions to generate the training set.

The contributions presented in Chapter 5 were published in the Proceedings of the Symposium on Combinatorial Search (2011) [53] and (2012) [60], and in the Proceedings of the International Conference on Automated Planning and Scheduling (2012) [52].

1.2.4 Finding Suboptimal Solutions for State-Space Search Problems

In Chapter 6 we present Stratified Tree Search (STS), a C&C search algorithm for finding suboptimal solutions for state-space search problems in cases in which other search algorithms tend to fail to find a solution in a reasonable amount of time. Namely, we make the following contributions.

- We empirically compare STS to standard search algorithms in problems with fairly large state spaces. Overall STS performs better than other search algorithms.
- STS has input parameters that offer a good tradeoff between solution quality and search speed. However, in some cases, STS does not provide a fine adjustment of this tradeoff. We present a simple variant of STS, named Beam STS (BSTS), that allows one to make fine adjustments of the tradeoff between solution quality and search speed. BSTS combines the ideas of Stratified Tree Search with those of beam search [5].

The contributions presented in Chapter 6 were published in the Proceedings of the Conference on Autonomous Agents and Multiagent Systems (2013) [59].

1.3 The Structure of this Dissertation

Before describing our C&C algorithms we review concepts of state-space search in Chapter 2. Then, this dissertation follows the structure used in Section 1.2 above. Namely, this dissertation is separated by problems and it contains one chapter for each problem. Finally, in Chapter 7 we mention the limitations of our C&C methods and we make concluding remarks.

Chapter 2

Background

In this chapter we review basic concepts of state-space search by using the sliding-tile puzzle [79] as a running example. We also provide definitions that will be used throughout this dissertation.

2.1 State-Space Search

Let us consider a classical example of a state-space problem: the sliding-tile puzzle [79]. Figure 2.1 shows two instances of the sliding-tile puzzle that has 8 tiles and a blank position. This puzzle is often referred to as the 8-puzzle. Our goal here is to "transform" a given instance into a goal instance by sequentially sliding the tiles into the blank position.

Figure 2.1: The 8-puzzle; a random state on the left and the goal state on the right.

We call different instances of the problem *states*. The 8-puzzle has a single goal state, which has the tiles ordered lexicographically from left to right and top to bottom, with no tile occupying the upper leftmost position, as shown in the 8-puzzle on the right of Figure 2.1. The act of sliding a tile into the blank position will be called an *action*. Note that at a given state s only certain actions are applicable to s. For instance, for the 8-puzzle on the left of Figure 2.1 one cannot slide tile 8 into the blank position because that tile is not adjacent to the blank position; only tiles 5, 6, and 7 could slide into the blank position.

We call the *underlying search tree* (UST) the tree rooted at a given start state representing

the state-space problem. In the UST the vertices represent states, and the edges connecting the vertices represent actions that transform one state into another. We call the vertices in the UST *nodes*. Each node in the UST represents a single state in the state-space. However, each state might be represented by more than one node in the UST as the same state might be generated by different paths from the root (start state). Nodes representing the same state in the UST are called *transpositions*. Nodes representing goal states are called *goal nodes*. When clear from context we will use the terms nodes and states interchangeably.

We say that the 8-puzzle's UST is *implicitly defined* because we have the vertices and edges in the UST defined by a compact set of rules, e.g., one can move a tile on the 8-puzzle only if this tile is adjacent to the blank. In order to navigate through the nodes of an implicitly defined UST one has to apply the set of actions available to a given state, which are given according to the rules describing the problem domain. Examples of *explicitly defined* USTs are those that arise in map-based pathfinding problems. In explicitly defined domains the graph representing the domain is loaded in memory and one can navigate through the vertices of the graph with the aid of specialized data structures.

The underlying search tree (UST) of a given start state is defined as follows.

Definition 1 (UST)**.** *Let $S = (N, E)$ be a directed and implicitly defined tree representing a state-space problem for a given start state. Here, N is its set of nodes and for each $n \in N$ $child(n) = \{n' | (n, n') \in E\}$ is its set of child nodes.*

Definition 2 (Solution Path)**.** *A solution path is a sequence of edges in the UST that connects the root of the UST to a goal node.*

Definition 3 (Solution Cost)**.** *Each edge in the UST has a cost. The cost of a solution path p is the sum of the costs of the edges in p.*

Definition 4 (Optimal Solution Cost)**.** *A solution path p is called optimal if there is no other path from the root to a goal node with cost lower than the cost of p.*

We say that node n in the UST is being *expanded* by a search algorithm when the algorithm *generates* the set $child(n)$ of nodes.

2.1.1 Breadth-First Search

Breadth-First Search (BFS) is a search algorithm that explores the UST to find a solution path. BFS starts by expanding the start state s^* and storing the set $child(s^*)$ of nodes in a first-in-first-out data structure we call OPEN. In every iteration of the algorithm BFS expands the first element

s in OPEN. During expansion BFS (i) adds s to a set we call CLOSED, and (ii) adds the nodes $\{n' \in child(s) | n' \notin$ CLOSED and $n' \notin$ OPEN$\}$ to OPEN. The set CLOSED prevents BFS from expanding transpositions. If the UST has unit-edge costs, BFS is guaranteed to find an optimal solution path.

Clearly the time and memory complexities of BFS are linear in the size of an UST with no transpositions. However, often in practical scenarios the size of the UST is much larger than one can afford to store or expand.

2.1.2 Guidance of Heuristic Functions

Due to memory or time requirements, strategies such as the one used by BFS are applicable only to problems with small state-spaces. For instance, the 8-puzzle has a small state-space with only $181,440$ states. If one is interested in solving larger problems such as the (5×5) sliding-tile puzzle (24-puzzle), which has more than 7.75×10^{24} reachable states, a better search strategy must be used.

A *heuristic function*, or simply heuristic, $h(\cdot)$, estimates the cost to reach a goal state from a given node in the search tree (cost-to-go). Heuristic search algorithms such as Iterative-Deepening A* (IDA*) [43] and Depth-First Branch and Bound (DFBnB) [4] use a heuristic function to guide their search to more promising parts of the state-space, avoiding unpromising states from being explored. IDA* and DFBnB using an *admissible* heuristic function are guaranteed to find optimal solutions.

Definition 5 (Admissible Heuristic)**.** *A heuristic function h is called admissible if for any node n it never overestimates the optimal cost-to-go from n.*

Another important definition we are going to use is the consistency of a heuristic function.

Definition 6 (Consistent Heuristic)**.** *Heuristic h is consistent iff $h(s) \leq c(s,t) + h(t)$ for all nodes s and t, where $c(s,t)$ is the cost of the cheapest path from s to t and $h(s)$ is the estimated cost-to-go from s to a goal node.*

2.1.3 Iterative-Deepening A*

IDA* searches the state-space in a depth-first manner, i.e., IDA* fully explores the subtree below a child of node n before exploring any other child of n. Figure 2.2 shows a search tree in which A is the root node and B and G are children of A. A depth-first search algorithm would explore the entire subtree rooted at B before exploring the subtree rooted at G. A depth-first search algorithm explores the search tree in Figure 2.2 in lexicographic order according to the label of the nodes.

Algorithm 1 Iterative-Deepening A*

Input: start state s^* and heuristic function h
Output: optimal solution cost from s^* to a goal node

```
1: d ← h(s*)
2: while true do
3:    (cost, goal) ← DFS(s*, d) // see Algorithm 2
4:    if goal is true then
5:       return cost
6:    else
7:       d ← cost
8:    end if
9: end while
```

Besides searching in depth-first order, IDA* uses a cost function based on a heuristic to guide its search. This function is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach node n from the root, and $h(n)$ is the estimated cost-to-go. Thus, $f(n)$ is the estimated cost of a solution path going through n. We now explain how IDA* uses f to guide its search.

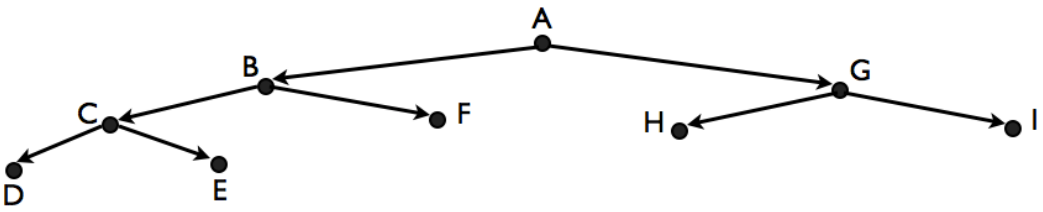

Figure 2.2: A depth-first search algorithm explores the nodes in the search tree in lexicographic order: A, B, C, D, E, F, G, H, I.

Given start state s^* and a heuristic function h, IDA* searches the UST by using a cost bound d. The value of d is initially set to the heuristic value of the start state. IDA* expands all nodes n it encounters such that $f(n) \leq d$. In case all nodes with f-value within cost bound d are expanded and a goal is not found, the cost bound d is increased by a minimum amount and the process is repeated. To be specific, the cost bound d is increased by setting it to the lowest f-value larger than d observed in the previous iteration of IDA*. IDA* using an admissible heuristic finds optimal solutions [43].

During an IDA* search a node $\hat{s}$ is not generated from s if $\hat{s}$ is the parent of s. For instance, if the state on the left side of Figure 2.1 was generated by moving tile 5 to the right, then IDA* will not move tile 5 back to the left when generating the children of that state. Otherwise IDA* would be generating a state that was already explored. This enhancement is called parent-pruning.

Algorithms 1 and 2 show the pseudocode. IDA* receives a start state s^* and heuristic function h and returns the optimal solution cost of a path from s^* to a goal state, if one exists. Although not shown in the pseudocode, the optimal solution path could be returned by tracing back the sequence

Algorithm 2 DFS

Input: node n and cost bound d
Output: cost bound c and variable $goal$ stating whether c is the optimal solution cost or not

```
1:  m ← ∞
2:  if f(n) > d then
3:      return (f(n), false)
4:  end if
5:  if n is a goal node then
6:      return (d, true)
7:  end if
8:  for each child n' of n do
9:      if n' is different than parent(n) then
10:         (cost, goal) ← DFS(n', d)
11:         if goal is true then
12:             return (cost, true)
13:         end if
14:         if m > cost then
15:             m ← cost
16:         end if
17:     end if
18: end for
19: return (m, false)
```

of actions taken to reach the goal node. Initially, the cost bound d is set to the heuristic value of the start state (line 1) and it is incremented by a minimum amount if a solution is not found (line 7).

Algorithm 2 shows the pseudocode for a depth-first search bounded by d. DFS implements parent-pruning in line 9, where it checks whether the node n' generated by n is the same node that generated n (n's parent in the search tree). DFS will explore n' only if n' is not the parent of n. DFS prunes nodes with f-value larger than d (line 2). In Algorithm 2 we store the lowest f-value larger than d in variable m (line 14).

2.1.4 Depth-First Branch and Bound

Depth-First Branch and Bound (DFBnB) also explores the UST in a depth-first manner while keeping track of the current best-known solution cost, denoted c^b. It uses an admissible heuristic function $h(\cdot)$, and is also guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the root node to n. When using an admissible heuristic, the value of $f(n)$ underestimates the cost of an optimal solution that goes through n. Therefore, whenever $f(n) \geq c^b$, n is pruned.

Algorithm 3 shows the pseudocode for DFBnB. Here DFBnB receives an initial solution with cost c^b which is derived by preprocessing (e.g., local search). Although in Algorithm 3 we only return the optimal solution cost, an optimal solution path could also be recovered by keeping in memory the incumbent least-cost path found by the algorithm. When the search finishes this incum-

Algorithm 3 DFBnB

Input: node n and a global variable with the cost of the best solution known c^b.
Output: optimal solution cost c^b.

```
1: for each child n' of n (sorted in increasing order according to the children's f-value) do
2:   if f(n') < c^b then
3:     if n' is a goal node then
4:       c^b ← f(n')
5:     else
6:       DFBnB(n')
7:     end if
8:   end if
9: end for
10: return c^b
```

bent solution path is guaranteed to be optimal.

2.2 Where Do Heuristics Come From?

One way of creating a heuristic function is to use domain knowledge. For example, a popular heuristic function for the sliding-tile puzzle is the sum of the Manhattan distance of the puzzle's tiles. For instance, the Manhattan distance heuristic of the instance on the lefthand side of Figure 2.1 is 15: 2 (tile 2) + 3 (tile 8) + 3 (tile 3) + 2 (tile 1) + 2 (tile 6) + 1 (tile 4) + 1 (tile 7) + 1 (tile 5).

In this section we describe general ways of creating heuristic functions without necessarily using domain-knowledge.

2.2.1 Pattern Databases

Culberson and Schaeffer [14] introduced the concept of a pattern database (PDB), a lookup table storing the optimal solution cost of states in an abstracted and smaller version of the original state-space. One can design an abstracted state-space of a problem by simplifying the original problem.

Example 1. *An abstracted space of the 8-puzzle can be obtained by making a set of tiles indistinguishable from each other. For instance, by making tiles 1, 2, and 3 indistinguishable from each other the abstracted state-space will be smaller than the original state-space as sets of states that differ from one another only by the positions of tiles 1, 2, and 3 will be mapped to single abstract states. Figure 2.3 shows the abstracted goal state when tiles 1, 2, and 3 are indistinguishable from each other.*

A lookup table with the optimal solution cost of the abstracted states is built by doing a complete search backward from the abstracted goal state as a preprocessing step.

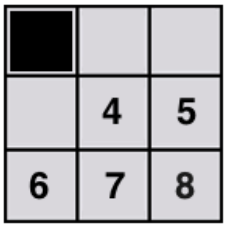

Figure 2.3: Example of an abstracted goal state.

Heuristic search algorithms use a PDB as a heuristic function by looking up the pre-computed values in the lookup table during search – every state n in the original state-space has a corresponding abstracted state s stored in the lookup table. The optimal solution cost of s computed with the backward search in the abstracted state-space offers a lower bound on the optimal cost-to-go of n in the original state-space [14]. PDBs represent a major advance in designing heuristic functions. For instance, PDBs allowed the first instances of the Rubik's Cube to be solved optimally [**?**]. Later, Felner et al. [19] developed additive PDBs, a more effective way of using a special class of abstractions to generate admissible heuristics by adding the heuristic value of different PDBs.

PDBs have shortcomings too. The preprocessing step of computing the lookup table can be time-consuming. Also, in order to be effective a PDB might require a considerable amount of memory for storing the lookup table.

2.2.2 Hierarchical Search

Hierarchical search algorithms compute the entries of the PDB as the search progresses [32, 33, 51]. This lazy computation of the PDB entries can be done efficiently during search by combining two ideas. The first idea is to use a hierarchy of abstractions. Hierarchical search algorithms search for the optimal solution cost of abstracted state s corresponding to node n in the original state-space by using a yet more abstracted state-space to generate a heuristic for s. That is, the optimal solution cost of a more abstracted state s' is used as heuristic to efficiently compute the optimal solution cost of s. An arbitrary number of layers of abstractions can be used by a hierarchical search algorithm. The last layer, which is the most abstracted one, is usually so small that optimal solution costs can be computed quickly without a heuristic. Caching is the second idea for making the lazy computation of the PDB entries in hierarchical search efficient. An efficient caching system must be used to avoid unnecessary search in the abstracted spaces.

While the hierarchical approach only stores in memory the abstracted states used during search, the approach might still use a considerable amount of memory.

2.2.3 Learning-Based Heuristics

The use of machine learning to create heuristics overcomes the memory limitation of abstraction-based heuristics. Learning heuristics has recently become popular, see for instance [17, 35, 76, 83]. Although learning-based heuristics overcome the memory problem, they are not guaranteed to be admissible. Thus, heuristic search algorithms using learning-based heuristics are not guaranteed to find optimal solutions.

One of the main challenges faced by such learning systems is to collect training instances; in order to learn heuristics one must know the solution cost of a sufficiently large number of problem instances. Jabbari Arfaee et al. [35] presented a learning system, Bootstrap, that generates training data through bootstrapping. Bootstrap tries to solve problem instances with a (possibly weak) initial heuristic h_0 within a time limit. The instances that the method manages to solve form a training set that is used to learn another heuristic h_1. The process is then repeated with h_1 replacing h_0, hoping that some of the instances not solved in the previous iteration will be solved and a new training set will be obtained. Jabbari Arfaee et al. showed that IDA* finds near-optimal solutions for state-space problems once a bootstrap heuristic is learned.

2.3 Type Systems for State-Space Problems

For convenience, instead of defining the type system as a partition of the states in the state space, as mentioned in Chapter 1, we define the type system as a partition of the nodes in the problem's UST.

Definition 7 (Type System)**.** *Let $S = (N, E)$ be a UST. $T = \{t_1, \ldots, t_n\}$ is a type system for S if it is a disjoint partitioning of N. If $s \in N$ and $t \in T$ with $s \in t$, we write $T(s) = t$.*

Example 2. *For the sliding-tile puzzle, for example, one could define a type system based on the position of the blank tile. In this case, two nodes n and n' would be of the same type if n has the blank in the same position as n', regardless of the configuration of the other tiles in the two nodes.*

In this dissertation all type systems we use are heuristic-based. The simplest heuristic-based type system we use is the one introduced by Zahavi et al. [84] in which two nodes n and n' are of the same type if they have the same heuristic value. We also introduce variations of Zahavi et al.'s type system. To be specific, in our type systems, in addition to accounting for the heuristic value of nodes n and n' when computing their type, we also account for the heuristic distribution of the children and sometimes even the grandchildren of n and n' in the UST when computing their type.

2.3.1 Type Systems Are Not Abstractions

A common misconception is to think of type systems as state-space abstractions. Prieditis [71] defines a state-space abstraction as a simplified version of the problem in which (1) the cost of the least-cost path between two abstracted states must be less than or equal to the cost of the least-cost path between the corresponding two states in the original state-space; and (2) goal states in the original state-space must be goal states in the abstracted state-space.

In contrast with state-space abstractions, a type system does not have these two requirements. A type system is just a partition of the nodes in the UST. It might not even be possible to represent a type system as a graph since a type system does not necessarily define the relation between the types. However, note that abstractions also offer a partition of the nodes in the UST. Therefore, even though type systems cannot necessarily be used as abstractions, abstractions can always be used as type systems.

2.4 Problem Domains

In this section we describe the problems we use as testbeds in our experiments. In addition to the problems we now describe, in Chapters 3 and 4 we also use optimization queries over graphical models such as finding the most likely explanation in Bayesian networks [69] as benchmark domains for some of our algorithms.

2.4.1 Sliding-Tile Puzzle

The sliding-tile puzzle [79] with parameters n and m consists of $n \times m - 1$ numbered tiles that can be moved in a grid. A state is a vector of length $n \times m$ in which component k names what is located in the k^{th} puzzle position (either a number in $\{1, \ldots, n \times m - 1\}$ representing a tile or a special symbol representing the blank). Every operator swaps the blank with a tile adjacent to it. The left part of Figure 2.4 shows the goal state that we used for the (4×4)-puzzle, also called 15-puzzle, while the right part shows a state created from the goal state by applying two operators, namely swapping the blank with tile 1 and then swapping it with tile 5. The number of states reachable from any given state is $(n \times m)!/2$ [3].

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

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

Figure 2.4: The goal state for the 15-puzzle (left) and a state two moves from the goal (right).

2.4.2 Pancake Puzzle

In the pancake puzzle [16] with parameter n, a state is a permutation of n numbered tiles and has $n-1$ successors, with the l^{th} successor formed by reversing the order of the first $l+1$ positions of the permutation ($1 \leq l \leq n-1$). The upper part of Figure 2.5 shows the goal state of the 15-pancake puzzle, while the lower part shows a state in which the first four positions have been reversed.

1	2	3	4	5	$\cdots$	14	15
4	3	2	1	5	$\cdots$	14	15

Figure 2.5: The goal state for the 15-pancake puzzle (above) and a state one move from the goal (below).

All $n!$ permutations are reachable from any given state.

2.4.3 Rubik's Cube

Rubik's Cube is a $3 \times 3 \times 3$ cube made up of 20 moveable $1 \times 1 \times 1$ "cubies" with colored stickers on each exposed face [46]. Each face of the cube can be independently rotated by 90 degrees clockwise or counterclockwise, or by 180 degrees. The left part of Figure 2.6 shows the goal state for Rubik's Cube while the right part shows the state produced by rotating the right face 90 degrees counterclockwise.

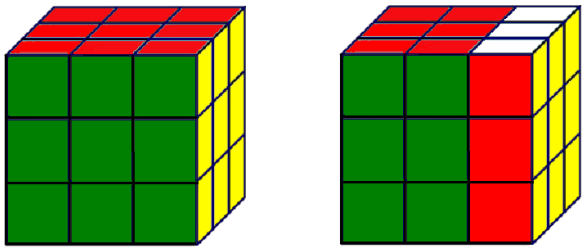

Figure 2.6: Rubik's Cube (modified from Zahavi et al. [84])

2.4.4 Towers of Hanoi

The goal in the Towers of Hanoi is to move all the disks from the original position onto a single specific peg. Only one disk can be moved at a time from the top of a peg onto another peg. A larger disk cannot be placed on top of a smaller disk. See Figure 2.7 for an example of the goal state of the Towers of Hanoi with 5 disks and 4 pegs. We ran experiments with the 12-disks and 4-pegs Towers of Hanoi, which has 4^{12} reachable states from the goal state [49]. The average branching factor of the 4 pegs Towers of Hanoi is about 3.766 [47].

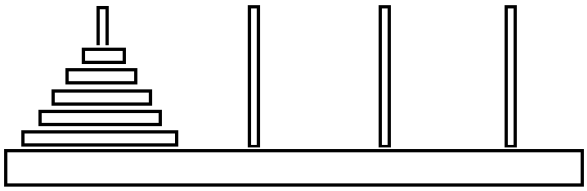

Figure 2.7: The 5-disks 4-pegs Towers of Hanoi.

2.4.5 Blocks World

In the blocks world domain one is given a set of n blocks and the goal is to have the blocks stacked up in a specific order. The blocks world is illustrated in Figure 2.8, where the blocks on the lefthand side of the figure represent some initial configuration of the problem and the blocks on the righthand side represent the goal configuration.

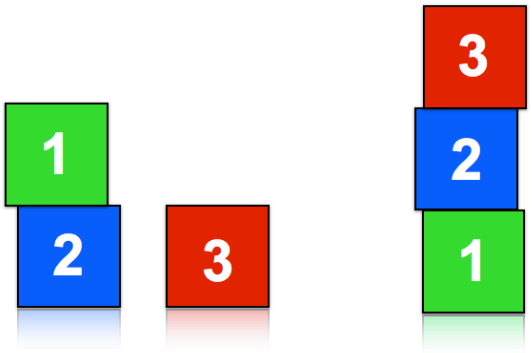

Figure 2.8: Blocks world with three blocks.

A solution for the problem shown in Figure 2.8 is to unstack block number 1 from block number 2; stack block number 2 on block number 1; finally, stack block number 3 on block number 2. We are interested in either optimal or near-optimal solutions for this problem.

Depending on the number n of blocks this domain can have very large state spaces. For instance, when using $n = 20$ this domain has approximately 10^{20} different states.

2.4.6 Discussion

These problems represent a large class of important real-world problems. For instance, the sliding-tile puzzle is a congested version of the pebble motion problem occurring in multi-robot motion planning; the blocks world can be seen as a warehouse problem in which a robot has to arrange the packages in a specific order before packing them into a truck so as to facilitate delivery.

Taken together, these domains offer a good challenge for state-space search algorithms as they have different properties. For instance, the sliding-tile puzzle has deeper solutions and a smaller branching factor than the other four domains. The pancake puzzle and the Rubik's Cube have a constant branching factor, the sliding-tile puzzle and the Towers of Hanoi have small variance in its branching factor, and the blocks world's branching factor can vary considerably from one state to another and can be in the hundreds when using 20 blocks.

Chapter 3

Predicting the Size of Iterative-Deepening-A* Search Trees

3.1 Introduction

A frequent impediment of the application of tree searching algorithms [68] is the inability to quickly predict the runtime of an algorithm on a particular problem instance [38]. While one problem instance might be solved in a blink of an eye, another instance of the same problem might take centuries.

One approach to predicting the runtime of search algorithms is to estimate the size of the algorithm's Expanded Search Tree (EST) [38]. In addition to the merit of estimating the search runtime, fast and accurate predictions of the EST size could also be used to fairly divide the search workload among different processors in a parallel computing setting. Such predictions could also be used to select the most suitable heuristic function to guide the search algorithm: is it better to use a slow and accurate heuristic, or a fast and inaccurate one?

In this chapter we are interested in predicting the EST size of a popular heuristic search algorithm, Iterative-Deepening-A* (IDA*) [43].

3.1.1 Problem Formulation

Given a directed and implicitly defined search tree representing a state-space problem [65], we want to estimate the size of the subtree expanded by IDA* with a given cost bound d, assuming no goal is found during that iteration. We call the nodes expanded by IDA* with cost bound d *the Expanded Search Tree* or, EST, for short.

Problem formulation. Let $S = (N, E)$ be a tree representing an EST where N is its set of nodes and for each $n \in N$ $child(n) = \{n' | (n, n') \in E\}$ is its set of child nodes. Our task is to estimate

the size of N without fully generating S.

3.1.2 Background

Korf, Reid, and Edelkamp [50] launched a line of research aimed at creating a method to predict exactly the IDA* EST size for given cost bound d and heuristic function h. This was in contrast with the traditional approach to search complexity analysis, which focused on "big-O" complexity typically parameterized by the accuracy of the heuristic [15, 22, 68, 70]. Korf, Reid, and Edelkamp developed a prediction formula, known as `KRE` as a reference to the author's names, for the special case of consistent heuristics. Zahavi et al. [84] created Conditional Distribution Prediction (`CDP`), an extension of `KRE` that makes accurate predictions when the heuristic employed is either consistent or inconsistent. `CDP` works by sampling the state space as a preprocessing step with respect to a *type system*, i.e., a partition of the nodes in the UST. The information learned during sampling is used to efficiently emulate the IDA* search tree and thus to approximate the number of nodes expanded on an iteration of the algorithm. `CDP` is reviewed in Section 3.2.

Independent of the `KRE` line of research, Knuth [38] created a method to efficiently predict the size of a search tree by making random walks from the root node. Knuth's assumption was that branches not visited would have the same structure as the single branch visited by the random walk. Despite its simplicity, Knuth proved his method to be efficient in the domains tested. However, as pointed out by Knuth himself, his method does not produce accurate predictions when the tree being sampled is unbalanced. Chen [11] extended Knuth's method to use a *stratification* of the nodes in the UST to reduce the variance of sampling. Like `CDP`'s type systems, the stratification used by Chen is a partition of the nodes in the UST. The method developed by Chen, Stratified Sampling, or `SS`, relies on the assumption that nodes in the same part of the partition will root subtrees of the same size. `SS` is reviewed in detail in Section 3.3.

3.1.3 Contributions

Most of the content of this chapter is published in the Proceedings of the Symposium on Combinatorial Search (2011) [54], in the Proceedings of the Conference on Artificial Intelligence (2012) [57], in the Artificial Intelligence journal (2013) [58].

In this chapter we make the following contributions to Korf et al.'s and Knuth's lines of research.

- We identify a source of prediction error that has hitherto been overlooked in the `CDP` system. We call it the "discretization effect". We also disprove the intuitively appealing idea, specifically asserted by Zahavi et al., that a "more informed" system cannot make worse predictions

than a "less informed" system.[1] The possibility of this statement being false follows directly from the discretization effect, because a more informed system is more susceptible to the discretization effect than a less informed one. We will show several cases of this statement being false and use the phrase "informativeness pathology" to refer to this situation.

- We present a method for counteracting the discretization effect, which we call "ϵ-truncation". One way to view ϵ-truncation is that it makes a prediction system less informed, in a carefully chosen way, so as to improve its prediction accuracy by reducing the discretization effect. In our experiments ϵ-truncation rarely degraded predictions; in the vast majority of cases it improved the prediction accuracy, often substantially.

- Another contribution to the CDP system is an algorithmic improvement that reduces its running time. Our CDP variant, named Lookup CDP (or L−CDP for short), decomposes a CDP prediction into independent subproblems. The solutions to these subproblems are computed in a preprocessing step and the results stored in a lookup table to be reused later during prediction. L−CDP can be orders of magnitude faster than CDP while it is guaranteed to produce the same predictions as CDP. Similar to a PDB [13], L−CDP's lookup table is computed only once, and the cost of computing it can be amortized by making predictions for a large number of instances. ϵ-truncation and L−CDP are orthogonal to each other as the former improves the prediction accuracy and the latter improves the prediction runtime of CDP.

- In addition to the contributions to the CDP prediction method, we connect the KRE line of research to that of SS. We do so by showing that the type systems employed by CDP can also be used by SS. Our empirical results show that SS employing CDP's type systems, which use a heuristic function, substantially improves upon the predictions produced by SS using the type system proposed by Chen.

- We make an empirical comparison of our enhanced version of CDP and SS using heuristic-based type systems. In our empirical comparison we consider scenarios that require (1) fast, and (2) accurate predictions. The first scenario represents applications that require less accurate but almost instantaneous predictions. For instance, quickly estimating the size of search subtrees would allow one to fairly divide the search workload among different processors in a parallel processing setting. The second scenario represents applications that require more accurate predictions but allow more computation time. Our experimental results suggest that

[1] "More informed" is defined formally in Definition 11 below.

if L−CDP's preprocessing time is acceptable or can be amortized, it is suitable for applications that require less accurate but very fast predictions, while SS is suitable for applications that require more accurate predictions but allow more computation time.

- We show how to use the k-means clustering algorithm for creating type systems for domains with real-valued heuristic functions. Our empirical results show that SS using a clustering-based type system tends to produce better predictions than the prediction methods that do not use a type system.

- We also present Active Stratified Sampling (Active SS), a prediction algorithm that uses the ideas of active sampling for predicting the size of IDA* search trees. In contrast with other active sampling approaches [18], it is not immediately clear that Active SS can perform better than regular SS due to its relatively expensive bookkeeping procedure. Our empirical results show that Active SS can perform better than regular SS.

3.1.4 Chapter Outline

The remainder of this chapter is organized as follows. We present the CDP prediction framework and our contributions to that framework in Section 3.2. In Section 3.3 we present SS and we explain how we connect SS to CDP. In Section 3.4 we make a comparison between our enhanced versions of CDP and SS. Then, we show how to create effective type systems for domains with real-valued heuristics with a clustering algorithm in Section 3.5. In Section 3.6 we present Active SS, a variation of SS that uses the ideas of active sampling to produce more accurate predictions of the size of the IDA* search tree. In Section 3.7 we mention the related work. Finally, we draw the conclusions of the chapter in Section 3.8.

3.2 The CDP Prediction Framework

The CDP prediction method uses the C&C paradigm to estimate the size of the IDA* EST. The accuracy of the CDP formula is based on the assumption that two nodes of the same type root subtrees of the same size.

IDA* with parent pruning will not generate a node $\hat{s}$ from s if $\hat{s}$ is the parent of s. Thus, the subtree below a node s depends on the parent from which s was generated. Zahavi et al. [84] use the information of the parent of a node s when computing s's type so that CDP is able to make accurate predictions of the number of nodes expanded on an iteration of IDA* when parent pruning is used.

Note that, as in Zahavi et al.'s work, all type systems we use with CDP are *heuristic-preserving*.

Definition 8 (Heuristic-Preserving Type System)**.** *A type system T is said to be heuristic-preserving if for every type, all nodes of that type have the same heuristic value. We then write $h(t)$ for any type $t \in T$ to denote the heuristic value of the nodes of type t.*

Definition 9. *Let $t, t' \in T$. For a given UST, $p(t'|t)$ denotes the average fraction of the children generated by a node of type t that are of type t' and b_t is the average number of children generated by a node of type t.*

Example 3. *If a node of type t generates 5 children on average ($b_t = 5$) and 2 of them are of type t' on average, then $p(t'|t) = 0.4$.*

CDP samples the state space in order to estimate $p(t'|t)$ and b_t for all $t, t' \in T$. CDP does its sampling as a preprocessing step and although type systems are defined for nodes in the UST, sampling is done before knowing the UST's start state s^*. This is achieved by considering a state s drawn randomly from the state space as the parent of nodes in a UST. As explained above, due to parent-pruning, CDP uses the information about the parent of a node when computing the type of the node. Therefore, when estimating the values of $p(t'|t)$ and b_t, the sampling is done based on the children of the state s drawn randomly from the state space, as though s and its children were part of a UST . We denote by $\pi(t'|t)$ and β_t the respective estimates thus obtained. The values of $\pi(t'|t)$ and β_t are used to estimate the number of nodes expanded on an iteration of IDA*. The predicted number of nodes expanded by IDA* with parent pruning for start state s^*, cost bound d, heuristic h, and type system T is formalized as follows.

$$\texttt{CDP}(s^*, d, h, T) = 1 + \sum_{s \in child(s^*)} \sum_{i=1}^{d} \sum_{t \in T} N(i, t, s, d)\,. \tag{3.1}$$

Here the outermost summation iterates over the children of the start state s^*. Assuming unit-cost edges, in the middle summation we account for g-costs from 1 to the cost bound d; any value of i greater than d would be pruned by IDA*. The innermost summation iterates over the types in T. Finally, $N(i, t, s, d)$ is the number of nodes n with $T(n) = t$ occurring at level i of the search tree rooted at s. A value of one is added to Equation 3.1 as CDP expands the start state so that the type of its children can be computed. $N(i, t, s, d)$ is computed recursively as follows.

$$N(1, t, s, d) = \begin{cases} 0 & \text{if } T(s) \neq t\,, \\ 1 & \text{if } T(s) = t\,, \end{cases} \tag{3.2}$$

The case $i = 1$ is the base of the recursion and is calculated based on the types of the children of the start state. For $i > 1$, the value $N(i, t, s, d)$ is given by

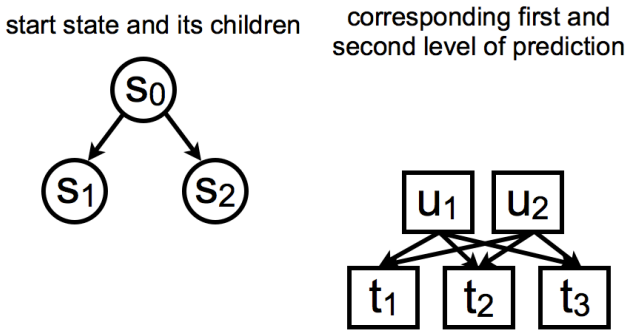

Figure 3.1: The first step of a CDP prediction for start state s_0.

$$\sum_{u \in T} N(i-1, u, s, d)\pi(t|u)\beta_u P(t, i, d)\,. \tag{3.3}$$

Here $\pi(t|u)\beta_u$ is the estimated number of nodes of type t a node of type u generates; P is a pruning function that is 1 if the cost to reach type t plus the type's heuristic value is less than or equal to the cost bound d, i.e., $P(t, i, d) = 1$ if $h(t) + i \leq d$, and is 0 otherwise.

Example 4. *Consider the example in Figure 3.1. Here, after sampling the state space to calculate the values of $\pi(t|u)$ and β_u, we want to predict the IDA* EST size with cost bound d for start state s_0. We generate the children of s_0, depicted in the figure by s_1 and s_2, so that the types that will seed the prediction formula can be calculated. Suppose that $T(s_1) = u_1$ and $T(s_2) = u_2$ and that IDA* does not prune s_1 or s_2, the first level of prediction will contain one node of type u_1 and one of type u_2, represented by the two upper squares in the right part of Figure 3.1. We now use the values of π and β to estimate the number of nodes of each type on the next level of search. For instance, to estimate how many nodes of type t_1 there will be on the next level of search we sum up the number of nodes of type t_1 that are generated by nodes of type u_1 and u_2. Thus, the estimated number of nodes of type t_1 at the second level of search is given by $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$. Note that $N(1, u_i, s_i, d) = 1$ for $i \in \{1, 2\}$, $N(1, u_1, s_2, d) = N(1, u_2, s_1, d) = 0$, and $N(1, u, s, d) = 0$ for all other pairs of values of u and s. Thus $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$ equals $\sum_{u \in T} N(1, u, s_1, d)\pi(t_1|u)\beta_u + \sum_{u \in T} N(1, u, s_2, d)\pi(t_1|u)\beta_u$.*

If $h(t_1) + 2$ (heuristic value of type t_1 plus the cost of reaching t_1) exceeds the cost bound d, then the number of nodes of type t_1 is set to zero, because IDA would have pruned those nodes. This process is repeated for all types at the second level of prediction. Similarly, we get estimates for the third level of the search tree. Prediction goes on until all types are pruned. The sum of the estimated number of nodes of every type at every level is the estimated number of nodes expanded*

by IDA with cost bound d for start state s_0.*

According to the formulae above and the example in Figure 3.1, in order to predict the number of nodes IDA* expands with a cost bound d, for every level $i \leq d$, CDP predicts how many instances *of each type* will be generated; i.e., it predicts a vector $(N[1], \ldots, N[|T|])$ of numbers of instances of each type on a level.[2] We will call such a vector a *type allocation vector*. The type allocation vector for the first level of prediction is computed from the types of the children of the start state (the $i = 1$ base case of the recursive calculation shown above). Once the allocation vector is calculated for the first level, the vector for the next level is calculated according to Equation 3.3. At level i, for each type t such that $h(t) + i$ exceeds the cost bound d, the corresponding entry in the type allocation vector, $N[t]$, is set to zero to indicate that IDA* will prune nodes of this type from its search. The prediction continues to deeper and deeper levels as long as at least one entry in the type allocation vector is greater than zero.

The prediction lookahead r. CDP is seeded with the types of the children of the start state s^*, as shown in Equation 3.1. Zahavi et al. [84] showed that seeding the prediction formula with nodes deeper in the search tree improves the prediction accuracy at the cost of usually increasing the prediction runtime. In this improved version of CDP one collects C_r, the set of nodes s such that s is at a distance $r < d$ from s^*. Then the prediction is made for a cost bound of $d - r$ when nodes in C_r seed CDP. In our experiments we also used this improved version of CDP.

3.2.1 Attempt to Improve CDP Predictions

Zahavi et al. derived a set of conditions for which CDP is guaranteed to make perfect predictions ([84], Section 4.5.1, p. 60). This set of conditions can be generalized with the definition of the *purity* of a type system, i.e., if a type system is *pure*, then CDP predictions are guaranteed to be perfect.

Definition 10. *A heuristic-preserving type system T is said to be pure if node n has exactly $p(t'|t) \times b_t$ children of type t' for all $t, t' \in T$ and all $n \in t$.*

Intuitively, there is no uncertainty in the prediction model when a pure type system is employed, and as stated by Zahavi et al., a simple proof by induction shows that a pure type system results in perfect predictions.

A trivial example of a pure type system is a one-to-one mapping from the original state space S to the type space T. In this case, every node in the search tree is of a different type. Such a type system is not of interest for large state spaces and d-values because the prediction calculations would

[2] We use $N[t]$ or $N(i, t)$ to denote $N(i, t, s^*, d)$ when i, s^* and d are clear from the context.

be too costly. Unfortunately, type systems that are pure and compact are hard to obtain in practice. For instance, Zahavi et al.'s basic "two-step" model defined below (in our notation) is not pure, as verified in their experiments.

$$T_h(s) = (h(parent(s)), h(s)), \tag{3.4}$$

where $parent(s)$ returns the parent of s in the UST.

In a first attempt to improve CDP's prediction accuracy we used "more informed" type systems, i.e., type systems that split every type in T_h into a set of types. Two new domain-independent type systems we introduce, which are "more informed" than T_h, are:

$$T_c(s) = (T_h(s), c(s,0), \ldots, c(s,H)), \tag{3.5}$$

where $c(s,k)$ is the number of children of s, considering parent pruning, whose h-value is k, and H is the maximum h-value observed in the sampling process; and

$$T_{gc}(s) = (T_c(s), gc(s,0), \ldots, gc(s,H)), \tag{3.6}$$

where $gc(s,k)$ is the number of grandchildren of s, considering parent pruning, whose h-value is k.

For instance, two nodes s and s' will be of the same T_c type (where c stands for *children*) if $h(parent(s)) = h(parent(s'))$ and $h(s) = h(s')$, and, in addition, s and s' generate the same number of *children* with the same heuristic distribution. Similarly, two nodes s and s' are of the same T_{gc} type (where gc stands for grandchildren) if besides matching on the information required by the T_c type system, s and s' generate the same number of *grandchildren* with same heuristic distribution.

The intuitive concept of one type system being "more informed" than another is captured formally as follows.

Definition 11. *Let T_1, T_2 be type systems. T_1 is a refinement of T_2, denoted $T_1 \preceq T_2$, if $|T_1| \geq |T_2|$ and for all $t_1 \in T_1$ there is a $t_2 \in T_2$ with $\{s \mid T_1(s) = t_1\} \subseteq \{s \mid T_2(s) = t_2\}$. If $t_1 \in T_1$ and $t_2 \in T_2$ are related in this way, we write $T_2(t_1) = t_2$.*

Note that $T_{gc} \preceq T_c \preceq T_h$, and so, by transitivity, $T_{gc} \preceq T_h$.

Example 5. *Consider the grid domain depicted in Figure 3.2. In this domain an agent lies on an infinite grid and wants to arrive at the goal position; Figure 3.2 shows the agent starting at position $(3,3)$ and the goal at position $(0,0)$. The agent can move to one of the four adjacent positions,*

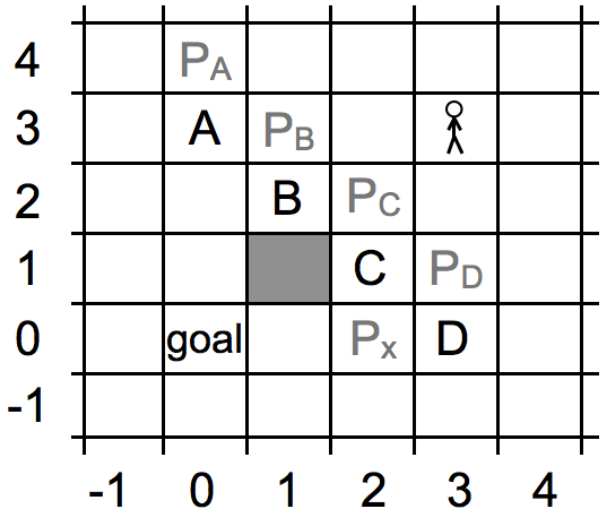

Figure 3.2: Grid example of type systems.

except when the position is blocked by a wall; in Figure 3.2 a gray square represents a wall. If Manhattan Distance is the heuristic function used in this domain, then the heuristic value of the agent's state is 6 (3 from the x-coordinate plus 3 from the y-coordinate).

Consider a type system in which nodes with the same heuristic value are of the same type. In this case, the nodes representing the states in which the agent is at positions A, B, C, and D would be of the same type. Note, however, that nodes representing these four states are not necessarily of the same T_h type. Recall that T_h uses both the heuristic value of the node and of the node's parent in the UST. A, B, C, and D will be of the same T_h type if they are generated by P_A, P_B, P_C and P_D, respectively. In this case, A, B, C, and D have a heuristic value of 3 and are generated by nodes with heuristic value of 4, resulting in the $(4, 3)$ type.

Consider again nodes A, B, C, and D when they are generated by parents P_A, P_B, P_C and P_D, respectively, and are therefore of the same type—$(4, 3)$—according to T_h. The T_c type system, which is a refinement of T_h, further partitions the nodes that are of the same type according to T_h. In our example, according to the T_c type system, nodes A and D are of different type than B and C. A and D are of the same T_c type because, with parent-pruning, they both generate two children with heuristic value of 4 (for A these are grid cells $(-1, 3)$ and $(1, 3)$) and one child with heuristic value of 2 (for A this is grid cell $(0, 2)$). B and C are of another T_c type: with parent-pruning both generate one child with heuristic value of 4 and one child with heuristic value of 2.

Intuitively, if $T_1 \preceq T_2$ one would expect predictions using T_1 to be at least as accurate as the predictions using T_2, since all the information that is being used by T_2 to condition its predictions

is also being used by T_1 ([84], p. 59). However, our experiments show that this is not always true. The underlying cause of poorer predictions by T_1 when $T_1 \preceq T_2$ is the discretization effect, which we will now describe.

3.2.2 The Discretization Effect

In this section we identify a source of error in CDP's predictions that has previously gone unnoticed, which we call the discretization effect. Understanding this source of error allows us to propose a method to counteract it—the ϵ-truncation method that is fully described in the next section.

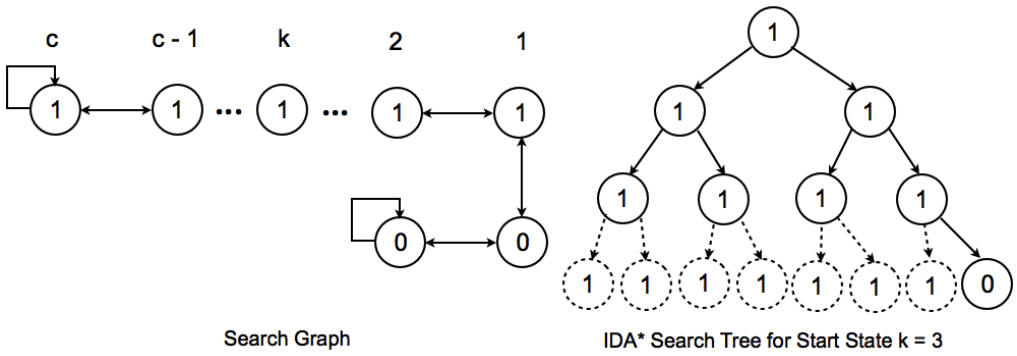

Figure 3.3: Example of the discretization effect. Numbers represent the heuristic value of a state, and states with heuristic value of zero are goal states. The search graph has c states with heuristic value of one and two goal states. The right part shows the search tree of the last iteration of IDA* with start state $k = 3$ and cost bound $d = 3$; dashed lines represent pruned nodes.

