Predicting Optimal Solution Cost with Bidirectional Stratified Sampling

Levi Lelis
Computing Science Dept.
University of Alberta
Edmonton, Canada T6G 2E8
(santanad@cs.ualberta.ca)

Roni Stern, Ariel Felner
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel 85104
(roni.stern@gmail.com)
(felner@bgu.ac.il)

Sandra Zilles
Computer Science Dept.
University of Regina
Regina, Canada S4S 0A2
(zilles@cs.uregina.ca)

Robert C. Holte
Computing Science Dept.
University of Alberta
Edmonton, Canada T6G 2E8
(holte@cs.ualberta.ca)

Abstract

Optimal planning and heuristic search systems solve state-space search problems by finding a least-cost path from start to goal. As a byproduct of having an optimal path they also determine the optimal solution cost. In this paper we focus on the problem of determining the optimal solution cost for a state-space search problem directly, *i.e.,* without actually finding a solution path of that cost. We present an efficient algorithm, `BiSS`, based on ideas of bidirectional search and stratified sampling that produces accurate estimates of the optimal solution cost. Our method is guaranteed to return the optimal solution cost in the limit as the sample size goes to infinity. We show empirically that our method makes accurate predictions in several domains. In addition, we show that our method scales to state spaces much larger than can be solved optimally. In particular, we estimate the average solution cost for the 6x6, 7x7, and 8x8 Sliding-Tile Puzzle and provide indirect evidence that these estimates are accurate.

Introduction

Given an admissible heuristic function, search algorithms such as A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) find a path from the start state to a goal state that minimizes the sum of the edge-costs in the path. A byproduct of finding such a path is that the optimal solution cost becomes known. Lelis, Stern, and Jabbari Arfaee (2011) pointed out that there are situations in which one is interested only in the optimal solution cost, the solution path is not required. For instance, for the purpose of bidding on a project, a construction company does not need to know the entire plan for the project, but only an accurate estimate of the project's cost. This allows the time and expenses required to calculate a complete plan in full detail to be saved in the event that the company's bid is not successful. Lelis *et al.* (2011) developed SCP, the current state-of-the-art solution cost predictor. In addition, they showed that an accurate prediction of solution cost can be used to substantially speed up certain algorithms for finding suboptimal solution paths.

Heuristic functions themselves provide estimates of the solution cost of a given problem instance. However, they are built to guide search algorithms and are applied to a large number of nodes during the course of solving an instance. It is absolutely essential, therefore, that a heuristic function be very quick to calculate. Often heuristics must trade accuracy for speed of computation so that they are useful in practice. By contrast, a *solution cost predictor* is applied only once for a given problem instance and it is expected to be as accurate as possible. Obviously, it should be faster than a search algorithm that finds a least-cost path. However, it is not required to be nearly as fast as a heuristic function.

In this paper we present a new algorithm for predicting the optimal solution cost of a problem instance. Our algorithm is a bidirectional adaptation of Chen's algorithm (1992), which estimates a numerical property of a search tree (such as its size) by stratified sampling. We call our method `BiSS`, for Bidirectional Stratified Sampling. `BiSS` has two advantages over SCP: (1) `BiSS` 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.

We empirically evaluated `BiSS` on small and large versions of three domains: the Blocks World, the Sliding-Tile puzzle, and the Pancake puzzle. Our experiments show that solution cost prediction using `BiSS` is orders of magnitude faster than finding optimal solutions by actually solving the problem. On the small versions of the domains `BiSS`'s accuracy is equal to or better than SCP's. SCP cannot be run on the large versions of the domains because its preprocessing step would be too time-consuming. By contrast, our experiments show that `BiSS` efficiently makes accurate predictions for large versions of these domains too. In addition, we show 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.