Consider the state space shown in the left part of Figure 3.3. Each circle is a state, and the number inside a circle is its heuristic value. All states in this space have two neighbors. There is a chain containing $c+2$ states, terminating at each end with a self-loop, with two goal states at one end of the chain having a heuristic value of 0 and the remaining c states having a heuristic value of 1. In this example we ignore parent pruning and we use the following type system: $T(s) = h(s)$. Hence there are only two types: t_0 is the type of the nodes with heuristic value 0, and t_1 is the type of the nodes with heuristic value 1. The key feature of this construction is that the probability of a node of type t_1 generating a node of type t_0 can be made arbitrarily small by making c sufficiently large. Table 3.1 shows the exact probabilities of each kind of transition between types. CDP's sampling would estimate these; Table 3.2 shows that the estimates (to two decimal places) based on sampling all 12 states when $c = 10$ are equal to the theoretical values.

We are interested in the prediction for a single start state, the k^{th} node with heuristic value 1. Note that the solution depth for this state is $d = k$. The IDA* search tree for start state $k = 3$ and cost bound $d = 3$ is shown in the right part of Figure 3.3. Dashed lines represent nodes that are

$p(t_0\|t_1)$	$\frac{1}{2c}$
$p(t_1\|t_1)$	$\frac{2c-1}{2c}$
$p(t_0\|t_0)$	$\frac{3}{4}$
$p(t_1\|t_0)$	$\frac{1}{4}$

Table 3.1: Exact type-transition probabilities for the graph shown in Figure 3.3.

generated but not expanded because their f-value exceeds d. When $d = k$, as in the figure, level i contains 2^i expanded nodes, for $0 \leq i < d$, and level d contains one expanded node (the one with heuristic value 0 is counted as being expanded), so the total number of expanded nodes is 2^d.

$p(t_0\|t_1)$	0.05
$p(t_1\|t_1)$	0.95
$p(t_0\|t_0)$	0.75
$p(t_1\|t_0)$	0.25

Table 3.2: Transition probabilities estimated by sampling the state space in Figure 3.3 when $c = 10$.

For start state $k = 4$, cost bound $d = 4$, and $c = 10$, IDA* expands $2^4 = 16$ nodes but CDP, using the exact transition probabilities (see Table 3.2), predicts it will expand 17.0264, an error of 1.0264. Table 3.3 summarizes CDP predictions at each level. We see that CDP's predictions of the total number of nodes expanded at each level (rightmost column) is perfect at all levels except the last. The cause of the error can be seen in the middle two columns, which show the predicted number of nodes expanded at each level of a particular type. At every level CDP is overestimating the number of nodes of type t_0 and correspondingly underestimating the number of nodes of type t_1. This occurs because one of the ten possible start states of type t_1 (namely, $k = 1$) would generate one child of type t_0, while all the others would generate none. This is represented in CDP's predictions by saying that every start state of type t_1 generates 0.1 children of type t_0. For start states other than $k = 1$, probability mass is being drawn away from the true prediction by an event that happens rarely for that type. Moreover, the error is compounded at each successive level in the tree: already at level 3, the percentage of nodes of type t_0 has risen to more than 10% of the nodes at that level, whereas at level 1 only 5% of the nodes are predicted to be of type t_0. These errors are invisible until IDA*'s pruning starts to treat nodes of type t_0 different than nodes of type t_1. In the example, this happens at level 4, where nodes of type t_1 are pruned but nodes of type t_0 are not.

Note that we would obtain a smaller prediction error if we were to totally ignore the rare event of t_1 generating a child of type t_0 by artificially setting the $p(t_0|t_1)$ entry in Table 3.2 to 0.0 and

level	t_0	t_1	total by level
0	0	1	1
1	0.1	1.9	2
2	0.34	3.66	4
3	0.876	7.124	8
4	2.0264	-	2.0264
total	3.3424	13.684	**17.0264**

Table 3.3: CDP prediction level by level when $c = 10$, $k = 4$, and $d = 4$. The dash represents a type and level that was pruned off.

renormalizing $p(t_1|t_1)$ (it would become 1.0). With this change CDP would get exactly the right total number of nodes on all levels except level 4, where it would predict that 0 nodes are expanded (since it thinks they are all of type t_1). This is an error of 1.00 compared to the error of 1.0264 obtained using the actual values in Table 3.2.

The method in the next section is based on this idea of altering the estimated probabilities so that rare events are ignored and common events correspondingly have their probabilities increased. However, we do this in a more sophisticated manner than having a global threshold to identify rare events. We use an optimization method to define a threshold, ϵ_i, to be used at level i. This allows different levels to have different definitions of "rare event". Applied to the example in this section, our method will set $p(t_0|t_1)$ to 0 at levels 0 to 3, but will leave it as 0.05 at level 4. This keeps the predictions perfect at levels 0 to 3 and predicts that there will be 0.8 ($0.05 \cdot 8 \cdot 2$) nodes expanded at level 4, for an error of 0.2.

3.2.3 The ϵ-Truncation Prediction Method

Our ultimate goal is to minimize the error in predicting the number of nodes expanded by IDA*. Section 3.2.2 suggests that this requires avoiding the discretization effect—by ignoring small probabilities of generating nodes of a certain type. The CDP system does not ignore small fractional numbers. By contrast, there is a chance that minimizing the absolute error for type allocation vectors might force the system to ignore some of these harmful small numbers.

It is hence natural to consider modifying the π-values used as estimates in CDP according to the following procedure (P1).

Procedure (P1):

For each type $u \in T$ do:

1. Compute a redistribution of $\pi(t|u)$ values, by solving the following optimization problem for each level i.

Find a type allocation vector $(a_1, \ldots, a_{|T|})$ *that minimizes:*

$$\sum_{t \in T} \sum_{j=0}^{\lceil N(i,u,s^*,d) \cdot \beta_u \rceil} pr(j, t, u, \lceil N(i, u, s^*, d) \cdot \beta_u \rceil) \cdot |(a_t - j)| \qquad (3.7)$$

subject to the following constraints:

$$\sum_{t \in T} a_t = \lceil N(i, u, s^*, d) \cdot \beta_u \rceil \text{ and } a_t \geq 0 \text{ for } t \in T\,.$$

Here $pr(j, t, u, N)$ is short for the estimated probability of generating exactly j children of type t from N many parents of type u, i.e.,

$$pr(j, t, u, N) = \pi(t|u)^j (1 - \pi(t|u))^{N-j}\,.$$

2. For each $t' \in T$, replace $\pi(t'|u)$ by $a_{t'} / \sum_{t \in T} a_t$.

However, Procedure (P1) is flawed. For any type t, it ignores the possibility that a large number of states of distinct types occurring at one level can all generate a state of type t with low probability, summing up to a large probability of having a state of type t at the next level.

Example 6. *Suppose at prediction level* i *one node each of 100 different types* $t_1, \ldots, t_{100}$ *occurs. Suppose further that each of these 100 types generates one node of type* t *with probability* 0.01*. Procedure (P1) would correspond to solving at least 100 optimization problems (one for each type), potentially replacing* $\pi(t|t_i)$ *by 0 for all* $i \in \{1, \ldots, 100\}$*. Consequently, the prediction system might suggest that no node of type* t *occurs at level* $i+1$*. However, it would be better to consider the interaction of the 100 types* $t_1, \ldots, t_{100}$ *at level* i *and to predict that one node of type* t *will occur at level* $i + 1$*.*

In order to take into account that, for some type t, nodes of different types t_1 and t_2 may both generate nodes of type t at any level of prediction, we need to reformulate Procedure (P1) using a system of "supertypes."

Supertypes

Intuitively, a supertype at level i is a set of pairs (t, u) of types where type t occurs at level i and type u is generated at level $i + 1$.

Example 7. *We now illustrate the concept of supertypes with the example of the grid domain shown in Figure 3.2. We adopt the* T_h *type system in this example. Consider that at a given level of search we see node* B *of type* $(4, 3)$*, with* B *being generated by moving the agent from* P_B *to* B*, and node*

D of type $(2,3)$, with D being generated by moving the agent from P_X to D. The pairs of types $\big((4,3),(3,4)\big)$ and $\big((2,3),(3,4)\big)$ will be in the same supertype as both B of type $(4,3)$ and D of type $(2,3)$ can generate a node of type $(3,4)$—from B the agent could move to P_C and from D the agent could move to P_D.

The following conditions should be met by the supertypes:

- If the set of types that can be generated by a type t_1 occurring at level i overlaps with the set of types that can be generated by a type t_2 occurring at level i—say both sets contain type u—then (t_1, u) and (t_2, u) will be in the same supertype.

  Consider for instance the four trees in Figure 3.4. Suppose nodes s_1 and s_2 of types $t_1 = T_{gc}(s_1)$ and $t_2 = T_{gc}(s_2)$, respectively, occur at level i of the prediction. The types t_1 and t_2 can potentially generate pairs of type $t_3 = T_{gc}(s_3)$ and pairs of type $t_4 = T_{gc}(s_4)$ at level $i+1$ (following the left branches of tree$_1$ and tree$_2$, framed in boxes in the figure). Hence we would like to put (t_1, t_3) and (t_2, t_3) in one supertype for level i; similarly we would put (t_1, t_4), and (t_2, t_4) in one supertype for level i.

- Any type u at level $i+1$ will be generated by a single supertype at level i. This is achieved by taking information from a coarser type system into account. If t_1 and t_2 at level i can generate type u_1 and u_2, respectively, at level $i+1$, and u_1 and u_2 are indistinguishable in a fixed *coarser* type system, then (t_1, u_1) and (t_2, u_2) will be in the same supertype for level i.

  In the example in Figure 3.4, both s_1 (of type t_1) and s_2 (of type t_2) generate nodes of type $T_c(s_1') = T_c(s_2')$ in the coarser type system T_c (see the framed boxes in tree$_1$ and tree$_2$). These state pairs at level $i+1$ (see the framed boxes in tree$_3$ and tree$_4$) could be of type t_3 or t_4 in the (refined) type system T_{gc}. Thus (t_1, t_3), (t_2, t_3), (t_1, t_4), and (t_2, t_4) will all be in a single supertype for level i.

Formally, we define supertype systems as follows.

Definition 12. *Let T, T' be type systems, $T \preceq T'$, and $t' \in T'$. For all i, the supertype $st(t', i, s)$ over T contains exactly the pairs $(t_1, t_2) \in T \times T$ for which $T'(t_2) = t'$ and t_1 occurs at level i starting the prediction from s. The supertype system $ST(i, s)$ over T with respect to T' is defined by $ST(i, s) = (st(t_1', i, s), \ldots, st(t_z', i, s))$ where $T' = \{t_1', \ldots, t_z'\}$. We write st instead of $st(t', i, s)$ whenever t', i, s are clear from context.*

Let T, T' be type systems such that $T \preceq T'$ that induce the supertype $ST(i, s^*)$. In order to adapt `CDP` and Procedure (P1) to supertypes, we estimate, for each node s^*, level i, cost bound d,

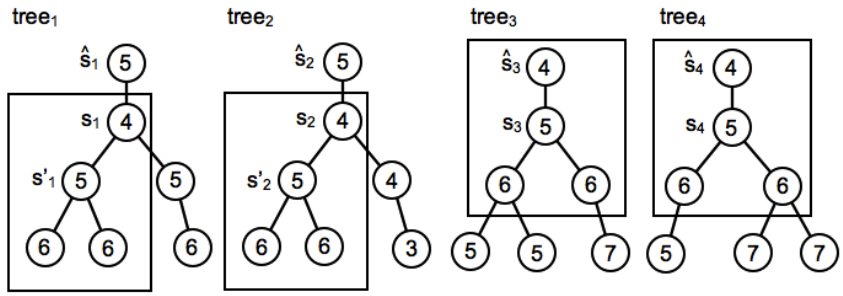

Figure 3.4: Four potential search subtrees for a search problem. Numbers inside the nodes denote h-values. For $j \in \{1, 2, 3, 4\}$, tree$_j$ determines the type $T_{gc}(s_j)$. The subtrees framed in boxes determine the type $T_c(s'_1) = T_c(s'_2) = T_c(s_3) = T_c(s_4)$.

type t, and supertype $st \in ST(i, s^*)$, the probability of generating a node of type t from a node of supertype st at level i. We denote this estimate by $\pi^{i,d}_{s^*}(t|st)$, defined by

$$\pi^{i,d}_{s^*}(t|st) = \frac{\sum_{\{t_p|(t_p,t)\in st\}} \pi(t|t_p)\beta_{t_p} N(i, t_p, s^*, d)}{\sum_{\{t_p|(t_p,t)\in st\}} \beta_{t_p} N(i, t_p, s^*, d)} .$$

We write $\pi(t|st)$ instead of $\pi^{i,d}_{s^*}(t|st)$, whenever i, d, and s^* are clear from the context.

The number of nodes $N_{ST}(i, st, s^*, d)$ of a supertype $st \in ST$ at a level i of prediction is given by

$$N_{ST}(i, st, s^*, d) = \sum_{(t_p,t_c)\in st} N(i, t_p, s^*, d)\beta_{t_p} . \tag{3.8}$$

We then reformulate the `CDP` formula equivalently, computing $N(i, t, s^*, d)$ by

$$N(i, t, s^*, d) = \sum_{st\in ST} N_{ST}(i-1, st, s^*, d)\pi^{i-1,d}_{s^*}(t|st)P(t, i, d) , \tag{3.9}$$

instead of Equation (3.3). Note that, in contrast with Equation (3.3), Equation (3.9) does not include any β-values as factors, due to the fact that those are already incorporated in the calculation in the N_{ST} values in Equation (3.8).

Procedure (P1) would then be adapted to the following procedure (P2).

Procedure (P2):

For each supertype $st \in ST$ at any level i do:

1. Compute a redistribution of $\pi(t|st)$ values, by solving the following optimization problem.

   Find a type allocation vector $(a_1, \ldots, a_{|T|})$ *that minimizes:*

$$\sum_{t \in T} \sum_{j=0}^{\lceil N_{ST}(i,st,s^*,d) \rceil} pr(j, t, st, \lceil N_{ST}(i, st, s^*, d) \rceil) \cdot |(a_t - j)| \qquad (3.10)$$

subject to the following constraints:

$$\sum_{t \in T} a_t = \lceil N_{ST}(i, st, s^*, d) \rceil \text{ and } a_t \geq 0 \text{ for } t \in T\,.$$

Here $pr(j, t, st, N)$ is short for the probability of generating exactly j children of type t from N many parents of *supertype* st, i.e.,

$$pr(j, t, st, N) = \pi(t|st)^j (1 - \pi(t|st))^{N-j}\,.$$

2. For each each $t' \in T$, replace $\pi(t'|st)$ by $a_{t'} / \sum_{t \in T} a_t$.

ϵ-Truncation as a Preprocessing Step

Ideally, one would now follow Procedure (P2) at every step of the prediction. However, although the optimization problem can be solved in polynomial time, solving distinct instances at every step of prediction is computationally prohibitive. For example, in an experiment we ran on the (4x4) 15 sliding-tile puzzle, following Procedure (P2) at every step of the prediction was almost three orders of magnitude slower than `CDP` without the optimization. We hence developed a method that sacrifices the optimality of the optimization problem in Procedure (P2) for feasibility, by redistributing $\pi(t|st)$ values and $\pi(t|u)$ values only in a pre-processing step. The goal of this pre-processing step is to find, for each level i, a cutoff value ϵ_i, below which $\pi(t|u)$ values will be set to zero. Our approach, called ϵ-truncation, can be summarized as follows.

1. As before, sample the state space to obtain $\pi(t|u)$ and β_u for each $t, u \in T$.

2. For each level $i \in \{1, \ldots, d\}$ compute a cutoff value ϵ_i. *(This step will be explained in detail below.)*

3. For each level i and each $t, u \in T$, replace the estimate $\pi(t|u)$ by an estimate $\pi_i(t|u)$ that is specific to level i. $\pi_i(t|u)$ is determined as follows.

   (a) If $\pi(t|u) < \epsilon_i$ then $\pi_i(t|u) = 0$.

   (b) If $\pi(t|u) \geq \epsilon_i$ then

   $$\pi_i(t|u) = \frac{\pi(t|u)}{\sum_{v \in T, \pi(v|u) \geq \epsilon_i} \pi(v|u)}\,.$$

   Thus the $\pi(t|u)$ values not smaller than ϵ_i are scaled so that they sum up to 1.

4. In computing CDP use $\pi_i(t|u)$ at level i instead of $\pi(t|u)$.

The key step in this process is Step 2, the calculation of the ϵ_i values.

2. For each level $i \in \{1, \ldots, d\}$ compute a cutoff value ϵ_i as follows.

   (a) Solve a (small) specified number z of instances of Equation (3.10) for level i.

   (b) For every previously estimated value $\pi(t|st)$, compute the fraction of times that this $\pi(\cdot|\cdot)$-value was set to zero in the z solutions to Equation (3.10) for level i.

   In Figure 3.5 the set of values that $\pi(t|st)$ can assume corresponds to the x-axis. The fraction of times a $\pi(\cdot|\cdot)$-value was set to zero is the corresponding value on the y-axis.

   (c) Compute a candidate cutoff value $\hat{\epsilon}_i$ as follows. $\hat{\epsilon}_i$ is the largest $\pi(t|st)$ for which all values $\pi \leq \pi(t|st)$ were set to zero in at least 50% of the z instances at level i.

   In Figure 3.5 this is the smallest x-value at which the curve intersects the horizontal y=0.5 line. We thus suggest to ignore (i.e., set to zero) only probabilities $\pi(\cdot|\cdot)$ that were set to zero in the majority *of the z solutions to the optimization problem.*

   (d) Compute the actual cutoff value ϵ_i as follows:

      i. If for each type $u \in T$ there is at least one type $t \in T$ such that $\pi(t|u) \geq \hat{\epsilon}_i$, let $\epsilon_i = \hat{\epsilon}_i$.

      In this case, it is safe to use $\hat{\epsilon}_i$ as a cutoff value, since for each $u \in T$ there will be some $\pi(\cdot|u)$-value that is not set to zero.

      ii. If for some type $u \in T$ there is no type $t \in T$ such that $\pi(t|u) \geq \hat{\epsilon}_i$, let ϵ_i be the largest value $\delta_i < \hat{\epsilon}_i$ such that, for all $u \in T$ there is some $t \in T$ such that $\pi(t|u) \geq \delta_i$.

      In this case, we cannot use $\hat{\epsilon}_i$ as a cutoff value, since this would imply setting all $\pi(\cdot|u)$ values to zero, for type u.

For illustration, Table 3.4 shows the ϵ_i values calculated using 10,000 randomly generated start states for the 8-puzzle with Manhattan Distance. The value of 0.05 for level 15 in Table 3.4 means that, out of the 10,000 searches, the majority of types that were generated with probability of 0.05 or lower at level 15 had their values of $p(\cdot|\cdot)$ set to zero by the optimization algorithm. In this table, as in all the experiments in this paper, the ϵ_i values approach zero as i gets larger.

CDP is applicable only in situations in which one is interested in making a large number of predictions, so that the time required for sampling is amortized. We show in the next section that the

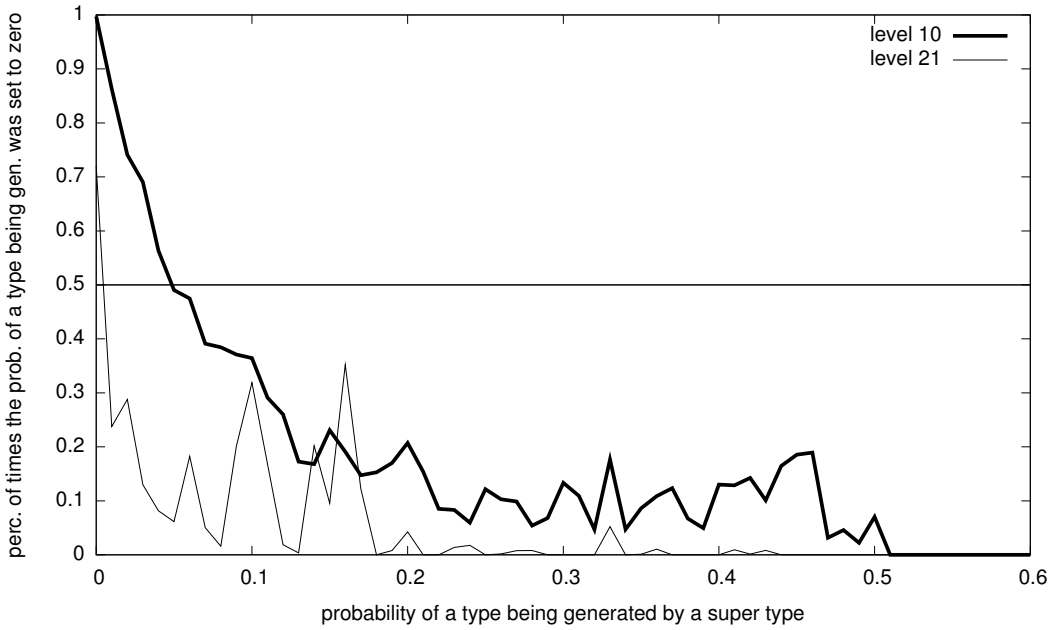

Figure 3.5: $\hat{\epsilon_i}$ calculation for $i=10$ and $i=21$ (8-puzzle with Manhattan Distance).

Level (i)	10	11-12	13	14	15	16-17	18-19	20-23	24
ϵ_i	0.05	0.07	0.08	0.07	0.05	0.04	0.03	0.01	0.00

Table 3.4: ϵ_i values for the 8-puzzle with Manhattan Distance.

ϵ-truncation procedure can substantially increase the accuracy of the CDP predictions. However, this improvement in accuracy comes at the cost of an increased preprocessing time. Namely, ϵ-truncation roughly doubles the preprocessing time. For instance, it takes approximately 10 hours to sample one billion random states to approximate the values of $p(t|u)$ and b_u for the 15-puzzle using the T_{gc} type system. The ϵ-truncation procedure adds another 15 hours of preprocessing for finding the value of ϵ_i. In the experiments described in the next section we assume one is interested in making a sufficiently large number of predictions that the preprocessing time required by ϵ-truncation is amortized.

3.2.4 Experimental Results on ϵ-Truncation

This section presents the results of experiments showing that: (a) refining a type system often reduces prediction accuracy; (b) ϵ-truncation often substantially improves predictions; (c) ϵ-truncation of a refinement of a type system usually gives greater improvements than ϵ-truncation of the basic type system, and (d) ϵ-truncation rarely reduces the prediction accuracy. Each experiment will use two type systems, a basic one and a refinement of the basic one, and will compare the predictions made by CDP with each type system and with ϵ-truncation applied to both type systems.

Experimental Setup

Our experiments are run on three domains: the sliding-tile puzzle, the pancake puzzle, and Rubik's Cube. For a description of these domains see Section 2.4.

The choice of the set of start states will be described in the specific sections below, but we always applied the same principle as Zahavi et al. [84]: start state s is included in the experiment with cost bound d only if IDA* would actually have used d as a cost bound in its search with s as the start state; Zahavi et al. called this selection of s and d the restricted selection and showed that if the selection of s and d is not restricted, the number of nodes expanded by the IDA* in the experiments would be substantially different than the number of nodes expanded by the algorithm in real situations (cf. Table 5 of Zahavi et al. [84]). Like Zahavi et al., we are interested in verifying the accuracy of the predictions in real situations, thus we also adopt the restricted selection. As mentioned in Section 3.2, unlike an actual IDA* run, we count the number of nodes expanded in the entire iteration for a start state even if the goal is encountered during the iteration.

The number of start states used to determine the ϵ_i values is closely related to the value of r that will be used in the experiment — recall that the value of r determines the level at which CDP collects states to seed the prediction. For example, the number of states at level 10 of the 8-puzzle is expected to be much lower than the number of states at level 25 of the 15-puzzle. Therefore, in order to find suitable ϵ-values for the 8-puzzle we have to use more start states than are required to determine ϵ-values for the 15-puzzle. The number of states used to determine the ϵ_i values is stated below for each experiment.

Error Measures. We report the prediction results using three different measures: *Relative Signed Error*, *Relative Unsigned Error*, and *Root Mean Squared Relative Error* (RMSRE).

- **Relative Signed Error** – For each prediction system we will report the ratio of the predicted number of nodes expanded, averaged over all the start states, to the actual number of nodes expanded, on average, by IDA*. Let PI be the set of problem instances used in an experiment; $\texttt{CDP}(s, d, h, T)$ is the predicted number of nodes expanded by IDA* for start state s, cost bound d, heuristic function h, and type system T; $A(s, d)$ is the actual number of nodes expanded by IDA* when s is the start state with cost bound d. The relative signed error for experiment with PI is calculated as follows.

$$\frac{\sum_{s \in PI} \texttt{CDP}(s, d, h, T)}{\sum_{s \in PI} A(s, d)}$$

This ratio will be rounded to two decimal places. Thus a ratio of 1.00 does not necessarily mean the prediction is perfect, it just means the ratio is closer to 1.00 than it is to 0.99 or 1.01. This ratio we call the relative signed error. It is the same as the "Ratio" reported by Zahavi et al. [84] and is appropriate when one is interested in predicting the total number of nodes that will be expanded in solving a set of start states. It is not appropriate for measuring the accuracy of the predictions on individual start states because overestimations are cancelled by underestimations. If these exactly balance out, a system will appear to have no error (a ratio of 1.00) even though there might be substantial error in every single prediction.

- **Relative Unsigned Error (RUE)** – To evaluate the accuracy of individual predictions, an appropriate measure is relative unsigned error, calculated as follows.

$$\frac{\sum_{s \in PI} \frac{|\mathtt{CDP}(s,d,h,T) - A(s,d)|}{A(s,d)}}{|PI|}$$

  A perfect score according to this measure is 0.00.

- **RMSRE** – Another error measure we use to compare predictions for individual start states is the *root mean squared relative error* (RMSRE), which is calculated as follows.

$$\sqrt{\frac{\sum_{s \in PI} (\frac{\mathtt{CDP}(s,d,h,T) - A(s,d)}{A(s,d)})^2}{|PI|}}$$

  A perfect score according to this measure is 0.00.

All of our experiments were run on an Intel Xeon CPU X5650, 2.67GHz.

Sliding-Tile Puzzles

We use three sizes of the sliding-tile puzzle: two that are small enough that the entire reachable portion of the state space can be enumerated and used in lieu of "sampling" — the 8-puzzle, and the (3×4)-puzzle — and one that is large enough to be non-trivial — the 15-puzzle. The small domains are an important element of the experiments because phenomena witnessed in them cannot be attributed to sampling effects.

We used the same type system as Zahavi et al. [84], which is a refinement of T_h we call $T_{h,b}$. $T_{h,b}$ is defined by $T_{h,b}(s) = (T_h, blank(parent(s)), blank(s))$ where $blank(s)$ returns the kind of location (corner, edge, or middle) the blank occupies in state s. For instance, if we assume that the lefthand state in Figure 2.1 (let us call it s) was generated by moving the 5 tile to the right, then we

would have the following $T_{h,b}$ type for s: $(14, 15, C, E)$. The 14 in the tuple stands for the heuristic value of the parent of s (we assume the heuristic being used is Manhattan Distance); the 15 stands for the heuristic value of s; the C tells us that the blank position of the parent of s was in a corner; finally, the E means that the blank is in an edge position in s.

For the (3x4)-puzzle there are two kinds of edge locations that $blank(s)$ needs to distinguish—edge locations on the short side (length 3) and edge locations on the long side (length 4). $T_{gc,b}$ is defined analogously. For square versions of the puzzle, T_{gc} is exactly the same as $T_{gc,b}$ and therefore $T_{gc} \preceq T_{h,b}$. However, for the (3x4)-puzzle, T_{gc} and $T_{gc,b}$ are not equal. We used the following coarse type systems to define the supertypes.

- for T_{gc} we use T_c;
- for $T_{gc,b}$ we use T_c augmented with the kind of blank location of the parent of the node;
- for $T_{h,b}$ we use $T_{p,b}(s) = (h(s), blank(s))$.

For the 8-puzzle we used 10,000 random start states to determine the ϵ_i values and every solvable state in the space to measure prediction accuracy. The upper part of Table 3.5 shows the results for the Manhattan Distance heuristic, which is admissible and consistent, with r=10. The bold entries in this and all other tables of results indicate the best predictions for a given error measure. Here we see a few cases of the informativeness pathology (Signed Error with $d \geq 21$): T_{gc}'s predictions are worse than $T_{h,b}$'s, despite its being a refinement of $T_{h,b}$. Applying ϵ-truncation substantially reduces T_{gc}'s prediction error for all three error measures.

We also ran experiments on the 8-puzzle using the inconsistent heuristic defined by Zahavi et al. [84]. Two pattern databases (PDB) [13] were built, one based on the identities of the blank and tiles 1–4 (tiles 5–8 were indistinguishable), and another based on the identities of the blank and tiles 5–8 (tiles 1–4 were indistinguishable). The locations in the puzzle are numbered in increasing order left-to-right and top-to-bottom and the first PDB is consulted for states having the blank in an even location; the second PDB is consulted otherwise. Since the blank's location changes parity every time it moves, we are guaranteed that the heuristic value of a child node will be taken from a different PDB than that of its parent. Heuristics defined by PDBs are admissible and consistent, but as we are alternating the lookup between two different PDBs the resulting heuristic is admissible but inconsistent. Again we used 10,000 random start states to determine the ϵ_i values and every solvable state in the space to measure prediction accuracy. The results of this experiment, with r=1, are shown in the middle part of Table 3.5. They exhibit the informativeness pathology and demonstrate that ϵ-

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}
Manhattan Distance. r=10.													
18	134.4	**1.00**	**1.00**	**1.00**	**1.00**	0.03	0.03	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
19	238.4	**1.00**	**1.00**	**1.00**	**1.00**	0.04	0.04	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
20	360.1	1.01	1.01	1.01	**0.99**	0.05	0.05	0.03	**0.02**	0.07	0.07	0.04	**0.03**
21	630.7	1.00	1.00	1.02	**0.99**	0.06	0.06	**0.03**	**0.03**	0.07	0.07	**0.04**	**0.04**
22	950.6	1.01	1.01	1.03	**0.98**	0.07	0.07	0.05	**0.04**	0.09	0.09	0.06	**0.05**
23	1,649.5	**1.00**	**1.00**	1.04	0.98	0.07	0.07	0.05	**0.04**	0.09	0.09	0.06	**0.05**
24	2,457.5	**1.01**	**1.01**	1.06	0.97	0.08	0.08	0.08	**0.05**	0.11	0.11	0.09	**0.07**
25	4,245.5	**1.00**	**1.00**	1.07	0.97	0.09	0.09	0.08	**0.05**	0.11	0.11	0.09	**0.06**
26	6,294.4	**1.00**	**1.00**	1.10	0.96	0.10	0.10	0.10	**0.06**	0.12	0.12	0.12	**0.08**
27	10,994.9	**0.99**	**0.99**	1.11	0.97	0.11	0.11	0.11	**0.06**	0.14	0.14	0.13	**0.08**
Inconsistent Heuristic. r=1.													
18	14.5	0.71	0.78	1.16	**1.02**	0.37	0.42	0.43	**0.36**	**0.55**	0.67	0.69	0.57
19	22.2	0.72	0.73	1.19	**1.01**	0.45	0.45	0.49	**0.39**	0.59	0.59	0.69	**0.54**
20	27.4	0.73	0.82	1.23	**0.98**	0.45	0.50	0.54	**0.40**	0.62	0.75	0.78	**0.57**
21	43.3	0.74	0.77	1.27	**0.99**	0.48	0.49	0.56	**0.40**	0.63	0.65	0.77	**0.53**
22	58.5	0.75	0.83	1.33	**0.95**	0.47	0.51	0.62	**0.40**	0.63	0.73	0.86	**0.54**
23	95.4	0.75	0.81	1.39	**0.97**	0.48	0.49	0.62	**0.38**	0.63	0.68	0.84	**0.50**
24	135.7	0.76	0.84	1.45	**0.92**	0.45	0.47	0.65	**0.37**	0.58	0.66	0.90	**0.49**
25	226.7	0.76	0.83	1.51	**0.97**	0.43	0.42	0.65	**0.33**	0.55	0.57	0.88	**0.44**
26	327.8	0.76	0.85	1.57	**0.91**	0.41	0.39	0.67	**0.30**	0.49	0.51	0.90	**0.40**
27	562.0	0.76	0.85	1.63	**0.98**	0.39	0.36	0.66	**0.26**	0.46	0.44	0.86	**0.34**
Same Inconsistent Heuristic. r=10.													
18	14.5	0.88	0.90	**1.00**	**1.00**	0.04	0.04	**0.01**	**0.01**	0.11	0.10	**0.03**	**0.03**
19	22.2	0.87	0.88	**1.00**	**1.00**	0.06	0.06	**0.01**	**0.01**	0.13	0.12	**0.05**	**0.05**
20	27.4	0.86	0.87	**1.01**	**1.01**	0.07	0.07	**0.03**	**0.03**	0.15	0.14	**0.07**	**0.07**
21	43.3	0.86	0.87	**1.02**	**1.02**	0.09	0.09	**0.04**	**0.04**	0.16	0.16	**0.09**	**0.09**
22	58.5	0.85	0.85	**1.03**	**1.03**	0.11	0.10	**0.07**	**0.07**	0.18	0.17	**0.12**	**0.12**
23	95.4	0.84	0.85	1.05	**1.04**	0.13	0.12	**0.09**	**0.09**	0.19	0.18	0.14	**0.13**
24	135.7	0.83	0.83	1.07	**1.05**	0.15	0.15	0.11	**0.10**	0.21	0.20	0.15	**0.14**
25	226.7	0.82	0.82	1.10	**1.06**	0.17	0.17	0.13	**0.12**	0.22	0.22	0.17	**0.15**
26	327.8	0.81	0.82	1.13	**1.07**	0.19	0.19	0.16	**0.13**	0.23	0.23	0.19	**0.16**
27	562.0	0.81	0.80	1.17	**1.08**	0.21	0.21	0.18	**0.13**	0.24	0.24	0.22	**0.16**

Table 3.5: CDP with ϵ-truncation on the 8-puzzle.

truncation can substantially reduce prediction error. ϵ-truncation produces slightly worse predictions when using the coarser type system for some of the cost bounds when measuring unsigned error or RMSRE (see d=20, 21, 22, 23, 24, 25, 26 in the middle part of Table 3.5). Note, however, that this decrease in accuracy is not observed when signed error is measured. Large overestimations and underestimations of the actual number of nodes expanded might cancel each other out when the signed error is measured, giving the impression that the predictions are accurate even if they are not. The decrease in performance caused by ϵ-truncation disappears for larger values of r as shown in the bottom part of Table 3.5. The improvements in accuracy by ϵ-truncation are still observed in the bottom part of Table 3.5.

For the (3x4)-puzzle we used 10 random start states to determine the ϵ_i values and 10,000 to measure prediction accuracy. The upper part of Table 3.6 shows the results for Manhattan Distance. Both the unsigned error and the RMSRE for $T_{gc,b}$ are very close to those for $T_{h,b}$'s, suggesting that being more informed provides no advantage. ϵ-truncation substantially improves $T_{gc,b}$'s predictions in all three error measures. The lower part of the table is for Manhattan Distance multiplied by 1.5, which is inadmissible and inconsistent. Here $T_{gc,b}$'s predictions are considerably more accurate than $T_{h,b}$'s and are substantially improved by ϵ-truncation. In both cases ϵ-truncation did not modify the

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$	$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$	$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$
Manhattan Distance.													
33	30,461.9	1.02	1.02	1.01	**1.00**	0.02	0.02	**0.01**	**0.01**	0.02	0.02	**0.01**	**0.01**
34	49,576.8	1.02	1.02	1.02	**1.01**	0.02	0.02	0.02	**0.01**	0.03	0.03	0.02	**0.01**
35	80,688.2	1.04	1.04	1.04	**1.01**	0.03	0.03	0.03	**0.02**	0.04	0.04	0.03	**0.02**
36	127,733.4	1.05	1.05	1.05	**1.02**	0.05	0.05	0.04	**0.02**	0.06	0.06	0.05	**0.03**
37	201,822.7	1.07	1.07	1.08	**1.03**	0.06	0.06	0.06	**0.03**	0.08	0.08	0.07	**0.04**
38	327,835.3	1.09	1.09	1.11	**1.04**	0.08	0.08	0.09	**0.04**	0.10	0.10	0.10	**0.05**
39	478,092.5	1.12	1.12	1.15	**1.05**	0.11	0.11	0.13	**0.06**	0.13	0.13	0.14	**0.07**
40	822,055.4	1.16	1.16	1.20	**1.07**	0.14	0.14	0.17	**0.08**	0.17	0.17	0.19	**0.09**
41	1,163,312.1	1.20	1.20	1.26	**1.10**	0.17	0.17	0.23	**0.10**	0.21	0.21	0.25	**0.12**
42	1,843,732.2	1.27	1.27	1.34	**1.13**	0.23	0.23	0.30	**0.13**	0.27	0.27	0.32	**0.15**
Manhattan Distance multiplied by 1.5.													
33	926.2	1.05	1.04	1.01	**1.00**	0.02	0.02	**0.00**	**0.00**	0.04	0.03	**0.01**	**0.01**
34	1,286.9	1.06	1.06	1.02	**1.00**	0.03	0.02	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
35	2,225.6	1.09	1.08	1.03	**1.00**	0.05	0.05	0.02	**0.01**	0.07	0.07	0.03	**0.02**
36	2,670.7	1.11	1.10	1.04	**0.99**	0.06	0.05	0.02	**0.01**	0.09	0.09	0.04	**0.03**
37	3,519.5	1.14	1.13	1.06	**0.99**	0.08	0.08	0.04	**0.02**	0.12	0.11	0.06	**0.04**
38	5,570.8	1.19	1.18	1.09	**0.98**	0.12	0.12	0.06	**0.03**	0.16	0.16	0.08	**0.05**
39	6,983.8	1.23	1.22	1.12	**0.97**	0.15	0.15	0.08	**0.03**	0.20	0.20	0.11	**0.06**
40	9,103.3	1.29	1.28	1.18	**0.97**	0.19	0.19	0.12	**0.05**	0.25	0.25	0.16	**0.08**
41	13,635.3	1.36	1.36	1.24	**0.96**	0.27	0.27	0.18	**0.06**	0.33	0.33	0.21	**0.09**
42	16,634.2	1.43	1.43	1.30	**0.95**	0.32	0.32	0.22	**0.07**	0.39	0.39	0.27	**0.10**

Table 3.6: CDP with ϵ-truncation on the (3x4)-puzzle. r=20.

predictions for T_h.

For the 15-puzzle, we used 5 random start states to determine the ϵ_i values and 1,000 to measure prediction accuracy. To define $\pi(t|u)$ and β_t, one billion random states were sampled and, in addition, we used the process described by Zahavi et al. [84] to non-randomly extend the sampling: we sampled the child of a sampled state if the type of that child had not yet been sampled. We call this process *biased sampling*. Table 3.7 shows the results when Manhattan Distance is the heuristic and $T_{h,b}$ and T_{gc} are the type systems. Here again we see the informativeness pathology ($T_{h,b}$'s predictions are better than T_{gc}'s) which is eliminated by ϵ-truncation. Like for the (3x4)-puzzle, ϵ-truncation does not modify the predictions when using the coarser type system.

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}
50	8,909,564.5	1.16	1.16	1.18	**1.08**	0.09	0.09	0.10	**0.05**	0.12	0.12	0.12	**0.07**
51	15,427,786.9	1.15	1.15	1.19	**1.07**	0.11	0.11	0.12	**0.07**	0.13	0.13	0.14	**0.08**
52	28,308,808.8	1.25	1.25	1.28	**1.14**	0.14	0.14	0.17	**0.09**	0.18	0.18	0.20	**0.11**
53	45,086,452.6	1.23	1.23	1.29	**1.13**	0.16	0.16	0.20	**0.11**	0.20	0.20	0.23	**0.13**
54	85,024,463.5	1.36	1.36	1.41	**1.22**	0.21	0.21	0.27	**0.15**	0.26	0.26	0.30	**0.17**
55	123,478,361.5	1.36	1.36	1.45	**1.24**	0.24	0.24	0.31	**0.17**	0.29	0.29	0.34	**0.20**
56	261,945,964.0	1.44	1.44	1.54	**1.30**	0.28	0.28	0.39	**0.21**	0.35	0.35	0.43	**0.25**
57	218,593,372.3	1.43	1.43	1.57	**1.32**	0.33	0.33	0.45	**0.26**	0.40	0.40	0.49	**0.30**

Table 3.7: CDP with ϵ-truncation on the 15-puzzle. Manhattan Distance. r=25.

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c	$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c	$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c
50	562,708.5	0.55	0.24	1.77	**1.20**	537.97	124.62	1.29	**1.17**	3,157.70	733.36	2.14	**2.08**
51	965,792.6	0.70	0.31	1.39	**1.04**	812.37	157.73	1.32	**1.12**	6,449.94	1,236.62	2.30	**1.92**
52	1,438,694.0	**0.96**	0.43	1.68	1.23	513.99	151.99	1.52	**1.35**	2,807.23	696.84	2.51	**2.34**
53	2,368,940.3	**1.29**	0.58	1.75	1.32	694.34	216.27	1.56	**1.26**	5,665.10	1,696.83	2.56	**2.05**
54	3,749,519.9	1.64	**0.73**	2.03	1.54	647.24	226.79	1.77	**1.53**	3,309.93	1,054.35	2.75	**2.46**
55	7,360,297.6	1.90	**0.86**	2.07	1.59	650.59	246.84	1.72	**1.35**	5,080.16	1,900.50	2.68	**2.12**
56	12,267,171.0	2.30	**1.03**	2.19	1.61	927.71	367.99	2.16	**1.86**	6,380.99	2,454.03	3.53	**3.41**
57	23,517,650.8	2.69	**1.21**	2.29	1.78	600.13	243.38	2.02	**1.55**	3,819.40	1,522.75	3.08	**2.40**

Table 3.8: CDP with ϵ-truncation on the 15-puzzle. Inconsistent Heuristic. r=1.

Like for the 8-puzzle, an inconsistent heuristic for the 15-puzzle was created with one PDB based on the identities of the blank and tiles 1–7, and another that kept the identities of the blank and tiles 9–15, exactly as used by Zahavi et al. (see their Table 11). We alternate the PDB that is used for the heuristic lookup depending on the position of the blank as described for the 8-puzzle. The results with $T_{h,b}$ and T_c as type systems are shown in Table 3.8. Here we see that even though $T_{h,b}$ has a reasonable signed error, it has in fact a very large unsigned error and RMSRE, and once again ϵ-truncation produced substantial improvement in prediction accuracy—in this case for both the coarse and refined type systems. These prediction results could be improved by increasing the r-value used. However, we wanted our results to be comparable to those in Zahavi et al.'s Table 11.

Pancake Puzzle

For the 15-pancake puzzle, we used 10 random start states to determine the ϵ_i values and 1,000 to measure prediction accuracy. We used T_h and T_c as the type systems. The coarser type systems used to define the supertypes for T_h and T_c were T_p and T_h, respectively. To define $\pi(t|u)$ and β_t, 100 million random states were sampled and, in addition, we used the biased sampling process described for the 15-puzzle. The results with r=4 and a PDB heuristic that keeps the identities of the smallest eight pancakes are shown in the upper part of Table 3.9. The lower part of the table shows results when the heuristic is multiplied by 1.5 (and hence is inadmissible and inconsistent). In both cases T_c outperforms T_h but is also substantially improved by ϵ-truncation. As in the previous experiment, here ϵ-truncation does not modify the predictions for the coarser type system.

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c
Admissible and Consistent Heuristic													
11	44,771.2	1.12	1.12	1.06	**1.00**	0.51	0.51	0.19	**0.13**	0.59	0.59	0.22	**0.15**
12	346,324.5	1.15	1.15	1.07	**0.98**	0.59	0.59	0.23	**0.14**	0.70	0.70	0.27	**0.18**
13	2,408,281.6	1.27	1.27	1.14	**1.01**	0.63	0.63	0.25	**0.15**	0.74	0.74	0.30	**0.19**
14	20,168,716.0	1.37	1.37	1.19	**1.05**	0.67	0.67	0.28	**0.17**	0.78	0.78	0.33	**0.21**
15	127,411,357.4	1.60	1.60	1.30	**1.15**	0.76	0.76	0.32	**0.20**	0.84	0.84	0.37	**0.25**
The heuristic above multiplied by 1.5													
12	188,177.1	1.99	1.99	1.25	**1.13**	1.50	1.50	0.50	**0.37**	1.78	1.78	0.62	**0.47**
13	398,418.8	2.12	2.12	1.31	**1.12**	1.61	1.61	0.52	**0.39**	2.08	2.08	0.74	**0.54**
14	3,390,387.6	2.31	2.31	1.37	**1.11**	1.62	1.62	0.50	**0.32**	1.96	1.96	0.68	**0.44**
15	6,477,150.7	2.23	2.23	1.27	**0.98**	1.73	1.73	0.54	**0.36**	2.23	2.23	0.75	**0.49**
16	16,848,215.1	2.79	2.79	1.49	**1.12**	1.97	1.97	0.55	**0.37**	2.46	2.46	0.75	**0.47**

Table 3.9: `CDP` with ϵ-truncation on the 15-pancake puzzle. r=4.