Related Work

Korf, Reid, and Edelkamp (2001) provided a method that predicts the number of nodes expanded by IDA*, for a given cost bound, when the heuristic being used is consistent. They pointed out that predicting the optimal solution cost of a problem instance, or even the average optimal solution cost

of a domain, was an open problem. Zahavi *et al.* (2010) provided a different method, CDP, for the same task. CDP is more accurate and applicable to inconsistent heuristics as well as consistent ones. At the core of CDP is the definition of a "type system" for nodes in a search tree. The type of a node reflects properties of the node itself (*e.g.,* its heuristic value, the position of the blank in the Sliding-Tile puzzle) and of the nodes near it in the search tree (*e.g.,* its parent and children). In a preprocessing phase, CDP samples the state space to build an estimate of $p(t|u)$, the probability that a node of type t will be generated when a node of type u is expanded. With this information CDP constructs a tree that emulates the IDA* search tree, but in which nodes represent types instead of individual states.

Lelis *et al.* (2011) presented SCP, an algorithm based on CDP that predicts the cost of the optimal solution of a problem instance. Their key idea was to include in the type system a "goal type", which is the type of all the goal states and no other states. Whenever the goal type is generated in CDP's tree with a probability higher than a given threshold, SCP halts and returns the cost of the path to this goal node in its tree as the predicted solution cost.

The Knuth-Chen Method

Our method, BiSS, is based on a method by Knuth (1975) that was later improved by Chen (1992). In this section, we describe their method; in the next section we describe how it can be adapted to do optimal solution cost prediction.

Knuth (1975) presents a method to predict the size of a search tree by repeatedly performing a random walk from the start state. Each individual random walk is called a *probe*. Knuth's method assumes that all branches have similar structure in terms of the branching factors along the path. Thus, walking on one path is enough to derive an estimation of the structure of the entire tree. Despite its simplicity, his method provided accurate predictions in the domains tested. However, Knuth himself pointed out that his method was not effective when the tree being sampled is unbalanced. Chen (1992) addressed this problem with a stratification of the search tree through a type system to reduce the variance of the probing process. Chen's concept of a type system (he called types "stratifiers") is defined as follows.

Definition 1. *Let $S(s^*)$ be the set of nodes in the search tree rooted at s^*. $T = \{t_1, \ldots, t_n\}$ is a type system for $S(s^*)$ if it is a disjoint partitioning of $S(s^*)$. For every $s \in S(s^*)$, $T(s)$ denotes the unique $t \in T$ with $s \in t$.*

Chen's Stratified Sampling (SS) is a general method for approximating any function of the form

$$\varphi(s^*) = \sum_{s \in S(s^*)} z(s),$$

where z is any function assigning a numerical value to a node. $\varphi(s^*)$ represents a numerical property of the search tree rooted at s^*. 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 $s \in S(s^*)$, then $\varphi(s^*)$ is the size of the tree. If $z(s)$ returns 1 if s is a goal node and 0 otherwise, then $\varphi(s^*)$ is the number of goal nodes in the search tree.

Algorithm 1 Stratified Sampling

1: *input:* root s^* of a tree and a type system T
2: *output:* an array of sets A, where $A[i]$ is the set of (node,weight) pairs $\langle s, w \rangle$ for the nodes s expanded at level i.
3: $A[0] \leftarrow \{\langle s^*, 1 \rangle\}$
4: $i \leftarrow 0$
5: **while stopping condition** is false **do**
6:     **for** each element $\langle s, w \rangle$ in $A[i]$ **do**
7:         **for** each child $\hat{s}$ of s **do**
8:             **if** $A[i+1]$ contains an element $\langle s', w' \rangle$ with $T(s') = T(\hat{s})$ **then**
9:                 $w' \leftarrow w' + w$
10:                 with probability w/w', replace $\langle s', w' \rangle$ in $A[i+1]$ by $\langle \hat{s}, w' \rangle$
11:             **else**
12:                 insert new element $\langle \hat{s}, w \rangle$ in $A[i+1]$.
13:             **end if**
14:         **end for**
15:     **end for**
16:     $i \leftarrow i + 1$
17: **end while**

Instead of traversing the entire tree and summing all z-values, SS assumes 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.

Given a node s^* and a type system T, SS estimates $\varphi(s^*)$ as follows. First, it samples the tree rooted at s^* and returns a set A of *representative-weight* pairs, with one such pair for every unique type seen during sampling. In the pair $\langle s, w \rangle$ in A for type $t \in T$, s is the unique node of type t that was expanded during search and w is an estimate of the number of nodes of type t in the search tree rooted at s^*. $\varphi(s^*)$ is then approximated by $\hat{\varphi}(s^*, T)$, defined as

$$\hat{\varphi}(s^*, T) = \sum_{\langle s, w \rangle \in A} w \cdot z(s),$$

Algorithm 1 shows SS in detail. For convenience, the set A is divided into subsets, one for every layer in the search tree; $A[i]$ is the set of representative-weight pairs for the types encountered at level i.

In SS the types are required to be partially ordered: a node's type must be strictly greater than the type of its parent. Chen suggests that this can always be guaranteed by adding the depth of a node to the type system and then sorting the types lexicographically. In our implementation of SS, types at one level are treated separately from types at another level by the division of A into the $A[i]$. If the same type occurs on different levels the occurrences will be treated as though they were different types – the depth of search is implicitly added to the type system.

Representative nodes from $A[i]$ are expanded to get representative nodes for $A[i+1]$ as follows. $A[0]$ is initialized to contain only the root of the search tree to be probed, with
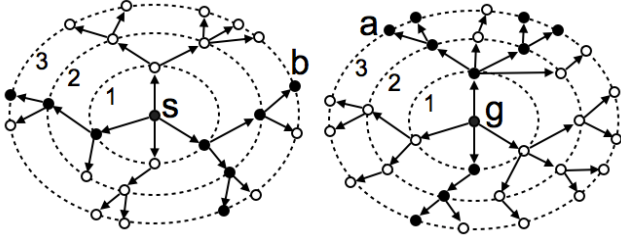
Figure 1: Hypothetical example of bidirectional sampling.

weight 1 (Line 3). In each iteration (Lines 6 through 15), all nodes in $A[i]$ are expanded. The children of each node in $A[i]$ are considered for inclusion in $A[i+1]$. If a child $\hat{s}$ has a type t that is already represented in $A[i+1]$ by another node s', then a *merge* action on $\hat{s}$ and s' is performed. In a merge action we increase the weight in the corresponding representative-weight pair of type t by the weight $w(s)$ of $\hat{s}$'s parent s (from level i) since there were $w(s)$ nodes at level i that are assumed to have children of type t at level $i+1$. $\hat{s}$ will replace the s' according to the probability shown in Line 10. Chen (1992) proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, we move to the next iteration. In Chen's original version of SS, the process continued until $A[i]$ was empty; Chen was assuming the tree naturally had a bounded depth.

One run of the SS algorithm is called a *probe*. Chen proved that the expected value of $\hat{\varphi}(s^*, T)$ converges to $\varphi(s^*)$ in the limit as the number of probes goes to infinity.

Using SS for Optimal Solution Cost Prediction

One approach to predicting the optimal solution cost of a given problem instance is to 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 probes 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 more than double the optimal cost.

Bidirectional Solution Cost Prediction

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 when one iteration of the loop in lines 5 to 16 of Algorithm 1 has finished. One "step" in a particular direction thus 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 1 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.[1] 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 states of the same type might occur close to the start and the goal even if the start and goal are very far apart. In Figure 1, 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 much 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 2. *(match) – For any n and m we say that $\mathcal{T}^F[n]$ and $\mathcal{T}^B[m]$ match if $\mathcal{T}^F[n+r] \cap \mathcal{T}^B[m-r] \neq \emptyset$ for all $r \in \{0, 1, \cdots, K\}$ 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 2, can be fully tested. This concept of match is illustrated in Figure 2 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_r 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 backwards search.

If a match occurs at step n from the start state and at step m from the goal state, then the searches terminate and $n+m$

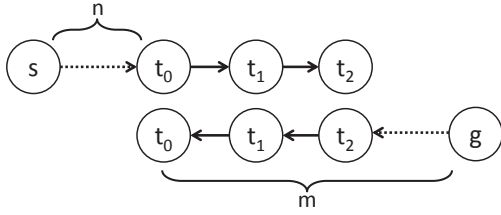[1]The correct stopping condition is more complex (Holte 2010).