Rubik's Cube

For the $3 \times 3 \times 3$ Rubik's Cube we used 10 random start states to determine the ϵ_i values and 1,000 to measure prediction accuracy. We sampled 100 million random states and, in addition, we used the biased sampling process described for the 15-puzzle. The random states were generated by random

walks from the goal state, whose length was randomly selected between 0 and 180 steps. We prune redundant moves in the main search as described by Korf [46], which reduces the branching factor from 18 to approximately 13.35. Korf considered two kinds of redundant move pruning. First, he noted that twisting the same face twice in a row leads to redundant states; second, twisting the front face and then the back face leads to the same state as twisting the faces in opposite order. We used the same procedure described by Zahavi et al. [84] to implement redundant move pruning during sampling: the last operator in the random walk is used as a basis to prune redundant moves. We used T_h and T_c as the type systems. The coarser type systems used to define the supertypes for T_h and T_c were T_p and T_h, respectively. The heuristic we used was a PDB of the 8 corner cubies [46] over an abstraction on the sides of the puzzle. The abstraction was built by mapping three colors to one color and the other three colors to a second color, in such a way that, in the abstract goal state, two opposite sides of the cube always differ in color.

		Signed Error				Unsigned Error				RMSRE			
d	IDA*	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c
Admissible and Consistent Heuristic													
9	119,506.2	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
10	1,626,583.9	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
11	21,985,207.8	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
12	295,893,415.9	0.93	0.93	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
The heuristic above multiplied by 1.5													
9	7,515.5	0.91	0.91	**0.98**	**0.98**	0.08	0.08	**0.02**	**0.02**	0.09	0.09	**0.03**	**0.03**
10	51,616.2	0.91	0.91	**0.98**	**0.98**	0.08	0.08	**0.02**	**0.02**	0.10	0.10	**0.03**	**0.03**
11	685,630.9	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.03**	**0.03**	0.09	0.09	**0.04**	**0.04**
12	8,674,465.2	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.02**	**0.02**	0.09	0.09	**0.03**	**0.03**
13	116,376,337.0	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.02**	**0.02**	0.08	0.08	**0.03**	**0.03**

Table 3.10: `CDP` with ϵ-truncation on the $3 \times 3 \times 3$ Rubik's Cube. r=3.

The upper part of Table 3.10 shows the results while using the PDB heuristic just described. Here, T_h results in fairly accurate predictions, which are further improved when T_c is used. ϵ-truncation does not modify the predictions in this case.

The lower part of Table 3.10 shows the results when the PDB heuristic is multiplied by 1.5. The resulting heuristic is inadmissible and inconsistent. Again, `CDP` makes very accurate predictions and ϵ-truncation does not modify the results.

3.2.5 Lookup CDP

We have demonstrated that ϵ-truncation improves the accuracy of the `CDP` predictions. We now present Lookup `CDP` (`L-CDP`), a variant of `CDP` that improves its runtime; it can be orders of magnitude faster than `CDP`. `L-CDP` takes advantage of the fact that the `CDP` predictions are decomposable into independent subproblems. The number of nodes expanded by each node s in the outermost summation in Equation 3.1 can be calculated separately. Each pair (t, d) where t is a

type and d is a cost bound represents one of these independent subproblems. In the example of Figure 3.1, the problem of predicting the number of nodes expanded by IDA* for start state s_0 and cost bound d could be decomposed into two independent subproblems, one for $(u_1, d-1)$ and another for $(u_2, d-1)$; the sum of the solutions to these subproblems plus one (for the start state being expanded) gives the solution to the initial problem. In L−CDP, the predicted number of nodes expanded by each pair (t, d) is computed with dynamic programming in a preprocessing step and stored in a lookup table. The number of entries stored in the lookup table depends on the number of types $|T|$ and on the number of different cost bounds d. For instance, the type system we use for the 15-pancake puzzle has approximately 3,000 different types, and the number of different cost bounds in this domain is 16, which results in only $3,000 \times 16 = 48,000$ entries to be precomputed and stored in memory. If the values of d are not known *a priori*, L−CDP can be used as a *caching* system. In this case L−CDP builds its lookup table as the user asks for predictions for different start states and cost bounds. Once the solution of a subproblem is computed, its result is stored in the lookup table and it is never computed again.

The following procedure summarizes L−CDP.

1. As in CDP, we sample the state space to approximate the values of $p(t'|t)$ and b_t and to compute the ϵ_i-values needed for ϵ-truncation.

2. We compute the predicted number of nodes expanded for each pair (t, d) and store the results in a lookup table. This is done with dynamic programming: pairs (t, d) with smaller values of d are computed first. This way, when computing the (t, k)-values for a fixed k, we can use the (t, k')-values with $k' < k$ that were already computed.

3. For start state s^* and cost bound d we collect the set of nodes C_r. Then, for each node in C_r with type t, we sum the entries of the $(t, d-r)$-values from our lookup table. This sum added to the number of nodes expanded while collecting the nodes in C_r is the predicted number of nodes expanded by IDA* for s^* and d.

The worst-case time complexity of a CDP prediction is $O(|T|^2 \cdot (d-r) + Q_r)$ as there can be $|T|$ types at a level of prediction that generate $|T|$ types on the next level. $d-r$ is the largest number of prediction levels in a CDP run. Finally, Q_r is the number of nodes generated while collecting C_r. The time complexity of an L−CDP prediction (Step 3 above) is $O(Q_r)$ as the preprocessing step has reduced the L−CDP computation for a given type to a constant-time table lookup. The preprocessing L−CDP does is not significantly more costly than the preprocessing CDP does because the runtime

of the additional preprocessing step of L−CDP (Step 2 above) is negligible compared to the runtime of Step 1 above. Both CDP and L−CDP are only applicable when one is interested in making a large number of predictions so that the preprocessing time is amortized.

3.2.6 Experimental Results on Lookup CDP

We now compare the prediction runtime of CDP with that of L−CDP. Note that the accuracy of both methods is the same as they make exactly the same predictions. Thus, here we only report prediction runtime. We ran experiments on the 15-puzzle, 15-pancake puzzle, and Rubik's Cube using the consistent heuristics described in Section 3.2.4. We used a set of 1,000 random start states to measure the runtime for each of the domains.

Table 3.11 presents the average prediction runtime in seconds for L−CDP and CDP for different values of r and d. The bold values highlight the faster predictions made by L−CDP. For lower values of r, L−CDP is orders of magnitude faster than CDP. However, as we increase the value of r the two prediction systems have similar runtime. For instance, with the r-value of 25 on the 15-puzzle L−CDP is only slightly faster than CDP as, in this case, collecting C_r dominates the prediction runtime.

3.3 Stratified Sampling

We now review a method introduced by Knuth [38] that was later improved by Chen [12] which can be used to predict the IDA* EST size.

The Knuth-Chen method, Stratified Sampling (SS), also uses the C&C paradigm. Chen used the term stratifiers instead of type systems. Like CDP, SS also assumes that nodes of the same type root subtrees of the same size.

Knuth [38] presents a method to predict the EST size by repeatedly performing a random walk from the start state. Each random walk is called a *probe*. Knuth's method assumes that all branches have a structure similar to that of the path visited by the random walk. Thus, walking on one path is enough to predict the structure of the entire tree. Knuth noticed that his method was not effective when the tree being sampled is unbalanced. Chen [12] addressed this problem with a stratification of the search tree through a type system (or stratifier) to reduce the variance of the probing process.

Given an EST $S = (N, E)$ and a numerical function z over N, SS approximates a function

$$\varphi(S) = \sum_{s \in N} z(s)\,.$$

15-puzzle						
	$r = 5$		$r = 10$		$r = 25$	
d	L−CDP	CDP	L−CDP	CDP	L−CDP	CDP
50	**0.0001**	0.3759	**0.0060**	0.3465	**3.0207**	3.1114
51	**0.0002**	0.4226	**0.0065**	0.3951	**4.3697**	4.4899
52	**0.0001**	0.4847	**0.0074**	0.4537	**6.9573**	7.1113
53	**0.0002**	0.5350	**0.0071**	0.5067	**9.1959**	9.3931
54	**0.0002**	0.6105	**0.0073**	0.5805	**14.5368**	14.8017
55	**0.0000**	0.6650	**0.0077**	0.6369	**17.4313**	17.7558
56	**0.0003**	0.7569	**0.0082**	0.7257	**27.6587**	28.1076
57	**0.0001**	0.7915	**0.0079**	0.7667	**23.4482**	23.8874
15-pancake puzzle						
	$r = 1$		$r = 2$		$r = 4$	
d	L−CDP	CDP	L−CDP	CDP	L−CDP	CDP
11	**0.0001**	0.0121	**0.0003**	0.0106	**0.0037**	0.0087
12	**0.0000**	0.0278	**0.0006**	0.0257	**0.0109**	0.0261
13	**0.0001**	0.0574	**0.0005**	0.0555	**0.0279**	0.0665
14	**0.0001**	0.1019	**0.0007**	0.1006	**0.0563**	0.1358
15	**0.0001**	0.1587	**0.0008**	0.1578	**0.0872**	0.2241
Rubik's Cube						
	$r = 2$		$r = 3$		$r = 4$	
d	L−CDP	CDP	L−CDP	CDP	L−CDP	CDP
9	**0.0012**	0.0107	**0.0090**	0.0156	**0.0319**	0.0344
10	**0.0014**	0.0287	**0.0174**	0.0415	**0.1240**	0.1328
11	**0.0013**	0.0549	**0.0182**	0.0695	**0.2393**	0.2645
12	**0.0014**	0.0843	**0.0180**	0.0992	**0.2536**	0.3065

Table 3.11: L−CDP and CDP runtime (seconds).

For instance, if $z(s)$ is the cost of processing node s, then $\varphi(S)$ is the cost of traversing the tree. If $z(s) = 1$ for all $s \in N$, then $\varphi(S)$ is the size of S.

Instead of traversing the entire tree and summing all z-values, SS assumes that subtrees rooted at nodes of the same type will have equal values of φ and so only one node of each type, chosen randomly, is expanded. This is the key to SS's efficiency since the search trees of practical interest have far too many nodes to be examined exhaustively.

Similar to CDP's pure type systems, we define a *perfect type system* for SS as follows.

Definition 13 (perfect type system)**.** *A type system T is perfect for a tree S iff for any two nodes n_1 and n_2 in S, if $T(n_1) = T(n_2)$, then the two subtrees of S rooted at n_1 and n_2 have the same value of φ.*

Definition 14 (monotonic type system)**.** *[12] A type system is monotonic for S if it is partially ordered such that a node's type must be strictly greater than its parent's type.*

Each probe p is described by a set A_p of representative/weight pairs $\langle s, w \rangle$, where s is a representative for the type $T(s)$ and w captures the estimated number of nodes of that type in S. For each

Algorithm 4 Stratified Sampling, a single probe

Input: root s^* of a tree, type system T, and cost bound d.
Output: array of sets A, where $A[i]$ is the set of pairs $\langle s, w\rangle$ for the nodes s expanded at level i.

```
1:  initialize A[0] ← {⟨s*, 1⟩}
2:  i ← 0
3:  while i is less than search depth do
4:    for each element ⟨s, w⟩ in A[i] do
5:      for each child s'' of s do
6:        if h(s'') + g(s'') ≤ d then
7:          if A[i + 1] contains an element ⟨s', w'⟩ with T(s') = T(s'') then
8:            w' ← w' + w
9:            with probability w/w', replace ⟨s', w'⟩ in A[i + 1] by ⟨s'', w'⟩
10:         else
11:           insert new element ⟨s'', w⟩ in A[i + 1]
12:         end if
13:       end if
14:     end for
15:   end for
16:   i ← i + 1
17: end while
18: return A
```

probe p and its associated set A_p a prediction can be computed as:

$$\hat{\varphi}^{(p)}(S) = \sum_{\langle s,w\rangle \in A_p} w \cdot z(s)\,.$$

Algorithm 4 describes SS for a single probe. The set A is organized into "layers", where $A[i]$ is a set of node/weight pairs at level i of the search tree – processing level i fully before $i+1$ forces the type system to be monotonic. $A[0]$ is initialized to contain the root node with weight 1 (line 1).

In each iteration (lines 4 through 10), all nodes in $A[i]$ are expanded. The children of each node in $A[i]$ are considered for inclusion in $A[i+1]$ if they are not to be pruned by the search algorithm based on cost bound d (line 7). If a child s'' of node s has a type t that is already represented in $A[i+1]$ by node s', then a *merge* action on s'' and s' is performed. In a merge action we increase the weight in the corresponding representative-weight pair of type t by the weight w of s''. s'' will replace s' according to the probability shown in line 9. Chen [12] proved that this scheme reduces the variance of the estimation scheme. The nodes in A form a *sampled subtree* of the EST.

Clearly, SS using a perfect monotonic type system would produce an exact prediction in a single probe. In the absence of that we treat $\hat{\varphi}(S)$ as a random variable; then, if $E[\hat{\varphi}(S)] = \varphi(S)$, we can approximate $\varphi(S)$ by averaging $\hat{\varphi}^{(p)}$ over multiple sampled probes. And indeed, Chen [12] proved the following theorem.

Theorem 1. *[12] Given a set of probes $p_1, \cdots, p_u$ from a search tree S, and given a monotonic type system T, the average $\frac{1}{u}\sum_{j=1}^{u} \hat{\varphi}^{(p_j)}(S)$ converges to $\varphi(S)$.*

The following lemmata will be used in later chapters for analyzing some of our C&C algorithms.

Lemma 1. *Let S be a search tree sampled by* SS *using any type system. Every node $n \in S$ has a nonzero probability of being expanded in a probe of* SS.

Proof. The proof is by induction. The base case deals with the root of S, which by definition is always expanded in any probe. We now assume that node $\hat{n} \in S$ is expanded in a probe. Every child n of $\hat{n}$ in S has a nonzero probability ($w > 0$ in line 9 of Algorithm 4) of becoming the representative node of type $T(n)$. Therefore, n has a nonzero probability of being expanded. □

Lemma 2. *Given a set of probes $p_1, \cdots, p_u$ from a search tree S, as the number of probes u goes to infinity, the probability of any node n in S being expanded by at least one probe goes to one.*

Proof. Let $P(n)$ be the probability of node $n \in S$ being expanded in a probe. According to Lemma 1 we have that $P(n) > 0$. The probability of n not being sampled in u probes is $(1-P(n))^u$, which goes to zero as u goes to infinity. This implies that the probability of n being expanded in at least one probe goes to one as u goes to infinity. □

Inconsistent heuristics. In the original SS, the process continues until $A[i]$ is empty; Chen was assuming the tree was naturally bounded. Chen used SS's approximation of the number of nodes in a search tree whose f-value did not exceed the cost bound d as an approximation of the number of nodes expanded by IDA* with cost bound d. However, when an inconsistent heuristic is used, there can be nodes in the search tree whose f-values do not exceed the cost bound d but are never expanded by IDA* as one of their ancestors had an f-value that exceeded d. Predictions made by SS as described by Chen [12] will overestimate the number of nodes expanded by IDA* when an inconsistent heuristic is used. We modify SS to produce more accurate predictions when an inconsistent heuristic is employed by adding line 7 in Algorithm 4. Now a node is considered by SS only if all its ancestors are expanded.

Parent pruning. In this chapter we are interested in using SS to predict the number of nodes expanded by IDA* with parent pruning. Like CDP, when IDA* uses parent pruning, SS makes more accurate predictions if using type systems that account for the information of the parent of a node. Thus, here we also use type systems that account for the information about the parent of node s when computing s's type (see Equations 3.4, 3.5 and 3.6 on page 26).

3.3.1 Better Type Systems for SS

The prediction accuracy of SS, like that of CDP, depends on the type system used to guide its sampling [11]. Chen suggests a type system that counts the number of children a node generates as a general type system to be used with SS. We now extend Chen's general type system to include information about the parent of the node so it makes more accurate predictions when parent pruning is considered. We define it as $T_{nc}(s) = nc(s)$, where $nc(s)$ is the number of children a node s generates accounting for parent-pruning. Recall that in our implementation of SS the depth of search is implicitly considered in any type system.

In order to reduce the variance of the size of subtrees rooted at nodes of the same type it is useful to include the *heuristic value* of the node in the type system. Intuitively, subtrees rooted at nodes with higher heuristic value are expected to have fewer nodes when compared to subtrees rooted at nodes with lower heuristic value as IDA* prunes nodes with higher heuristic value more quickly.

3.3.2 Comparison of SS with Different Type Systems

We now show empirically that using a type system that accounts for the information provided by a heuristic function instead of Chen's substantially improves SS's predictions.

Definition 15 (Domination)**.** *We say that a prediction system V dominates another prediction system V' if V is able to produce more accurate predictions in equal or less time than V'; we also say that V dominates V' if V is able to produce equally or more accurate predictions in less time than V'.*

In our tables of results we highlight the runtime and error of a prediction system if it dominates its competitor. The results presented in this section experimentally show that SS employing type systems that account for the heuristic value dominates SS employing the general type system introduced by Chen on the domains tested.

Here we measure prediction accuracy with the Relative Unsigned Error (RUE). In this experiment we also aim to show that SS produces accurate predictions when an inconsistent heuristic is employed. We show results for SS using T_{nc}, which does not account for any heuristic value, and another type system (T_h, T_c, or T_{gc}) that accounts for at least the heuristic value of the node and its parent. The results were averaged over 1,000 random start states. The number of probes used in each experiment is shown in parentheses after the name of the type system used.

The results for the 15-puzzle when using the inconsistent heuristic created by Zahavi et al. [84] and defined in Section 3.2.4 are presented in the upper part of Table 3.12. We chose the number of probes so that we could show the dominance of T_h over T_{nc}. For T_h we used 50 probes in each

prediction, while for T_{nc} we used 5,000. Given the same number of probes as T_h (50), T_{nc} was faster than T_h, but produced predictions with error approximately three times higher than T_h. When the number of probes was increased to improve accuracy, T_{nc} eventually became slower than T_h before its accuracy equalled T_h's. In addition to RUE, we show in our tables the cost bound d, the number of nodes expanded by IDA* for a given cost bound (column IDA*), and the runtime in seconds.

In Table 3.12 we see that when using a type system that considers the information provided by a heuristic function SS produces more accurate predictions in less time than when employing T_{nc}. The dominance of SS employing the type systems that account for the heuristic values over T_{nc} is also observed in experiments run on the 15-pancake puzzle and on Rubik's Cube. For both 15-pancake puzzle and Rubik's Cube we used the consistent heuristics defined in Section 3.2.4. Improvements over T_{nc} were observed not only when using T_h or T_c, but also when using T_{gc}, in all three domains.

15-puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	T_{nc} (5,000)	T_h (50)	T_{nc} (5,000)	T_h (50)
50	562,708.5	1.9816	**0.3559**	0.31	**0.20**
51	965,792.6	2.0834	**0.4118**	0.27	**0.18**
52	1,438,694.0	2.1905	**0.4579**	0.27	**0.18**
53	2,368,940.3	2.3058	**0.5260**	0.33	**0.20**
54	3,749,519.9	2.4465	**0.5685**	0.29	**0.19**
55	7,360,297.6	2.5575	**0.6927**	0.33	**0.21**
56	12,267,171.0	2.6160	**0.6923**	0.30	**0.18**
57	23,517,650.8	2.8032	**0.8150**	0.36	**0.23**
15-pancake puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	T_{nc} (1,000)	T_c (1)	T_{nc} (1,000)	T_c (1)
11	44,771.2	0.1134	**0.0067**	0.19	**0.13**
12	346,324.5	0.1310	**0.0181**	0.31	**0.14**
13	2,408,281.6	0.1536	**0.0426**	0.40	**0.15**
14	20,168,716.0	0.1768	**0.0850**	0.43	**0.18**
15	127,411,357.4	0.1974	**0.1401**	0.49	**0.19**
Rubik's Cube					
		Runtime (s)		Error (RUE)	
d	IDA*	T_{nc} (40)	T_h (10)	T_{nc} (40)	T_h (10)
9	119,506.2	0.0061	**0.0027**	0.31	**0.15**
10	1,626,583.9	0.0071	**0.0032**	0.37	**0.15**
11	21,985,207.8	0.0086	**0.0057**	0.40	**0.16**
12	295,893,415.9	0.0099	**0.0064**	0.27	**0.14**

Table 3.12: SS employing different type systems.

3.4 Experiments with the Enhanced Versions of CDP and SS

Here we empirically compare our enhanced versions of CDP and SS: L−CDP with ϵ-truncation and SS using CDP's type systems. We analyze two scenarios and in both scenarios we assume

the user is interested in making predictions for a large number of problem instances, so that the preprocessing time of CDP is amortized. In the first scenario, after preprocessing, we are interested in making predictions very quickly. In the second scenario, we allow the prediction algorithms more computation time, expecting to get more accurate predictions. We run experiments on the 15-puzzle, 15-pancake puzzle and Rubik's Cube with the consistent heuristics described in Section 3.2.4.

3.4.1 Fast Predictions

We obtained fast predictions by using a low p-value for SS (number of probes), and a low r-value for CDP (prediction lookahead). The results are shown in Table 3.13. The value in parentheses after the algorithm's name indicates the value of r for L−CDP and the number of probes for SS. L−CDP is able to make almost instantaneous predictions even when using a large type system. On the other hand, SS does the sampling for each problem instance separately during prediction. Thus, in order to make fast predictions with SS we must use a smaller type system. We used T_h for SS in all three domains. For L−CDP we used T_{gc} in the experiment on the 15-puzzle, and T_c on the 15-pancake puzzle and Rubik's Cube. Given the same type system as L−CDP, SS was in some cases even more accurate than L−CDP but always about 1,000 times slower; when it was speeded up (by being given the T_h type system) to be within an order of magnitude or two of L−CDP, its predictions were far worse. In all three domains L−CDP dominates SS.

3.4.2 Accurate Predictions

The results for accurate predictions are shown in Table 3.14. For these experiments, we used more informed type systems for both CDP and SS, namely T_{gc} for the 15-puzzle and T_c for the 15-pancake puzzle and Rubik's Cube. We also increased the value of r used by L−CDP to increase its prediction accuracy.

Often the error of the CDP predictions increases as we increase the cost bound. For instance, the CDP error shown in Table 3.14 for the 15-puzzle is 0.05 for $d = 50$, and it grows to 0.26 for $d = 57$. SS's error increased only by 0.01 for the same cost bounds. Recall that CDP samples the state space in a preprocessing step to approximate the values of $p(t|u)$ and b_u, and that these values might be different from the actual values of $p(t|u)$ and b_u of the search tree. CDP is domain-specific, instead of instance-specific. We conjecture that noisy values of $p(t|u)$ and b_u used by CDP insert errors in early stages of the prediction that compound as the depth increases. SS on the other hand is instance-specific and only nodes that are part of the search tree for the given instance are considered for sampling. SS has a similar error when predicting the size of shallow and deep search trees. For

15-puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	L−CDP (5)	SS (5)	L−CDP (5)	SS (5)
50	8,909,564.5	**0.0001**	0.0151	**0.62**	0.93
51	15,427,786.9	**0.0002**	0.0167	**0.60**	0.99
52	28,308,808.8	**0.0001**	0.0188	**0.60**	0.84
53	45,086,452.6	**0.0002**	0.0192	**0.57**	0.98
54	85,024,463.5	**0.0002**	0.0215	**0.58**	0.87
55	123,478,361.5	**0.0000**	0.0223	**0.58**	1.11
56	261,945,964.0	**0.0003**	0.0243	**0.56**	0.73
57	218,593,372.3	**0.0001**	0.0241	**0.63**	0.74

15-pancake puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	L−CDP (2)	SS (5)	L−CDP (2)	SS (5)
11	44,771.2	**0.0003**	0.0012	**0.22**	0.36
12	346,324.5	**0.0006**	0.0017	**0.22**	0.38
13	2,408,281.6	**0.0005**	0.0029	**0.22**	0.44
14	20,168,716.0	**0.0007**	0.0041	**0.21**	0.34
15	127,411,357.4	**0.0008**	0.0057	**0.22**	0.47

Rubik's Cube					
		Runtime (s)		Error (RUE)	
d	IDA*	L−CDP (2)	SS (10)	L−CDP (2)	SS (10)
9	119,506.2	**0.0012**	0.0027	**0.05**	0.15
10	1,626,583.9	**0.0014**	0.0032	**0.05**	0.15
11	21,985,207.8	**0.0013**	0.0057	**0.05**	0.16
12	295,893,415.9	**0.0014**	0.0064	**0.04**	0.14

Table 3.13: Fast predictions. L−CDP and SS.

the 15-puzzle and 15-pancake puzzle SS dominates CDP for larger cost bounds and it is no worse than CDP for lower cost bounds. Rubik's Cube turned out to be an easy domain in which to make predictions. Both CDP and SS make almost perfect predictions in this domain.

3.4.3 Experiments on Large State Spaces

We now evaluate both CDP with ϵ-truncation and SS on larger state spaces, namely the 24-puzzle and the 60-pancake puzzle.

In the experiments in this section we do not use L−CDP. This is because we use a relatively large value of r to produce accurate predictions. Recall that L−CDP and CDP take approximately the same amount of time to produce predictions for larger values of r (see Table 3.11 in Section 3.2.6).

24-puzzle. For the 24-puzzle we used the 6-6-6-6 disjoint PDBs [48]. One single random instance was used to compute the ϵ_i-values. We used a value of r of 25 and it took about 36 hours to sample one billion states for CDP. SS used 50 probes. Finally, for both CDP and SS we used the T_{gc} type system. Table 3.15 shows the prediction results for the number of nodes generated during IDA*

15-puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	L-CDP (25)	SS (5)	L-CDP (25)	SS (5)
50	8,909,564.5	3.0207	0.8765	0.05	0.09
51	15,427,786.9	4.3697	0.9715	0.07	0.08
52	28,308,808.8	6.9573	**1.1107**	0.09	**0.09**
53	45,086,452.6	9.1959	**1.1767**	0.11	**0.09**
54	85,024,463.5	14.5368	**1.3577**	0.15	**0.10**
55	123,478,361.5	17.4313	**1.3940**	0.17	**0.10**
56	261,945,964.0	27.6587	**1.6438**	0.21	**0.10**
57	218,593,372.3	23.4482	**1.5258**	0.26	**0.10**

15-pancake puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	L-CDP (5)	SS (3)	L-CDP (5)	SS (3)
11	44,771.2	0.0095	0.0180	0.09	0.07
12	346,324.5	0.0341	0.0500	0.10	0.09
13	2,408,281.6	0.1084	0.1176	0.11	0.09
14	20,168,716.0	0.2898	**0.2321**	0.13	**0.10**
15	127,411,357.4	0.6071	**0.3813**	0.16	**0.11**

Rubik's Cube					
		Runtime (s)		Error (RUE)	
d	IDA*	L-CDP (5)	SS (20)	L-CDP (5)	SS (20)
9	119,506.2	**0.0802**	0.2668	**0.01**	0.02
10	1,626,583.9	0.4217	0.7231	0.01	0.01
11	21,985,207.8	1.6155	1.5098	0.01	0.01
12	295,893,415.9	3.1221	2.5269	0.01	0.01

Table 3.14: Accurate predictions. L-CDP and SS.

searches on 200 start states. The trend that was observed in the experiment on the 15-puzzle shown in Table 3.14 is also observed here: as the search gets deeper, CDP's prediction accuracy worsens. SS, on the other hand, makes accurate predictions across different cost bounds. For instance, for a cost bound of 110 SS has a relative unsigned error of only 0.10.

This experiment also shows that the prediction methods studied in this paper can be substantially faster than IDA* performing the actual search.[3] For instance, IDA* takes approximately 90 hours on average to execute an iteration with cost bound of 110. SS takes only 37 seconds on average to make predictions with the same cost bound — a speedup of more than 8,700 times compared to the runtime of the actual IDA* search. CDP is not as fast as SS but it is still substantially faster than the actual IDA* search as CDP takes only little more than one hour on average to make predictions with the cost bound of 110.

60-pancake. For the 60-pancake puzzle we used the GAP heuristic [28]. SS tends to perform better on the 60-pancake puzzle when using the T_h type system rather than when using the T_c or

[3] Rong Zhou provided the number of nodes generated by IDA* in this experiment.

24-puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	CDP (25)	SS (50)	CDP (25)	SS (50)
90	164,814,526.6	30.9482	**7.5034**	0.20	**0.03**
92	368,992,103.4	66.6174	**8.9480**	0.34	**0.03**
94	1,985,011,441.3	178.7885	**12.5902**	0.65	**0.04**
96	4,874,007,803.3	277.2403	**16.2077**	1.07	**0.03**
98	11,015,303,521.6	455.4678	**20.2952**	1.68	**0.04**
100	11,976,556,484.1	684.3925	**22.7853**	2.48	**0.04**
102	27,500,453,677.2	1,058.3840	**26.0549**	3.76	**0.05**
104	108,902,222,694.8	1,643.4890	**30.8277**	5.68	**0.05**
106	204,754,382,723.4	2,055.6937	**33.7998**	8.15	**0.06**
108	277,502,287,352.6	2,335.7943	**33.6668**	11.47	**0.08**
110	1,954,871,642,630.4	4,161.4029	**36.8365**	20.39	**0.10**

Table 3.15: CDP and SS on the 24-puzzle using the 6-6-6-6 PDB.

the T_{gc} type systems. This is because the 60-pancake puzzle has a relatively large branching factor, namely 59. The larger branching factor slows down the T_c and T_{gc} type computation due to the lookahead these type systems perform. The T_c and T_{gc} type computation is also slower for CDP on the 60-pancake puzzle. However, in CDP, most of the expensive type computations are done as a preprocessing step, during sampling. Therefore, we use T_c for CDP and T_h for SS. CDP sampled 10 million start states in 94 hours. The CDP sampling with the T_c type system takes this long due to the large branching factor of the 60-pancake puzzle. As in the other experiments in this paper, we assume that the time required for sampling is amortized over a large number of predictions. For CDP we used 5 problem instances to compute the ϵ_i-values and an r-value of 8. For SS we used 400 probes.

60-pancake puzzle					
		Runtime (s)		Error (RUE)	
d	IDA*	CDP (8)	SS (400)	CDP (8)	SS (400)
55	4,661,209.3	0.0407	0.1193	0.21	0.20
56	21,878,193.1	0.0970	0.1506	0.21	0.20
57	40,279,688.4	**0.1435**	0.1726	**0.22**	0.22
58	82,790,542.6	**0.1575**	0.1860	**0.21**	0.21
59	242,822,659.9	0.2599	0.2325	0.19	0.21

Table 3.16: CDP and SS on the 60-pancake puzzle using the GAP heuristic.

The results shown in Table 3.16 are averages over 340 start states. Both CDP and SS are able to quickly make accurate predictions on the 60-pancake puzzle — the relative unsigned error is at most 0.22 and the runtime is at most 0.26 seconds for both algorithms. IDA* using the GAP heuristic takes approximately 30 seconds on average to solve a random instance of the 60-pancake puzzle. Thus CDP and SS produce accurate predictions more quickly than IDA* can solve the problem. The

prediction methods are much faster if we compare the time required for IDA* to finish a complete iteration for a given cost bound — recall that IDA* terminates an iteration as soon as a goal is found. For instance, for a cost bound of 59, IDA* takes approximately 37 minutes to finish a complete iteration, ignoring the goal if one is found. Both CDP and SS make predictions for the same cost bound in about a quarter of a second — a speedup of more than 9,000 times over the runtime of IDA*.

3.4.4 Discussion

Our empirical results in Section 3.2.4 showed that by carefully ignoring rare events ϵ-truncation can substantially improve the accuracy of CDP's predictions. We conjecture that these harmful rare events come from noisy values of $p(t|u)$: the values of $p(t|u)$ represent the type transition probability averaged across the state space, which can be different from the type transition probability averaged across the search tree for a particular start state. As we mentioned before, Chen [12] proved that the expected value of an SS prediction is the actual number of nodes expanded, i.e., SS is an unbiased estimator. This is because SS samples the search tree being approximated. The same can not be said about CDP. We observed empirical evidence that ϵ-truncation reduces the prediction error inserted by noisy $p(t|u)$-values, but it does not guarantee unbiased predictions. On the other hand, being domain-specific allows CDP to store the prediction results in a lookup table as a preprocessing step and produce predictions much more quickly than SS.

We showed that SS using heuristic-based type systems can be used to make accurate predictions of the IDA* EST size, even on very large state spaces. We observed that SS can produce predictions much more quickly than IDA* can solve problem instances on the 24-puzzle and on the 60-pancake puzzle. For instance, SS is approximately 8,700 times faster than IDA* on the 24-puzzle using the 6-6-6-6 disjoint PDBs as heuristic function.

In contrast with CDP, to the best of our knowledge there is no general and efficient way of preprocessing SS's predictions so that it produces almost instantaneous predictions. In fact, any preprocessing done for SS before knowing the start state would make SS quite similar to CDP.

3.5 Type Systems for Domains with Real-Valued Heuristics

Our experimental results presented in Section 3.4.3 show that SS using heuristic-based type systems is able to produce good predictions in domains with integer-valued heuristics. Nevertheless, in domains with real-valued heuristics, such as optimization queries over graphical models [69], a type system based on the comparison of floating-point heuristics might be too large to be practical.

Algorithm 5 k-means clustering for nodes in the search tree

Input: set of nodes $\mathcal{X}$ and number of clusters $k > 0$.
Output: partitions $\mathcal{X}$ into k disjoint groups $\mathcal{X}_k$.

1: create k different centroids.
2: **while** centroid assignment has not converged **do**
3: assign each node $n \in \mathcal{X}$ to the closest centroid.
4: adjust each centroid to be the mean of the value of their assigned nodes.
5: **end while**

3.5.1 The Constant C Approach

One could create effective heuristic-based type systems for domains with real-valued heuristics by multiplying the h-values by a constant C and using the integer part of the resulting number to define the type system. Different values of C result in type systems of different sizes. Although this approach works well in some cases (we use this approach in Chapter 4), it has a high experimental cost as it can be hard to find suitable values of C. Moreover, this approach usually requires different values of C for different domains.

We propose a domain-independent method based on the k-means clustering algorithm [64] for creating type systems. In contrast with the constant C approach, *clustering-based* type systems allow one to control exactly the number of nodes SS expands at every level of search, independently of the domain.

3.5.2 Clustering-Based Type Systems

We treat the problem of creating type systems as a clustering problem. A clustering algorithm groups together objects that are more alike according to a measure of similarity. In a *clustering-based* type system, nodes n and m are of the same type if they are in the same cluster. We use the words cluster and type interchangeably hereafter.

Algorithm 5 shows the k-means clustering algorithm for nodes in the search tree. It receives as input a set of nodes $\mathcal{X}$ encountered at a given level of a SS probe and a number of clusters $k > 0$. Algorithm 5 partitions $\mathcal{X}$ into k disjoint clusters $\mathcal{X}_k$. The k-means algorithm requires a measure of similarity to compute the distance between the nodes in $\mathcal{X}$ and the k centroids (line 3). We use the absolute difference between the nodes's f-values as the measure of similarity. For nodes $m, n, v \in \mathcal{X}$, according to our measure of similarity, m is closer to n than to v if $|f(m) - f(n)| < |f(m) - f(v)|$. By using a clustering-based type system one is able to control the number of node expansions at every level of the search tree. Specifically, SS expands at most k nodes at every level of search.

It is worth noting that, although only the h-value appears in the definition of the type systems used in the previous sections of this chapter, we were implicitly considering the node's f-value in those cases. This is because we ran experiments on domains with unit-edge costs and by considering the level of the search tree as part of the type systems, like CDP and SS do, we were implicitly considering the node's f-value.

Before we empirically evaluate SS using clustering-based type systems we introduce an algorithm we call Active SS, which is the application of active sampling to search trees.

3.6 Active Stratified Sampling

Active Stratified Sampling (Active SS) is a variation of SS that uses the idea of active sampling [18] in search trees. With the aim of producing better predictions of the EST size, Active SS allocates more samples to the types with higher variance. In contrast with other active sampling methods, it is not immediately clear that Active SS will perform better than SS. This is because Active SS deals with search trees and it requires a relatively expensive procedure to keep track of the variance of the types.

3.6.1 Sampling with Multiple Representatives

Active SS uses a variation of SS introduced by Chen [11] that explores multiple representative nodes of the same type in a single probe.

Chen suggested to add a random number to the type of a node so that SS expands multiple representative nodes of the same type. For instance, in the multiple-representative setting, the T_h type system is augmented with a random integer ∇ drawn from the interval $[1, M]$, for some integer M. SS using T_h in the multiple-representative setting expands at most M nodes of each T_h type. We define $T_r(s) = (T(s), \nabla)$ as the multiple representative version of a base type system T.

Purdom's Algorithm

The idea of sampling multiple representatives was first introduced by Purdom [72] as an improvement to Knuth's algorithm. Random walks are unlikely to sample deep branches of tall and skinny search trees as a probe is likely to quickly encounter a leaf node. As a result, one has to perform a large number of probes with Knuth's algorithm to produce accurate predictions. Instead of expanding one node at random at every level of search, Purdom's algorithm expands at most $M > 1$ nodes at random at every level of search. Purdom explained that by sampling multiple nodes one would increase the probability of reaching deeper branches of the search tree. Note that Purdom's strategy is

different than performing M independent probes with Knuth's algorithm. This is because one could sample the same branch multiple times across different probes. By sampling multiple branches at once, one does not sample the same branch more than once within a probe, which increases the chances of a probe reaching deeper branches of the EST. The same argument applies to SS.

Note that the strategy of expanding multiple representative nodes does not preclude SS neither Purdom's algorithm from performing multiple probes.

Unified Framework

Algorithm 6 describes a single probe of SS using a multiple-representative clustering-based type system. The main differences between Algorithm 4 and Algorithm 6 are the following. In the latter we do not insert one node of each type directly into A, but we first insert all nodes into set $\mathcal{X}$ with the weight of their parents (lines 5–11). We then use Algorithm 5 to cluster the nodes in $\mathcal{X}$ into k different types (line 12). In line 13 we select the maximum number of representative nodes m_t of each type t that will be expanded; for now we define $m_t = q/k$ for all t. Here, q is an input parameter representing the total number of extra nodes SS is allowed to expand at every level in addition to one node of each type. The type of a node s is defined by the cluster $\mathcal{X}_t$ s belongs to, and also by the random number between 1 and m_t (line 15).

This version of SS, which is described in Algorithm 6, generalizes all three algorithms: Knuth's, Purdom's and Chen's. If $k = 1$ and $q = 0$, then Algorithm 6 behaves like Knuth's algorithm as it expands a single node at every level of search. If $k = 1$ and $q > 0$, then Algorithm 6 behaves like Purdom's algorithm as it expands at most $q + 1$ nodes at every level of search. If $k > 1$ and $q > 0$, then Algorithm 6 behaves like SS with multiple representatives employing a clustering-based type system.

The differences between regular SS, which we call Passive SS hereafter, and Active SS are the following. First, instead of sampling q/k extra nodes per type, Active SS uses the Adaptive Allocation of Samples algorithm (AA) [18] for allocating the q extra samples among different types, with the aim of reducing the variance of the estimates. Second, Active SS calls a bookkeeping procedure after each probe to collect the statistics required by AA about the types in the EST. We now describe the AA and the bookkeeping procedures.

3.6.2 Adaptive Allocation of Samples

Let $\hat{Y}_t^i$ be the estimated size of the subtree rooted at a node of type t at level i of the EST. SS approximates the value of $\hat{Y}_t^i$ for all i and t encountered in the EST as a byproduct of its estimation

Algorithm 6 Stratified Sampling with Clustering-Based Types and Multiple Representatives, a single probe

Input: start state s^*, number of types k, number of samples q, cost bound d, and a type system T.
Output: array of sets A, where $A[i]$ is the set of pairs $\langle s, w\rangle$ for the nodes s expanded at level i. A sampled subtree A of the EST.

$A[0] \leftarrow \{\langle s^*, 1\rangle\}$
$\mathcal{X} \leftarrow \emptyset$
$i \leftarrow 0$
while i is less then search depth **do**
  for each element $\langle s, w\rangle$ in $A[i]$ **do**
    for each child s'' of s **do**
      if $h(s'') + g(s'') \leq d$ **then**
        insert $\langle s'', w\rangle$ in $\mathcal{X}$
      end if
    end for
  end for
  partition nodes in $\mathcal{X}$ into k types $\mathcal{X}_t$ (Algorithm 5)
  $m_t \leftarrow$ number of representative nodes for each type t (dependent on k and q)
  for each element $\langle s, w\rangle$ in $\mathcal{X}$ **do**
    $T(s) \leftarrow \big(t, \text{random } (1, m_t)\big)$, for $s \in \mathcal{X}_t$
    if $A[i+1]$ contains an element $\langle s', w'\rangle$ with $T(s') = T(s)$ **then**
      $w' \leftarrow w' + w$
      with probability w/w', replace $\langle s', w'\rangle$ in $A[i+1]$ by $\langle s, w'\rangle$
    else
      insert new element $\langle s, w\rangle$ in $A[i+1]$
    end if
  end for
  $i \leftarrow i + 1$
  clear $\mathcal{X}$
end while

of the EST's size, $\hat{\varphi}(S)$. The value of $\hat{Y}_t^i$ estimated by SS depends on the representative node n of type t it expands at level i. Thus, we treat $\hat{Y}_t^i$ as a random variable. Clearly, $E[\hat{\varphi}(S)] = E[\hat{Y}_{t_{s^*}}^1]$, where t_{s^*} is the type of the start state s^* at the first level of the EST.

Intuitively, due to the tree structure, the variance of the estimated value of $E[\hat{Y}_{t_{s^*}}^1]$ produced by SS correlates with the variance of the estimated values of $E[\hat{Y}_t^i]$ for all i and t encountered in the EST. Thus, we are interested in reducing the variance of the estimates of $E[\hat{Y}_t^i]$ for all i and t.

Problem Formulation

Given a collection of nodes $\mathcal{X}$ at level i of the EST and a k-disjoint partition of $\mathcal{X}$, we are interested in allocating the q samples among the k types in $\mathcal{X}$ in a way that reduces the variance of the estimates of $E[\hat{Y}_t^i]$.

Algorithm 7 Adaptive Allocation of Samples

Input: number of samples q, a type system T, and a collection of types $U \subseteq T$ found at level i of the EST

Output: number of samples m_t for each $t \in U$

1: **for** each t in U **do**

2: compute the empirical standard deviation $\hat{\sigma}_t$:

$$\hat{\sigma}_t \leftarrow \sqrt{\frac{1}{B^i_{t,p}} \sum_{a=1}^{B^i_{t,p}} (\hat{Y}^i_{t,a})^2 - \left(\frac{1}{B^i_{t,p}} \sum_{a=1}^{B^i_{t,p}} \hat{Y}^i_{t,a} \right)^2}$$

3: compute the approximated probability mass pr_t:

$$pr_t \leftarrow \frac{w_t}{\sum_{u \in U} w_u}$$

4: **end for**

5: **for** each t in U **do**

6: compute the number of samples m_t:

$$m_t \leftarrow \begin{cases} q/|U| & \text{if } \sum_{u \in U} pr_u \hat{\sigma}_u = 0\,, \\ \frac{pr_t \hat{\sigma}_t}{\sum_{u \in U} pr_u \hat{\sigma}_u} \cdot q & \text{otherwise} \end{cases}$$

7: **end for**

The Algorithm

AA, described in Algorithm 7, allocates the number of samples for each type t proportionally to $\hat{Y}^i_t$'s variance (we describe how to get the samples on which $\hat{Y}^i_t$ is based in our bookkeeping procedure below). Types with higher variance should be sampled more often to reduce the variance of the predictions.

AA takes as input a collection of types U and a number of extra samples q to be allocated among each type in U. AA returns the number of samples m_t that will be allocated for each type t in U. First, AA computes the empirical standard deviation $\hat{\sigma}_t$ for each type $t \in U$ (line 2). The value of $\hat{\sigma}_t$ is computed based on different samples of $\hat{Y}^i_t$. Here, $\hat{Y}^i_{t,a}$ is the a-th sample and $B^i_{t,p}$ is the total number of samples of type t at level i observed until probe p.

AA assumes that the probability mass of each type $t \in U$, pr_t, is known. Although we do not know the pr_t-values exactly, we have estimates of their values given by the weight associated with each type (line 3). Recall that the weight associated with each type t is the estimated number of nodes of type t (see Algorithm 4). Once the values of $\hat{\sigma}_t$ and pr_t are computed, AA determines the number of samples m_t according to the equation shown in line 6.