Figure 2: 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_r denotes just one of the types in the corresponding set. A match occurs for $K = 2$ as $t_r \in \mathcal{T}^F[n+r] \cap \mathcal{T}^B[m-r]$ for $r \in \{0, 1, 2\}$.

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 $K = 0$).

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 $\mathcal{T}_i^B[m_i]$, for $0 \leq m \leq max_i\{m_i\}$. 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.

Theoretical Analysis

Assuming a BiSS probe always terminates we now prove that the predictions of BiSS never overestimate the optimal solution cost when the number of probes goes to infinity.

Lemma 1. *Given start state s, goal state g, type system T, any value $\gamma \in [0, 1]$, and a number p of probes,* BiSS *almost surely outputs an estimate $\hat{c}^*$ with $\hat{c}^* \leq c^*$ as $p \to \infty$.*

Proof. Let $BFSF[n]$ be the set of types of all nodes distance n from the start state, and $BFSB[m]$ be the sets of types of all nodes distance m from the goal state (using inverse operators). For instance, in Figure 1, $BFSB[1]$ is the set of types of the states one step from g. In SS every node in the tree rooted at s can be expanded with nonzero probability. By the law of large numbers, as $p \to \infty$, at least one node of each type at each level of the tree rooted at s will be expanded by SS, and the same argument is also true for the tree rooted at g. Therefore, as $p \to \infty$, $\mathcal{T}_*^F[n]$ will equal $BFSF[n]$ and $\mathcal{T}_*^B[m]$ will equal $BFSB[m]$.

If there exists a path from s to g with cost c^*, then, for some r, bidirectional BFS would find a state that occurs both in the forward frontier of depth r starting from s and in the backward frontier of depth r' starting from g, where $r' \in \{r, r-1\}$ and $c^* = r + r'$. This means that $\mathcal{T}_*^F[r]$ and $\mathcal{T}_*^B[r']$ have at least one type in common. Hence, for any $\gamma \in [0, 1]$, as $p \to \infty$, BiSS almost surely finds the level n and the level m for which $\mathcal{T}_*^F[n]$ and $\mathcal{T}_*^B[m]$ match with respect to γ. Since the candidate values for n and m are gradually increased, the first such values n and m found must fulfill $\hat{c}^* = n + m \leq c^*$. $\square$

By mapping the goal state g to a special unique type, named *goal type* (Lelis *et al.* 2011), and setting $\gamma = 1.0$, we prove our prediction algorithm to be optimal as $p \to \infty$.

Definition 3. *(goal type)* – *The goal type is a type $t^g \in T$ with $t^g = \{g\}$, i.e., $T(g) = t^g$ and $T(s) \neq t^g$ for $s \neq g$.*

Theorem 1. *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,* BiSS *almost surely outputs an estimate $\hat{c}^*$ with $\hat{c}^* = c^*$ as $p \to \infty$.*

Proof. $\hat{c}^* = n + m$ where $BFSF[n]$ and $BFSB[m]$ is the first match found by BiSS for $\gamma = 1.0$.

$n + m \leq c^*$ almost surely as $p \to \infty$ follows from Lemma 1.

To prove $n + m \geq c^*$, note that $BFSB[0]$ contains only the goal type t^g. Thus, with $\gamma = 1.0$, a match between $BFSF[n]$ and $BFSB[m]$ occurs only if $t^g \in BFSF[n + m]$. Since t^g contains only the goal state g, g must be on a path of cost $m + n$ from s. Since c^* is the optimal solution cost for s, this implies $m + n \geq c^*$.

Consequently, $m + n = c^*$. $\square$

The proof of Theorem 1 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 $\tau^F[.]$ and $\tau^B[.]$ for all values of r (as in Definition 2) 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. As the difference of the number of steps between $\tau^F[.]$ and $\tau^B[.]$ will be at most one 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 $O(C \cdot |T|)$.