The m_t-values in Algorithm 7 are not defined as integers. Etore and Jourdain [18] show a complex scheme to compute integer values of m_t that sum up to q. Grover [23] conjectured that rounding down the m_t-values and allocating the remaining samples to the options with the largest

Algorithm 8 Active SS Bookkeeping

Input: sampled tree A, cost bound d, and type systems T and T_r, where T_r is T augmented with a random integer.

Output: a collection Z_u^i for each $u \in T$ of $\hat{Y}_t^i$-values with $t \in T_r$ encountered in A.

```
1:  for i ← tree depth to 1 do
2:    for each node s in A[i] do
3:      Ŷ^i_{T_r(s)} ← 1
4:      for each child s'' of s do
5:        if h(s'') + g(s'') ≤ d then
6:          Ŷ^i_{T_r(s)} ← Ŷ^i_{T_r(s)} + Ŷ^{i+1}_{T_r(s'')}
7:        end if
8:      end for
9:      insert Ŷ^i_{T_r(s)} in Z^i_{T(s)}
10:   end for
11:   i ← i + 1
12: end for
```

allocation ratios would perform as well as the original AA. We round up the m_t-values to the smallest integer greater than m_t because our samples are relatively cheap to obtain.

3.6.3 Bookkeeping

We now show how to compute the values of $\hat{Y}_{t,a}^i$ used in Algorithm 7. SS estimates the size of the EST by summing up the w-values in A. We are able to compute the values of $\hat{Y}_t^i$ for all i and t with some bookkeeping. This is done with dynamic programming as shown in Algorithm 8.

Algorithm 8 receives as input the sampled subtree A produced in a single probe of SS, the cost bound d and two type systems: T and T_r. Recall that T_r is T augmented with a random integer ∇. Algorithm 8 returns one collection Z_u^i for each $u \in T$ of $\hat{Y}_t^i$-values with $t \in T_r$ encountered in A. We iterate over the different levels of A in reverse order, i.e., we go from the deepest level of the sampled subtree to the root (line 1). Then, we compute the values of $\hat{Y}_t^i$ based on the values of $\hat{Y}_t^{i+1}$ already computed (line 6).

Note that the $\hat{Y}_t^i$-values produced by Algorithm 8 are for types $t \in T_r$, but in Algorithm 7 we are interested in the $\hat{Y}_u^i$-values for types $u \in T$. That is why Z_u^i is indexed by types $u \in T$ and not by types $t \in T_r$.

3.6.4 Overall Active Stratified Sampling

In summary, Active SS is described by Algorithms 6, 7 and 8. Active SS uses Algorithm 7 to define the values of m_t (line 13 of Algorithm 6). Note that like Passive SS, Active SS also uses the value of q/k to set the values of m_t during its first probe. Once the first probe is finished and the first $\hat{Y}_t^i$-values are stored in memory, Active SS calls Algorithm 7 to define the m_t-values. Active SS

calls Algorithm 8 after each probe to store the new $\hat{Y}_t^i$-values in memory.

Passive SS expands one node of each type at every level of the EST. In addition to that, in our implementation, Active SS iterates once again over the nodes in the sampled tree A in Algorithms 7 and 8. Thus, an Active SS probe takes about twice as long as a Passive SS probe. In the next section we verify empirically that Active SS's computational overhead can result in better predictions.

3.6.5 Experiments with Active SS and Clustering-Based Type Systems

We run experiments on optimization problems over graphical models such as finding the most likely explanation in Bayesian networks [69]. Specifically, we are interested in predicting the EST size of Depth-First Branch and Bound (DFBnB) with the mini-bucket heuristic [36] while proving a given solution to be optimal. As we explain in detail in Chapter 4, because we provide the optimal solution cost and DFBnB does not detect transpositions, our experimental setup is exactly the same as the one used in Section 3.4 when predicting the size of the IDA* search tree for a given cost bound.

Our experiments are run on three domains: protein side-chain prediction (pdb),[4] randomly generated grid networks (grids), and computing haplotypes in genetic analysis (pedigree). In total, we have 164, 56 and 13 problems, for pdb, grids and pedigree, respectively.

In our experiments we measure the error and the runtime of different prediction schemes. The prediction error is measured with the relative unsigned error. We repeat each prediction task 10 times and, in our plots, we show the average relative unsigned prediction error on the y-axis and runtime in seconds on the x-axis for different numbers of probes. In addition to the average error, assuming the error follows a normal distribution, we also show the 95% confidence interval with error bars. Note that in some cases the error bars can hardly be noticed.

We compare three algorithms: Active SS, Passive SS, and Purdom's algorithm. In preliminary results, Purdom's algorithm produced much better predictions than Knuth's algorithm. Thus, we compare Active and Passive SS using clustering-based type systems to Purdom's algorithm, the best prediction algorithm, to the best of our knowledge, that does not use a type system.

We use the following set of input values: $k = \{2, 3, 4, 5, 15, 25\}$ and $q = \{10, 100, 150\}$. Note that for Purdom's algorithm k always equals 1. The number of probes used in our experiments depends on the algorithm as the runtime of each probe differs from algorithm to algorithm. We choose the number of probes so that we can compare the accuracy and the runtime of the different prediction methods. First, we show average results for the different values of q and k (Figures 3.6,

[4]Note that when we write PDB with capital letters we mean pattern database, and when we write pdb with uncapitalized letters we mean protein side-chain prediction.

3.7 and 3.8). Then, we show results for different values of k and number of probes p, for $q = 100$ (Figure 3.9). Finally, we show results for different values of q and p, for $k = 25$ (Figure 3.10). The results for $q = 100$ and $k = 25$ are representative in the sense that they are qualitatively similar to the prediction results when q and k assume other values.

We are interested in predictions that can be produced much quicker than the time required to run the actual search. Thus, we choose the input parameters p, q and k in a way that the algorithms can quickly produce predictions. The average search times (DFBnB's runtime to prove a solution to be optimal) for the pdb, grids and pedigree domains is 11,008, 25,591, and 5,710,850 seconds, respectively. On the same machine, the predictions are produced within 8 seconds—a small fraction of the actual DFBnB runtime.

Averaged Results

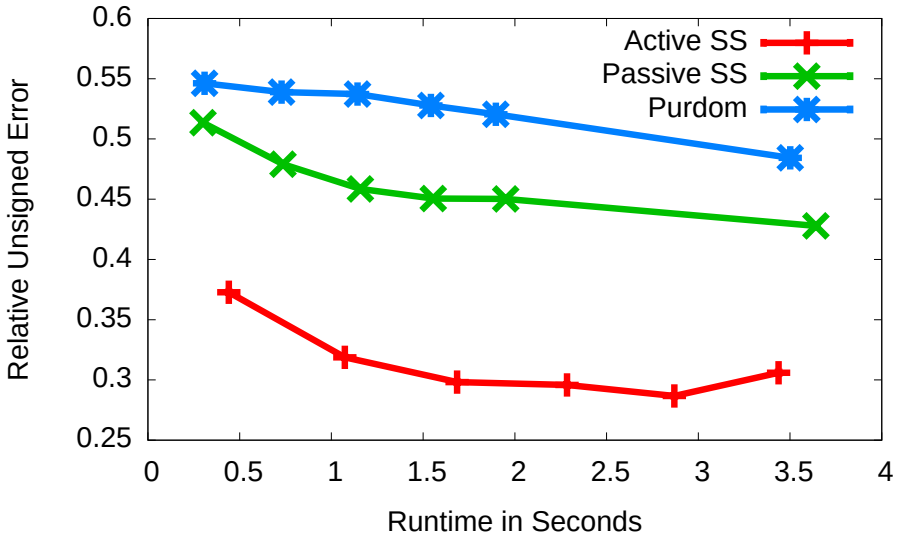

Figure 3.6: Average results for Active SS, Passive SS and Purdom's algorithm on the pdb domain.

Figures 3.6, 3.7, 3.8 present the average results. Here, each point in the plots represents the prediction error and runtime averaged over all different values of q and k for a different value of p (number of probes). Active SS performs best on average in all three domains: it produces more accurate predictions in less time than Passive SS and Purdom's algorithm. In the pdb domain the difference between Active SS and other schemes is larger. Passive SS using clustering-based type systems performs better than Purdom's algorithm in two out of three domains. In the pedigree domain Passive SS and Purdom's algorithm produce predictions of similar quality.

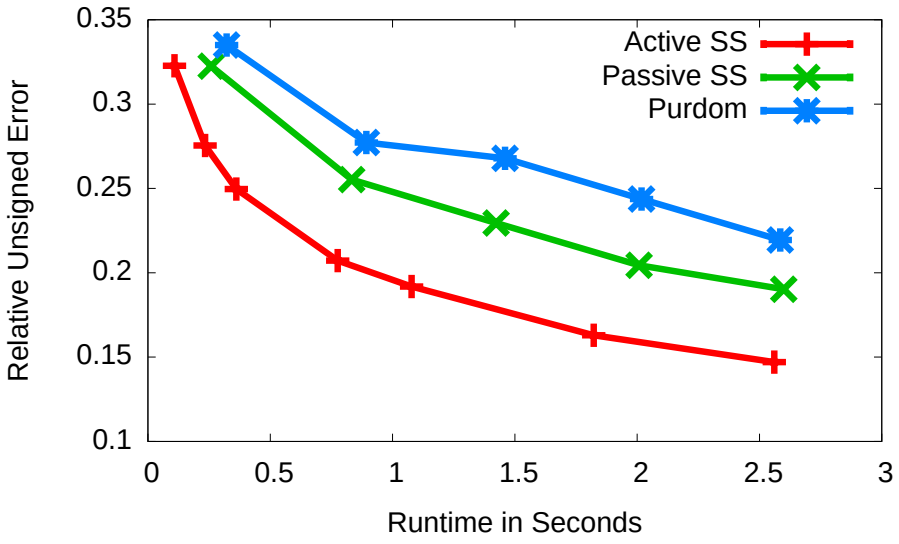

Figure 3.7: Average results for Active SS, Passive SS and Purdom's algorithm on the grids domain.

Empirical Study of the Parameter q

Figure 3.9 shows the prediction results for q in $\{10, 100, 150\}$ and $k = 25$. As we increase q, the predictions tend to become more accurate at the cost of increasing their runtime. Active SS outperforms Passive SS and Purdom's algorithm except in the pedigree domain with q-values of 100 and 150, where the predictions produced by Active SS and Purdom's algorithm are of similar quality.

The results in Figure 3.9 suggest that Passive SS is competitive with Purdom's algorithm for lower values of q, but is outperformed for larger values of q. As an illustrative example, consider the following case. For very large values of q, all three algorithms will expand a number of nodes close to the actual EST size, and Purdom's algorithm should be preferred in such cases as it does not have the computational overhead the SS algorithms have for computing the types. However, in practice, in order to obtain fast predictions, one should use manageable values of q. Our results suggest that Active SS is the algorithm of choice for lower values of q.

Empirical Study of the Parameter k

Figure 3.10 shows the empirical results for the k in $\{2, 5, 15\}$ and $q = 100$. Since k always equals 1 in Purdom's algorithm, we repeat the curve for Purdom's algorithm in plots of the same domain.

The results in Figure 3.10 suggest that Active SS is more robust than Passive SS to variations in the value of k. Although Active SS presents some improvement when increasing the value of k, the gain is not substantial (e.g., in the pdb domain the prediction error goes from 0.27 to 0.22 when

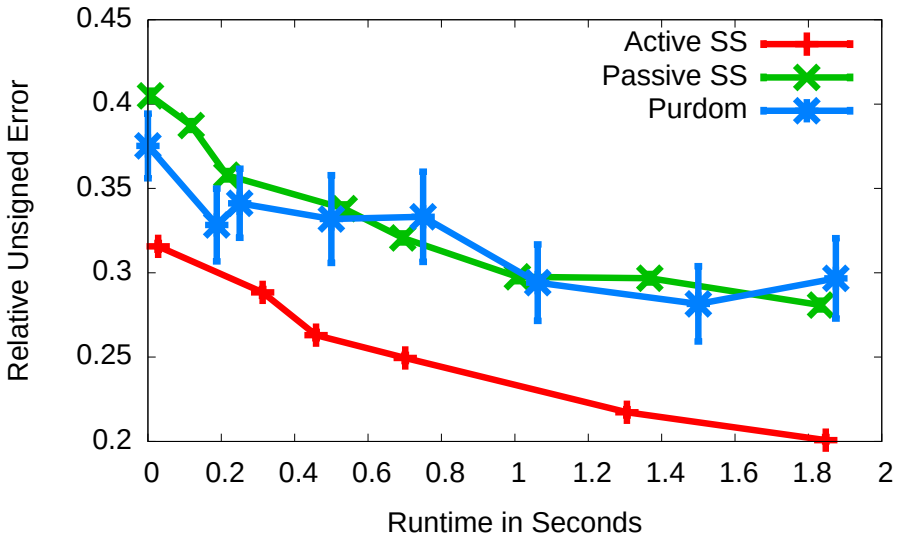

Figure 3.8: Average results for Active SS, Passive SS and Purdom's algorithm on the pedigree domain.

increasing k from 2 to 15 in predictions produced in approximately 2 seconds). Note that for larger values of k the runtime of the k-means algorithm used for defining the type system increases. We conjecture that, despite the increase in runtime for computing the types, Active SS is able to use the extra knowledge provided by the larger number of clusters to allocate more samples in important parts of the search tree, i.e., the parts in which the uncertainty is higher. The same phenomenon is not observed with Passive SS. For instance, in the case of the grids domain, Passive SS tends to produce worse predictions as we increase the value of k.

3.6.6 Discussion

The bookkeeping required by the AA algorithm is not expensive for the general case of stratified sampling because one has immediate access to the value of a sample [2]. By contrast, the bookkeeping required by AA in the context of search trees is relatively expensive as one has to re-expand the nodes expanded by SS to compute the value of the samples. Thus, one probe of Active SS is about twice as slow as a probe of Passive SS. Our experimental results showed that although the bookkeeping of Active SS is somewhat expensive, because it carefully chooses the parts of the search tree that should be sampled more often, it produces better predictions than Passive SS.

Active SS is general and could be applied to other problems. For instance, Bresina et al. [7] used Knuth's algorithm to measure the expected solution quality of scheduling problems. One could also use Active SS to reduce the variance of Bresina et al.'s algorithm. Assuming a type system is available, instead of measuring the variance of the size of the subtree rooted at nodes of the same

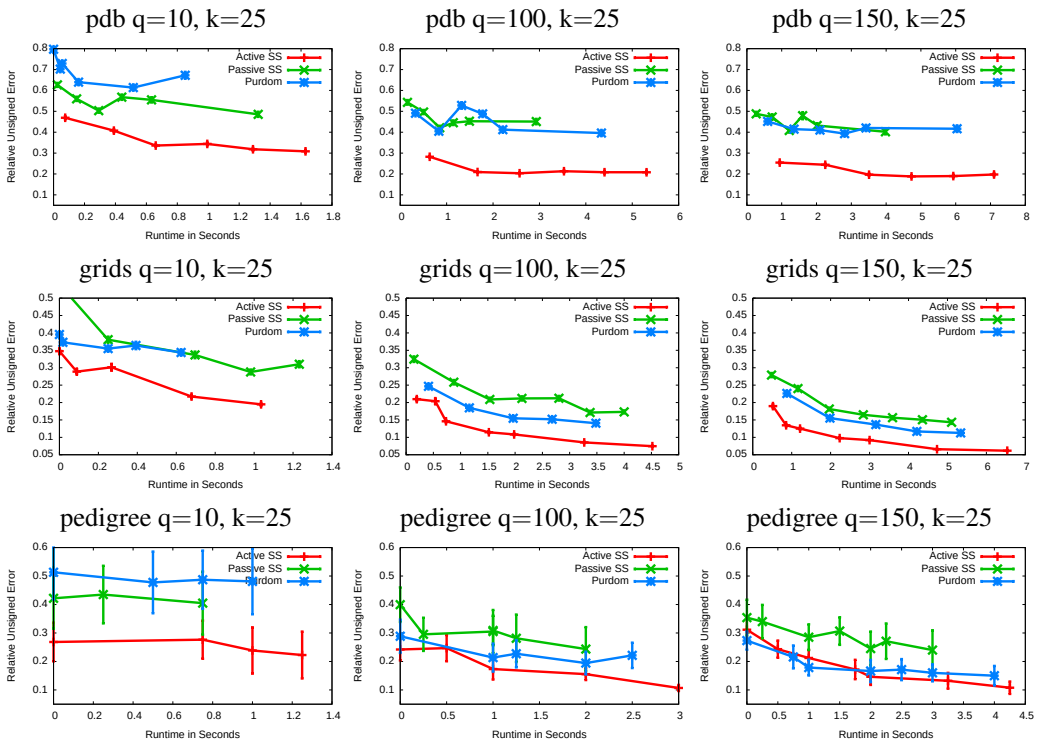

Figure 3.9: Active SS, Passive SS and Purdom's algorithm for different values of q.

type, one would measure the variance of the quality of the solution found in the subtrees rooted at nodes of the same type.

Passive SS produces better predictions than Active SS when "very good" type systems are available. As an illustrative example, Passive SS performs better than Active SS when the type system used is perfect, i.e., nodes of the same type root subtrees of the same size. Clearly, with a perfect type system a single probe of Active or Passive SS suffices to produce a perfect prediction. Therefore, one will be better off with the fastest algorithm. As a practical example, we showed that Passive SS produces near-perfect predictions on domains with integer-valued heuristics such as the Sliding-Tile puzzle and the Pancake puzzle. We do not expect Active SS to present gains over Passive SS in those domains.

3.7 Related Work

Many other algorithms were developed based on Knuth's ideas for predicting the EST size. For instance, Kilby et al. [37] introduced an online estimator of the size of backtrack search trees of branch-and-bound search algorithms. Later Haim and Walsh [24] used Kilby et al.'s method as a feature for their machine-learned online algorithm for estimating the runtime of SAT solvers.

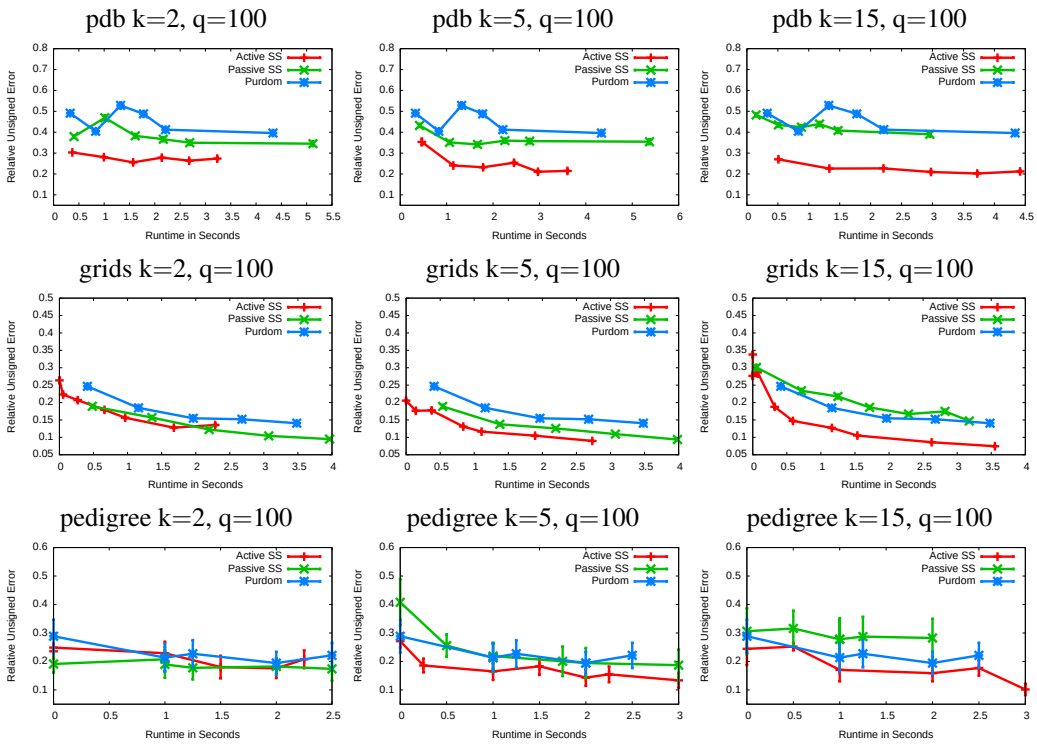

Figure 3.10: Active SS, Passive SS and Purdom's algorithm for different values of k.

Allen and Minton [1] adapted Knuth's algorithm for constraint satisfaction problems; Lobjois and Lemaître [62] used Knuth's algorithm to select the most promising branch-and-bound algorithm for a given problem; Bresina et al. [7] used Knuth's algorithm to measure the expected solution quality of a scheduling problem. All these algorithms could potentially benefit from the idea of using a heuristic function (or some other source of information) to define type systems to reduce the variance of random probing.

Haslum et al. [26] used KRE to evaluate different PDB heuristics for domain-independent planning. They posed the problem of selecting good abstractions to construct pattern databases as an optimization problem. A hill climbing search algorithm is then used with the KRE formula as an evaluation function. Other methods such as CDP and SS could also be used for this purpose.

Breyer and Korf [8] showed how to use KRE to make accurate predictions of the number of nodes expanded on average for the special case of consistent heuristics by the A* algorithm for the 15-puzzle. In order to make predictions of the number of nodes expanded by A*, due to the transposition detection the algorithm does, one needs to know the number of nodes at a level i in the brute-force search graph [8]. For domains in which the search graph cannot be enumerated, accurately predicting the number of nodes expanded by A* remains an open problem.

Burns and Ruml [10] presented IM, a prediction method that works in domains with real-valued edge costs. IM was developed to make online estimations of the number of nodes expanded by IDA* as the algorithm searches. Burns and Ruml's goal was to avoid the poor performance of IDA* in domains with real-valued edge costs by setting a cost bound d that would make IDA* expand an exponentially larger number of nodes in each iteration. IM works by learning the variation of the f-value between a node and its children. Like CDP, this is done based on a type system, i.e., IM learns the value Δ by which the f-value changes when a node of type u generates a node of type t. In fact, CDP can be seen as a special case of IM, when the edges have unitary cost. The difference between CDP and IM is that IM implicitly incorporates the cost to generate a child in its type system, CDP assumes in its formulas that the cost is always one. Not surprisingly, Burns and Ruml verified empirically that in domains with unit edge-costs IM and CDP produce predictions with indistinguishable accuracy. Like CDP, IM could also benefit from ϵ-truncation.

3.8 Conclusions

In this chapter we improved the runtime and prediction accuracy of CDP and SS, two algorithms that were developed independently of each other for predicting the number of nodes expanded on an iteration of a backtrack search algorithm such as IDA*.

For the CDP algorithm, we have identified a source of prediction error that had previously been overlooked, namely, that low probability events can degrade predictions in certain circumstances. We call this the discretization effect. This insight led us to the ϵ-truncation method for altering the probability distribution used for making predictions at level i of the search tree by setting to zero all probabilities smaller than ϵ_i, an automatically derived threshold for level i. Our experimental results showed that more informed type systems for prediction often suffer more from the discretization effect than less informed ones, sometimes leading to the pathological situation that predictions based on the more informed system are actually worse than those based on the less informed system. In our experiments ϵ-truncation rarely degraded predictions; in the vast majority of cases it improved predictions, often substantially. In addition, we presented L−CDP, a variant of CDP that can be orders of magnitude faster than CDP and is guaranteed to make the same predictions as CDP.

For the SS algorithm, we showed that type systems employed by CDP can also be used as stratifiers for the SS algorithm. Our empirical results showed that SS employing CDP's type systems substantially improves the predictions produced by SS as presented by Chen.

We compared empirically our enhanced version of CDP and SS using heuristic-based type sys-

tems. Our experimental results point out that if CDP's preprocessing time is acceptable or can be amortized, it is suitable for applications that require less accurate but very fast predictions, while SS is suitable for applications that require more accurate predictions but allow more computation time.

We presented the clustering-based type systems as an alternative to the constant C approach for creating effective type systems for domains with real-valued heuristics. We tested the clustering-based type systems with SS by comparing it to Purdom's algorithm, the best method, to the best of our knowledge, that does not use a type system. Our results showed that, although SS using clustering-based type system has the overhead of computing the types with the k-means algorithm, it tends to produce better predictions than Purdom's algorithm.

Finally, we presented Active SS, a prediction algorithm that uses the ideas of active sampling in the context of search trees. In contrast with other active sampling approaches, it is not immediately clear that Active SS can perform better than Passive SS due to its relatively expensive bookkeeping procedure. Our empirical results showed that Active SS can perform better than Passive SS.

Chapter 4

Predicting the Size of Depth-First Branch and Bound Search Trees

4.1 Introduction

A frequently used heuristic search algorithm for solving combinatorial optimization problems is Depth-First Branch-and-Bound (DFBnB) [4]. Similarly to IDA*, predicting the DFBnB EST size could facilitate the choice of a heuristic on an instance by instance basis. Or, in the context of parallelizing search, a prediction scheme could facilitate load-balancing by partitioning the problem into subproblems of similar EST sizes [66].

4.1.1 Problem Formulation

Given a directed and implicitly defined search tree representing a state-space problem [65] rooted at start state s^*, called the underlying search tree (UST), we are interested in estimating the size of a subtree which is expanded by a search algorithm, called the Expanded Search Tree (EST), while seeking an optimal solution.

Problem formulation. Let $S = (N, E)$ be a tree representing an EST where N is its set of nodes and for each $n \in N$ $child(n) = \{n' | (n, n') \in E\}$ is its set of child nodes. Our task is to estimate the size of N without fully generating S.

4.1.2 Background

In Chapter 3 we showed that several methods, such as `SS`, have been developed for predicting the size of the search tree of backtracking and heuristic search algorithms such as IDA* [44]. These prediction schemes work by sampling a small part of the EST and extrapolating from it. The challenge in applying these sampling techniques to DFBnB lies in their implicit assumption of the "stable children" property. Namely, for every node in the EST, the set of EST children can be

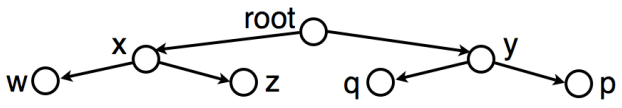

Figure 4.1: Example of an underlying search tree (UST).

determined at the time of sampling. In the case of DFBnB, however, the set of children in the EST depends on c^b , the cost value of the current best solution, which controls the pruning but is generally not known at prediction time because it can change during search (see Algorithm 3 on page 12).

The stable children property. A hidden assumption made by SS is that it knows the set of child nodes in the EST for every node in the EST. If used for predicting the DFBnB EST size, SS would have to assume that child nodes are pruned only if their f-value is greater than or equal to the initial upper bound c^b, which is accurate for algorithms such as IDA*, for which the cost bound is fixed. However, as acknowledged by Knuth [38], this sampling scheme would not produce accurate predictions of the EST generated by DFBnB because the exact child nodes of nodes expanded by DFBnB are not known unless we fully run the algorithm.

Definition 16 (stable children property)**.** *Given an EST $S = (N, E)$ implicitly specified. The stable children property is satisfied iff for every $n \in N$ in S along a path leading from the root, the set $child(n)$ in N can be determined based only on the information along the path.*

Proposition 1. *The stable children property for the EST of DFBnB is guaranteed if and only if the initial upper bound c^b is already optimal.*

Proof. By counter example. Assume a fixed bound c^b input to SS for estimating the search tree expanded by DFBnB exploring the search space in Figure 4.1. We assume DFBnB follows a left-first node ordering. Let c_{left} be the cost of the best solution found when exploring the left hand-side of this search tree. When the search algorithm considers child y of the root, then if $f(y) \geq c_{left}$, y will not be expanded, and otherwise it will. In the former case, the EST will have only x as a child to the root and in the latter the EST will have both x and y as child nodes to the root; distinguishing between these two cases is not possible without actually executing DFBnB on the left-hand side of the search tree. □

Weighted Backtrack Estimator

Kilby et al. [37] presented the Weighted Backtrack Estimator (WBE), a method that ensures the stable children property by running alongside the DFBnB search and using the explored branches to

predict unvisited ones and through that the DFBnB EST size. Because WBE visits the nodes in the EST in the same order DFBnB does, it knows exactly the value of c^b when visiting each node in the tree. This strategy ensures that the stable children property holds for WBE. Moreover, because it fully explores the EST, WBE is guaranteed to produce perfect predictions when the search finishes.

On the down side, as we show in our empirical results, WBE usually only produces accurate predictions when the DFBnB search is about to finish. In practical scenarios one is interested in accurate predictions that can be produced much quicker than the actual search.

4.1.3 Contributions

Most of the content of this chapter is published in the Proceedings of the International Joint Conference on Artificial Intelligence (2013) [56]. In this chapter we make the following contributions.

- We present *Two-step Stratified Sampling* (TSS), an algorithm for predicting the EST size of DFBnB. The algorithm performs multiple "SS runs" followed by a constrained DFBnB execution and exploits memory to cope with the stable children issue. We show that when given sufficient time and memory the prediction produced by TSS converges to the actual EST size.

- We apply our prediction scheme to optimization queries over graphical models, such as finding the most likely explanation in Bayesian networks [69] (known as MPE or MAP). In particular, we are interested in predicting the EST size of DFBnB using the mini-bucket heuristic [36]. In addition to comparing against pure SS, we compare TSS to WBE. Empirical results show that, if memory allows, our prediction is effective and overall far superior to earlier schemes.

4.1.4 Chapter Outline

The remainder of this chapter is structured as follows. In Section 4.2 we explain Two-Step Stratified Sampling (TSS). The empirical results are presented in Section 4.3, which is followed by a discussion in Section 4.4. Finally, we make concluding remarks in Section 4.5.

4.2 Two-Step Stratified Sampling

In order to cope with the lack of the stable children property, we extend SS to approximate the actual set of child nodes in DFBnB's EST using a two-step algorithm.

SS was presented in Chapter 3, but we repeat it here in Algorithm 9 with the following minor changes to accommodate predictions of the DFBnB EST size.

- We note that the output of Algorithm 9, A, forms a subtree ST of the UST.
- We use the term "upper bound" instead of the term "cost bound" used in Chapter 3.
- We implement pruning with a "less than" sign (see line 6 of Algorithm 9) instead of a "less than or equal to" sign used in Chapter 3. This is because now we know a solution with cost c^b and DFBnB is interested in expanding nodes that could lead to solutions strictly better than the current best solution.

In the **first step** we use SS to generate a subtree ST of the UST, assuming a fixed upper-bound c^b which is derived by some preprocessing (e.g., local search). In the **second step** we prune ST by simulating a run of DFBnB restricted to nodes in ST. As we are going to see later in the chapter, our predictions will be more accurate when ST is closer to the UST. However, our algorithm will require more memory and time to produce predictions for larger subtrees ST.

This two-step procedure yields our two-step Stratified Sampling algorithm (TSS). In contrast to SS, TSS is guaranteed to eventually converge to correct predictions of the size of the EST of DFBnB. TSS also uses the C&C paradigm and it assumes that nodes of the same type root isomorphic subtrees. This is because TSS is interested not only in the size of the search tree, but also in the distribution of solution costs in a given subtree.

For clarity, we describe a naive version of the second step of TSS first, and then present an improved version.

4.2.1 Naive Second Step

Algorithm 10 describes a naive second step of TSS. It uses an array $SIZE[\cdot]$ indexed by types, where $SIZE[t]$ is the estimated size of the subtree rooted at node s representing type t in ST. Like DFBnB, we explore the UST in a depth-first manner, pruning nodes based on c^b (line 7) and updating the value of c^b (line 3). However, in contrast to DFBnB, we restrict the algorithm to only expand node s representing the type of child node s' in ST (line 8). Hence Algorithm 10 explores only one node of each type in a depth-first fashion while keeping track of the value of the upper bound c^b. If the value of $SIZE[T(s)]$ has not been computed, i.e., $SIZE[T(s)] = 0$, then TSS computes it by calling $SecondStep$ recursively (line 10).

Limitation of the Naive Approach

Algorithm 10 corrects (to some extent) for SS's lack of access to the actual child nodes in EST as it searches depth-first and updates the upper bound c^b. However, Algorithm 10 is not sufficiently

Algorithm 9 Stratified Sampling, a single probe

Input: root s^* of a tree, type system T and initial upper bound c^b.
Output: array of sets A, where $A[i]$ is the set of pairs $\langle s, w\rangle$ for the nodes s expanded at level i, which forms a subtree ST of the search tree being sampled.

```
1: initialize A[0] ← {⟨s*, 1⟩}
2: i ← 0
3: while i is less then search depth do
4:    for each element ⟨s, w⟩ in A[i] do
5:       for each child s'' of s do
6:          if h(s'') + g(s'') < c^b then
7:             if A[i + 1] contains an element ⟨s', w'⟩ with T(s') = T(s'') then
8:                w' ← w' + w
9:                with probability w/w', replace ⟨s', w'⟩ in A[i + 1] by ⟨s'', w'⟩
10:            else
11:               insert new element ⟨s'', w⟩ in A[i + 1]
12:            end if
13:         end if
14:      end for
15:   end for
16:   i ← i + 1
17: end while
```

sensitive to the bound updates when estimating the size of subtrees rooted at nodes of the same type.

Example 8. *Assume two nodes n_1 and n_2 of type $T(n_1) = T(n_2)$ that root identical subtrees in the UST; given an initial upper bound c^b, DFBnB might expand fewer nodes when exploring n_2's subtree than when exploring n_1's subtree if n_1 is explored first, because it might find a solution that yields a tighter upper bound c'. Thus, when exploring n_2's subtree the value of c' found in n_1's subtree will allow extra pruning.*

4.2.2 Using Histograms

In order to handle the above problem and to get a more accurate knowledge we propose to collect more information as follows. We consider the distribution of f-values in the subtree rooted at the nodes representing types in our prediction scheme. The distribution of f-values under a node n is stored as a *histogram*, which will initially be indexed by types.

Definition 17 (Histogram)**.** *Let ST be the sampled tree generated by* SS *based on type system T and initial upper bound c^b. The histogram of a type t, $hist_t$, is a set of pairs (k, r) for each observed f-value k in the subtree rooted at node $n \in ST$ whose type is t , and r is the estimated number of nodes whose f-value is k in n's subtree.*

Example 9. *Consider the UST shown in Figure 4.1 and a type system where $T(x) = T(y)$, $f(x) = f(y) = 1$, $f(w) = f(q) = 3$, $f(z) = f(p) = 2$, and an initial upper bound $c^b = 4$.* SS

Algorithm 10 SecondStep(s^*, T, ST, c^b), naive version

Input: root of a tree s^*, type system T, and sampled tree ST, initial upper bound c^b
Output: estimated size of the search tree rooted at s^*

```
1: Initialize once: ∀t ∈ T, SIZE[t] ← 0.
2: if s* is a solution node with cost better than c^b then
3:    update upper bound c^b
4: end if
5: SIZE[T(s*)] ← 1
6: for each child s' of s* do
7:    if g(s') + h(s') < c^b then
8:       s ← node representing T(s') ∈ ST
9:       if SIZE[T(s)] = 0 then
10:         SecondStep(s, T, ST, c^b)
11:      end if
12:      SIZE[T(s*)] ← SIZE[T(s*)] + SIZE[T(s)]
13:   end if
14: end for
15: return SIZE[T(s*)]
```

produces a sampled tree ST with either the right or the left branch, since $T(x) = T(y)$. Assume SS *produces the left branch with nodes x, w, and z. After exploring the subtree rooted at x, we store the histogram: $hist_{T(x)} = \{(1,1),(2,1),(3,1)\}$, as the subtree contains one node whose f-value is 1 (node x), one with f-value of 2 (node z), and one with f-value of 3 (node w). Let us ignore how the histogram is computed for now (this is explained in detail in Algorithm 11 below). We also update the upper bound c^b to 2, which is the value of the solution found in node z.*

When we backtrack to the subtree rooted at y, we use the histogram $hist_{T(x)}$ to estimate the size of the subtree of y because $T(x) = T(y)$. This is done by summing up all the r-values of the entries of $hist_{T(x)}$ whose f-value is less than 2, the current upper bound. Thus, we estimate that the subtree rooted at node y has one node (the entries $(2,1)$ and $(3,1)$ of $hist_{T(x)}$ are pruned), which is exactly DFBnB's EST size. Using Algorithm 10 would yield $SIZE(T(x)) = 3$, however.

4.2.3 Union over Multiple Sampled Trees

Before describing the full second step of TSS, we propose another extension of SS that is orthogonal to the use of histograms. Instead of applying the second step to a single subtree generated by SS, we propose to run m independent probes of SS and take the union of the m subtrees into a single tree called $UnionST$. For every path in $UnionST$ there exists a path in at least one of the sampled trees. The aim is to provide the second step of TSS with a subtree that is closer to the UST. The number of probes of SS is a parameter that controls the first step. In the second step we compute histograms for nodes in $UnionST$. $UnionST$ could have multiple nodes representing the same type. Thus, in contrast with Example 9, in the second step of TSS we are going to index the histograms by nodes

Algorithm 11 SecondStepHist($s^*, T, UnionST, c^b$)

Input: root of a tree s^*, type system T, $UnionST \leftarrow$ union of m sampled trees, initial upper bound c^b

Output: histogram $hist_{s^*}$ and estimate of the EST's size.

1: Initialize once: $\forall s \in UnionST, hist_s \leftarrow \emptyset$.
2: $hist_{s^*} \leftarrow \{(f(s^*), 1)\}$
3: **if** s^* is a solution node with cost better than c^b **then**
4: update lower bound c^b
5: **end if**
6: **for** each child s of s^* **do**
7: **if** $g(s) + h(s) < c^b$ **then**
8: **if** s is not in $UnionST$ **then**
9: $s \leftarrow$ random $n \in UnionST$ with $T(n) = T(s)$
10: **end if**
11: **if** $hist_s \neq \emptyset$ **then**
12: $histPruned_s \leftarrow prune(hist_s, c^b)$
13: $hist_{s^*} \leftarrow hist_{s^*} + histPruned_s$
14: **else**
15: $hist_s \leftarrow SecondStepHist(s, T, UnionST, c^b)$
16: $hist_{s^*} \leftarrow hist_{s^*} + hist_s$
17: **end if**
18: **end if**
19: **end for**
20: return $hist_{s^*}$ and the sum of the r-values of $hist_{s^*}$

in $UnionST$ instead of types and allow multiple histograms for the same type. We now describe the actual second step of TSS.

4.2.4 Overall TSS

Algorithm 11 describes the second step of TSS. It includes a recursive procedure that computes the histograms for nodes in $UnionST$ that are generated in the first step. The sum of the r-values of the histogram of the start state is the TSS prediction of the size of the EST.

TSS first generates $UnionST$ which is the union of m trees ST generated by SS. It then applies DFBnB restricted to the nodes in $UnionST$ while computing the histograms and doing the appropriate pruning. The histogram of the start state, $hist_{s^*}$, initially contains only one entry with value of one for the f-value of s^* (line 2). The histogram $hist_{s^*}$ is computed recursively by computing the histogram of each child s of s^* and combining these histograms' entries into a single histogram (lines 13 and 16).

The combined histogram $hist_3$ of two histograms $hist_1$ and $hist_2$ has one entry for each k-value found in $hist_1$ and $hist_2$. The r-value of each k in $hist_3$ is the sum of the corresponding r-values of $hist_1$ and $hist_2$.

If the histogram of s was already computed and stored in memory, *i.e.,* $hist_s \neq \emptyset$ (line 11),

then we "prune" $hist_s$ according to the *current* c^b (line 12), as shown in Example 9. The function $prune(\cdot,\cdot)$ receives as parameters the histogram $hist_s$ and the current upper bound c^b. It returns a histogram, $histPruned_s$, that contains all entries in $hist_s$ for which the k-value is lower than c^b, as illustrated in Example 9.

If TSS generates a node s that is in $UnionST$, then we use the histogram of s itself as an estimate of the size of the subtree rooted at s. If s is not in $UnionST$, we use the histogram of any node representing $T(s)$ in $UnionST$, chosen randomly (line 9).

4.2.5 Type Systems

As described in Chapter 3, a type system can be defined based on any feature of the nodes. In our experiments we use the f-value of the nodes as types. Namely, nodes n and n' are of the same type if they are at the same depth and have the same f-value.

As discussed in Section 3.5, the cost function input and the derived heuristic bounds are real-valued in optimization problems over graphical models. As a result, a type system built on floating point equality might be far too large for producing efficient predictions. In this chapter we address this issue by using the constant C approach, i.e., we multiply an f-value by a constant C and truncate it to the integer portion to compute a node's type; different values of C yield type systems of different sizes. Note that one could also use the clustering-based type systems we described in Section 3.5.2 with TSS. Finally, we note in this context that we use the same strategy to control the size of the histograms, which are also determined by the number of different f-values.

4.2.6 Theoretical Analysis of TSS

We assume that all type systems used with TSS are f-preserving type systems, which we now define.

Definition 18 (f-Preserving Type System)**.** *A type system T is said to be f-preserving if for every $t \in T$, all nodes of type t have the same f-value.*

Proposition 2. *Let T be an f-preserving type system. Then, the size of any histogram computed by* TSS *is at most $|T|$.*

Proof. The largest size a histogram computed by TSS can assume is the number of different f-values encountered in the search tree SS samples. The minimum size of a f-preserving type system is the number of different f-values encountered in the search tree SS samples. This number is upper-bounded by $|T|$. □

TSS is efficient because it explores only one node of each type in the search tree sampled by a single probe of SS.

Theorem 2 (Complexity)**.** *Given a type system T and a number of probes m, the memory complexity of* TSS *is $O(m \times |T|^2)$, where $|T|$ is the size of T. The time complexity of* TSS *is $O(m \times |T|^2 \times b)$, where b is the branching factor of the UST.*

Proof. TSS stores in memory the subtree ST and one histogram $hist_t$ for each type t encountered in ST. The size of the type system $|T|$ bounds the size of ST and from Proposition 2 we know that $|T|$ also bounds the size of the histograms. Therefore, TSS's memory complexity is $O(m \times |T|^2)$.

In the first step TSS expands one node of each type while generating the subtree ST. In the second step TSS expands the same nodes expanded in the first step as it computes the histogram for each node in ST. While computing the histogram of each node in ST, TSS combines b different histograms, where b is the UST's branching factor. While combining two histograms TSS iterates once over each element of the histograms. Therefore, the time complexity of TSS is $O(m \times |T|^2 \times b)$. □

One could run TSS with $m = 1$ multiple times and average the results, like SS does with its probes. However, this approach is not guaranteed to converge to the correct value. As the number of probes m goes to infinity, TSS converges properly to the correct value, simply because the subtree accumulated in $UnionST$ approaches a superset of nodes in the actual DFBnB EST in the limit.

Lemma 3. *If $UnionST \supseteq EST$, Algorithm 11 computes the DFBnB EST size exactly.*

Proof. Because $UnionST$ contains all nodes in the EST, Algorithm 11 expands the same nodes and in the same order as DFBnB. Thus, Algorithm 11 stores one histogram for each node in the EST and computes exactly the size of the EST. □

Consequently, we have the following.

Theorem 3. *Let S be an EST generated by DFBnB using any heuristic function. The probability of* TSS *perfectly predicting the size of S goes to one as the number of probes m goes to infinity.*

Proof. According to Lemma 2, the probability of $UnionST \supseteq EST$ goes one as m goes to infinity. Then, it follows from Lemma 3 that the probability of TSS perfectly predicting the size of S goes to one. □

In practice, however, we are interested in evaluating predictions using a small m for time and memory efficiency reasons.

4.3 Experimental Results

We evaluate TSS and competing schemes by predicting the EST size of DFBnB when using the mini-bucket heuristic (BBMB) [36], on the same three domains used in Section 3.6.5: protein side-chain prediction (pdb), computing haplotypes in genetic analysis (pedigree), and randomly generated grid networks (grids). For each domain we experiment with different mini-bucket i-bounds, yielding different strengths of the heuristic. Every pair of problem instance and i-bound represents a different prediction problem. In total, we have 14, 26, and 54 problems, for pdb, pedigree, and grids, respectively. All experiments are run on 2.6 GHz Intel CPUs with a 10 GB memory limit. We present individual results on selected single problem instances and summary statistics over the entire set.

In addition to SS, we also compare TSS to WBE [37]. We implemented WBE in the context of BBMB. Kilby et al. presented another prediction algorithm, the Recursive Estimator (RE), whose performance was similar to WBE's in their experiments. Both WBE and RE were developed to predict the size of binary trees, but in contrast to WBE it is not clear how to generalize RE to the non-binary search trees we have in our domains.

4.3.1 Results on Single Instances

Figures 4.2, 4.3 and 4.4 show the prediction results for TSS, SS, and WBE on six problem instances for each problem domain. These instances are representative in the sense that they highlight different aspects of the prediction methods. For each plot we show the i-bound used. The x-axis represents the runtime in seconds required to produce the predictions. The DFBnB search time is implied by a plot's rightmost x-value. The y-axis represents, in log-scale, the ratio between the predicted and the actual EST size. A perfect prediction has a ratio of 1, indicated in each plot by a horizontal line. For TSS and SS we show the average ratio over five independent runs, i.e., we run TSS or SS five independent times and compute the average ratio as $\frac{predicted}{actual}$ if $predicted > actual$ and as $\frac{actual}{predicted}$, otherwise—that way we avoid overestimations and underestimations canceling each other out when averaging. We note that the cases in which TSS or SS underestimates the actual EST size are rare. WBE also almost always overestimates the actual EST size. Out of the 18 instances shown in Figures 4.2, 4.3 and 4.4, WBE underestimates the actual EST size only briefly on the 75-16-6 instance (see Figure 4.4) of the grids domain with $i = 10$.