Experimental Results

In this section we experimentally evaluate the accuracy and runtime of BiSS. Our experiments are run in three domains: the Blocks World, the Sliding-Tile puzzle, and the Pancake puzzle. These domains offer a good challenge for the optimal solution cost predictors as they have different properties. For instance, the Sliding-Tile puzzle has deeper solutions and a smaller branching factor than the other two domains. The Pancake puzzle has a constant branching factor, the Sliding-Tile puzzle has small variance in its branching factor, and the Blocks World's branching factor can vary considerably from one state to another. 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.

Our results show that: (a) BiSS's accuracy equals or exceeds SCP's in the smaller versions of the domains; (b) BiSS scales up much better than SCP and produces accurate predictions for the large versions of the domains; (c) BiSS is more accurate than the heuristics used to build its type system; (d) BiSS is considerably faster than solving the least-cost path problem; and (e) BiSS produces accurate predictions for state spaces too large to be solved optimally.

As stated in the introduction, 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 all our experiments, this heuristic was admissible. However, admissible heuristic functions are known as poor estimators of the optimal solution cost compared to inadmissible heuristics (Lelis *et al.* 2011). For examples of inadmissible heuristics see, *e.g.,* Bonet and Geffner (2001), Hoffmann and Nebel (2001), and Richter, Helmert, and Westphal (2008). We chose the Bootstrap heuristic (Jabbari Arfaee, Zilles, and Holte 2011) 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.* 2011).

Solution cost predictions are compared using *relative absolute error* (Lelis *et al.* 2011) for a set of optimal solution costs. For all start states with optimal solution cost X one computes the absolute difference of the predicted solution cost and X, adds these up, divides by the number of start states with optimal solution cost X and then divides by X. A system that makes perfect predictions will have a relative absolute error of 0.00.

We use the following type systems in our experiments.

$T_c(s) = (h(s), c(s, 0), \ldots, c(s, H))$, 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;

$T_{gc}(s) = (T_c(s), gc(s, 0), \ldots, gc(s, H))$, where $gc(s, k)$ is how many of s's grandchildren have heuristic value k.

Two nodes have the same type according to T_c if they have the same heuristic value and, for each k, they both have the same number of children with heuristic value k. T_{gc} additionally requires the same heuristic distribution for the grandchildren.

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 2), 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 same input parameter values Lelis *et al.* (2011) used. 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. The type system used in each experiment is specified below. When using a type system T for

Cost	Relative Absolute 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 1: 15 Blocks World

Cost	Relative Absolute 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 2: 20 Blocks World

Cost	Relative Absolute 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 3: 15-puzzle

BiSS we used the same T for SCP, but with the heuristic value of the parent node added, as required by SCP.

The results will be presented in tables like Table 1. The first column shows the optimal solution cost, followed by the relative absolute error of different predictors. We compare BiSS with SCP, 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 value in each row is in bold.

Blocks World

We ran experiments on small 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 (2001) 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.* (2011). 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 1. 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 2), 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%.

Sliding-Tile Puzzle

The smaller size of the sliding-tile puzzle we experimented with was the 4x4 puzzle (15-puzzle), which has 16!/2 reachable states from the goal state. Accuracy was measured over 1,000 uniformly random instances. We used T_{gc} with Man-

hattan Distance (MD) as the type system. Table 3 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 Korf and Felner's (2002) 6-6-6-6 disjoint pattern database with reflection along the main diagonal. We used T_{gc} with MD as the type system. Table 4 shows the results. BiSS is significantly more accurate than Bootstrap. For example, for instances with optimal solution cost of 100, BiSS's predictions are only 3 moves

Cost	Relative Absolute 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 4: 24-puzzle

$(0.03 * 100)$ different than the true optimal solution cost, on average, whereas Bootstrap's are 8 moves different.

Pancake Puzzle

Cost	Relative Absolute 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: 10-pancake

Cost	Relative Absolute 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 6: 35-pancake

For the pancake puzzle we also compare to the "GaP" heuristic (Helmert 2010), a highly accurate hand-crafted admissible heuristic for this domain.