Assuming the error follows a normal distribution, we show the 95% confidence interval with error bars (note that these are hardly noticeable in the plots). For each problem we first run a

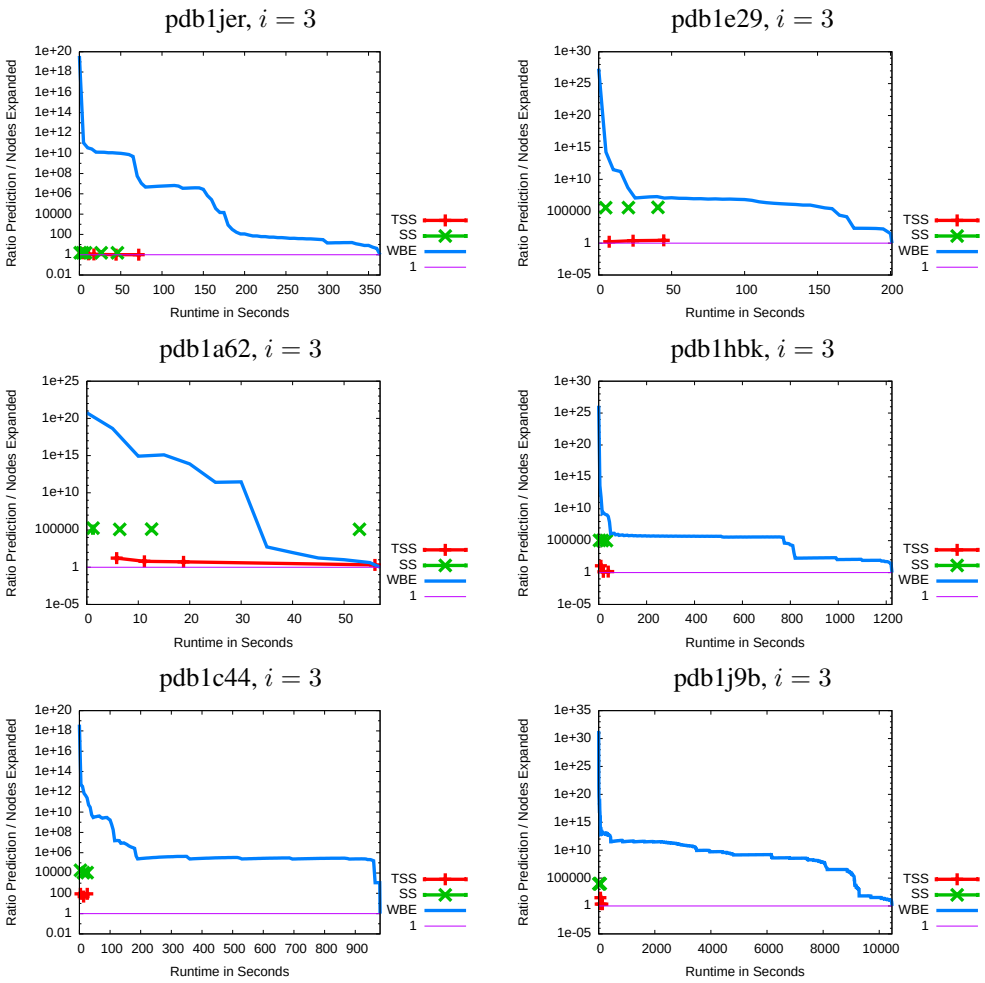

Figure 4.2: TSS, SS and WBE results on single instances of the pdb domain.

limited-discrepancy search [25] with a maximum discrepancy of 1 to find an initial bound c^b which is provided to both DFBnB and to the prediction algorithms. We use $C = 50$ for pdb and $C = 100$ for pedigree and grid instances. We control the prediction runtime and accuracy for TSS and SS with different number of probes, i.e., each point in the plots represents the result for a given number of probes. Note that some plots have more points than others. This is because TSS runs out of memory in a few cases. WBE was set to produce a prediction every 5 seconds.

4.3.2 Results on the Entire Set of Problems

Table 4.1 presents summary results across the entire set of benchmarks. We display the average ratio of the results of TSS using different numbers of probes m and different values of C averaged across different problems. We report TSS's running time through the average percentage of the DFBnB search time (column %). For instance, a %-value of 10 means that the prediction was produced in

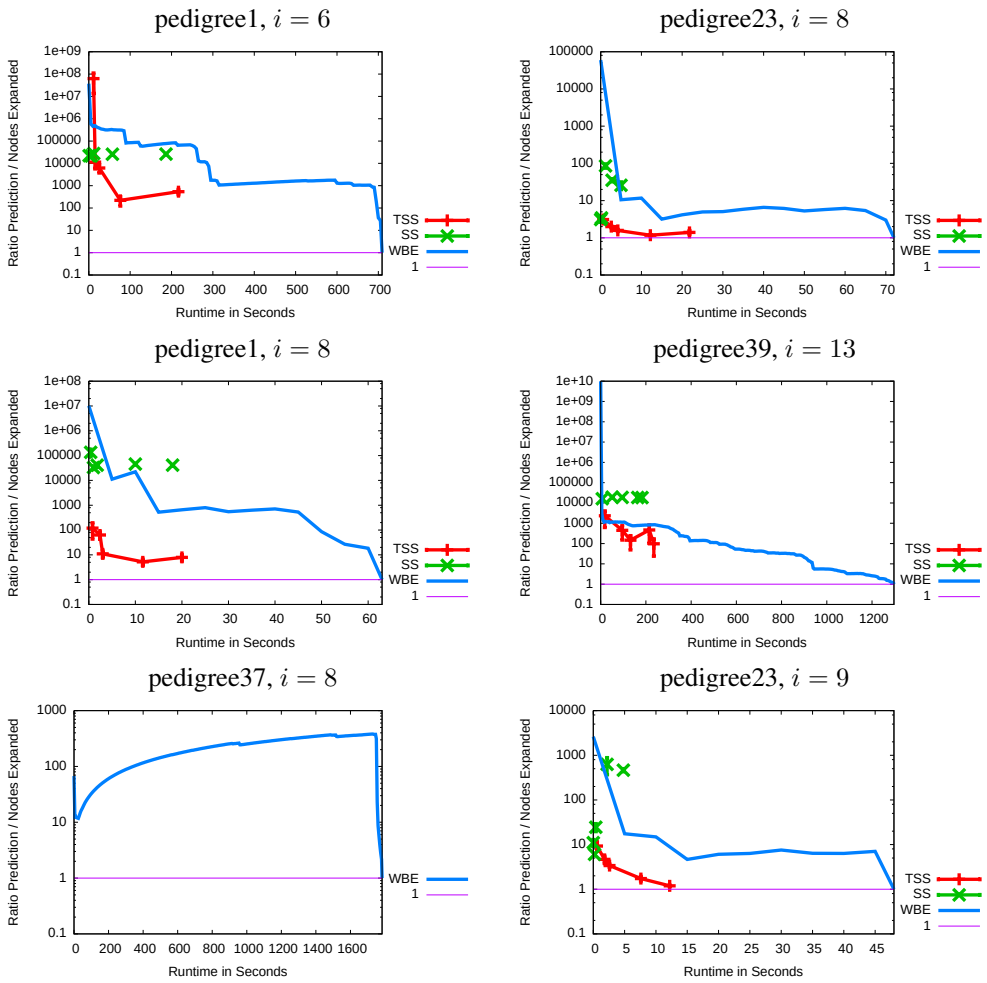

Figure 4.3: TSS, SS and WBE results on single instances of the pedigree domain.

10% of the full DFBnB runtime. For each prediction produced by TSS in x seconds we report one prediction produced by WBE in y seconds such that y is the lowest value greater than x (thereby giving WBE a slight advantage).

Column n shows TSS's coverage, i.e., the number of problems for which TSS is able to produce predictions. Namely, TSS does not produce results either if it takes more time than the actual DFBnB search or if it runs out of memory. Instances for which TSS did not produce results are not included in the averages in Table 4.1, since neither SS nor WBE produced reasonable results on those instances either.

Finally, we highlight in Table 4.1 those entries of TSS where TSS produces more accurate predictions in less time than WBE. SS is not included in the table because it produced unreasonable predictions even when granted more time than the actual DFBnB search.

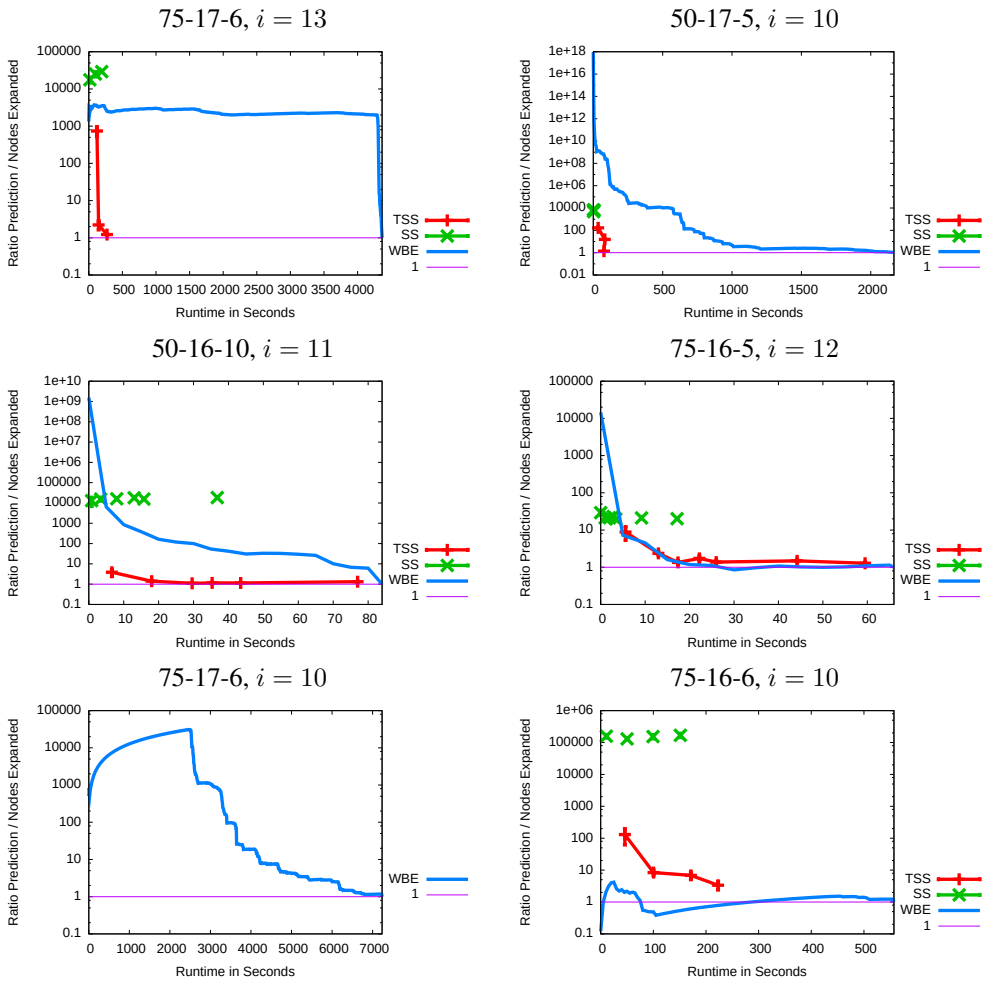

Figure 4.4: TSS, SS and WBE results on single instances of the grids domain.

4.4 Discussion

The results in Figures 4.2, 4.3 and 4.4 suggest that TSS is far superior to both WBE and SS in most of the cases. WBE often produces accurate predictions only when DFBnB is about to finish (DFBnB search time is implied by a plot's rightmost x-value). For instance, in Figure 4.4, on the 75-17-6 grid instance with $i = 13$, WBE is accurate only after more than one hour of computation time, while TSS is able to produce accurate predictions in a couple of minutes.

As anticipated, SS also tends to produce poor predictions. However, if the initial upper bound c^b is in fact the optimal cost, then SS will also produce accurate predictions – in this case the stable children property is satisfied because a full DFBnB run would never update c^b. Although rare, this is depicted in the plot of pdb1jer with $i = 3$ in Figure 4.2.

There are a few instances in which no method was able to produce accurate predictions. See for

pdb (total of 14 problems)															
	TSS ($C = 100$)		WBE			TSS ($C = 50$)		WBE			TSS ($C = 10$)		WBE		
m	ratio	%	ratio	%	n	ratio	%	ratio	%	n	ratio	%	ratio	%	n
1	**66.4**	**3.07**	8.05e+25	3.57	13	**7.34e+03**	**2.6**	2.31e+27	3.4	14	**1.73e+04**	**0.383**	9.02e+30	1.19	14
5	**14.3**	**8.26**	4.37e+20	9.21	12	**779**	**4.9**	8.21e+25	5.6	13	**5.19e+03**	**1.11**	1.55e+30	1.49	14
10	**12.1**	**13.5**	2.98e+20	14.3	10	**78.3**	**7.57**	4.39e+20	7.9	12	**4.34e+03**	**1.87**	1.1e+29	2.69	14
50	**1.1**	**12.2**	8.12e+07	12.5	2	**1.43**	**29.3**	7.59e+13	29.8	4	**174**	**4.96**	9.75e+25	5.37	13
100	-	-	-	-	0	**1.26**	**9.78**	1.34e+14	10.1	3	**3.88**	**8.55**	2.21e+25	9.31	11
pedigree (total of 26 problems)															
	TSS ($C = 1000$)		WBE			TSS ($C = 100$)		WBE			TSS ($C = 10$)		WBE		
m	ratio	%	ratio	%	n	ratio	%	ratio	%	n	ratio	%	ratio	%	n
1	**1.21e+04**	**15.9**	2.88e+04	29.8	12	**4.2e+06**	**2.38**	2.49e+08	14.5	15	**7.45e+06**	**8.47**	1.41e+09	11.4	21
5	**128**	**33.5**	8.58e+03	47.3	10	**1.34e+03**	**18.8**	4.83e+04	28.3	12	**3.17e+06**	**7.09**	7.22e+08	17.3	19
10	**80.4**	**30.4**	9.92e+03	36.7	8	**1.07e+03**	**22.3**	3.17e+04	28.8	12	**3.92e+06**	**5.67**	7.14e+08	14.7	18
50	**1.47**	**58.6**	3.94	81.2	4	**58.8**	**23.6**	3.87e+04	27.1	8	**4.37e+06**	**10.4**	1.3e+08	21.4	15
100	**1.52**	**72.7**	3.46	93.5	3	**72.4**	**42.3**	8.28e+03	54.3	8	6.46e+04	18.9	4.98e+04	32.2	13
grids (total of 54 problems)															
	TSS ($C = 100$)		WBE			TSS ($C = 50$)		WBE			TSS ($C = 10$)		WBE		
m	ratio	%	ratio	%	n	ratio	%	ratio	%	n	ratio	%	ratio	%	n
1	**373**	**20.4**	3.02e+07	34.2	50	**9.95e+03**	**12.7**	4.07e+07	29.6	51	**2.61e+06**	**1.61**	1.41e+10	14.9	53
5	**12.3**	**38.5**	5.53e+06	46.1	45	**196**	**25.4**	1.85e+07	37.8	49	**8.51e+05**	**4.39**	4.81e+08	19.3	53
10	**3.47**	**41.2**	1.23e+07	47.3	40	**24.9**	**31.1**	9.25e+06	40.7	46	**3.02e+05**	**7.89**	1.09e+08	26	53
50	**1.94**	**61.1**	9.75e+04	66.2	20	**3.09**	**44.8**	3.93e+05	49.6	33	**416**	**17.3**	4.49e+07	30.1	50
100	**2.22**	**69.6**	7.21e+04	71.4	12	**1.93**	**57.3**	1.26e+05	63	25	**134**	**23**	4.16e+07	33.4	48

Table 4.1: Summary of experiments on three domains. "ratio" denotes the average ratio between the predicted and the actual EST size, % is the corresponding average prediction time relative to the runtime of complete DFBnB.

example pdb1c44 with $i = 3$ in Figure 4.2 and pedigree1 with $i = 6$ in Figure 4.3.

Instance 75-16-6 with $i = 10$ in Figure 4.4 shows a rare case in which WBE produces better predictions than TSS. Instance 75-17-6 with $i = 10$ also in Figure 4.4 shows the WBE results for a case in which TSS runs out of memory. Note that although WBE produces predictions for this instance, it only produces accurate predictions when the search is about to finish.

Overall, we see that TSS is the only robust method which is able to produce good predictions in a timely fashion. In particular, Table 4.1 shows that TSS can produce predictions orders of magnitude more accurate than WBE on average. WBE yields accurate predictions on the pedigree domain, providing average ratios of 3.94 and 2.79. These are produced, however, when DFBnB is about to finish exploring the whole EST: 81% and 94% of the search is complete, respectively. Table 4.1 also demonstrates the tradeoff between quality and runtime of TSS: as expected, predictions are more accurate with time, namely they get more accurate when m and C grow. A similar expected tradeoff is also observed between accuracy and coverage (see column n).

The main difference between WBE and TSS is that the former is restricted to sample the branches of the EST currently explored by DFBnB, while the latter has the freedom to sample different parts of the EST guided by the type system. In fact, WBE and TSS could have almost opposite sampling behaviors if DFBnB is implemented to explore the subtrees rooted at nodes with lower f-value first (which is standard since nodes with lower f-value tend to be more promising). Thus, WBE tends to explore first the subtrees rooted at nodes of same or "similar" type, while TSS explores subtrees rooted at nodes of different types. Our experimental results suggest that TSS's diversity in sampling

can be effective in practice.

One might find paradoxical the fact that by running TSS we find tighter bounds c^b that can be used to feed DFBnB to search a smaller part of the UST, which automatically outdates the TSS prediction. In this case TSS is likely to provide an upper bound on the size of the DFBnB search tree. However, if one uses TSS to select parameters such as the heuristic function to be used in a set of similar problems of DFBnB-based solvers, then running TSS on part of the problem instances could be enough to determine the best heuristic function to be used in the entire set. For this kind of application the paradox automatically disappears.

4.5 Conclusions

In summary, our empirical results showed that, if memory allows, TSS produces very reasonable predictions, far superior to competing schemes. Yet, TSS is a memory-intensive method. With more time and memory it produces better results, with the memory bound effectively limiting its applicability. Secondly, the UST sampled by SS in the first step of TSS is sometimes far larger than the actual DFBnB EST. In such a case even the first step could take a prohibitive amount of time. We observed this phenomenon especially when no initial upper bound c^b was provided. Clearly, TSS works best in domains with high solution density.

In this chapter we presented *Two-Step Stratified Sampling* or TSS, a prediction algorithm able to produce good estimates of the size of the Expanded Search Tree (EST) of Depth-first Branch and Bound (DFBnB) for optimization problems over graphical models. Building upon Chen's Stratified Sampling (SS) [12], TSS modifies SS so it can handle the lack of the stable children property in the EST of DFBnB.

Chapter 5

Predicting Optimal Solution Cost of State-Space Search Problems

5.1 Introduction

As discussed in previous chapters, heuristic search algorithms such as IDA* [43] and DFBnB [4] are designed to find a least-cost path from a start state to a goal state. The optimal solution cost is found as a byproduct of finding an optimal solution path. In this chapter we are interested in applications in which one only needs to know the optimal solution cost or an accurate estimate of the optimal solution cost—the solution path is not needed. For example, consider an owner of a construction company that is required to quickly assess the monetary cost of a project for bidding purposes. In such a case, only the cost of executing the project is needed. The actual construction plan could be formulated later, if the bid is won.

Thus, an important question to be answered is the following. Can one accurately and efficiently predict the optimal solution cost of a problem without finding the sequence of actions from the start to a goal? In fact, Korf, Reid and Edelkamp [50] stated: "Predicting the optimal solution cost for a given problem instance (...) is an open problem". In this chapter we present two algorithms that efficiently and accurately predict the optimal solution cost of a problem instance.

5.1.1 Problem Formulation

Given a directed and implicitly defined search tree representing a state-space problem [65] rooted at start state s^*, called the underlying search tree (UST), we are interested in estimating the optimal solution cost of a path from s^* to a goal node without necessarily finding an actual path from s^* to a goal node. In this chapter we assume state-space problems with unit-edge costs.

5.1.2 Background

The heuristic function $h(\cdot)$ used by heuristic search algorithms is in fact an estimate of the optimal solution cost. A considerable amount of effort has been devoted to creating admissible heuristics [13, 29, 80, 82] and inadmissible heuristics [17, 35, 76, 81]. As shown in the experimental results of this chapter, admissible heuristics usually provide inaccurate predictions of the optimal solution cost as they are biased to never overestimate the actual cost. In some cases, even inadmissible heuristics are biased towards admissibility [17, 76].

Regardless of admissibility, heuristics share a property: the heuristic evaluation must be fast enough to be computed for every node generated during the search,[1] while the solution cost predictor is run only on the start state. In fact, often heuristic functions sacrifice accuracy for speed. By contrast, the algorithms presented in this chapter aim at accurately predicting the optimal solution cost of a problem instance. While algorithms for predicting the optimal solution cost can be viewed as a heuristic, they differ from a heuristic conceptually in that: 1) they are not required to be fast enough to guide search algorithms; 2) they do not favor admissibility; and 3) they aim at making accurate predictions and thus our measure of effectiveness is prediction accuracy, in contrast to the solution quality and search time used to measure the effectiveness of heuristic functions.

5.1.3 Contributions

The content of this chapter is published in the Proceedings of the Symposium on Combinatorial Search (2011) [53] and (2012) [60], and in the Proceedings of the International Conference on Automated Planning and Scheduling (2012) [52]. In this chapter we make the following contributions.

- We present Solution Cost Predictor (`SCP`), a prediction method based on `CDP`, able to produce fairly accurate predictions of the optimal solution cost of state-space search problems.
- We present Bidirectional Stratified Sampling (`BiSS`), a bidirectional variation of `SS` that produces good predictions of the optimal solution cost of state-space search problems. `BiSS` has two advantages over `SCP`: (1) it entirely avoids the time-consuming preprocessing required by `SCP`; and (2) unlike `SCP`, `BiSS` is guaranteed to return the optimal solution cost in the limit as its sample size goes to infinity. On the other hand, `BiSS` has two disadvantages `SCP` does not have: (1) `BiSS` as presented in this chapter requires the ability to reason backwards about an operator so that the backward search is doable; (2) `BiSS` only produces predictions for domains with single goal states.

[1] In some settings it is more efficient to perform heuristic calculations lazily during node expansion [74].

- We show empirically that BiSS scales to state spaces much larger than can be solved optimally. In particular, we predict the average solution cost for the Sliding-Tile puzzles up to the 8x8 configuration, which has more than 10^{88} reachable states, and provide indirect evidence that BiSS's predictions for these huge state spaces are accurate.

- Finally, we show how to quickly learn strong heuristics from predictions produced by BiSS. We show that it is possible to reduce the time required for learning strong heuristic functions from days to minutes by using BiSS's predictions to label the training set.

5.1.4 Chapter Outline

The remainder of this chapter is organized as follows. In Section 5.2 we present SCP, a solution cost predictor based on CDP. In Section 5.3 we introduce BiSS, a bidirectional variant of SS for predicting the optimal solution cost. In Section 5.4 we present the C&C assumption used to develop SCP and BiSS. In Section 5.5 we show how BiSS can be used for generating data for learning strong heuristic functions. Finally, in Section 5.6 we draw the concluding remarks.

5.2 Solution Cost Predictor

We now describe the Solution Cost Predictor, SCP, a C&C method based on CDP for estimating the optimal solution cost of state-space search problems. We state the C&C assumption SCP makes for the type system it uses later on in the chapter, in Section 5.4.

5.2.1 Preliminaries

SCP requires the type system to have special types containing only goal nodes. We call this kind of type a *goal type*, and define it as follows.

Definition 19 (Goal Type)**.** *A type $t_g \in T$ is a goal type if for all nodes $s \in t_g$, s is a goal node.*

During sampling, whenever we reach a state s, we perform a goal test on s. If s is a goal, then its type becomes a goal type. There can be as many goal types as different goal nodes in a UST.

We say that a type t generates a type t' if there exist two nodes s and s' such that $s \in t$, $s' \in t'$, and s is the parent of s' in the UST. We use the term *type space* to denote the graph whose vertices are the types, and where every two types t and t' have a directed edge between them if at least one node of type t generates at least one node of type t'. The weight of an edge between types t and t' in the type space is given by the probability of a node of type t generating a node of type t'; note that these probabilities are the $\pi(\cdot|\cdot)$-values learned during CDP's sampling.

Like the type systems considered in Chapter 3, the type systems we use in this chapter are heuristic-preserving (see Definition 8 in Chapter 3). Moreover, the type systems we consider are based on admissible heuristics. Thus, we have that $h(t_g) = 0$ for any goal type t_g.

In this chapter we also use the type systems defined in Equations 3.4, 3.5 and 3.6 of Chapter 3, repeated here in Equations 5.1, 5.2 and 5.3 for the reader's convenience.

$$T_h(s) = (h(parent(s)), h(s)), \tag{5.1}$$

where $parent(s)$ returns the parent of s in the UST.

$$T_c(s) = (T_h(s), c(s,0), \ldots, c(s,H)), \tag{5.2}$$

where $c(s,k)$ is the number of children of s, considering parent pruning, whose h-value is k, and H is the maximum h-value observed in the sampling process.

$$T_{gc}(s) = (T_c(s), gc(s,0), \ldots, gc(s,H)), \tag{5.3}$$

where $gc(s,k)$ is the number of grandchildren of s, considering parent pruning, whose h-value is k.

5.2.2 The SCP Prediction Formula

Like `CDP`, `SCP` also samples the state space of a given problem with respect to a type system as a preprocessing step to find the values of $\pi(t|u)$ and β_u (see Section 3.2). After sampling, `SCP` predicts the optimal solution cost for a given start state s^* based on the values of $\pi(t|u)$ and β_u.

The basic building block of `SCP` is a formula for estimating the probability of at least one goal node existing at a level of the UST (a goal node n exists at level i of the UST rooted at s^* if there is a path of cost i from s^* to n). `SCP` estimates the probability of a goal node existing at a level of search by approximating the probability of a node of a goal type existing at a level of search. This probability is estimated by extending the `CDP` prediction formula, as explained next.

The probability that a node of a goal type exists at the i^{th} level of the search tree depends on (a) the probability that nodes exist at level $i-1$ that can potentially generate a goal node; and (b) the probability that at least one node at level $i-1$ indeed generates a goal node. In general, we define $p(i,t,s^*,d)$ as the approximated probability of finding at least one node of type t, in a search tree rooted at state s^*, at level i, with cost bound d. While computing $p(i,t,s^*,d)$ we assume the variables $\pi(\cdot|\cdot)$ to be independent.

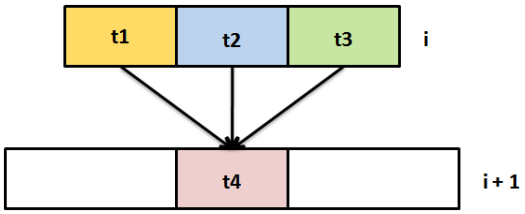

Figure 5.1: Types at level i are used to calculate the approximated probability of t_4 existing at level $i+1$.

Example 10. *We now illustrate how $p(i,t,s^*,d)$ is calculated with the example shown in Figure 5.1. Assume that only nodes of types t_1, t_2 and t_3 can generate a node of type t_4. In other words, for any type $t \notin \{t_1,t_2,t_3\}$, we have that $\pi(t_4|t) = 0$. A node of type t_4 exists at level $i+1$ iff at least one of the nodes of types t_1, t_2 or t_3 at the previous level generates one or more instances of the type t_4.*

We now describe exactly how $p(i,t,s^*,d)$ is calculated. Let $\phi(M,t,t')$ be the probability of M nodes of type t generating no nodes of type t', assuming $\beta_t = 1$. Given $\pi(t'|t)$, ϕ is computed as follows.

$$\phi(M,t,t') = (1-\pi(t'|t))^M$$

We define $p(i,t \rightarrow t',s^*,d)$ as the approximated probability of one or more nodes of type t existing at level i and generating at least one node of type t' at level $i+1$. $p(i,t \rightarrow t',s^*,d)$ is calculated from $p(i,t,s^*,d)$ and $\phi(N(i,t,s^*,d)\cdot\beta_t,t,t')$ as follows:

$$p(i,t \rightarrow t',s^*,d) = p(i,t,s^*,d)\big(1-\phi(N(i,t,s^*,d)\cdot\beta_t,t,t')\big) \tag{5.4}$$

Recall that $N(i,t,s^*,d)$ is the number of nodes of type t at level i of an IDA* search tree rooted at s^* with cost bound d. In Equation 5.4 the term we subtract from 1 is the probability that none of the $N(i,t,s^*,d)\cdot\beta_t$ nodes generated by nodes of type t are of type t'; this subtraction gives us the probability of at least one node of type t' being generated by nodes of type t.

Finally, $p(i,t,s^*,d)$ can be formally defined as follows.

$$p(i,t,s^*,d) = 1 - \prod_{u\in T}(1-p(i-1,u\rightarrow t,s^*,d)) \tag{5.5}$$

Here, the term $1-p(i-1,u\rightarrow t,s^*,d)$ gives us the approximated probability of no nodes of type t being generated at level i by nodes of type u. By multiplying this probability for every u in T we get the approximated probability of no nodes of type t being generated at level i. Finally, when

we subtract this resulting multiplication from 1 we get $p(i, t, s^*, d)$, the approximated probability of there existing at least one node of type t at level i.

Note that Equations 5.4 and 5.5 are recursive as $p(i, t \rightarrow t', s^*, d)$ depends on $p(i, t, s^*, d)$, which in turn depends on $p(i-1, u \rightarrow t, s^*, d)$. The base of the recursion is defined for $i = 1$ where we know exactly the probability of finding a node of a given type as this corresponds to the type of the children of the start state.

$$p(1, t, s^*, d) = \begin{cases} 1 & \text{if there exists a child } s \text{ of } s^* \text{ such that } T(s) = t\,, \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

We write $N(i, t)$, $p(i, t \rightarrow t')$, and $p(i, t)$ instead of $N(i, t, s^*, d)$, $p(i, t \rightarrow t', s^*, d)$, and $p(i, t, s^*, d)$ whenever s^* and d are clear from the context.

Next, we describe the SCP algorithm, which uses Equation 5.5 presented above for estimating the probability of an instance of type t existing at level i of a search tree bounded by cost d.

5.2.3 The SCP Algorithm

SCP predicts the optimal solution cost as follows:

1. First, set the cost bound d to the heuristic value of the start state.
2. For every level i, estimate the probability of finding a goal type at that level in a search tree bounded by cost d, using Equation 5.5.
3. Terminate returning d as the predicted optimal solution cost when a goal type exists at level i with probability higher than a threshold value c provided by the user. Otherwise, increase the cost bound d by one and go to step 2.

SCP searches iteratively in the type space, incrementing the cost bound by one in case a goal type is not found with probability higher than c. This process is similar to how IDA* searches in the original state space. Algorithm 12 and Algorithm 13 provide the pseudocode for SCP. The SCP algorithm starts in Algorithm 12. Initially, SCP uses the heuristic value of the start state s^* to initialize the cost bound d (line 1 of Algorithm 12). Initializing $N(1, t)$ and $p(1, t)$ (line 2) is done according to the children of the start state as shown in Equations 3.2 and 5.6 for $N(1, t)$ and $p(1, t)$, respectively. $N(1, t)$ is the number of nodes of type t among the children of s^*. $p(1, t)$ is one if there is an instance of type t among the children of s^*, or zero otherwise.

For every level i (ranging from 2 to d), SCP calculates the values of $N(i, t)$ and $p(i, t)$ of every type t by calling Algorithm 13 (line 6 of Algorithm 12). Algorithm 13 computes the values of

Algorithm 12 SCP

Input: start state s^*, threshold value c, type system T
Output: predicted optimal solution cost

```
1:  d ← h(s*)
2:  initialize N(1,t) and p(1,t)
3:  loop
4:    for i = 2 to d do
5:      for all t ∈ T do
6:        p(i,t) ← compute-probability(i,t,d) // see Algorithm 13
7:        if t is a goal type AND p(i,t) ≥ c then
8:          return d
9:        end if
10:     end for
11:   end for
12:   d ← d + 1
13: end loop
```

Algorithm 13 compute-probability

Input: level i, type $t \in T$, cost bound d
Output: the estimated probability of existing a node of type t at level i of the search tree

```
1:  if h(t) + i > d then
2:    p(i,t) = 0, N(i,t) = 0
3:  end if
4:  n ← 0
5:  p ← 1
6:  for all u ∈ T do
7:    n ← n + N(i−1,u) · π(t|u) · β_u
8:    p(i−1, u→t) ← p(i−1,u)(1 − φ(N(i−1,u) · β_t, π(u|t), 0))
9:    p ← p · (1 − p(i−1, u→t))
10: end for
11: p(i,t) = 1 − p
12: N(i,t) = n
13: return p(i,t)
```

$N(i,t)$ as shown in line 7 and $p(i,t)$ as shown in line 9. Algorithm 13 returns $p(i,t)$. Every node of type t at level i for a given cost bound d will be pruned by the search if $i + h(t) > d$. Thus, in such cases we set $N(i,t)$ and $p(i,t)$ to zero (line 2 in Algorithm 13).

After every call to Algorithm 13 for a type t, if t is a goal type and $p(i,t)$ is above the user-defined threshold parameter c, the prediction halts and d is returned (line **??**) as SCP's optimal solution cost prediction. The reason SCP returns d and not i is explained in the next section.

5.2.4 The Cost Bound Equality Property of SCP

Next, we show that if a heuristic-preserving type system is based on a consistent heuristic, the value of i when SCP reaches line 8 of Algorithm 12 will always be equal to the current cost bound d — we call this property the *cost bound equality property*.

Definition 20. *Let $u \in T$ and $i, q \geq 1$. We say that type $t \in T$ at level q cannot modify the value of $p(i,u)$ if the computation of $p(i,u)$ is independent of the value of $N(q,t)$, i.e., changes in $N(q,t)$ do not affect the value of $p(i,u)$.*

Let T be a heuristic-preserving type system built from a consistent heuristic $h(\cdot)$. Further, let $f(q,t) = q + h(t)$ where q is the level at which type $t \in T$ is observed in the search tree.

Lemma 4. *Let $t_g \in T$ be a goal type. Let $i, q \geq 1$, $i \geq q$. If $f(q,u) > i$ for type $u \in T$ at level q of the search tree, then u cannot modify the value of $p(i,t_g)$.*

Proof. Since T is a heuristic-preserving type system built from a consistent heuristic $h(\cdot)$, each type $t \in T$ can only generate types t' (i.e., $\pi(t'|t) > 0$) for which $h(t')$ is one of the values $h(t)$, $h(t)+1$, and $h(t) - 1$. In particular,

$$\pi(t'|t) > 0 \text{ implies } h(t') \geq h(t) - 1\,. \tag{5.7}$$

Let $f(q,u) > i$. Suppose u can modify the value of $p(i,t_g)$. Thus, type u can generate a goal type at level i. In particular, there must exist a type sequence of length $i - q + 1$, say $(t_{i-q}, t_{i-q-1}, t_{i-q-2}, \cdots, t_1, t_g)$, such that $t_{i-q} = u$ and $\pi(t_{i-q-1}|t_{i-q}) > 0, \cdots, \pi(t_g|t_1) > 0$. Since $h(t_g) = 0$, Equation 5.7 yields

$$h(u) = h(t_{i-q}) \leq h(t_g) + i - q = i - q\,.$$

The latter in turn implies $f(q,u) = q + h(u) \leq q + i - q = i$ in contradiction to $f(q,u) > i$. Hence u cannot modify the value of $p(i,t_g)$. □

Theorem 4. *If* `SCP` *uses a heuristic-preserving type system T built from a consistent heuristic h, then, for any value of c, the value of i in line 8 of Algorithm 12 is always equal to the current cost bound d.*

Proof. Let us first consider the case of the first iteration of the algorithm, when $d = h(s^*)$. By analogy with the proof of Lemma 4, when replacing q by 0, there exists a sequence of types $(t_i, t_{i-1}, \cdots, t_1, t_g)$ such that $\pi(t_{i-1}|t_i) > 0, \cdots, \pi(t_g|t_1) > 0$. Since h is consistent, Equation 5.7 yields $h(s) \geq h(parent(s)) - 1$ for every node s. Since $h(t_g) = 0$, we obtain $i \geq h(s^*) = d$. Line 4 of Algorithm 12 implies $i \leq d = h(s^*)$, and thus $i = d$.

Next, consider the case $d > h(s^*)$. During an iteration with $d = U$, `SCP` will expand all the types that were expanded during the iteration with $d = U - 1$, and also the types u with $f(u) = d$ that were not expanded in the previous iteration. Lemma 4 states that no type t with $f(q,t) = U$

can modify the probability $p(i, t_g)$ for goal type t_g at a level $i < U$. Therefore, for $i = U - 1$, if SCP does not find a goal type t_g with $p(i, t_g) \geq c$ in the iteration with $d = U - 1$, then SCP will not find a goal type t_g with $p(i, t_g) \geq c$ in the iteration with $d = U$ either. So, for the test in line 7 of Algorithm 12 to be successful, i has to equal d.

□

Type Systems Based on Inconsistent Heuristics

If the type system is based on an inconsistent heuristic, Theorem 4 does not necessarily hold. That is, type $u \in T$ at level q of the search tree with $f(q, u) > i$ can potentially modify the value of $p(i, t_g)$ for goal type t_g. Thus, when using a type system built from an inconsistent heuristic, if returning i instead of d, Algorithm 12 could return a value smaller than the current cost bound d. In fact, SCP could even return a value of i that is lower than the heuristic value of the start state.

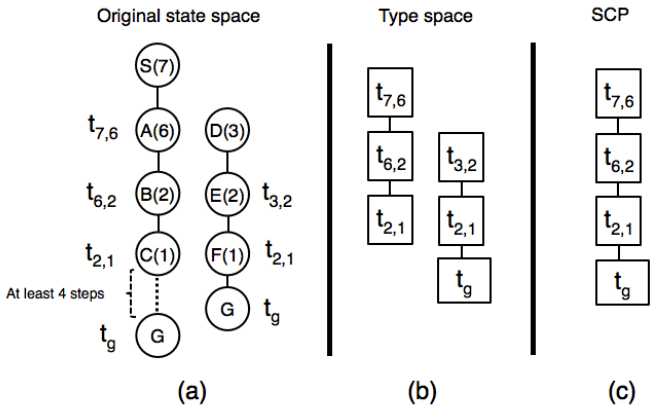

Figure 5.2: States and edges in the original state space, their corresponding types in the type system space, and a possible "shortcut" path in the type system space.

Example 11. *Consider the example shown in Figure 5.2. Here we use the T_h type system. We represent a type of a node n as $t_{x,y}$, where x is the heuristic value of the parent of n, and y is the heuristic value of n. Figure 5.2(a) shows states and edges of two paths to the goal in the original state space. The value in brackets in each circle is the heuristic value of the corresponding state – note the heuristic inconsistency between nodes A and B, when the heuristic value changes by more than one (edge cost). Figure 5.2(b) shows the corresponding types for the states and edges shown in Figure 5.2(a). Now, assume that* SCP *is run on state A from Figure 5.2(a). State A has a heuristic*

value of 6, and thus the optimal solution cost is at least 6. However, SCP *might return 3, since as shown in Figure 5.2(c) there is a path of cost 3 in the type system space from type $t_{7,6}$ – during sampling,* SCP *has seen a node of type $t_{7,6}$ that generates a node of type $t_{6,2}$, which generates a node of type $t_{2,1}$, and in a different part of the state space* SCP *has sampled a node of type $t_{2,1}$ that generates a goal node.*

We have observed in our experiments that returning d instead of i in line 8 of Algorithm 12 yields slightly more accurate predictions when the type system is built from an inconsistent heuristic.

5.2.5 Reducing the Size of Type Systems Built from Inconsistent Heuristics

We observed in our experiments that the number of types in a type space can be too large to be practical when using an inconsistent heuristic. This is because with an inconsistent heuristic, the number of different combinations of heuristic values in subtrees considered in type systems such as T_c or T_{gc} is much larger than what is observed when a consistent heuristic is used. Recall that when the heuristic is consistent the heuristic value of nodes connected by an edge in the search tree differ by at most one, when the heuristic is inconsistent, heuristic values of neighbor nodes might differ by any value, which increases the size of the T_c and T_{gc} type systems.

When using a type system that is very large, it may be difficult to properly sample all types. For example, a T_c type system for the 15-pancake puzzle was built based on the inconsistent heuristic that takes the maximum of the regular and the dual lookup of a PDB [20, 85]. Even after sampling 100 million random states and using biased sampling, there were types that were not sampled.

We use a process similar to bidirectional pathmax (BPMX) [20] to shrink the size of type systems based on inconsistent heuristics. This is done as follows. For every sampled state s, the heuristic values of its children are increased by propagating the heuristic values from other nodes while computing the type of s.

Example 12. *Figure 5.3 illustrates this process. The children of the highlighted state in (a) with heuristic values of 2 and 1 can be raised to 4 as there is a child with heuristic value of 6. This is computed by subtracting the cost of the shortest path between the two siblings from the highest heuristic value among the children ($6 - 2 = 4$). The same process is applied to the highlighted state in (b): the states with heuristic value of 1 can be raised to 4. Therefore, two states that would belong to different types in T_c after applying the heuristic propagation now belong to the same type. Consequently, this process reduces the size of the type system.*

Although not shown in the example, we also use the heuristic value of the sampled node (the

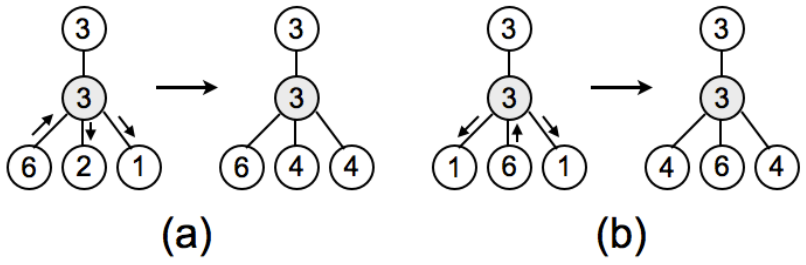

Figure 5.3: Example of how we readjust the heuristic values of the children of the highlighted state while computing that state's type. This process reduces the size of a type system. The highlighted states in (a) and (b) end up being of the same type after propagating the heuristic values among the children. The heuristic propagation could be extended to the highlighted states and their parents, but by propagating only among the children we get the desired reduction of the type system size.

nodes highlighted in Figure 5.3) and of the parent of the sampled node to adjust the heuristic value of the children of the sampled node. In this case we propagate the heuristic value of the parent or of the grandparent downward to the children. Note that we could use a similar strategy to increase the heuristic value of the highlighted nodes in Figure 5.3 and further shrink the size of the type system. However, we get type systems of good sizes by adjusting only the value of the children of the highlighted nodes. We use this technique only with the T_c type system.

5.2.6 SCP Runtime Behavior

The runtime behavior of SCP is exactly the same as that of CDP and it depends on the size of the type system used. That is, in the worst-case SCP will expand $|T|$ types at every level of prediction, and it will generate another $|T|$ types on the next level of prediction.

Often the number of nodes expanded by a search algorithm for finding the optimal solution path grows exponentially with the solution cost. By contrast, assuming a predicted optimal solution cost of c^*, SCP will expand only $c^* \times |T|$ types to produce a prediction.

5.2.7 Accuracy Experiments

We test SCP with different type systems based on different heuristic functions. We compare the accuracy of SCP with the accuracy of the heuristic functions used to define the type systems used by SCP. Finally, in one of the tested domains we compare SCP to a Bootstrap heuristic [35], a machine-learned estimator found in the literature. The system for generating Bootstrap heuristics will be described later in the chapter, in Section 5.5.1.

Experimental Setup

Problem domains. We run experiments on three domains: the 15-puzzle, 15-pancake puzzle, and Towers of Hanoi (12-disk, 4-peg). For a description of these domains see Section 2.4.

Parameter setting. In all the experiments in this section we set the user-defined threshold parameter c to 0.99. In Section 5.2.8 we empirically study the influence of different c-values on the prediction accuracy. Analogously, in Section 5.2.9 we show empirically the effect of different r-values and in Section 5.2.10 the effect of ϵ-truncation in SCP's prediction accuracy.

The depth of the prediction lookahead (i.e, the r-value shown in Section 3.2) was set to 1 ($r = 1$) in most of the experiments in this section; on the 15-puzzle we also experiment with $r = 25$. When experimenting with $r > 1$ we also make a prediction lookahead for the heuristic estimates, i.e., use the lowest heuristic value among the nodes at distance r from the start state. In Section 5.2.9 we show experiments with different values of r and we analyze how this parameter affects the runtime and accuracy of SCP's predictions.

Error measure. We use the relative unsigned error to measure the prediction accuracy. The relative unsigned error of an instance with optimal cost C and predicted cost P is $\frac{|P-C|}{C}$, i.e., the absolute difference between the predicted and the optimal cost, normalized by the optimal cost. A perfect score according to this measure is 0.00. Note that the relative unsigned error represents the percentage by which a predictor overestimates (or underestimates) the actual optimal solution cost. For instance, an error of 0.1 for a single prediction represents a prediction that overestimates (or underestimates) the optimal solution cost by 10%. In our plot of results we present the relative unsigned error in terms of percentage.

Results are presented in plots such as Figure 5.4. The x-axis groups start states by their optimal solution costs and the y-axis represents the relative unsigned error of the predictions. The error bars represent the 95% confidence interval based on the assumption that the prediction error for a given optimal solution cost follows a normal distribution.

Type systems. Ideally we would employ the T_{gc} type system in all our experiments as it strictly contains more information than T_c and T_h. However, depending on the domain and on the heuristic used, a T_{gc} type system can have a prohibitively large number of types, which prevents sampling of all types in a reasonable amount of time. Therefore, the choice of the type system used in each experiment is closely related to (1) the heuristic function used and also to (2) the problem domain. For instance, employing T_{gc} with an inconsistent heuristic for the 15-puzzle the size of the type

system could become too large for sampling to be done in a reasonable amount of time. Usually inconsistent heuristics produce larger T_c and T_{gc} type systems as there is a larger variety of heuristic values among the children and grandchildren of a node. The branching factor of a domain also influences the size of a type system. For instance, the 15-pancake puzzle, which has a much larger branching factor than the 15-puzzle, will likely have larger T_c and T_{gc} type systems due to the larger number of heuristic values considered. The type system used in each experiment is specified below.

The Sliding-Tile Puzzle

For the 15-puzzle we solved optimally and performed the `SCP` prediction for 1,000 random solvable states to measure prediction accuracy. To define $\pi(t|u)$ and β_t, one billion random states were sampled and, in addition, we use the biased sampling process described in Chapter 3, in which we sample the child of a sampled state if the type of that child had not yet been sampled.

Type Systems. Predictions with three different type systems were performed. We use the following type systems/heuristic functions.

- Manhattan Distance (MD) – This is a popular and easy-to-implement heuristic function for the sliding-tile puzzles. It sums the Manhattan Distance of the individual tiles to their goal position (excluding the blank tile). This heuristic is consistent and we use the T_{gc} type system with it. The predictions using this type system will be referred to by SCP-MD.

- 7-8 Additive PDBs (APDB) – The 7-8 additive pattern database is an effective heuristic for the 15 puzzle [19, 48]. It consists of the sum of two disjoint pattern databases, one based on tiles 1-7 and the other based on tiles 8-15. The APDB is inconsistent[2] and the type system we use with it is the T_c type system. Predictions using this type system will be referred to by SCP-APDB.

- Double Inconsistent PDB (DPDB) – This is the same heuristic used by [84]. Two PDBs were created, one based on tiles 1-7 and another one based on tiles 9-15. For states with the blank in a location with an even number according to the goal state (see Figure 2.4) the first PDB is consulted, the other PDB is consulted otherwise. As the blank always changes from even to odd or odd to even from one state to a neighbor, the PDB that is consulted alternates from parent to child. This generates inconsistency. The type system we use is T_c. Predictions using this type system will be referred to by SCP-DPDB.

[2] See footnote 1 in [19]. We built the same PDBs.

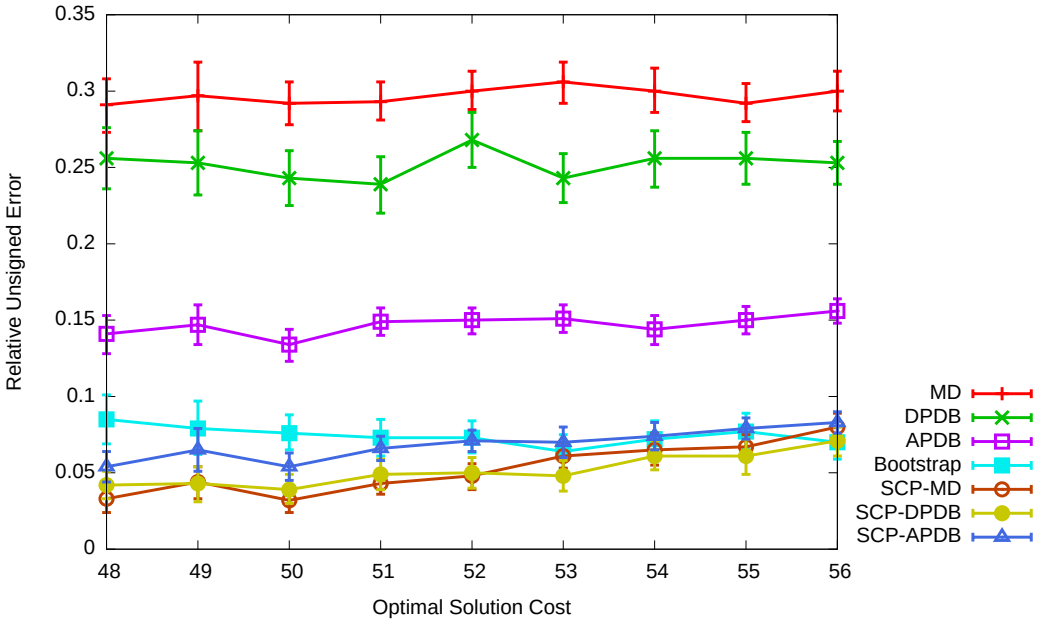

Figure 5.4: 15-puzzle.

In addition to comparing SCP predictions with the heuristics used to build the type systems, we compare SCP predictions with the inadmissible Bootstrap heuristic [35]. The Bootstrap algorithm iteratively improves an initial heuristic function by solving a set of successively more difficult training problems in each iteration. This process was shown to create effective heuristics for a variety of domains [35, 81].

Figure 5.4 presents the results. First, the results show the well-known fact that the Manhattan Distance (MD) heuristic is less accurate than the Additive PDB (APDB) heuristic. Second, SCP is able to produce accurate predictions, with errors less than 10% of the optimal solution cost. Note that by comparison, all the tested admissible heuristics, MD, DPDB and APDB, tend to make inaccurate estimates, having an error of approximately 30%, 25% and 15%, respectively. This corresponds to the intuitive observation mentioned earlier in the chapter that admissible heuristics tend to make poor estimates as they are biased to never overestimate the optimal cost. It is clear in this experiment that SCP using a type system based on a heuristic function produces more accurate predictions than the heuristic itself. We also observe in Figure 5.4 that SCP's prediction error is about half of Bootstrap's for problems with optimal solution cost of 51 or less.

According to the results shown in Figure 5.4 the accuracy of the heuristic used to build a type system does not seem to affect SCP's prediction accuracy much. SCP-MD, SCP-DPDB, and SCP-APDB are of similar accuracy, even though APDB is more accurate than MD and DPDB. While

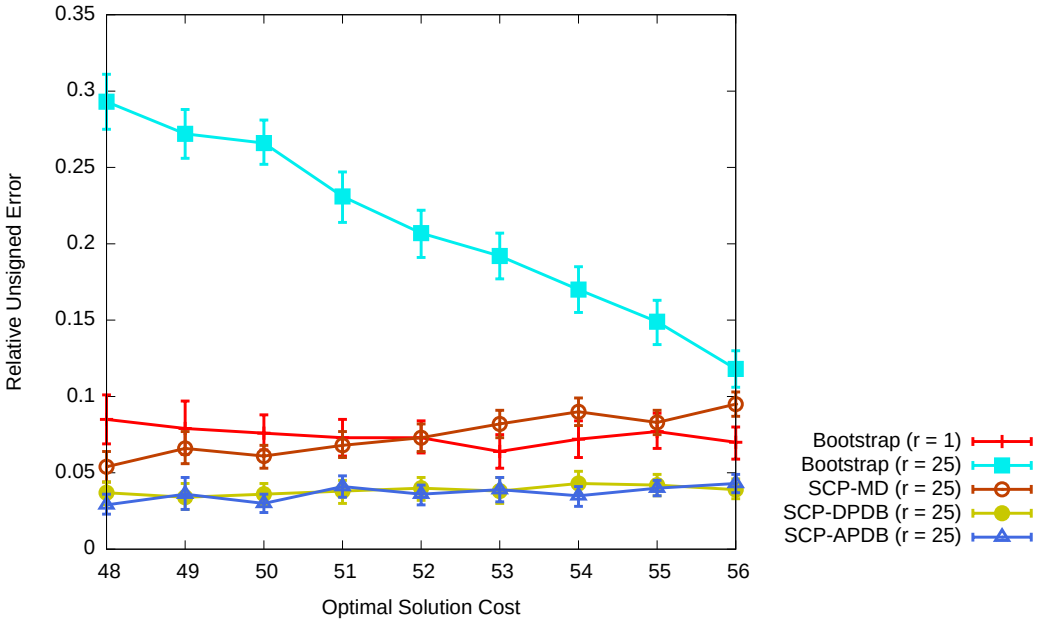

Figure 5.5: 15-puzzle (r=25).

there is no definite explanation for this phenomenon, the results indicate that a type system does not necessarily have to be built from an accurate heuristic to allow SCP to make accurate predictions.

We also compare SCP's prediction accuracy with the accuracy of Bootstrap when using the prediction lookahead. Figure 5.5 presents the results. For convenience we repeat the results of Bootstrap with no prediction lookahead (r=1) shown in Figure 5.4. It is interesting to see that the prediction lookahead substantially reduces Bootstrap's prediction accuracy. SCP, on the other hand, tends to benefit from the prediction lookahead as it is seeded with the exact distribution of types at distance r from the start state. SCP-DPDB and SCP-APDB are more accurate than Bootstrap using either $r = 1$ or $r = 25$. In Section 5.2.9 we study the tradeoff between prediction accuracy and prediction runtime of the prediction lookahead.

The Pancake Puzzle

For the 15-pancake puzzle we use 1,000 random solvable states to measure prediction accuracy. To define $\pi(t|u)$ and β_t, 100 million random states were sampled and, in addition, we use the biased sampling process described in Chapter 3. In this experiment, in order to sample the goal types and states in the neighborhood of the goal types, 10,000 out of the 100 million states sampled were generated with random walks of length 10 from the goal state.

Type Systems. As with the 15-puzzle, we use three different heuristic functions to define the

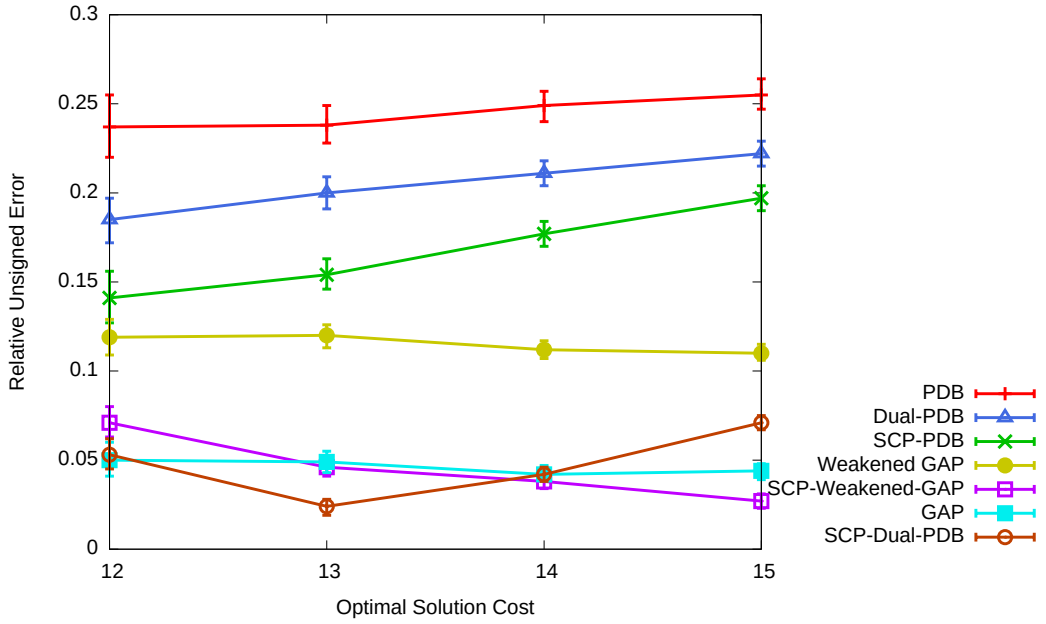

Figure 5.6: 15-pancake puzzle.

type systems used in this experiment. The type systems used were based on the T_c type system.

- PDB – We created a PDB based on the location of the eight leftmost pancakes. The resulting PDB is consistent. The predictions using this type system will be referred to by SCP-PDB.

- Dual-PDB – For this type system we use a heuristic that makes a regular and a dual lookup on a PDB based on the leftmost eight pancakes and returns the maximum of them [20, 85]. The resulting heuristic is inconsistent. For the Dual-PDB there were types that were not sampled even after sampling 100 million random states. Thus, we used the process described in Section 5.2.5 to shrink the size of the type system so that all types could be sampled at least once with 100 million random states.

- Weakened GAP – GAP is a very accurate hand-crafted consistent heuristic for the pancake puzzle [28].[3] As the GAP heuristic already provides accurate cost estimates, it is not interesting to build a type system for `SCP` with such an accurate heuristic. Thus, we use a weakened version of it to build a type system. In our weakened version of GAP we use the number of adjacent pancakes whose number differs by more than one except for the rightmost pancake. We show the accuracy of GAP and its weakened version.

[3] See also http://tomas.rokicki.com/pancake/

Figure 5.6 shows the results for the 15-pancake puzzle. As seen in the previous experiment, using SCP with a type system based on a given heuristic produces substantially more accurate predictions than using the heuristic itself as a predictor. This can be seen by the difference between PDB and SCP-PDB, Dual-PDB and SCP-Dual-PDB, and Weakened GAP and SCP-Weakened-GAP. SCP using a type system built from Dual-PDB is accurate and competitive with GAP.

The results in Figure 5.6 also show that the PDB based on the eight leftmost pancakes is the least accurate estimator, giving estimates with errors of about 25%. The type system built with the consistent PDB clearly fails to offer a good partition of the state space. As a consequence, SCP-PDB is the least accurate of the SCP predictions. SCP-Dual-PDB and SCP-Weakened-GAP on the other hand produce very accurate predictions – the errors indicate that the predictions are less than one move longer than the average optimal number of moves. Interestingly, SCP with Dual-PDB as well as SCP with Weakened GAP produce predictions of similar accuracy even though the estimates of Dual-PDB are much less accurate than the estimates of Weakened GAP.

Towers of Hanoi

For the 12-disk 4-peg Towers of Hanoi we used 5,000 random solvable states to measure prediction accuracy. To define $\pi(t|u)$ and β_t, one million random states were sampled and, in addition, we used the biased sampling process described in Chapter 3. Random instances for sampling were generated with random walks from the goal with a random length between 100 and 10,000 steps, while the random instances used to measure the accuracy were generated with random walks of fixed length of 500 steps. The type systems used in this experiment were all based on T_{gc}.

Type Systems. We use two different type systems based on different PDB heuristics. In our implementation a state of the 12-disk 4-peg Towers of Hanoi is represented with 48 binary variables, one variable for each possible peg a disk might be on (12 for each of the 4 pegs). The variable v_{dp} is set to one in a state if disk d is on peg p, and to zero otherwise. The way we build simplified versions of the puzzle to create PDB heuristics is by *projecting out* some of these variables. When we project out a variable v_{dp} of a state we cannot distinguish whether v_{dp} carries a value of zero or one. The more variables we project out the more "simplified" will be the resulting puzzle. The choice of the PDBs we use for this domain in our experiments is arbitrary. That is, we arbitrarily selected two sets of variables to be projected out so that we would have two different PDBs. The type system we use is T_{gc}.

- PDB1 – We created a PDB built by projecting out 20 of the 48 state variables. Namely, we

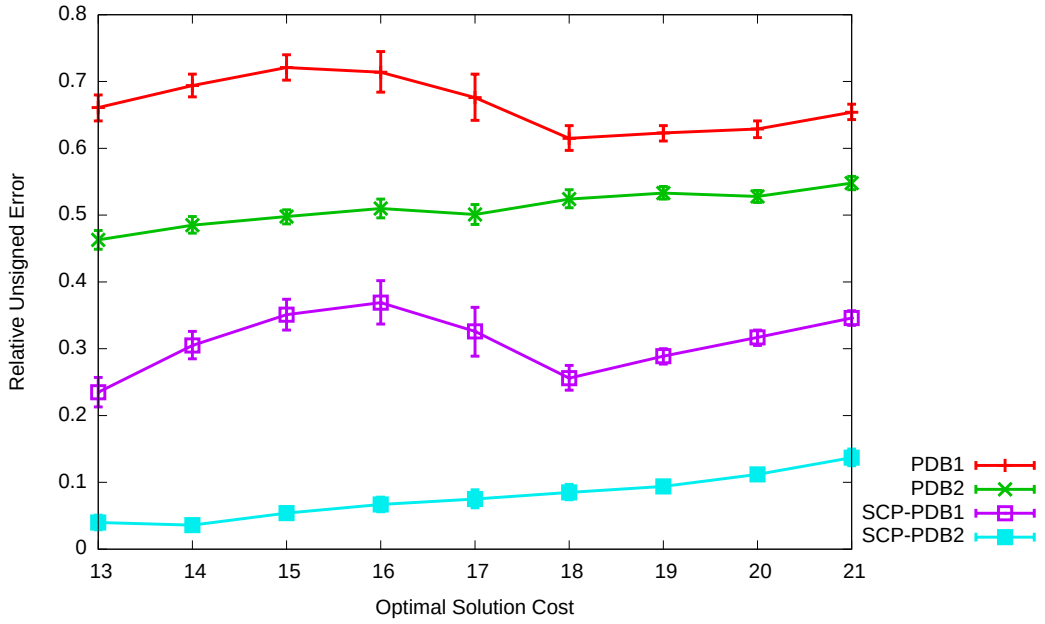

Figure 5.7: 12-disk 4-peg Towers of Hanoi.

project out the even disks from pegs 1 and 2, disks 2, 8, and 10 from peg 3, and disks 2, 4, 6, 8, and 10 from peg 4. The resulting PDB is consistent. Predictions using this type system will be referred to by SCP-PDB1.

- PDB2 – We created another PDB by projecting out 19 of the 48 state variables. Namely, we projected out from pegs 1 and 2 the same disks we did for PDB1; from peg 3 we projected out the same disks as in PDB1 with the exception of projecting out disk 12 instead of disk 8; from peg 4 we also projected out the same disks as in PDB1, with the exception of disk 10, which was not projected out.

Figure 5.7 shows the results for the Towers of Hanoi. The least accurate estimations of the solution cost are given by PDB1, followed by PDB2. A major improvement is observed with the SCP predictions. For instance, the error drops from around 50% to about 5% when comparing PDB2 with SCP-PDB2.

Discussion

There are several trends that are observed in all three domains. First, for every domain and every heuristic, using SCP with a given heuristic always produces predictions that are substantially more accurate than the heuristic used to build the type system. This shows the benefit of using SCP to predict the optimal cost over using heuristics. Furthermore, we have observed that SCP is able to

make accurate predictions of the optimal solution cost even when the type system being employed is built from an "inaccurate" heuristic function. We have also observed in some cases that SCP might produce inaccurate predictions, such as SCP-PDB1 in Figure 5.7, where the error is about 30% of the optimal solution cost. However, even in that case SCP is substantially more accurate than the heuristic function used to build the type system, which in that case produces estimates with errors of about 70% of the optimal solution cost.

5.2.8 Empirical Study of the Threshold Parameter

In Section 5.2.7 SCP was tested for a fixed set of parameters. Namely, we used a threshold parameter c of 0.99. In this section we discuss the effect of this c-value on the accuracy of SCP. We experimentally show the algorithm's accuracy with a c-value of 0.80, 0.85, 0.90, 0.95, and 0.99.

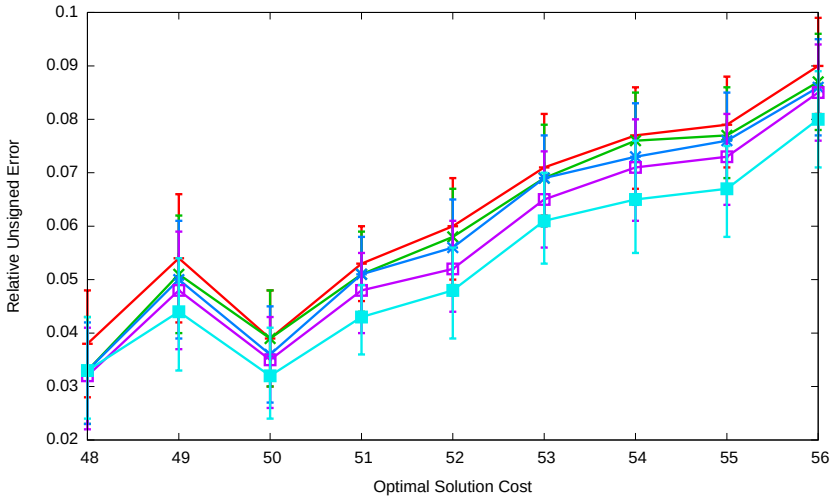

Figure 5.8: Robustness to the parameter c for the 15-puzzle. SCP-MD.

Figures 5.8, 5.9, and 5.10 present SCP's prediction errors for different c-values for the 15-puzzle. Different curves correspond to different values of c. As can be observed, the effect of different c values is minor, and the curves are clustered together; there is no substantial difference in prediction accuracy for the different c-values used. The accuracy of SCP is relatively robust to the choice of c. Similar results were observed on the 15-pancake-puzzle and on the 12-disk 4-peg Towers of Hanoi.

5.2.9 Empirical Study of the Prediction Lookahead

CDP is known to improve its accuracy by increasing the r-value [84]. Here we observe a similar effect for SCP. To focus only on the effect of the r-values, we did not use ϵ-truncation in the following experiment.

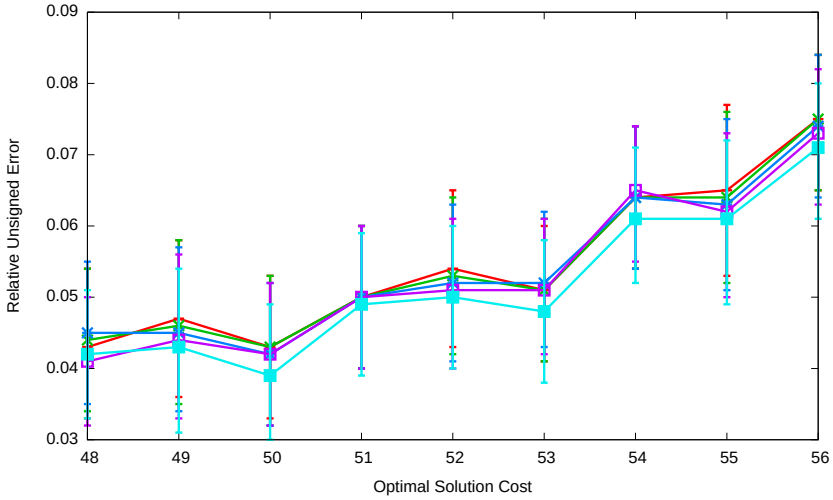

Figure 5.9: Robustness to the parameter c for the 15-puzzle. SCP-DPDB.

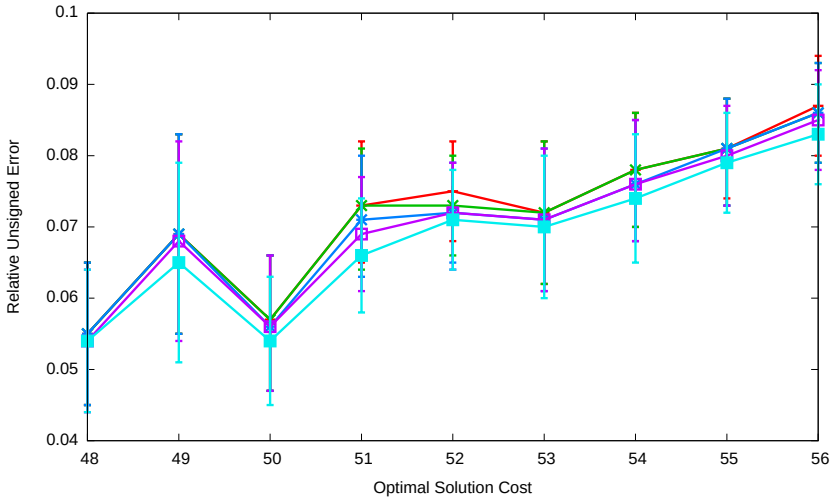

Figure 5.10: Robustness to the parameter c for the 15-puzzle. SCP-APDB.

Table 5.1 shows prediction results of SCP using the T_{gc} type system with MD as the heuristic function for five different values of r. Each row in Table 5.1 presents results for start states with a given optimal solution cost, shown in the first column of the table. Besides the relative unsigned error (Error), we also show the maximum (t_{max}) and the minimum (t_{min}) of the prediction runtime in seconds of start states in a set of 1,000 random problem instances.

The first observation in Table 5.1 is that the error decreases as the value of r increases. Larger values of r result in SCP being seeded with types of nodes farther from the start state. Like CDP, SCP emulates a search tree in the type space where states in the actual state space are compressed into types. While this compression into types makes the process of predicting the optimal solution cost

	$r=1$			$r=9$			$r=12$			$r=15$			$r=21$		
Cost	Error	t_{min}	t_{max}	Error	t_{min}	t_{max}	Error	t_{min}	t_{max}	Error	t_{min}	t_{max}	Error	t_{min}	t_{max}
48	0.071	0.51	0.93	0.074	0.25	0.52	0.071	0.20	0.49	0.068	0.20	0.56	0.061	0.21	1.16
49	0.092	0.55	0.99	0.092	0.19	0.73	0.088	0.18	0.55	0.083	0.18	0.53	0.076	0.16	1.25
50	0.091	0.46	0.99	0.083	0.26	0.61	0.078	0.22	0.55	0.076	0.20	0.58	0.068	0.25	1.76
51	0.099	0.54	1.04	0.097	0.26	0.65	0.090	0.18	0.56	0.087	0.21	0.58	0.078	0.26	1.59
52	0.106	0.52	1.02	0.101	0.27	0.70	0.097	0.18	0.57	0.093	0.15	0.60	0.084	0.28	1.88
53	0.118	0.48	0.98	0.111	0.29	0.72	0.107	0.23	0.55	0.102	0.14	0.59	0.091	0.15	2.29
54	0.122	0.51	1.01	0.116	0.30	0.69	0.112	0.25	0.61	0.111	0.24	0.69	0.097	0.22	2.25
55	0.121	0.48	1.10	0.111	0.32	0.73	0.108	0.28	0.68	0.103	0.23	0.69	0.093	0.30	2.64
56	0.130	0.48	1.11	0.125	0.24	0.77	0.123	0.27	0.63	0.117	0.26	0.70	0.106	0.30	2.04

Table 5.1: Effect of different r-values on SCP's prediction accuracy and runtime for the 15-puzzle. We show the relative unsigned error and the minimum and maximum runtimes in seconds for a set of 1,000 random start states.

faster, it makes the prediction subject to error. For instance, SCP's predicted distribution of types at distance x from the start state (i.e., the values of $N(x,t)$ and $p(x,t)$) is only an approximation of the actual distribution. When using $r = x$, SCP uses the actual distribution of types at that level. Therefore, larger values of r reduce the error of the type distribution at a given distance from the start state, which, as suggested by Table 5.1, also reduce the prediction error.

On the other hand, larger values of r increase the prediction runtime. In the extreme case, if one makes $r = c^*$, where c^* is the optimal solution cost of a given problem instance, the SCP prediction is guaranteed to correctly predict the optimal solution cost as a goal node will be expanded while collecting the nodes at distance r from the start state. However, all nodes n in the UST for which $f(n) \leq c^*$ will have to be expanded. It can be observed that the maximum runtime when $r = 21$ is substantially higher than the other maximum runtimes.

However, contrary to common intuition, SCP is not fastest when $r = 1$. SCP makes quicker predictions for the r-values of 9, 12, and 15. The type system's compression does not payoff for the first levels of search – the number of types is roughly the same as the number of nodes in the search tree and initially it is cheaper to expand the nodes in the actual state space rather than the types in the type space.

5.2.10 Empirical Study of ϵ-truncation with SCP

Next, we check the effect of ϵ-truncation on the accuracy of SCP predictions. We isolate the effect of ϵ-truncation, in the subsequent experiment by setting the r-value to one. Table 5.2 shows the relative unsigned error of SCP with and without ϵ-truncation, on 1,000 random 15-puzzle instances. Results are shown for each of the type systems described for the 15-puzzle in Section 5.2.7. For convenience we drop the prefix "SCP" from the name of the type systems, and, in addition, we add an ϵ to the name of the type system if SCP uses ϵ-truncation. We highlight an ϵ-truncation entry in

Cost	MD	ϵ-MD	DPDB	ϵ-DPDB	APDB	ϵ-APDB
48	0.071	**0.033**	0.051	**0.042**	0.083	**0.054**
49	0.092	**0.044**	0.054	**0.043**	0.098	**0.065**
50	0.091	**0.032**	0.055	**0.039**	0.086	**0.054**
51	0.099	**0.043**	0.062	**0.049**	0.102	**0.066**
52	0.106	**0.048**	0.072	**0.050**	0.102	**0.071**
53	0.118	**0.061**	0.069	**0.048**	0.102	**0.070**
54	0.122	**0.065**	0.081	**0.061**	0.105	**0.074**
55	0.121	**0.067**	0.085	**0.061**	0.111	**0.079**
56	0.130	**0.080**	0.091	**0.071**	0.118	**0.083**

Table 5.2: Effect of ϵ-truncation on SCP's prediction accuracy for the 15-puzzle.

the table if it is at least as accurate as its counterpart.

As can be observed in Table 5.2, ϵ-truncation substantially improves the prediction accuracy of SCP in the 15-puzzle. ϵ-truncation was designed to carefully ignore rare and harmful events observed during the CDP sampling (see Section 3.2.3). For instance, if a type t rarely generates a type t', then CDP improves its prediction accuracy by completely ignoring this rare event. The rare events also seem to be harmful to SCP as carefully ignoring them improves SCP's prediction accuracy. We conjecture that rare events create "shortcuts" to the goal type in the type space, making SCP find the goal type prematurely.

In some domains the rare events are not observed, and in such domains ϵ-truncation does not change CDP's prediction accuracy. We observed the same phenomenon in our experiments with SCP. ϵ-truncation does not change SCP's prediction accuracy for the 15-pancake puzzle and for Towers of Hanoi when using the heuristic functions described above. Moreover, like with CDP, ϵ-truncation has a larger impact on the SCP predictions for lower values of r.

5.3 Bidirectional Stratified Sampling

In this section we present another C&C algorithm, a bidirectional adaptation of SS, which we call Bidirectional Stratified Sampling (BiSS) for predicting the optimal solution cost of a problem instance. We state the C&C assumption BiSS makes for the type system it uses in Section 5.4

BiSS has two advantages over SCP: (1) it entirely avoids the preprocessing required by SCP; and (2) unlike SCP, it is guaranteed to return the optimal solution cost in the limit as its sample size goes to infinity. These two advantages allow BiSS to produce accurate predictions even for very large state spaces, as our empirical results demonstrate.

On the other hand, as we shall see, BiSS has two disadvantages over SCP: (1) it requires there to be a single goal state and is therefore not suitable for domains in which a set of goal conditions is

given instead of an actual goal state; and (2) in our implementation of BiSS we assume the ability to reason backwards about an operator so that the backward search is doable.

5.3.1 Using SS for Optimal Solution Cost Prediction

Before explaining BiSS we mention another possible approach for using SS for predicting the optimal solution cost: one could run SS and have it stop when a goal state is generated. The cost of the path found to the goal state is an upper bound on the optimal solution cost, so the minimum of these upper bounds over a set of runs gives an estimate of the optimal solution cost. However, in an experiment we ran on the 24-puzzle using the same heuristic function and the same number of probes we use in our experiments below, the predictions produced by this approach were less accurate than the relatively inaccurate Manhattan Distance heuristic. We are going to use such approach with a few enhancements in the next chapter, when we discuss algorithms for finding suboptimal paths from the start state to a goal state.

5.3.2 The BiSS Prediction Algorithm

BiSS is a bidirectional variant of SS for predicting optimal solution costs. It interleaves the execution of two copies of SS, one proceeding forwards from the start state, the other proceeding backwards (using inverse operators) from the goal state. We switch between the two searches after completing an SS "step" in a given direction. One "step" in a particular direction corresponds to the expansion of all the representative nodes at a given level. When referring to the array A in the SS algorithm, we will use a superscript to distinguish the array used in the forward search (A^F) from the one used in the backward search (A^B). For example, $A^B[3]$ is the set of (node,weight) pairs for the nodes expanded at level 3 of the backward search.

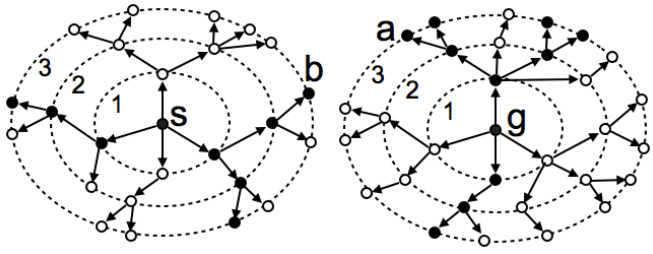

Figure 5.11: Hypothetical example of bidirectional sampling.

Figure 5.11 illustrates the situation after three steps in each direction. Nodes around both the start state s and goal state g are shown. The filled nodes are those that BiSS expands in its first three steps from s and its first three steps from g.

Stopping Condition

The stopping condition for bidirectional state-space search, when an optimal solution path is required, involves testing if a state has been generated in both directions.[4] Since A^F and A^B contain individual states that have been generated by SS in each direction, the same test could be used in BiSS. However, $A^F[n]$ and $A^B[m]$ contain only one state of each type, chosen at random, so if the number of distinct types is much smaller than the number of states this test is doomed to failure unless the number of probes is prohibitively large. We therefore base our stopping condition on the set of types that have occurred at each level of the searches and define $\mathcal{T}^F[n] = \{T(s)|\langle s, w\rangle \in A^F[n]\}$, the set of types of the nodes expanded at level n by the copy of the SS algorithm searching forward from the start state, and $\mathcal{T}^B[m] = \{T(s)|\langle s, w\rangle \in A^B[m]\}$.

The naive stopping condition would be to stop as soon as $\mathcal{T}^F[n]$ and $\mathcal{T}^B[m]$ have a type in common, where n and m are the most recently generated levels. The problem with this approach is that, often in practice, the type system being used is inaccurate and states of the same type might occur close to the start and the goal even if the start and goal are far apart. In Figure 5.11, for example, states a and b might have the same type ($T(a) = T(b)$) even though the actual distance between start and goal is greater than 6 (the combined distance from start to a and from goal to b).

We therefore use a more elaborate condition to decide when to stop the bidirectional search, requiring the type sets at the frontiers of the two searches to overlap for several consecutive levels. We call this stopping condition a *match* between the two searches, defined as follows.

Definition 21. ***(match)*** *– For any n and m we say that $\mathcal{T}^F[n]$ and $\mathcal{T}^B[m]$ match if $\mathcal{T}^F[n + v] \cap \mathcal{T}^B[m - v] \neq \emptyset$ for all $v \in \{0, 1, \cdots, K\}$ where $K = \max\{\lfloor \gamma \cdot m \rfloor, 1\}$. Here $\gamma \in [0, 1]$ is an input parameter.*

After each step in each direction we test if the same type occurs in both $\mathcal{T}^F[n]$ and $\mathcal{T}^B[m]$, where n and m are the most recently generated levels in the respective search directions. If this happens, we extend the forward search up to level $n+K$ so that a match, as defined in Definition 21, can be fully tested. This concept of match is illustrated in Figure 5.12 for $K = 2$. Each circle in the figure represents a set of types at a level of search ($\mathcal{T}^F[.]$ or $\mathcal{T}^B[.]$); each t_v denotes just one of the types in the corresponding set. The forward search has a state of type t_0 at level n; the backward search has a state of the same type at level m. The forward search continues for K more levels, producing (among others) a node of type t_1 at level $n + 1$ and a node of type t_2 at level $n + 2$. This yields a match since there are nodes of type t_1 and t_2 at levels $m - 1$ and $m - 2$, respectively, of the

[4] The correct stopping condition is more complex [31].

backwards search.

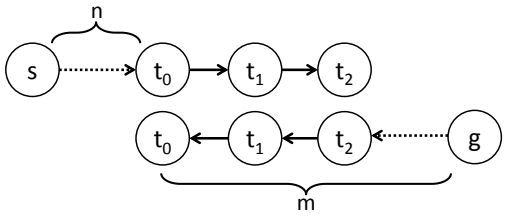

Figure 5.12: Illustration of a match for $K = 2$. Each circle represents a set of types at a level of search ($\mathcal{T}^F[.]$ or $\mathcal{T}^B[.]$); each t_v denotes just one of the types in the corresponding set. A match occurs for $K = 2$ as $t_v \in \mathcal{T}^F[n+v] \cap \mathcal{T}^B[m-v]$ for $v \in \{0, 1, 2\}$.

If a match occurs at step n from the start state and at step m from the goal state, then the searches terminate and $n+m$ is returned as an estimate of the optimal solution cost. If a match does not occur, then the searches resume from levels $n+1$ and m, or from levels n and $m+1$ depending on which frontier advanced last before checking for the match.

When a type system makes use of properties of the children and/or grandchildren of a node the definition of match only makes sense if the children/grandchildren are computed in the backward search using the forward version of the operators. Otherwise, the forward and backward searches might assign different types to the same state, thus making it impossible for a match to occur (even if $K = 0$). Also, in contrast with `SCP`, we do not use the information of the parent of a node n when computing n's type. We define the type systems used in our experiments with `BiSS` below.

Multiple Probes

The procedure just described represents one probe of our algorithm. We now describe how the information obtained from a set of p probes can be aggregated to produce a more accurate solution cost prediction. Let the type frontiers generated by probe i be denoted $\mathcal{T}_i^F[n_i]$ and $\mathcal{T}_i^B[m_i]$, where n_i is the depth of the last level generated in the forward direction by probe i and m_i is the depth of the last level generated in the backwards direction by probe i. Let $\mathcal{T}_*^F[n]$ denote the union of all the $\mathcal{T}_i^F[n]$, for $0 \leq n \leq max_i\{n_i\}$ and let $\mathcal{T}_*^B[m]$ denote the union of all the $\mathcal{T}_i^B[m]$, for $0 \leq m \leq max_i\{m_i\}$. We treat $\mathcal{T}_i^F[n] = \emptyset$ if `BiSS` did not reach level n during the i-th probe of the forward search. Likewise, we treat $\mathcal{T}_i^B[m] = \emptyset$ if `BiSS` did not reach level m during the i-th probe of the backward search.

To compute the final estimate of the optimal solution cost we set m and n to zero and gradually increment them, checking for a match between $\mathcal{T}_*^F[n]$ and $\mathcal{T}_*^B[m]$ after each increment; $n+m$ is returned as the predicted optimal solution cost when the first match occurs.

Chen assumes an `SS` probe eventually terminates by reaching leaf nodes of the search tree. We

also assume that each of BiSS's probes eventually terminates. In our case a probe will finish if it either reaches leaf nodes ($A^F[n]$ or $A^B[m]$ is empty), or if a match is found between the forward and backward frontiers. If the former happens, it means this BiSS probe predicts there is no path from start to goal. If the latter happens, this BiSS probe produces an estimate of the optimal solution cost. In all our experiments every BiSS probe finished by finding a match between the forward and backward frontiers.

5.3.3 Theoretical Analysis

Assuming a BiSS probe always terminates, we now prove that, as the number of probes goes to infinity, the probability of BiSS producing perfect predictions approaches one.

Definition 22. *Let $BFSF[n]$ be the set of types of all nodes at distance n from the start state, and $BFSB[m]$ be the set of types of all nodes at distance m from the goal state.*

For instance, in Figure 5.11, $BFSB[1]$ is the set of types of the states one step from g.

Lemma 5. *Given any value $\gamma \in [0, 1]$, for every level n expanded by BiSS in the forward search and every level m expanded by BiSS in the backward search, the probability of $\mathcal{T}_*^F[n]$ being equal to $BFSF[n]$ and $\mathcal{T}_*^B[m]$ being equal to $BFSB[m]$ approaches one as the number of probes p goes to infinity.*

Proof. Let $NE[n]$ be the set of nodes at level n of the UST rooted at the start state. According to Lemma 2 the probability of BiSS's forward search expanding every node in $NE[n]$ goes to one as the number of probes goes to infinity. The same argument is also true for BiSS's backward search and the UST rooted at the goal state. Therefore, the probability of $\mathcal{T}_*^F[n]$ being equal to $BFSF[n]$ and $\mathcal{T}_*^B[m]$ being equal to $BFSB[m]$ approaches one as the number of probes p goes to infinity. □

Lemma 6. *Given start state s with optimal solution cost c^*, goal state g, type system T, any value $\gamma \in [0, 1]$, and a number p of probes, the probability of BiSS producing an estimate $\hat{c}^*$ with $\hat{c}^* \leq c^*$ approaches one as the number of probes p goes to infinity.*

Proof. If there exists a path from s to g with cost c^*, then, for some v, bidirectional Breadth-First Search would find a state that occurs both in the forward frontier of depth v starting from s and in the backward frontier of depth v' starting from g, where $v' \in \{v, v-1\}$ and $c^* = v + v'$. This means that $BFSF[v]$ and $BFSB[v']$ have at least one type in common. Hence, for any $\gamma \in [0, 1]$, as the number of probes p goes to infinity, it follows from Lemma 5 that BiSS finds the level n and

the level m for which $\mathcal{T}_*^F[n]$ and $\mathcal{T}_*^B[m]$ match with respect to γ with probability approaching one. Since the candidate values for n and m are gradually increased, the first such values n and m found must fulfill that the probability of having $\hat{c}^* = n + m \leq c^*$ approaches one. □

By mapping the goal state g to a special unique goal type (see Definition 19), denoted here as t_g, and setting $\gamma = 1.0$, we prove that, as the number of probes p goes to infinity, the probability of BiSS producing a perfect prediction goes to one.

Theorem 5. *Given a start state s, a goal state g, a type system T mapping g to a goal type, $\gamma = 1.0$, and a number p of probes, the probability of* BiSS *producing an estimate $\hat{c}^*$ with $\hat{c}^* = c^*$ approaches one as the number of probes p goes to infinity.*

Proof. Let $\hat{c}^* = n + m$ where $\mathcal{T}_*^F[n]$ and $\mathcal{T}_*^B[m]$ is the first match found by BiSS for $\gamma = 1.0$. The probability of having $n + m \leq c^*$ approaches one as p goes to infinity follows from Lemma 6.

We now prove that the probability of having $n+m \geq c^*$ also approaches one as p goes to infinity Note that $\mathcal{T}_*^B[0]$ contains only the goal type t_g. Thus, with $\gamma = 1.0$, a match between $\mathcal{T}_*^F[n]$ and $\mathcal{T}_*^B[m]$ occurs only if $t_g \in \mathcal{T}_*^F[m+n]$. Since t_g contains only the goal state g, g must be on a path of cost $m + n$ from s, which according to Lemma 5 happens with probability approaching one as p goes to infinity. Since c^* is the optimal solution cost for s, this implies that, in this case, $m+n \geq c^*$. Consequently, the probability of having $m + n = c^*$ approaches one as p goes to infinity. □

The proof of Theorem 5 assumes that BiSS's probing expanded states of all possible types in every level before checking for a match between $\mathcal{T}_*^F[.]$ and $\mathcal{T}_*^B[.]$. This theorem proves that BiSS correctly predicts the optimal solution cost when $\gamma = 1.0$ and the number of probes goes to infinity. In the next section we show empirically that BiSS also produces accurate predictions with a limited number of probes and lower γ-values.

Time Complexity

What dictates the time complexity of BiSS is $|T|$, the size of the type system being used, b, the problem's branching factor, p, the number of probes, and $C = max_i\{n_i + m_i\}$, the largest $n_i + m_i$ value returned by the probes. We assume the representative-weight pairs (maintained by all the collections such as $A^F[.]$, $A^B[.]$) are stored in a hash table and that the insert and search operations on the table are made in constant time. We further assume a probe will terminate with a match of the two frontiers. BiSS generates $|T| \cdot b$ nodes at each step of the forward or backward frontiers in the worst case. Therefore, BiSS generates up to $|T| \cdot b \cdot C$ nodes during each probe. In the worst case, when checking for a match between the two frontiers there will be a nonempty intersection between

$\mathcal{T}^F[.]$ and $\mathcal{T}^B[.]$ for all values of v (as in Definition 21) except the last one. When $\gamma = 1.0$ this results in $|T| \cdot C^2$ comparisons until the match is found and the probe terminates. Therefore, in the worst case BiSS has a running time of $O(p \cdot (|T| \cdot b \cdot C + |T| \cdot C^2))$.