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 uniformily random instances to compute the accuracy of the estimators. The heuristic used to construct the type systems was the maximum of the regular and the dual lookups (Zahavi et al. 2008) of a pattern

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

Table 7: BiSS runtime for $p = 2$.

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 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 (Yang *et al.* 2008), which we used to construct a "coarser" version of the T_c type system and to provide features for the Bootstrap heuristic (Jabbari Arfaee *et al.* 2011). 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 three of the children of a node, instead of taking into account the heuristic values of all the children. Table 6 shows the results. For an optimal solution cost larger than 32, the results for BiSS, Bootstrap, and GaP are very similar. Here we observe that all three predictors are very accurate in this domain, and BiSS was never worse than any other tested predictor. Moreover, BiSS tends to have a higher percentage of problem instances predicted perfectly. Note that as the Pancake puzzle gets larger the relative absolute error of the GaP heuristic gets smaller.

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 7 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 7 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 7 (Jabbari Arfaee *et al.* 2011). The advantage becomes even more evident when comparing to optimal solving time. For instance, Korf *et al.* (2001) 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 very accurate predictions of the optimal solution cost when using Manhattan Distance to build its type system.

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 8. 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 8.

Parberry (1995) 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 8, 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 8: Predicted average optimal solution cost for very large Sliding-Tile Puzzles.

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.

According to Lemma 1 for $\gamma < 1.0$ `BiSS` will have a zero or negative relative signed error in the limit of large p, and we have observed this trend in experiments that space limits prevent us from describing in detail. Hence, for any setting of γ other than 1.0, increasing p beyond a certain value will cause signed error, and therefore also absolute 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.

Limitations of BiSS

`BiSS` 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. Another limitation of our method is that it assumes the inverse of the operators to be available, as it is assumed by any bidirectional search algorithm. In cases where the inverse operators are not known, `SCP` is still the state-of-the-art solution cost predictor. Even though the current version of `BiSS` does not handle non-unit edge costs, we believe it could be extended to do so. For this purpose, `BiSS` would have to sample the *uniform-cost* search trees. Implementing and experimenting in domains with non-unit edge costs is also a subject of future work.

Conclusion

Optimal planning and heuristic search systems solve state-space search problems by finding a least-cost path from start to goal. As a byproduct of having an optimal path they also determine the optimal solution cost. However, there are situations in which the optimal path is not needed – one is interested only in the optimal solution cost. In this paper we presented `BiSS`, an efficient algorithm that accurately predicts the optimal solution cost without finding a least-cost path from start to goal. `BiSS` is based on ideas of bidirectional search and stratified sampling.

`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. It is much faster than actually solving the least-cost path problem. We showed empirically that `BiSS` makes very accurate predictions in several domains. `BiSS`'s predictions, with an appropriate setting of its parameters, were never worse than `SCP`'s in our experiments and were sometimes much better. `BiSS` scales much better than `SCP`. Finally, we showed it could be applied to state spaces much larger than can be solved optimally in a reasonable time.

References

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

Chen, P.-C. 1992. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing* 21:295–315.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In Felner, A., and Sturtevant, N. R., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search*, 109–110. AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 46–51.

Jabbari Arfaee, S.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence* 175(16-17):2075–2098.

Knuth, D. E. 1975. Estimating the efficiency of backtrack programs. *Mathematics of Computation* 29:121–136.

Korf, R., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artificial Intelligence* 129:199–218.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible treesearch. *Artificial Intelligence* 27:97–109.

Lelis, L.; Stern, R.; and Jabbari Arfaee, S. 2011. Predicting optimal solution costs with conditional probabilities. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 100–107.

Parberry, I. 1995. A real-time algorithm for the $(n^2$-1)-puzzle. *Information Processing Letters* 56(1):23–28.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In Fox, D., and Gomes, C. P., eds., *AAAI*, 975–982. AAAI Press.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1–2):119–153.

Yang, F.; Culberson, J. C.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence* 172(4–5):514–540.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of IDA* using conditional distributions. *Journal of Artificial Intelligence Research* 37:41–83.