Memory Complexity

The size of the type system $|T|$ and $C = max_i\{n_i + m_i\}$ determine the memory complexity of BiSS. We again assume a probe will always finish with a match between the two frontiers. In the worst case there will be $|T|$ states at each level of both forward and backward frontier. We can approximate the number of representative-weight pairs to be stored in memory when $\gamma = 1.0$ as $C \cdot |T| + \frac{C}{2} \cdot |T|$. The first term in the sum accounts for the pairs in the forward frontier, and the second for the pairs in the backward frontier. Recall that the memory requirement for the forward frontier is larger as this is the frontier we advance while looking for a match. Thus, BiSS has a worst-case memory complexity of $O(C \cdot |T|)$.

5.3.4 Experimental Results

In this section we evaluate empirically the accuracy and runtime of BiSS.

Experimental Setup

Problem domains.

Our experiments are run in three domains: the Blocks World, the Sliding-Tile puzzle, and the Pancake puzzle. We use two sizes for each domain, a smaller size used to compare BiSS to SCP and a larger size to demonstrate the scalability of BiSS. SCP cannot be run on the large versions of the domains as its preprocessing step would be prohibitively time-consuming.

Heuristic functions. As stated earlier in the chapter, while not designed to, heuristic functions themselves can be used as predictors of the optimal solution cost if they are applied to the start state. They are typically faster but less accurate than predictors designed exclusively to predict the optimal solution cost. To show this we also compare the accuracy of BiSS's predictions with the accuracy of two heuristic functions. First, it is natural to compare BiSS to the heuristic used to define its type system. In our experiment, this heuristic is always admissible. However, as we observed in our experiments with SCP, admissible heuristic functions are known as poor estimators of the optimal solution cost compared to inadmissible heuristics. For examples of inadmissible heuristics see, e.g., Bonet and Geffner [6], Hoffmann and Nebel [30], and Richter, Helmert, and Westphal [75]. Like we did in the SCP experiments, we choose the Bootstrap heuristic [35] to represent the class

of inadmissible heuristics for two reasons. First, IDA* with the Bootstrap heuristic was found to produce near-optimal solutions while expanding relatively few nodes, which suggests the heuristic is providing accurate estimates of the optimal solution cost. Second, the Bootstrap heuristic was shown to be superior to some of the inadmissible heuristics mentioned above on the Blocks World (Jabbari Arfaee et al. [35]).

Type systems. We use the type systems defined in Equations 3.5 and 3.6. However, in contrast with the experiments with CDP and SCP, we do not consider the heuristic value of the parent of node n when computing n's type. We redefine such type systems as T'_c and $T'gc$ below.

$$T'_c(s) = (h(s), c(s,0), \ldots, c(s,H)), \tag{5.8}$$

where $h(s)$ is the heuristic value of node s, $c(s,k)$ is how many of s's children have heuristic value k, and H is the maximum heuristic value a node can assume;

$$T'_{gc}(s) = (T'_c(s), gc(s,0), \ldots, gc(s,H)), \tag{5.9}$$

where $gc(s,k)$ is how many of s's grandchildren have heuristic value k.

We run BiSS with the same set of input parameters for all the experiments. In particular, we use 2 probes and $\gamma = 0.5$. As K also depends on the number of steps m (see Definition 21), BiSS is able to make accurate predictions in domains with different average solution costs while using the same γ-value. SCP was run with the input parameter that produced the best results among a large set of parameters we tested. For BiSS the type system and the set of input parameters (p and γ) were chosen so that BiSS would make predictions quickly. For instance, BiSS's predictions are more accurate using the larger T'_{gc} type system. However, using T'_{gc} in domains with large branching factor could result in very large type systems, which would result in slow prediction computations. Thus, T'_c will be preferred in that case. Besides T'_c and T'_{gc} one could also create type systems "in between" those two by evaluating only a subset of the children or a subset of the grandchildren of a node while calculating its type. Another alternative for controlling the size of the type system is to use the clustering-based type systems described in Chapter 3. The type system used in each experiment is specified below. When using a type system T for BiSS we used the same T for SCP, but with the heuristic value of the parent node added.

As in the SCP experiments, predictions are compared using relative unsigned error

The results will be presented in tables like Table 5.3. The first column shows the optimal solution cost, followed by the relative unsigned error of different predictors. We compare BiSS with SCP,

Cost	Relative Unsigned Error				Percentage Correct		
	BiSS	SCP	BS	h	BiSS	SCP	BS
16	**0.00**	0.06	0.44	0.06	**100**	0	0
17	**0.00**	0.04	0.33	0.18	**100**	40	0
18	**0.02**	0.03	0.28	0.19	**75**	43	0
19	**0.01**	0.07	0.22	0.24	**73**	2	0
20	**0.01**	0.09	0.20	0.26	**77**	1	0
21	**0.02**	0.12	0.15	0.30	**70**	0	0
22	**0.02**	0.14	0.12	0.33	**70**	0	1
23	**0.01**	0.17	0.08	0.36	**74**	0	18
24	**0.01**	0.18	0.06	0.39	**81**	0	11
25	**0.01**	0.20	0.04	0.41	**78**	0	10
26	**0.01**	0.22	0.02	0.43	**84**	0	70
27	**0.00**	0.21	0.03	0.45	**100**	0	30
28	**0.00**	0.20	0.05	0.48	**100**	0	0

Table 5.3: BiSS and SCP predictions on the15 blocks world

Bootstrap (BS), and the heuristic (h) used to define the type system for BiSS and SCP. We also show the percentage of problem instances for which a predictor makes perfect predictions ($\hat{c}^* = c^*$). The best values in each row are in bold.

Blocks World

We ran experiments on smaller problems with 15 blocks and larger ones with 20 blocks, using 1,000 uniformly random instances to measure the accuracy of the predictors. Optimal solutions were obtained with Slaney and Thiebaux's [78] solver. In both cases, for ease of comparison to SCP and Bootstrap, the goal state is fixed and has all blocks in a single stack. The type system used is T'_c, built with the very weak "Out of Place" heuristic, which counts the number of blocks not in their goal position, cf. Jabbari Arfaee et al. [35]. SCP was run with the type system T'_c, sampling 10 million states in its preprocessing step, and using 10 uniformly random instances to calculate the required ϵ-values.

Our results on 15 blocks are shown in Table 5.3. BiSS is very accurate for this domain; its predictions are nearly perfect. Bootstrap and SCP's errors vary considerably with the optimal solution cost of the problem instances and are much higher than BiSS's error. For 20 blocks (Table 5.4), again BiSS makes nearly perfect predictions and is far more accurate than the Bootstrap heuristic. BiSS's predictions are also substantially more accurate than the values of the heuristic used to build the type system. For example, for problem instances with optimal solution cost of 37 BiSS makes perfect predictions, while the heuristic has an error of 47%.

Cost	Relative Unsigned Error			Percentage Correct	
	BiSS	BS	h	BiSS	BS
22	**0.00**	0.50	0.09	**100**	0
24	**0.02**	0.29	0.19	**50**	0
25	**0.02**	0.26	0.22	**50**	0
26	**0.03**	0.26	0.25	**38**	0
27	**0.01**	0.22	0.27	**72**	0
28	**0.02**	0.18	0.29	**59**	0
29	**0.01**	0.15	0.32	**66**	0
30	**0.02**	0.13	0.34	**57**	0
31	**0.01**	0.10	0.36	**63**	0
32	**0.01**	0.08	0.38	**68**	6
33	**0.01**	0.06	0.40	**68**	7
34	**0.01**	0.04	0.42	**70**	12
35	**0.01**	0.03	0.43	**79**	11
36	0.01	**0.00**	0.45	80	**95**
37	**0.00**	0.03	0.47	**100**	28
38	**0.00**	0.04	0.49	**100**	0

Table 5.4: BiSS predictions on the 20 Blocks World

Sliding-Tile Puzzle

The smaller size of the sliding-tile puzzle we experimented with was the 4x4 puzzle (15-puzzle). Accuracy was measured over 1,000 uniformly random instances. We used T'_{gc} with Manhattan Distance (MD) as the type system. Table 5.5 shows the results. MD underestimates the actual solution cost by about 30%. The Bootstrap heuristic, SCP and BiSS with our default set of parameters (2 probes and $\gamma = 0.5$) are all very accurate for this domain. SCP is slightly more accurate than BiSS with small costs but the trend shifts for larger costs. However, in results not shown, if the number of probes and the γ-value are increased, BiSS and SCP make predictions of similar accuracy for the small costs too. Both predictors are more accurate than the Bootstrap heuristic.

The larger puzzle version used, the 5x5 puzzle (24-puzzle), has $25!/2$ states reachable from the goal. Here accuracy was measured over 433 uniformly random start states, which were solved optimally using the 6-6-6-6 disjoint pattern database with reflection along the main diagonal [42]. We used T'_{gc} with MD as the type system. Table 5.6 shows the results. BiSS is substantially more accurate than Bootstrap. For example, for instances with optimal solution cost of 100, BiSS's predictions are only 3 moves ($0.03 * 100$) different than the true optimal solution cost, on average, whereas Bootstrap's are 8 moves different.

Pancake Puzzle

For the pancake puzzle we also compare to the "GaP" heuristic [28], a highly accurate hand-crafted admissible heuristic for this domain.

Cost	Relative Unsigned Error				Percentage Correct		
	BiSS	SCP	BS	h	BiSS	SCP	BS
48	0.08	**0.06**	0.09	0.29	13	**15**	5
49	0.07	**0.05**	0.08	0.30	17	**20**	11
50	0.07	**0.04**	0.08	0.29	14	**21**	2
51	0.07	**0.03**	0.07	0.29	8	**43**	7
52	0.07	**0.03**	0.07	0.30	23	**42**	6
53	0.07	**0.03**	0.06	0.31	19	**32**	4
54	0.07	**0.03**	0.07	0.30	6	**37**	9
55	0.06	**0.03**	0.08	0.29	10	**35**	4
56	0.07	**0.04**	0.07	0.30	9	**23**	6
57	0.06	**0.04**	0.06	0.30	10	**23**	13
58	0.08	**0.04**	0.07	0.29	2	**23**	6
59	0.05	**0.04**	0.08	0.27	7	**10**	0
60	0.06	**0.06**	0.07	0.28	8	**16**	8
61	**0.05**	0.06	0.08	0.28	**23**	11	0
62	**0.06**	**0.06**	0.09	0.28	0	**7**	3
63	**0.04**	0.06	0.08	0.27	**17**	5	0

Table 5.5: BiSS and SCP predictions on the 15-puzzle

The smaller size of the pancake puzzle we used was the 10-pancake puzzle, which has 10! states reachable from the goal state. We used 5,000 uniformly random instances to compute the accuracy of the predictions. The heuristic used to construct the type systems was the maximum of the regular and the dual lookups [85] of a pattern database built by keeping the identity of the four smallest pancakes and turning the other pancakes into "don't cares". Here the type system used is T_c'. Table 5.7 shows that even GaP is not as accurate as BiSS for the 10-pancake.

The larger size of the pancake puzzle we used was the 35-pancake, which has 35! states reachable from the goal state. We used 1,000 uniformly random instances to measure accuracy and solved them optimally using the GaP heuristic. The heuristic used in the experiment was the 5-5-5-5-5-5-5 additive pattern database heuristic [82], which we used to construct a "coarser" version of the T_c' type system and to provide features for the Bootstrap heuristic [35]. Even though very accurate, BiSS's prediction computations were slow when using the T_c' type system. In order to speed up the predictions, we reduced the size of the type system by accounting for the heuristic value of only the three children generated by the first three actions available to a node according to our implementation. This is in contrast with the regular T_c' type system which accounts for the heuristic values of all the children. Table 5.8 shows the results. For an optimal solution cost larger than 32, the results for BiSS, Bootstrap, and GaP are similar. Here we observe that all three methods produce accurate predictions in this domain, and BiSS was never worse than any other tested method. Moreover, BiSS tends to have a higher percentage of problem instances predicted perfectly. Note that as the Pancake puzzle gets larger the relative unsigned error of the GaP heuristic gets smaller.

Cost	Relative Unsigned Error			Percentage Correct	
	BiSS	BS	h	BiSS	BS
82	**0.05**	0.14	0.24	**0**	**0**
84	**0.04**	0.09	0.25	**0**	**0**
86	**0.04**	0.07	0.26	0	**11**
88	**0.06**	0.09	0.25	**12**	0
90	**0.05**	0.07	0.28	**0**	**0**
92	**0.05**	0.07	0.25	**19**	3
94	**0.04**	0.07	0.26	**18**	0
96	**0.05**	0.08	0.25	**13**	2
98	**0.03**	0.07	0.26	**20**	5
100	**0.03**	0.08	0.26	**26**	2
102	**0.04**	0.08	0.25	**11**	0
104	**0.03**	0.08	0.25	**25**	0
106	**0.04**	0.08	0.25	**10**	0
108	**0.03**	0.08	0.24	**37**	5
110	**0.03**	0.09	0.24	**17**	0
112	**0.04**	0.07	0.25	**0**	**0**
114	**0.03**	0.08	0.24	**0**	**0**
116	**0.04**	0.10	0.20	**12**	0
118	**0.02**	0.10	0.23	**33**	0
120	**0.04**	0.09	0.21	**0**	**0**

Table 5.6: BiSS predictions on the 24-puzzle

Cost	Relative Unsigned Error				Percentage Correct		
	BiSS	SCP	GaP	h	BiSS	SCP	GaP
7	**0.04**	0.12	0.08	0.18	**73**	25	53
8	**0.02**	0.05	0.07	0.21	**84**	60	48
9	0.05	**0.04**	0.07	0.23	58	**67**	46
10	**0.06**	0.07	**0.06**	0.25	**49**	34	46

Table 5.7: BiSS and SCP predictions on the 10-pancake

Empirical Runtime

BiSS's runtime is polynomial in the size of the type system, the predicted solution cost, the number of probes, and the branching factor. Table 5.9 shows how this time complexity translates into actual runtime by showing the fastest (min), slowest (max), and the average (mean) prediction runtimes for the set of problem instances used in the accuracy experiment above. Table 5.9 shows the runtime for $p = 2$ and $\gamma = 0.5$ (using an Intel Xeon CPU X5650, 2.67GHz).

BiSS is thus considerably faster than solving the problem suboptimally; the mean times for Bootstrap (also using a 2.67GHz machine) to suboptimally solve one single instance were 3h 49 mins, 14 mins 5 s, and 2 h 29 mins, respectively, for the three domains in Table 5.9 [35]. The advantage becomes even more evident when comparing to optimal solving time. For instance, Korf et al. [50] predicted it would take 51,454 years to solve one 24-puzzle instance of average solution cost with IDA* guided by Manhattan Distance. BiSS takes about 30 seconds on average to make

Cost	Relative Unsigned Error				Percentage Correct		
	BiSS	BS	GaP	h	BiSS	BS	GaP
29	**0.03**	**0.03**	0.04	0.21	**50**	33	16
30	**0.03**	0.05	**0.03**	0.18	**36**	18	18
31	**0.02**	0.04	0.03	0.19	**36**	27	23
32	**0.01**	0.03	0.02	0.19	**62**	30	23
33	**0.02**	0.03	**0.02**	0.19	**38**	33	34
34	**0.02**	0.03	**0.02**	0.19	**39**	32	37
35	**0.02**	**0.02**	**0.02**	0.19	34	**42**	36
36	0.02	**0.01**	0.03	0.19	33	**52**	0

Table 5.8: BiSS predictions on the 35-pancake

Domain	min	max	mean
20 Blocks World	26s	57s	41s
24 Sliding-Tile Puzzle	18s	48s	30s
35 Pancake Puzzle	19s	30s	24s

Table 5.9: BiSS runtime for $p = 2$, $\gamma = 0.5$.

very accurate predictions of the optimal solution cost when using Manhattan Distance to build its type system.

5.3.5 Predictions for Very Large State Spaces

We also used BiSS (again using $p = 2$, $\gamma = 0.5$, and the T'_{gc} type system with MD) to predict the optimal solution cost of problem instances for the n^2 Sliding-Tile puzzle with $n \in \{6, 7, 8\}$, i.e., state spaces much too large to be solved optimally by any known technique in a reasonable time. The number of instances for which predictions were made and the average time (in minutes) taken by BiSS to compute one prediction are shown in the first two rows of Table 5.10. We have no way to verify the accuracy of the individual predictions directly, but we did devise a way to evaluate the accuracy of the average predicted optimal solution cost on these sets of instances; the average predictions are shown in the third row of Table 5.10.

Parberry [67] proved lower and upper bounds for the average solution cost of the n^2-puzzle to be cubic in n. Thus one way to estimate the average solution cost for the Sliding-Tile puzzle is to fit a third-order polynomial to the known average solution costs and then infer the unknown average solution costs. The average solution cost for the (2x2), (3x3), and (4x4) puzzles are roughly 3, 22, and 53, respectively. The average solution cost obtained from solving more than 400 instances of the (5x5) puzzle is approximately 101. The third-order polynomial fit for these data is $0.8333 \cdot n^3 - 1.5 \cdot n^2 + 10.6667 \cdot n - 19$. The results for the polynomial fit, shown in the final row of Table 5.10, are very close to BiSS's average predictions, suggesting that BiSS's individual predictions are also accurate.

	Configuration		
	6x6	7x7	8x8
# instances	1,000	650	50
BiSS time (mins)	6	18	80
BiSS avg predicted cost	172	280	423
Polynomial predicted cost	171	268	397

Table 5.10: BiSS's predicted average optimal solution cost for very large state spaces.

5.3.6 Parameter Selection

In our experiments we fixed the number of probes p to 2 and the confidence parameter γ to 0.5. How would BiSS's accuracy be affected by different settings of these parameters? We use the *relative signed error* to better understand the impact of different p and γ on BiSS's predictions. The relative signed error is calculated by summing the difference between the predicted cost with the actual optimal solution cost for each problem instance. This sum is then divided by the sum of the actual costs. A system that always underestimates the actual optimal solution cost will have a negative relative signed error.

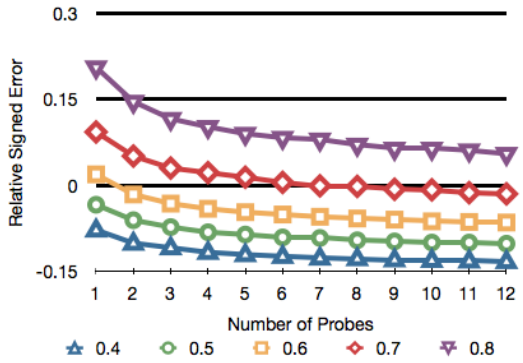

Figure 5.13: Each curve represents the relative signed error for different values of γ; for larger number of probes, the predictions tend to have a negative signed error.

According to Lemma 6 for $\gamma < 1.0$ BiSS will have a zero or negative relative signed error in the limit of large p. This trend is illustrated in an experiment on the 15-puzzle shown in Figure 5.13, where each curve shows the relative signed error of the predictions for different values of γ. As we increase the number of probes, the relative signed error of BiSS becomes negative, independently of the value of γ. Hence, for any setting of $0 < \gamma < 1.0$, increasing p beyond a certain value will cause signed error, and therefore also unsigned error, to increase. With sufficiently small values of γ BiSS will almost always underestimate the optimal solution cost, so it will have a negative signed error even when $p = 1$, which will only get worse as p is increased. For larger values of γ BiSS will overestimate the optimal solution cost when $p = 1$ so its signed error will be positive.

Increasing p will drive the signed error towards 0, i.e., increase the accuracy of the predictions, until p is large enough that the signed error becomes negative. At this point further increases of p will cause accuracy to get worse. This gives some guidance as to how these two parameters might be set automatically: find a value of γ that is small (so that prediction is fast) but sufficiently large that `BiSS` with $p = 1$ overestimates, and then increase p until `BiSS` begins to underestimate.

5.4 The C&C Assumption for Predicting Optimal Solution Costs

`CDP` and `SS`, presented in Chapter 3, assume that nodes of the same type root subtrees of the same size. `TSS`, introduced in Chapter 4, assumes that nodes of the same type root isomorphic subtrees. The assumption `SCP` and `BiSS` make for predicting the optimal solution cost is not as intuitive. That is why we chose to state their assumption after fully explaining the two algorithms.

`SCP` and `BiSS` assume that subtrees R_m and R_n rooted at nodes n and m of the same type have the same set of types at any level i. Formally, let S_m^i and S_n^i be the set of nodes encountered at level i of subtrees R_m and R_n, respectively. For type system T and level i, `SCP` and `BiSS` assume that,

$$\{T(s)|s \in S_m^i\} = \{T(s)|s \in S_n^i\} \tag{5.10}$$

When using a type system that has goal types and obeys Equation 5.10 for all nodes m and n of the same type, both `SCP` and `BiSS` are able to produce perfect predictions of the optimal solution cost.

5.5 Learning Strong Heuristic Functions from Predictions

The use of machine learning to learn a heuristic function recently became popular, see for instance [17, 35, 76, 83]. One of the main challenges faced by such learning systems is to collect training instances; in order to learn heuristics one must know the solution cost of a sufficiently large number of problem instances. We now describe one successful system developed for learning strong heuristic functions for large state spaces called Bootstrap, which we used in other experiments in this chapter.

5.5.1 The Bootstrap Learning System

Jabbari Arfaee et al., [35] presented a learning system, Bootstrap, that generates training data through bootstrapping. Bootstrap tries to solve problem instances with a (possibly weak) initial heuristic h_0 within a time limit. The instances that the method manages to solve form a training

set that is used to learn another, stronger heuristic h_1. The process is then repeated with h_1 replacing h_0, hoping that some of the instances not solved in the previous iteration will be solved and a new training set will be obtained. Experiments showed that IDA* finds near-optimal solutions for state-space problems once a Bootstrap heuristic is learned.

We refer to Jabbari Arfaee et al.'s system as the *Bootstrap system* and to the heuristics learned by their system as the *Bootstrap heuristics*.

The Bootstrap Drawback

Although the Bootstrap system is able to learn strong heuristic functions for large state spaces, the process of learning could be time-consuming. For example, the Bootstrap system takes two days to learn a strong heuristic for the Rubik's Cube. The bootstrap process is time-consuming for two reasons. First, Bootstrap spends a substantial amount of time trying and failing to solve problem instances to form the training set. This procedure is time-consuming because it is hard to separate the hard-to-solve from the easy-to-solve instances. It is difficult to make this separation even when using `CDP` or `SS` to predict the number of nodes expanded by IDA* while solving the instances. In addition to being able to predict the size of the search tree, in order to be able to separate the easy from the hard instances one has to know the optimal solution cost of those instances. `SCP` or `BiSS` could be used to estimate the optimal solution cost, but even accurate predictions of the optimal solution cost (e.g., predictions with relative unsigned error within 0.02) could be misleading. For instance, if `BiSS` predicts the optimal solution cost of a problem instance of the 24-puzzle is 100 when it is actually 102, the instance is in fact much harder to solve than we predict it is. This is because the size of the search tree usually grows exponentially with the solution cost.

The second reason for the Bootstrap learning process being slow is that it ultimately learns from the easy instances first. Therefore, it can take several iterations until a heuristic is created that is able to solve hard problem instances quickly.

5.5.2 Using BiSS to Learn Heuristic Functions

We propose `BiSS-h`, a different approach to learning heuristics, with the goal of reducing learning time. Instead of using search to solve problem instances to generate a training set, we use `BiSS` predictions to generate the training set.

Algorithm 14 presents `BiSS-h`. It takes as input a value γ and a type system T, both required by `BiSS`. Recall that γ dictates how much the forward and the backward frontiers must overlap so that a match occurs. Higher values of γ will require a larger overlap of the frontiers, thus the predictions

Algorithm 14 `BiSS-h`

Input: value of γ, type system T
Output: heuristic function h

1: Collect a set I of problem instances
2: $Tr \leftarrow \emptyset$
3: **for** each $i \in I$ **do**
4: $\quad c_i \leftarrow \texttt{BiSS}(i, \gamma)$
5: $\quad Tr \leftarrow \{Tr, (i, c_i)\}$
6: **end for**
7: learn h from Tr

will tend to overestimate the optimal solution cost for a fixed number of probes. A training set with solution costs that overestimate the actual values will bias the heuristic function learned from that set to also overestimate its estimations of the cost-to-go. Again we use $\gamma = 0.5$.

In line 1 `BiSS-h` collects the instances that will form the training set. In our experiments we use a general method that collects instances by performing random walks from the goal and also by generating truly random instances when possible. We mix instances generated by random walks of short length with random instances so that the training set will contain both easy and hard problem instances. For each instance i in I we generate one training pair (i, c_i). This is in contrast with the Bootstrap method, which uses, for every bootstrap instance solved by the method, all the nodes on the path to the goal. By doing so Bootstrap balances the number of easy and hard instances in its training set. As `BiSS` does not solve a problem instance, it does not generate a path from start to goal. Thus, we add easy instances to our training set with the random walks of short length. After we collect the set I and `BiSS` makes predictions c_i for each instance $i \in I$, the resulting training set Tr is used for learning a heuristic function h.

Using Neural Networks

The learning algorithm we use in our experiments is the same as that used by Jabbari Arfaee et al.: a neural network (NN) with one output neuron representing cost-to-goal and three hidden units trained using standard backpropagation and mean squared error (MSE). Training ended after 500 epochs or when MSE < 0.005. The time required to train a NN is only a few seconds, thus we ignore this time when computing the learning time for both Bootstrap and `BiSS-h`. For Bootstrap and `BiSS-h` we always used exactly the same set of features, which we describe below.

5.5.3 Experimental Evaluation of BiSS-h

Our experiments are on the 24-puzzle, the 35 pancake puzzle, and Rubik's Cube. For a description of these domains see Section 2.4.

Experimental Setup

We compare BiSS-h solely to Bootstrap since (1) we evaluate its learning time and (2) in terms of search performance, IDA* using a Bootstrap heuristic outperforms standard suboptimal search algorithms on the domains tested [35]. All our experiments are run on a machine with 2.6 GHz clockspeed. Both systems are tested on the same test instances.

As we are interested in learning effective heuristics quickly, we use with BiSS-h the smallest training sets reported by Jabbari Arfaee et al. [35] (500 training instances). Smaller training sets result in shorter learning time.

Evaluation metric. We evaluate Bootstrap and BiSS-h primarily on the basis of the time required to learn a heuristic. To ensure that the heuristics learned are of roughly the same quality, we also measure the average solving time and average suboptimality of the solutions. We compute the suboptimality for one problem instance as follows. We divide the cost of the solution found by the optimal cost and then subtract one from the result of the division and multiply by 100. For instance, a value of suboptimality of 5.7 for learning algorithm X means in average the solutions found by IDA* using the heuristic learned by X were 5.7% longer than optimal. We present units of time in our tables of results with the letters d, h, m, and s after the numbers, indicating days, hours, minutes, and seconds, respectively. All heuristics were tested using the same implementation of IDA*.

24-puzzle. The features used for the NN were Manhattan Distance (MD), number of out-of-place tiles, position of the blank, and five 4-tile pattern PDBs. BiSS uses the T'_{gc} type system with MD to predict the optimal solution cost of the instances in I. For this domain, 400 out of the 500 training instances were generated with random walks from the goal. The length of each random walk was chosen randomly between 1 and 50 moves. The 100 remaining training instances were generated randomly. Korf and Felner's 50 problem instances [42] constitute the test set.

35 Pancake Puzzle. The input features for the NN were seven 5-token PDBs (but instead of using the regular lookup from the PDBs, we use the maximum of the regular and the dual lookups [85]), a binary value indicating whether the middle pancake is out of place, and the number of the largest out-of-place pancake. BiSS uses the T'_c type system with the additive PDBs as heuristic functions [82]. The training instances for the 35 Pancake Puzzle are generated exactly in the same way they are generated for the 24-puzzle. We use 50 random instances as the test set.

Rubik's Cube. The features used for Rubik's Cube are the three PDBs introduced by Korf [**?**]. The test set consist of Korf's 10 instances [**?**]. BiSS uses the T'_c type system with the three PDBs

introduced by Korf [**?**]. However, instead of taking the maximum of the PDBs, as Korf uses to solve instances of the puzzle, we use the values of each PDB separately in the type system. For this domain, we generate easy and hard training instances solely with random walks from the goal. The length of each random walk for Rubik's Cube was chosen randomly between 1 and 80 moves.

Experimental Results

Table 5.11 shows the `BiSS-h` experimental results .We observe that with `BiSS-h` we reduce the time required for learning from the order of hours and days to minutes, while keeping the quality of the learned heuristics roughly the same. An important fact to note from the data in Table 5.11 is that this is the first time, to the best of our knowledge, that near-optimal solutions have been found for Rubik's Cube in substantially less time than optimal solutions – the fastest known average optimal solving time for random instances of the Rubik's Cube is due to Zahavi et al. [85]: 12 hours, 16 minutes and 41 seconds on average. Moreover, if using 200 training instances instead of 500 we can learn a strong heuristic in only 21 minutes for this domain.

Algorithm	Learning Time	Solving Time	Suboptimality
24-puzzle			
BST	11h 43m	64s	5.7
`BiSS`	17m	77s	5.0
35 Pancake Puzzle			
BST	1d 11h	48s	5.5
`BiSS`	33m	33s	5.0
Rubik's Cube			
BST	2d	8,253s	4.0
`BiSS`	53m	2,148s	11.4

Table 5.11: We observe major reductions in learning time when using `BiSS` for learning heuristic functions.

Solving single instances. Bootstrap's learning times are prohibitive when one is interested in solving a single problem instance. To address this problem, Jabbari Arfaee et al. suggested a method that interleaves learning heuristics and solving the given target instance.

By using the interleaving method the Bootstrap system finds solutions of similar quality of those presented in Table 5.11, but in much less time. On average, it solves an instance of the 24-puzzle in 14 minutes and 5 seconds; an instance of the 35 Pancake Puzzle in 1 hour and 42 minutes; and an instance of the Rubik's Cube in 10 hours and 54 minutes [35]. Only on the 24-puzzle is the interleaving system of Bootstrap slightly faster than `BiSS-h` for learning a general-purpose heuristic function. Our results suggest that it tends to be faster to use `BiSS-h` to learn a general-purpose

heuristic function $h(\cdot)$ and then use $h(\cdot)$ to solve the problem instance than to use Bootstrap's interleaving process.

The Selection of Training Instances Matters

We ran additional experiments with `BiSS-h` on the 20 blocks world and they showed that the strategy for collecting the training instances matters. IDA* with the heuristic `BiSS-h` learns using the same strategy as in the other experiments (i.e., using a mix of easy and hard training instances) solved only 10 out of the 50 test instances with a time limit of one hour per instance. `BiSS-h` clearly failed to learn a strong heuristic in this case considering that IDA* solves the same instances in about 30 seconds on average when using a Bootstrap heuristic. We then ran our system again but, instead of using the training set consisting of random walk instances and truly random instances, we used the instances used by Bootstrap on its last iteration, including all the instances on a path to the goal found by Bootstrap. With these instances our system learned a strong heuristic. This result suggests that the training instances play a crucial role on the quality of the heuristic learned.

5.6 Conclusions

In some real world scenarios it might be sufficient to know the solution cost of a problem. Classical search algorithms find the solution cost by finding an optimal path from start to goal. Heuristic functions estimate the cost of such a path, but are required to be fast enough to be calculated for many nodes during the search.

In this chapter we presented `SCP` and `BiSS`, two algorithms designed for predicting the optimal solution cost of a problem. While such methods can be viewed as heuristic functions, they differ from heuristics conceptually in that: 1) they are not required to be fast enough to guide search algorithms; 2) they do not favor admissibility; and 3) they aim at making accurate predictions and thus our measure of effectiveness is prediction accuracy, in contrast to solution quality and search time used to measure the effectiveness of heuristic functions.

We showed empirically that `SCP` makes fairly accurate predictions in most of the domains tested. We also studied the impact of the parameters required by `SCP` on the prediction accuracy. Namely, we empirically studied the impact of the threshold parameter c, of the ϵ-cuts, and of the prediction lookahead r on the prediction accuracy. Our results suggested that (1) for any value of c between 0.8 and 0.99 `SCP` makes accurate predictions; (2) ϵ-truncation can substantially improve the prediction accuracy; and (3) the prediction lookahead can improve the prediction accuracy at the cost of increasing the runtime.

We showed that BiSS does not require preprocessing and is guaranteed to return the optimal solution cost in the limit as the number of its probes goes to infinity. We showed empirically that BiSS makes accurate predictions in several domains while using the same set of input parameters. We also studied empirically how to set BiSS's input parameters p and γ. When verifying its accuracy, our experiments showed that BiSS scales much better than SCP. We also showed BiSS could be applied to state spaces much larger than can be solved optimally in a reasonable time.

Finally, we presented an application of a solution cost predictor. Namely, we presented BiSS-h, a learning system that uses a solution cost predictor instead of a search algorithm to generate the training set required to learn heuristics. BiSS-h is able to quickly learn effective heuristics. Our system reduces the time required previously to learn effective heuristics from days to minutes.

Our empirical results with BiSS-h on the 20-blocks world clearly suggest that by carefully selecting training instances we could learn much stronger heuristics. To the best of our knowledge, our learning system is the only one that gives the flexibility of choosing which instances are used for training. Previous systems are restricted to the instances that are solvable by a search method to form the training set. As BiSS can predict the optimal solution cost of virtually any problem instance, our system opens up the possibility of applying active learning techniques [77] – a subject of future work.

Chapter 6

Stratified Tree Search

6.1 Introduction

Many interesting real-world problems can be cast as a state-space search problem. In Section 2.4 we describe several domains we use as benchmark problems for our algorithms and how they represent a broad class of important problems. Heuristic search algorithms are frequently used for solving state-space problems. In this chapter we present algorithms for finding suboptimal solution paths from a given start state to a goal state.

6.1.1 Problem Formulation

Given a directed and implicitly defined search tree representing a state-space problem [65] rooted at start state s^*, the least-cost path planning problem is the task of finding an optimal solution path from s^* to a goal node. In this chapter we are interested in finding suboptimal solutions for the least-cost path planning problem. Although we are looking for suboptimal solutions, we are interested in finding solutions of good quality, i.e., solutions as near to optimal.

6.1.2 Background

As mentioned in earlier chapters, heuristic search algorithms use a heuristic function to guide their search to find solutions for state-space problems. A heuristic function $h(\cdot)$ estimates the cost-to-go of a node, i.e., it estimates the optimal cost of a solution path from a given node. Intuitively, nodes with lower heuristic value are more promising than nodes with higher heuristic value as they seem to be closer to a solution. Recall that heuristic search algorithms such as IDA* [43] use the function $f(s) = g(s) + h(s)$ to guide their search, where $g(s)$ is the cost from a start state s^* to node s in the search tree. $f(s)$ is an estimate of the lowest cost of all solutions for s^* that go through s. IDA* usually explores fewer nodes while searching for a solution if $h(\cdot)$ offers accurate estimates of the

cost-to-go.

Beam search [5] uses heuristic functions in a different way to guide search. Beam search selects for expansion the B nodes with the lowest f-value at a given level of search, where the "beam width" B is an input parameter. Thus, in this case, the accuracy of the heuristic estimates is no longer important. For beam search, ideally, a heuristic function will correctly rank nodes in the search tree: nodes closer to a goal receive a lower heuristic value than nodes farther from a goal. In contrast with IDA*, here the absolute value is not important. The existence of different strategies for using a heuristic function allows one to choose the appropriate algorithm for existing heuristic functions in practical scenarios.

In this chapter we present a heuristic search algorithm, Stratified Tree Search (`STS`), that uses a heuristic function in yet a different way. Our goal was to develop a general-purpose heuristic search algorithm for finding near-optimal solutions in relatively large domains when only weak heuristics were available. When we refer to weak heuristics we mean heuristics that (1) do not provide accurate estimates of the cost-to-go and (2) do not provide an accurate ranking of the nodes in the search tree. Analogously, when we refer to strong heuristics we mean heuristics that either (1) provide accurate estimates of the cost-to-go or (2) provide an accurate ranking of the nodes in the search tree.

While other heuristic search algorithms rely on the estimates of the cost-to-go or on the ranking provided by a heuristic for the nodes in the search tree, `STS` uses a heuristic to group together nodes with similar solution cost. Consider the following example.

Example 13. *Suppose an agent wants to find the shortest path from its current location to the closest grocery store. If it is known that heading south or west from its current location will lead to paths of the same length, then only one of the two initial directions (south or west) must be further explored for planning, the other direction can be ignored. Note that we do not need to know the estimated solution length by heading south or west nor which direction looks more promising—we only need to know which directions lead to solutions of the same cost.*

6.1.3 Contributions

The content of this chapter is published in the Proceedings of the Conference on Autonomous Agents and Multiagent Systems (2013) [59]. In this chapter we make the following contributions.

- We present `STS`, which is the application of the ideas of Chen's `SS` for finding suboptimal paths from start to goal in state-space search problems. `STS` also uses a type system and it assumes that nodes of the same type lead to solutions of the same cost. As we explain

later in the chapter, STS has Breadth-First Search and random walk search as special cases. STS using different type systems results in different search algorithms in the spectrum of possibilities between Breadth-First Search and random walks.

- We empirically compare STS to BULB [21], a variation of beam search equipped with backtracking, and Weighted IDA* (WIDA*) [45], two standard suboptimal heuristic search algorithms that also scale to state spaces as large as the ones we use as testbeds, and also to IDA*-BST, which is IDA* using the Bootstrap heuristic [35].

- STS offers a good tradeoff between solution quality and search speed by varying the size of the type system in most of the cases. However, in some cases, STS does not provide a fine adjustment of this tradeoff. We present a simple variant of STS, named Beam STS (BSTS), that allows one to make fine adjustments of the tradeoff between solution quality and search speed. BSTS combines the ideas of Stratified Tree Search with those of beam search.

6.1.4 Chapter Outline

This chapter is organized as follows. In Section 6.2 we present the "vanilla" STS and show that STS can perform much better than traditional heuristic search algorithm on classical state-space problems such as the Blocks World when only weak heuristics are available. In Section 6.3 we present Beam STS, a variation of STS that uses the ideas of beam search. We also show experimental results comparing Beam STS with our vanilla STS.

6.2 Stratified Tree Search

STS is a heuristic search algorithm that uses the C&C paradigm. STS assumes that nodes of the same type lead to solutions of the same cost, thus only one node of each type must be expanded. The STS algorithm is the application of Chen's ideas [11, 12] for finding suboptimal solution paths for state-space search problems. Recall that Chen's SS uses a type system to efficiently estimate the size of backtrack search trees. Chen assumed that nodes of the same type at a level of the search tree would root subtrees of the same size. Therefore, by sampling only one node of each type SS efficiently estimates the size of a search tree.

Chen also showed that SS can be used to measure any property of backtrack search trees [11]. Therefore, in theory, SS is applicable for finding suboptimal solutions for the least-cost path planning problem. However, Chen assumed trees with bounded depth, and applied SS in domains with high solution density. For instance, Chen used SS for finding an approximation of the longest path

for the uncrossed knight's tour problem. In the uncrossed knight's tour problem a knight is placed on a chess board and the goal is to find the longest tour that a knight in chess can make, with its chess-like moves (L-shaped), without crossing cells already visited. It is easy to obtain some solution in this domain as every leaf node (i.e., nodes not generating children) is a solution.

STS is able to find suboptimal solutions in combinatorial domains with unbounded depth that are much larger than the domains Chen used as testbeds. In experiments on the 35 Pancake puzzle, 20-blocks world, and 24-puzzle SS using the type systems proposed by Chen did not find any solution on 10 random instances of each domain with a three-hour time limit per instance; STS finds near-optimal solutions in seconds in all three domains for all instances tested (see Section 6.2.4 below). The main reason STS works better than SS in these larger domains with lower solution density is because STS uses a heuristic function to define its type system and guide its search.

STS resembles beam search in that it expands nodes in a level by level manner and in that it ignores a set of the nodes at every level of search. However, instead of expanding the B best nodes according to the heuristic, STS expands one node of each type at every level of search. STS assumes that nodes of the same type lead to solutions of the same cost.

Figure 6.1 illustrates how STS compares to Breadth-First Search and beam search. The gray rectangles represent nodes expanded by each search algorithm in different levels of the UST. From Figure 6.1 we note the following.

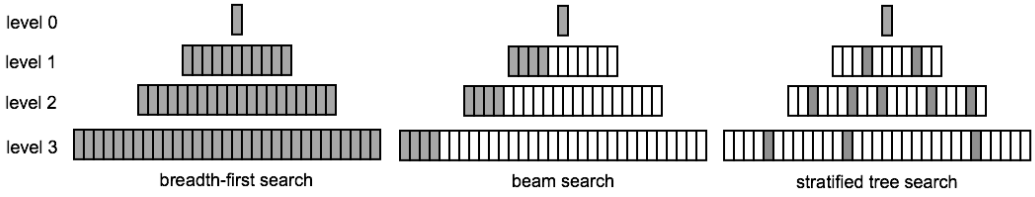

Figure 6.1: Illustration of the differences between Breadth-First Search, beam search and STS.

- Breadth-First Search is not guided by a heuristic function and it explores all nodes at every level of the UST. Being exhaustive prevents Breadth-First Search from scaling to larger domains due to the prohibitively large number of nodes it explores and stores in memory.

- Beam search explores only the B best nodes according to the heuristic function at every level of the search tree. While such a strategy can be quite effective when the heuristic is informative, there are cases, which we discuss later in the chapter, in which the heuristic could be misleading.

Algorithm 15 Stratified Tree Search

Input: start state s, goal state g, and type system T
Output: path from s to g

```
1:  C ← {⟨s, 1⟩}
2:  i ← 0
3:  while true do
4:     empty N
5:     (C, P) ← STS-Expansion(C, i) // see Algorithm 16
6:     if P is not empty then
7:        return P
8:     end if
9:     if C is empty then
10:       return failure
11:    end if
12:    i ← i + 1;
13: end while
```

- STS explores different parts of the UST guided by the type system. In contrast to beam search, STS also expands nodes that do not look promising according to the heuristic function. This strategy can be quite effective when the only heuristics available are too weak to guide traditional heuristic search algorithms.

Algorithms 15 and 16 present STS. The reader will notice that this vanilla version of STS we now present is quite similar to SS, shown in Algorithm 4 in Chapter 3.

STS receives as input a start state s, a goal state g, and a type system T, and it returns a path from s to g. STS assumes that nodes of the same type will lead to solutions of the same cost. Under this assumption, exploring only one node of each type is enough to find the optimal solution path to the goal. In Algorithm 16 STS keeps only two frontiers in memory, one for the current level of search (C), and another one for the next level of search (N). However, in our implementation of STS we keep all the levels in memory so that the path from start to goal can be recovered. STS stores pairs $\langle s, w \rangle$ in the frontiers, where s is a node at a given level of search such that $T(s) = t$, and w is a weight associated with s. Again we call such a pair the representative pair of type t, as STS keeps at most one such pair for every $t \in T$ in each level. The value of w in a representative pair for type t is the estimated number of nodes of type t that exist at that level of search. Such weights were used by SS to estimate the size of a search tree. In SS, the sum of all w values gives an estimate of the size of a search tree. STS uses w to sample the search tree uniformly.

Algorithm 15 starts by inserting the start state with a weight of one into the current frontier C. As in Breadth-First Search, STS iterates through an entire level before moving to the next level of search. In the current implementation of STS we assume edges with unit cost. STS generalizes

Algorithm 16 STS-Expansion

Input: layer C and cost i
Output: a tuple (N, P) with layer N and a solution path P

```
1: for each element ⟨s, w⟩ in C do
2:   for each child ŝ of s do
3:     if ŝ equals g then
4:       return (∅, P), where P is the solution path to ŝ
5:     end if
6:     if N contains an element ⟨s′, w′⟩ with T(s′) = T(ŝ) then
7:       w″ ← w′ + w
8:       with probability w/w″, replace ⟨s′, w′⟩ in N by ⟨ŝ, w″⟩
9:     else
10:      insert new element ⟨ŝ, w⟩ in N.
11:    end if
12:  end for
13: end for
14: return (N, ∅)
```

to domains with non-unit edge costs if instead of iterating over levels STS iterates over layers of nodes of same g-cost. In Algorithm 16, STS selects for expansion a state s in a representative pair $\langle s, w\rangle$ in C. If the goal is found among the children of s, then STS returns the path through s with cost $i + 1$. As we mentioned earlier, although not shown in the pseudocode, we keep in memory all levels expanded by STS so that the solution path can be recovered. If the goal is not found, each child $\hat{s}$ of s is considered for insertion in N. When STS generates a node $\hat{s}$ of type t and there is no representative pair for t in N, then $\hat{s}$ is inserted with the weight of its parent in N. If there is a representative pair $\langle s', w'\rangle$ with $T(\hat{s}) = T(s') = t$ in N, then STS adds w to w' — STS estimated there were w' nodes of type t in N, and now w nodes in C generate a node of type t in N. STS replaces $\langle s', w'\rangle$ with the new representative pair $\langle \hat{s}, w' + w\rangle$ with probability $w/(w' + w)$. After expanding all nodes in representative pairs at a level of search STS starts expanding the next level of search.

Note that every node seen during search may potentially become the node in a representative pair. Nodes that were generated from parents with higher values of w have a higher chance of becoming a representative in exact proportion to their frequency of occurrence. STS samples the search tree more uniformly by weighting nodes in this way.

Multiple Probes. The process just described represents one probe of STS. A probe finishes when STS either finds a goal or reaches leaf nodes. In the latter case, C will be empty and the algorithm will return failure. One could run multiple probes and possibly improve the performance of STS by returning the solution with minimum cost found. An experiment illustrating the tradeoff between solution cost and runtime is discussed later (see Table 6.2).

Transposition Detection. The only kind of transposition detection we implemented in STS was parent-pruning, i.e., STS does not generate node $\hat{s}$ from s if $\hat{s}$ is the parent of s in the search tree. Transpositions can hurt the efficiency of a search algorithm because they could make the algorithm explore the same state multiple times. Later in this chapter we equip STS with a system for detecting transpositions.

Restarts. In some cases STS might be misled by the type system to explore large parts of the state-space that do not contain goal nodes. In this case a probe could take an arbitrarily large amount of time to finish. One can use a restart strategy during a probe to prevent STS from getting stuck in hopeless parts of the state-space. A restart strategy St is defined as $St = \{x_1, x_2, x_3, \dots\}$. A STS probe using St searches for a solution for x_1 time steps before restarting from scratch and searching for a solution for another x_2 time steps; this process goes on until the probe finishes. Luby et al. [63] present a generic restart strategy that can be used in STS's probes. In Section 6.2.2 we assume a restart strategy is being used. However, we did not have to use restarts in our initial set of experiments, STS always quickly found near-optimal solutions in the domains tested. Later in this chapter, when we use STS on domain-independent planning problems we equip STS with a restarting strategy.

6.2.1 Type Systems

In general, a type system can use any information about a node to define its type. For instance, Chen suggests a general type system which counts how many children a node generates. Thus, in this case, two nodes are of the same type if they generate the same number of nodes (this is the type system we used with SS in the experiment mentioned above in which SS did not find any solution with a three-hour time limit). Like we did with SS, TSS, and BiSS in the previous chapters, we include the information provided by a heuristic function in the type systems STS uses. Namely, we use with STS the following type system we used in Chapter 5 with BiSS.

$$T'_c(s) = (h(s), c(s,0), \dots, c(s,H)), \tag{6.1}$$

where $h(s)$ is the heuristic value of node s, $c(s,k)$ is how many of s's children have heuristic value k, and H is the maximum heuristic value a node can assume. We use variants of T'_c in our experiments.

Beam search algorithms explore more nodes per level of search by increasing the value of B. Analogously, STS explores more nodes per level by increasing the size of the type system. One could use the same strategy discussed in Section 3.6.1 for SS in which one increases the size of a

type system T by appending a random integer to the information considered by T. For instance, one could define $T_r(s) = (T'_c(s), \nabla)$, where ∇ is a random integer ranging from 1 to M. Thus, STS using T_r expands at most M different nodes for each type $u \in T'_c$. Larger values of ∇ will make STS explore more nodes during search. We do not use this strategy in our experiments

6.2.2 Theoretical Analysis

We use the definition of an algorithm being *probabilistically approximately complete* [34] to study the asymptotic behavior of STS.

Definition 23. *Let $P_A(s^* \leq x)$ be the probability of search algorithm A finding a path from start state s^* to a goal state in time less than or equal to x. A is called probabilistically approximately complete if, and only if, for any solvable start state s^*, $\lim_{x\to\infty} P_A(s^* \leq x) = 1$.*

Theorem 6. STS *with restarts is probabilistically approximately complete.*

Proof. Every node reachable from s^* has a non-zero probability of being expanded by STS. This is because every state s of a given type u has a non-zero probability of being the representative state of u. Therefore, if the number of allowed restarts is unbounded, $\lim_{x\to\infty} P_{\texttt{STS}}(s^* \leq x) = 1$. □

Similarly, we say that STS is *probabilistically approximately optimal*, i.e., since an optimal path to a goal state has a non-zero probability of being explored, in the limit as the number of probes goes to infinity STS almost surely finds the optimal solution as long as the number of allowed restarts is unbounded.

Memory usage. STS keeps in memory one node for each representative pair in C and N. Thus, the maximum number of nodes stored in a frontier is $|T|$, the size of the type system employed. If d is the depth at which the solution is found, then STS keeps in memory at most $|T| \times d$ nodes.

Runtime behavior. STS generates at most $|T| \times b$ nodes at a level, where b is the branching factor. Again considering that the solution is found at depth d, STS generates at most $|T| \times b \times d$ nodes.

6.2.3 The Quality-Speed Tradeoff

WIDA* and beam search have parameters – the weight w by which WIDA* multiplies the heuristic value and the number of nodes B beam search expands at each level of search – that can be varied to trade off solution quality and speed. Analogous to this, by varying the number of probes p and the size of the type system, STS can trade solution quality for search speed. For instance, STS becomes Breadth-First Search when it uses a type system that assigns a different type to every node at a level

of search. In this case, STS finds optimal solutions in a single probe, but it will not scale to large state spaces due to memory restrictions. The other extreme is when STS assigns every node to the same type. In this case STS becomes a random walk. By varying the size of the type system one gets a "new" search algorithm in the large spectrum of possibilities between Breadth-First Search and random walks. The number of probes used can also give some flexibility on the quality of the solutions found by STS and search speed. By increasing the number of probes and taking the best solution found by them, one will reduce the variance of the quality of the solution found by STS.

Experiments with Blocks World

We illustrate the behavior of STS when varying the size of the type system and the number of probes with experiments on two different state-space representations of the blocks world domain [78]. We use two representations of the blocks world because we are also interested in studying the effect of the solution depth on the suboptimality of STS's solutions. In one representation an action corresponds to the act of moving a block from the top of a stack to the top of another stack (or to the table). In this representation the branching factor is quadratic in the number of blocks. The other representation we use has a robot hand to pick up and put down blocks. In this representation an action corresponds to the act of picking up or putting down a block, thus the branching factor is only linear on the number of blocks. Clearly, the solution length of the representation with the hand is exactly twice as long the solution length without the hand — two actions in one representation correspond to one action in the other. We call the version with the hand the "deep blocks world" (DBW) and the other the "shallow blocks world" (SBW).

In this experiment the results are averaged over 500 random start states with 20 blocks. These instances were solved optimally with a specialized solver by Slaney and Thiébaux [78] so that the suboptimality of STS's solutions could be computed exactly. We compute the suboptimality for a problem instance by dividing the solution cost STS found by the optimal solution cost of that problem instance, subtracting one from the result of the division and then multiplying by 100. The suboptimality we report in our tables of results is the average of the suboptimalities for the set of problem instances (denoted by Sub.). Besides the average suboptimality, we also show the average runtime (Time) in seconds. In addition to the average values we show the standard deviation of the STS results.

We use several variations of T_c. The T_c type system includes the heuristic distribution of the children of a node. In our variations of T_c we limit the number of operators to be applied to a node when computing its type. Type system T_x is the type system that allows x operators. For instance,

Shallow Blocks World ($p = 1$)			Deep Blocks World ($p = 1$)		
Type	Sub. (%)	Time (s.)	Type	Sub. (%)	Time (s.)
T_{100}	4.0 ± 4.6	7.93 ± 0.60	T_{16}	80.8 ± 230.6	0.79 ± 0.28
T_{120}	3.5 ± 4.0	10.22 ± 0.75	T_{17}	25.2 ± 71.6	0.76 ± 0.32
T_{140}	2.9 ± 3.4	12.56 ± 0.84	T_{18}	12.8 ± 24.0	0.80 ± 0.09
T_{160}	2.6 ± 3.4	14.12 ± 0.93	T_{19}	7.8 ± 15.4	0.83 ± 0.09
T_{180}	2.4 ± 3.3	14.91 ± 1.01	T'_c	5.0 ± 3.7	0.80 ± 0.07

Table 6.1: STS with different type systems.

Shallow Blocks World (T_{140})			Deep Blocks World (T_{18})		
p	Sub. (%)	Time (s.)	p	Sub. (%)	Time (s.)
1	2.9 ± 3.4	12.56 ± 0.84	1	12.8 ± 24.0	0.80 ± 0.09
2	2.0 ± 2.8	25.34 ± 1.61	2	8.7 ± 20.0	1.54 ± 0.14
3	1.8 ± 2.6	38.13 ± 2.43	3	7.1 ± 12.8	2.30 ± 0.29
4	1.4 ± 2.4	50.51 ± 3.16	4	5.8 ± 9.7	3.05 ± 0.30
5	1.1 ± 2.1	63.43 ± 3.90	5	4.3 ± 5.5	3.76 ± 0.29

Table 6.2: STS with different number of probes.

if a node s normally generates 200 children and we are using a type system T_{50} that limits the number of operators to 50, then when computing $T_{50}(s)$ we only use the heuristic value of the 50 nodes generated by applying the first 50 operators applicable to s. Type systems that allow more operators to be applied tend to be larger than type systems that allow fewer operators as they take into account more information about the neighborhood of a node. Therefore, by varying the number of allowed operators we control the size of a T_c type system. The tradeoff is clearly observed in the experiments on the SBW shown on the left part of Table 6.1: as the size of the type system increases the suboptimality decreases and the runtime increases. This tradeoff is not guaranteed to occur in general, as demonstrated by the results on the DBW, shown on the right part of Table 6.1. Although T'_c is larger than T_{18}, STS finds solutions in 0.8 seconds on average when using either type system. This fact is explained by the suboptimality of the solutions. STS expands more nodes per level when using T'_c because T'_c is larger than T_{18}. However, STS searches deeper when using T_{18} because STS finds longer solutions: solutions 12.8% longer than optimal when using T_{18} compared to solutions 5% longer than optimal when using T'_c. The nodes expanded by STS while searching deeper in the search tree compensate for the fewer nodes expanded per level when using T_{18}.

Table 6.2 shows experiments when the type system is fixed (T_{140} for the SBW and T_{18} for the DBW) and we vary the number of probes. As p increases, the total runtime obviously must increase and the suboptimality will usually decrease. As when varying the size of a type system, it is not always true that increasing the number of probes will decrease suboptimality.

At first glance STS seems to perform better on the DBW than on the SBW. For instance, com-

paring T_{100} of the SBW with T'_c of the DBW we note that STS produces solutions of only slightly higher suboptimality and it is almost ten times faster on the latter. However, when we increased the number of probes on the DBW so that the solutions were of equal suboptimality in both representations, we noticed that STS could be slower on the DBW. For instance, we slowly increased the number of probes for STS using T_{16} on the DBW and we noted that STS eventually became slower than STS using T_{100} on the SBW and still produced solutions of higher suboptimality. In particular, STS with 20 probes using T_{16} on the DBW produces solutions $6.6\% \pm 12.9\%$ longer than optimal in 11.71s. $\pm$ 3.04s. We conjecture that STS tends to find solutions with higher suboptimality in domains with deeper solutions.

Here is an observation that supports our conjecture. STS seems to be more sensitive to variations in the size of the type system on the DBW. When using T_{16} STS finds solutions 80% longer than optimal on average, but when using the larger T'_c the solutions are only 5% longer than optimal on average. This phenomenon is not observed on the results on the SBW: the average suboptimality does not change dramatically when varying the size of the type system. One possible explanation for this phenomenon is the following. Let q be the probability of STS selecting the node on an optimal path at a given level of search and d be the optimal solution length. Then, q^d is the probability of STS finding the optimal path in a probe. If q drops from 0.995 to 0.990, say, by decreasing the size of the type system, then the probability of STS finding the optimal solution on the SBW drops from 0.86 to 0.73 — a decrease of 13% — considering solutions with 30 moves. With the same change in q, the probability of STS finding the optimal solution on the DBW drops from 0.74 to 0.54 — a decrease of 20% — considering solutions with 60 moves. STS is more sensitive to changes in q for larger values of d. For simplicity in this illustrative example we assumed q does not correlate with the problem's branching factor. However, the value of q will tend to be lower in problems with larger branching factor.

Note that depending on the values of q and d it might take a prohibitively large number of probes to reduce the suboptimality of STS's solutions to an acceptable value.

6.2.4 Empirical Comparison to Other Heuristic Search Algorithms

While comparing STS to other heuristic search algorithms we emulate an environment in which only weak heuristics are available. However, at the same time, we wanted to test STS in domains for which optimal solutions could be obtained, either by search algorithms using strong heuristics or by specialized solvers, so that suboptimality could be computed exactly.

The experiments described in this section are run on the 20-SBW (SBW with 20 blocks), on the

35 Pancake puzzle, and on the 24-puzzle.

20-SBW

The results are averaged over 50 random start states. We compute the suboptimality of STS's solutions by optimally solving the problems with a specialized solver [78]. We use the T_{180} type system explained above in this domain. The weak heuristic function we used for STS in this experiment is the number of blocks out of place.

35 Pancake Puzzle

We used 50 random instances solved optimally with IDA* using the hand-crafted GAP heuristic [28] as our test set. The heuristic we used for the search algorithms in this experiment was the maximum of the dual lookup [85] of a set of seven 5-pancake PDBs. We used T_7 as the type system. If using the heuristic values of the 7 children separately in the type system STS would find paths with lower suboptimality, but it would require more time to search. Thus, the result would not be comparable to other heuristic search algorithms. Instead of using the heuristic value of each child separately as in T'_c, we use the sum of the heuristic value of all 7 children to summarize the information.

24-Puzzle

We used the standard 50 instances solved optimally by Korf and Felner [42] for testing. Five 4-tile PDBs, and Manhattan Distance (MD) were used as heuristic functions in this domain. The heuristic we used was the sum of all PDBs and MD. We used the T'_c type system for STS. In addition to the information contained in T'_c, when computing the type of node s we also included the kind of location of the blank (the blank can occupy one of the following kinds of positions: corner, edge, or middle) of the parent of s and of s itself. For WIDA* we used the maximum of the PDBs and MD – when using the sum of the heuristics as we did for BULB and STS, WIDA* (or even IDA*) found very suboptimal and thus not comparable solutions.

These three domains offer a good challenge to STS because the UST of all three domains do not contain leaf nodes, i.e., nodes that do not generate children, thus an STS probe will only finish if it reaches the goal.

Experimental Setup

We compare STS to WIDA* and BULB [21], a variation of beam search with limited discrepancy backtracking, and also to IDA*-BST, which is IDA* using the strong inadmissible bootstrap heuristic [35]. All our experiments were run on a 2.6 GHz machine and all four algorithms were run on

the same test instances. The maximum number of nodes stored by BULB was set to 6 million nodes in all experiments.

In our experiments we chose the type system and the number of probes used by STS, the value of w used by WIDA*, and the value of B used by BULB so that the results were comparable. A result is comparable either if two algorithms are similar in one of the dimensions (suboptimality or runtime) and different in the other dimension, or if one algorithm outperforms the other in both dimensions. The procedure we used to find the values of w, B, the number of operators considered by the T_c' type system, and the number of probes p was to set them initially to some value and slowly vary them until the results were comparable. We state the values of w, B, and p used in each domain in the table of results below. In our table of results we highlight the best results in bold. For IDA*-BST we used 5,000 bootstrap instances and used the last heuristic produced by the method after days of bootstrap learning.

Experimental Results

Table 6.3 presents the results. We do not present the number of nodes generated because they are not comparable across different algorithms. For instance, while STS generates many fewer nodes than any other algorithm in all domains tested, STS is not necessarily the fastest one, as shown by the results on the 24-puzzle. This is because the node generation in STS is more expensive than the node generation in the other algorithms because of the type computation.

Our empirical results show that (1) on the 20-SBW and on the 35-pancake STS substantially outperforms WIDA*, BULB, and IDA*-BST; (2) on the 24-puzzle, STS outperforms WIDA*, but is outperformed by BULB and IDA*-BST. WIDA* does not have entries for the 20-SBW because it was not able to solve any problem instance with a time limit of more than one day per instance (we tried the following values of w: 1.0, 1.5, 2.0, $\cdots$, 10).

	35-Pancake Puzzle			20-Shallow Blocks World			24-puzzle		
Algorithm	Parameter	Sub. (%)	Time (s.)	Parameter	Sub. (%)	Time (s.)	Parameter	Sub. (%)	Time (s.)
WIDA*	w=1.4	12.7	11.42	-	-	-	w=1.9	34.2	253.30
BULB	B=5,000	16.5	34.26	B=5,000	86.0	31.79	B=12,000	**7.5**	16.02
IDA*-BST	-	15.4	21	-	9.6	23.00	-	8.1	**9.65**
STS	p=4	**8.8 $\pm$ 3.7**	**5.22 $\pm$ 0.26**	p=4	**2.4 $\pm$ 3.3**	**14.91 $\pm$ 1.01**	p=1	27.6 $\pm$ 9.2	15.58 $\pm$ 2.45

Table 6.3: Comparison of STS with some traditional heuristic search algorithms.

Our experiments show that STS can be substantially better than other heuristic search algorithms. For instance, on the 20-SBW STS is so accurate that 70% of the test instances were solved optimally. STS is also accurate on the 35-Pancake puzzle and it is substantially better than its competitors. However, STS is outperformed on the 24-puzzle by BULB and IDA*-BST. The 24-puzzle

is a domain with deep solutions (average solution depth of approximately 100). As we have conjectured, STS is more likely to find lower quality solutions in domains with large solution length. Nevertheless, STS is better than WIDA* on the 24-puzzle.

Although WIDA* performs worse than STS in all three domains and worse than BULB in two of the domains, WIDA* solves a harder problem as it finds solutions with bounded suboptimality. BULB and IDA*-BST do not have quality bounds on their solutions. STS only guarantees optimality in the limit as the time allowed for searching goes to infinity.

We have observed in experiments not shown in Table 6.3 that STS performs better when employing strong heuristics to define its type system. However, STS is certainly not the algorithm of choice when strong heuristics are available. For instance, in an experiment we performed on the pancake puzzle using the strong, hand-crafted GAP heuristic [28], WIDA* was always better than STS. A search algorithm should concentrate its search effort on nodes with lower heuristic value when using strong heuristics. STS ignores the fact that the heuristic might be strong and it distributes its search effort equally among nodes with different heuristic values.

6.3 Beam Stratified Tree Search

STS usually offers a good tradeoff between solution quality and search speed by varying the size of the type system and the number of probes, as discussed in Section 6.2.3. However, in some cases, varying the number of operators available when computing a node's T'_c type might not be enough to get the desirable tradeoff between solution quality and search speed. For example, one can imagine a domain in which for a fixed number of probes STS could find solutions of low quality with a T_5 type system and be too slow to be practical with a T_6 type system.

We present Beam STS (BSTS), a variant of STS that allows one to make fine adjustments on the tradeoff between solution quality and search speed. Similar to beam search, BSTS only expands B nodes at every level of search. However, instead of expanding the best B nodes according to the heuristic, BSTS expands the representative node of the "best" B types at every level of search; the other representative nodes are pruned. We use the heuristic to define the set of best B types. Let T be a T'_c type system and n and n' be nodes in a search tree. We say that type $T(n)$ is better than a type $T(n')$ if n generates a child that has a heuristic value lower than the heuristic value of any children of n'. Intuitively, $T(n)$ is better than $T(n')$ if n generates a child that is more promising than any children generated by n' according to the heuristic function. We break ties arbitrarily when selecting the best B types at a level of search.

In contrast to STS, BSTS has the advantage that one can specify exactly the maximum number of nodes expanded at every level of search. On the downside, although BSTS still has the diversity of STS, it relies on the ranking provided by the heuristic function to select the best B types. However, our empirical results show that, due to the diversity of nodes expanded, BSTS does not suffer from the inaccuracy of the heuristic function as traditional beam search does. In fact, BSTS and beam search can also have almost opposite behaviors: all nodes expanded by beam search could have similar heuristic value whereas the nodes expanded by BSTS could have different heuristic values.

The beam strategy in BSTS allows one to adjust how much the algorithm should trust the heuristic. For lower values of B BSTS trusts more the heuristic, and thus exploits the information provided by the heuristic; for higher values of B the algorithm expands nodes with higher heuristic value, that do not look promising according to the heuristic, and thus the algorithm explores the state space.

Another approach one could use to control the number STS's node expansions is to use the clustering-based type systems introduced in Chapter 3. Note that BSTS and STS using clustering-based type systems could have different search strategies. For lower values of B BSTS will tend to trust more the information provided by the heuristic function. Clustering-based type systems will distribute STS's search effort equally among nodes with different heuristic values.

6.3.1 Empirical Evaluation of BSTS

We verify the effectiveness of BSTS empirically by testing it with different values of p and B on the same domains we tested STS on. We use the same heuristic functions used with STS. We use the T_c' type system for both 35-Pancake Puzzle and 20-Shallow Blocks World.

For the 24-puzzle we use the T_{ggc} type system. In addition to the information that the T_c' type system takes into account, when computing the type of node n, the T_{ggc} type system also accounts for the heuristic distribution and the number of grandchildren and great-grandchildren n generates. In the case of the T_{ggc} type system, we say that type $T(n)$ is better than a type $T(n')$ if n generates a great-grandchild that has a heuristic value lower than the heuristic value of any of the great-grandchildren of n'. STS would find solutions very close to optimal, but it would be too slow to be practical if using the type systems we use with BSTS in this experiment. BSTS is able to find solutions quickly while using these type systems because the number of types expanded at every level is bounded by the input parameter B.

The results of BSTS are shown in Table 6.4. For the values of B reported in Table 6.4 BSTS always quickly found a path to the goal. As in the STS experiments, we did not use a restart strategy in this experiment. We observe in Table 6.4 that in all three domains as we increase the

		35-Pancake Puzzle		20-Shallow Blocks World		24-puzzle	
p	B	Sub. (%)	Time (s.)	Sub. (%)	Time (s.)	Sub. (%)	Time (s.)
1	200	20.3 ± 6.9	1.83 ± 0.13	1.3 ± 2.3	3.15 ± 0.74	55.5 ± 30.2	1.67 ± 0.28
1	300	18.4 ± 5.6	2.53 ± 0.17	1.3 ± 2.0	4.69 ± 1.00	44.7 ± 20.9	2.63 ± 0.39
1	400	16.9 ± 5.1	3.27 ± 0.18	1.3 ± 1.9	6.28 ± 1.26	37.7 ± 14.3	3.11 ± 0.37
1	500	16.3 ± 4.7	3.81 ± 0.24	1.2 ± 2.1	7.98 ± 1.43	43.0 ± 16.4	4.14 ± 0.46
1	600	15.7 ± 6.1	4.53 ± 0.26	1.2 ± 1.9	9.39 ± 1.59	38.8 ± 14.7	4.83 ± 0.53
1	700	14.4 ± 4.5	5.10 ± 0.26	1.2 ± 2.2	10.98 ± 1.66	35.0 ± 13.2	5.52 ± 0.61
1	800	13.8 ± 4.9	5.91 ± 0.34	1.2 ± 2.0	12.63 ± 1.62	39.2 ± 21.8	6.64 ± 1.17
1	900	14.6 ± 5.5	5.80 ± 0.32	1.5 ± 2.7	14.45 ± 1.66	35.0 ± 14.5	7.58 ± 0.88
1	1000	13.3 ± 4.2	7.02 ± 0.34	1.1 ± 2.0	15.86 ± 1.65	33.2 ± 13.1	8.38 ± 0.92
5	200	12.8 ± 4.1	8.84 ± 0.45	0.4 ± 1.3	16.11 ± 3.72	29.4 ± 8.9	8.12 ± 0.79
5	300	11.3 ± 3.3	13.01 ± 0.57	0.3 ± 1.2	23.89 ± 5.05	28.8 ± 9.0	12.27 ± 1.36
5	400	10.5 ± 3.3	16.54 ± 0.58	0.6 ± 1.6	30.93 ± 6.14	24.8 ± 7.4	16.53 ± 1.56
5	500	9.9 ± 3.3	19.34 ± 0.67	0.4 ± 1.2	38.71 ± 6.96	24.5 ± 8.9	20.07 ± 2.01
5	600	9.5 ± 3.3	22.62 ± 0.79	0.4 ± 1.3	46.73 ± 7.73	24.6 ± 8.0	24.16 ± 2.29
5	700	8.9 ± 3.7	26.36 ± 0.85	0.3 ± 1.0	54.42 ± 8.06	23.4 ± 8.7	29.29 ± 2.91
5	800	8.6 ± 3.0	29.86 ± 1.56	0.4 ± 1.0	63.45 ± 8.23	24.3 ± 7.7	32.96 ± 2.93
5	900	8.8 ± 3.4	33.05 ± 1.21	0.3 ± 0.9	70.46 ± 8.27	24.6 ± 7.9	37.15 ± 3.45
5	1000	8.3 ± 3.5	34.76 ± 1.22	0.4 ± 1.1	80.71 ± 8.63	23.5 ± 8.1	40.53 ± 4.02

Table 6.4: Beam STS with different values of p and B.

number of probes p and the beam size B the suboptimality decreases and the runtime increases. The high suboptimality values for lower values of B suggest that a node n might be worth expanding even when n does not generate descendants that are considered promising according to the heuristic function. These results reinforce the principle of STS for diversifying search.

Most of the entries in Table 6.4 are not comparable to those in Table 6.3 in the sense that there is no clear winner in terms of both runtime and suboptimality. However, we observe that in a few cases for the 24-puzzle and for the 20-SBW, BSTS is better than STS. For instance, on the 24-puzzle, BSTS with $p = 5$ and $B = 400$ finds solutions of lower suboptimality quicker than STS. On the 20-SBW BSTS is better than STS for any value of B lower than 1,000 when $p = 1$.

BSTS is competitive with STS and it is substantially better than BULB (the beam search algorithm used in our experiments). The most impressive result is on the 20-SBW for $p = 1$ and $B = 200$: BSTS is 10 times faster and finds solutions of 66 times better quality than BULB.

The 35-Pancake Puzzle is the only domain tested in which BSTS is worse than STS. BSTS and STS expand approximately the same number of nodes per level to produce solutions 8.8% longer than optimal on the 35-Pancake Puzzle – 700 nodes per level (information not shown in the tables). The time required to compute the type of a node is reduced by reducing the number of operators available as fewer nodes are expanded per type computation. When computing a node's T'_c type one has to expand 34 nodes (branching factor of the 35-Pancake Puzzle). However, if using, say, the T_5

type system, then only 5 nodes have to be expanded for each type computation. The T_5 represents a saving of 31 node expansions per type computation over T_c'. Therefore, because BSTS and STS expand the same number of nodes per level and the type computation of BSTS is slower than that of STS, BSTS tends to be slower than STS.

Note that BSTS and STS could have been given the same type system in this experiment; in this case the time required to compute a node's type would be the same for both algorithms. However, that was not our goal in this experiment. We wanted to use BSTS as a way of making fine adjustments to the tradeoff between solution quality and search speed without restricting the number of operators available when computing a node's type. Our results showed that BSTS can be quite effective.

6.4 Conclusion

In this chapter we presented STS and BSTS, two C&C suboptimal heuristic search algorithms that use a different search strategy than traditional heuristic search methods. STS and BSTS use a partition of the nodes in the search tree through a type system to guide their search, which is in contrast with the estimated cost-to-go used by traditional heuristic search algorithms. Our algorithms assume that nodes of the same type will lead to solutions of the same cost. Empirical results showed that this strategy can be effective even when the only heuristics available are too weak to guide traditional heuristic search algorithms.

Chapter 7

Conclusion

In this chapter we recap the contributions presented in this dissertation. We also point out the weaknesses of our C&C methods which reflect in possibilities of future work.

7.1 Contributions

In this dissertation we showed that the C&C paradigm is general and can be used to solve different problems. In particular, we presented C&C solutions for four problems: (i) IDA* tree size prediction; (ii) DFBnB tree size prediction; (iii) optimal solution cost prediction (iv) suboptimal state-space path-planning.

As we explained in Chapter 1, C&C algorithms are divided into two parts: the cluster part, which is defined by a type system, and a conquer part. In this dissertation we introduced several effective variations of heuristic-based type systems. However, our major contributions were in conquer part of the C&C paradigm by designing algorithms that employ type systems. In fact, it is strikingly interesting that the C&C algorithms we studied and introduced use essentially the same cluster strategy, i.e., type system, to solve different problems. CDP, SS, TSS, SCP, BiSS and STS all use variations of type systems based on heuristic functions.

Here is a summary of our contributions.

7.1.1 Predicting the size of the IDA* search tree

In Chapter 3 we connected and advanced CDP and SS. We made the following contributions.

- For the CDP algorithm we presented:
  - ϵ-truncation, a method for improving the accuracy of the CDP predictions that works by carefully ignoring rare events observed during sampling.

  - Lookup CDP, a method for speeding up the CDP predictions by precomputing and storing in a lookup table the different possible prediction outcomes according to the different types in the type space.

- For the SS algorithm we showed:

  - that the type systems developed for CDP can also be applied to SS to improve its predictions.

  - how to use active sampling [18] in SS to improve the accuracy of its predictions.

- We also showed how to use clustering through the k-means algorithm for controlling the size of heuristic-based type systems in domains with real-valued heuristic functions.

- Our empirical results showed that our enhanced versions of CDP and SS represent the current state of the art for predicting the size of IDA* search trees.

7.1.2 Predicting the Search Tree Size of Depth-First Branch and Bound

In Chapter 4 we used the C&C paradigm for developing an algorithm for predicting the DFBnB search tree size.

- As anticipated by Knuth, we showed that the methods developed for predicting the IDA* search tree size do not produce good predictions of the DFBnB search tree size.

- We introduced TSS, a C&C algorithm that adapts SS for predicting the DFBnB search tree size.

- Our empirical results on optimization queries over graphical models showed that TSS is the only method able to produce accurate predictions.

7.1.3 Predicting the Optimal Solution Cost of State-Space Search Problems

In Chapter 5 we presented C&C algorithms for predicting the optimal solution cost of state-space search problems. We made the following contributions.

- We presented SCP, a C&C prediction method based on CDP, able to produce fairly accurate predictions of the optimal solution cost of state-space search problems.

- We presented BiSS, a C&C bidirectional variation of SS that produces good predictions of the optimal solution cost of state-space search problems.

- We showed empirically that BiSS scales to state spaces much larger than can be solved optimally. In particular, we predicted the average solution cost for the sliding-tile puzzles up to the 8x8 configuration, which has more than 10^{88} reachable states, and provided indirect evidence that BiSS's predictions for these huge state spaces are accurate.
- We showed how to learn strong heuristics from predictions produced by BiSS. We showed that it is possible to reduce the time required to learn strong heuristic functions from days to minutes by using BiSS's predictions to generate the training set.

7.1.4 Finding Suboptimal Solutions for State-Space Search Problems

In Chapter 6 we presented STS, a C&C search algorithm for finding suboptimal solutions for state-space search problems in cases where only weak heuristic functions are available. We made the following contributions.

- We empirically compared STS to standard search algorithms in problems with fairly large state spaces. Overall STS performed better than other search algorithms.
- STS has input parameters that offer a good tradeoff between solution quality and search speed. However, in some cases, STS does not provide a fine adjustment of this tradeoff. We presented a simple variant of STS, BSTS, that allows one to make fine adjustments of the tradeoff between solution quality and search speed.

7.2 Weaknesses of Our C&C Methods

Here is a list of the main weaknesses of the C&C methods discussed in this dissertation.

7.2.1 CDP and SS for Predicting the Size of IDA* Search Trees

Solving a problem instance. Both CDP and SS were designed for predicting the number of nodes expanded on an iteration of IDA* with cost bound d. Note, however, that we are not able to predict the number of nodes expanded by IDA* to solve a particular problem instance x. This is because we do not know a priori the optimal solution cost c^* for solving x. We were unable to produce good predictions of the number of nodes expanded by IDA* for solving problem instances of the 15-puzzle when using BiSS to estimate the value of c^* and SS to estimate the tree size. Although the predictions produced by BiSS are accurate for the purpose of learning heuristics, because the IDA* search tree grows exponentially with the solution cost, small prediction errors on the solution

cost prediction could result in a large difference between the predicted number of nodes expanded and the actual number of nodes expanded by IDA* for solving a particular problem instance.

This is a weakness that dates from the work by Korf et al. [50] as they limited themselves to predicting the number of nodes expanded by IDA* for a given cost d. Predicting the number of nodes expanded by IDA* for solving a given problem instance remains as an open problem.

Transposition detection. Another weakness of CDP and SS is that they assume IDA* does not detect transpositions. However, IDA* could be implemented with tables for transposition detection [73]. It is not clear how to adapt the prediction methods to account for transpositions.

Sampling strategy of CDP. CDP samples the state space as a preprocessing step. We showed that this allows the CDP predictions to be precomputed and stored in a lookup table, resulting in very fast predictions later on. On the down side, this sampling strategy assumes that different search trees will have similar structure. One could imagine state spaces in which different search trees could have completely different structures; we conjecture that CDP would produce inaccurate predictions in such cases. In fact, we observed in our experiments that CDP does not produce predictions as accurate as SS does in some of the domains tested. We believe this difference in accuracy is related to the different sampling strategies adopted by the two algorithms.

7.2.2 TSS for Predicting the Size of DFBnB Search Tree

Memory Requirement. TSS is guaranteed to produce perfect predictions of the DFBnB tree size in the limit, as the number of probes goes to infinity. However, in order to have this guarantee, TSS has to store in memory the union of the sampled trees expanded across different probes. This union of the sampled trees could require a prohibitive amount of memory in some cases.

Strong C&C assumption. In contrast with CDP and SS that assume that nodes of the same type root subtrees of the same size, TSS assumes that nodes of the same type root isomorphic subtrees. This is because TSS is interested not only in the size of the search tree, but also in the distribution of solution costs in a given subtree. Clearly the assumption TSS makes is much stronger than the assumption CDP and SS make. This partially explains why we are able to produce more accurate predictions of the IDA* search tree size than the DFBnB search tree size. On the positive side, in contrast with CDP and SS that predict the IDA* search tree size for a given cost bound d, TSS produces predictions of the DFBnB search tree size to solve a particular problem instance.

7.2.3 SCP and BiSS for Predicting the Optimal Solution Cost

Quality Bounds. Neither SCP nor BiSS provide bounds on the quality of their predictions. Quality bounds could be critical in real-world applications in which one needs to know how inaccurate the predictions can be. Currently one has to trust the empirical accuracy of the prediction methods.

Accuracy and Scalability of SCP. SCP also suffers from the sampling strategy inherited from CDP. Namely, it samples the state space in a preprocessing step and it assumes the information learned during sampling will be helpful across different problem instances. We conjecture this assumption does not hold for domains with irregular structure such as the blocks world. In contrast with the sliding-tile puzzle and the pancake puzzle, the blocks world's branching factor could change substantially in different parts of the state space. Our empirical results showed that SCP does not produce accurate predictions for instances of the blocks world. Moreover, SCP's sampling process is time-consuming and it could take a prohibitive amount of time to sample a sufficiently large number of states in domains with large state spaces such as the 24-puzzle.

Parameter selection of BiSS. Although we showed that BiSS is able to produce accurate predictions in different problem-domains while using the same set of input parameters, one could imagine a domain in which BiSS would not produce good predictions using our suggested set of parameters.

7.2.4 STS for Finding Suboptimal Solutions

Optimal solutions and quality bounds. STS does not find optimal solutions. Moreover, although STS found near-optimal solutions in the domains we tested, it does not provide quality bounds for the solutions found.

Strong heuristics. STS is not the algorithm of choice when strong heuristics are available, although it also benefits from strong heuristics. This is because STS expands both nodes that look promising according to the heuristic and nodes that do not look promising according to the heuristic. If a strong heuristic is available the search algorithm should exploit the information provided by the heuristic.

7.3 Final Thoughts

In this dissertation we showed that the Cluster-and-Conquer paradigm for solving state-space problems is general and can be applied to a variety of such problems.

Our modified versions of CDP and SS represent the current state of the art to predict the size

of the IDA* search tree. TSS is the only known algorithm able to produce good predictions of the size of the DFBnB search tree in a reasonable amount of time. In fact, since the seminal work of Knuth [38] on predicting the search tree size, we believe we were the first to make a successful step towards adapting his approach to produce accurate predictions of the size of the DFBnB search tree. SCP and BiSS were the first algorithms designed to accurately predict the optimal solution cost of state-space problems. We also showed that we can reduce the time required for learning heuristic functions from days to minutes by using BiSS to label the training set. Finally, STS has shown to be a promising alternative for state-space search when the only heuristics available are too weak to effectively guide existing algorithms.

We hope the ideas presented in this dissertation will inspire others to use the C&C paradigm to design effective algorithms for solving other interesting computational problems.

Bibliography

[1] J. A. Allen and S. Minton. Selecting the right heuristic algorithm: Runtime performance predictors. In *Proceedings of the Canadian Conference on Artificial Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 1996.

[2] András Antos, Varun Grover, and Csaba Szepesvári. Active learning in multi-armed bandits. In *Proceedings of the Conference on Algorithmic Learning Theory*, volume 5254 of *Lecture Notes in Artificial Intelligence*, pages 287–302. Springer, 2008.

[3] A. F. Archer. A modern treatment of the 15-puzzle. *American Mathematical Monthly*, 106:793–799, 1999.

[4] E. Balas and P. Toth. Branch and bound methods. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kart, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, 1985.

[5] R. Bisiani. *Encyclopedia of Artificial Intelligence*, chapter Beam Search, pages 56–58. Wiley & Sons, 1987.

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

[7] J. L. Bresina, M. Drummond, and K. Swanson. Expected solution quality. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1583–1591, 1995.

[8] T. Breyer and R. Korf. Recent results in analyzing the performance of heuristic search. In *In Proceedings of the First International Workshop on Search in Artificial Intelligence and Robotics*, page 2431, 2008.

[9] N. Burch, R. C. Holte, M. Müller, D. O'Connell, and J. Schaeffer. Automating layouts of sewers in subdivisions. In *Proceedings of the European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 655–660. IOS Press, 2010.

[10] E. Burns and W. Ruml. Iterative-deepening search with on-line tree size prediction. In *Proceedings of the International Conference on Learning and Intelligent Optimization*, pages 1–15, 2012.

[11] P.-C. Chen. *Heuristic Sampling on Backtrack Trees*. PhD thesis, Stanford University, 1989.

[12] P.-C. Chen. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[13] J. C. Culberson and J. Schaeffer. Searching with pattern databases. In *Proceedings of the Canadian Conference on Artificial Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 1996.

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

[15] H. T. Dinh, A. Russell, and Y. Su. On the value of good advice: The complexity of A* search with accurate heuristics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1140–1145. AAAI Press, 2007.

[16] H. Dweighter. Problem E2569. *American Mathematical Monthly*, 82:1010, 1975.

[17] M. Ernandes and M. Gori. Likely-admissible and sub-symbolic heuristics. In *Proceedings of the European Conference on Artificial Intelligence*, pages 613–617, 2004.

[18] P. Etore and B. Jourdain. Adaptive optimal allocation in stratified sampling methods. *Methodology and Computing in Applied Probability*, 12(3):335–360, 2010.

[19] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[20] A. Felner, U. Zahavi, J. Schaeffer, and R. C. Holte. Dual lookups in pattern databases. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 103–108, 2005.

[21] D. Furcy and S. Koenig. Limited discrepancy beam search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 125–131, 2005.

[22] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, 1979.

[23] V. Grover. Active learning and its application to heteroscedastic problems. Master's thesis, University of Alberta, 2008.

[24] S. Haim and T. Walsh. Online estimation of SAT solving runtime. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 4996 of *Lecture Notes in Computer Science*, pages 133–138. Springer, 2008.

[25] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–613, 1995.

[26] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1007–1012. AAAI Press, 2007.

[27] A. Heifets and I. Jurisica. Construction of new medicines via game proof search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1564–1570. AAAI Press, 2012.

[28] M. Helmert. Landmark heuristics for the pancake problem. In *Proceedings of the Symposium on Combinatorial Search*, pages 109–110. AAAI Press, 2010.

[29] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

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

[31] R. C. Holte. Common misconceptions concerning heuristic search. In *Proceedings of the Symposium on Combinatorial Search*, pages 46–51. AAAI Press, 2010.

[32] R. C. Holte, J. Grajkowski, and B. Tanner. Hierarchical heuristic search revisited. In Jean-Daniel Zucker and Lorenza Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium*, volume 3607 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2005.

[33] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 530–535, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[34] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[35] S. Jabbari Arfaee, S. Zilles, and R. C. Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[36] K. Kask and R. Dechter. A general scheme for automatic search heuristics from specification dependencies. *Artificial Intelligence*, pages 91–131, 2001.

[37] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1014–1019. AAAI Press, 2006.

[38] D. E. Knuth. Estimating the efficiency of backtrack programs. *Math. Comp.*, 29:121–136, 1975.

[39] R. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[40] R. Korf. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 910–916. AAAI Press, 2000.

[41] R. Korf. A new algorithm for optimal bin packing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 731–736. AAAI Press, 2002.

[42] R. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22.

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

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

[45] R. E. Korf. Linear-space best-first search: Summary of results. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 588–588. AAAI Press, 1992.

[46] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 700–705. AAAI Press, 1997.

[47] R. E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM*, 55(6):26:1–26:40, 2008.

[48] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[49] R. E. Korf and A. Felner. Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2324–2329, 2007.

[50] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence*, 129(1-2):199–218, 2001.

[51] B. J. Larsen, E. Burns, W. Ruml, and R. Holte. Searching without a heuristic: Efficient use of abstraction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 114–120. AAAI Press, 2010.

[52] L. Lelis, R. Stern, A. Felner, S. Zilles, and R. C. Holte. Predicting optimal solution cost with bidirectional stratified sampling. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 155–163. AAAI Press, 2012.

[53] L. Lelis, R. Stern, and S. Jabbari Arfaee. Predicting solution cost with condiditional probabilities. In *Proceedings of the Symposium on Combinatorial Search*, pages 100–107. AAAI Press, 2011.

[54] L. Lelis, S. Zilles, and R. C. Holte. Improved prediction of IDA*'s performance via ϵ-truncation. In *Proceedings of the Symposium on Combinatorial Search*, pages 108–116. AAAI Press, 2011.

[55] L. H. S. Lelis. Active stratified sampling with clustering-based type systems for predicting the search tree size of problems with real-valued heuristics. In *Proceedings of the Symposium on Combinatorial Search*, pages 123–131. AAAI Press, 2013.

[56] L. H. S. Lelis, L. Otten, and R. Dechter. Predicting the size of depth-first branch and bound search trees. In *International Joint Conference on Artificial Intelligence*, page to appear, 2013.

[57] L. H. S. Lelis, S. Zilles, and R. C. Holte. Fast and accurate predictions of IDA*'s performance. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 514–520. AAAI Press, 2012.

[58] L. H. S. Lelis, S. Zilles, and R. C. Holte. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence*, pages 53–76, 2013.

[59] L. H. S. Lelis, S. Zilles, and R. C. Holte. Stratified Tree Search: a novel suboptimal heuristic search algorithm. In *Proceedings of the Conference on Autonomous Agents and Multiagent Systems*, pages 555–562, 2013.

[60] L. H. S. Lelis, S. Jabbari Arfaee S. Zilles, and R. C. Holte. Learning heuristic functions faster by using predicted solution costs. In *Proceedings of the Symposium on Combinatorial Search*, pages 166–167. AAAI Press, 2012.

[61] Y. Li, J. J. Harms, and R. Holte. IDA* MCSP: a fast exact MCSP algorithm. In *ICC*, pages 93–99. IEEE Press, 2005.

[62] L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 353–358. AAAI Press, 1998.

[63] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *IPL: Information Processing Letters*, 47, 1993.

[64] J. B. McQueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[65] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[66] L. Otten and R. Dechter. A case study in complexity estimation: Towards parallel branch-and-bound over graphical models. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 665–674, 2012.

[67] I. Parberry. A real-time algorithm for the (n^2-1)-puzzle. *Information Processing Letters*, 56(1):23–28, 1995.

[68] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley, 1984.

[69] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[70] I. Pohl. Practical and theoretical considerations in heuristic search algorithms. *Machine Intelligence*, 8:55–72, 1977.

[71] A. E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12(1–3):117–141, August 1993.

[72] P. W. Purdom. Tree size by partial backtracking. *SIAM Journal of Computing*, 7(4):481–491, 1978.

[73] A. Reinefeld and T. A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[74] S. Richter and M. Helmert. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 273–280, 2009.

[75] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 975–982. AAAI Press, 2008.

[76] M. Samadi, A. Felner, and J. Schaeffer. Learning from multiple heuristics. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 357–362. AAAI Press, 2008.

[77] B. Settles. Active learning literature survey. Technical Report 1648, Computer Sciences. University of Wisconsin–Madison, 2010.

[78] J. Slaney and S. Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1–2):119–153, 2001.

[79] J. Slocum and D. Sonneveld. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.

[80] N. R. Sturtevant, A. Felner, M. Barrer, J. Schaeffer, and N. Burch. Memory-based heuristics for explicit state spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 609–614, 2009.

[81] J. Thayer, A. Dionne, and W. Ruml. Learning inadmissible heuristics during search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 250–257, 2011.

[82] F. Yang, J. C. Culberson, R. C. Holte, U. Zahavi, and A. Felner. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662, 2008.

[83] S. W. Yoon, A. Fern, and R. Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, 2008.

[84] U. Zahavi, A. Felner, N. Burch, and R. C. Holte. Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research*, 37:41–83, 2010.

[85] U. Zahavi, A. Felner, R. C. Holte, and J. Schaeffer. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4–5):514–540, 2008.