Artificial Intelligence 230 (2016) 51–73

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces

Levi H.S. Lelis [a,*], Roni Stern [b], Shahab Jabbari Arfaee [c], Sandra Zilles [d], Ariel Felner [b], Robert C. Holte [c]

[a] Departamento de Informática, Universidade Federal de Viçosa, Viçosa, MG, Brazil
[b] Information Systems Engineering, Ben Gurion University, Beer-Sheva, Israel
[c] Computing Science Department, University of Alberta, Edmonton, AB, Canada
[d] Department of Computer Science, University of Regina, Regina, SK, Canada

ARTICLE INFO

Article history:
Received 1 September 2014
Received in revised form 7 September 2015
Accepted 29 September 2015
Available online 3 October 2015

Keywords:
Heuristic search
Solution cost prediction
Stratified sampling
Type systems
Learning heuristic functions

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 algorithm, BiSS, which is a hybrid of bidirectional search and stratified sampling that produces accurate estimates of the optimal solution cost. BiSS is guaranteed to return the optimal solution cost in the limit as the sample size goes to infinity. We show empirically that BiSS produces accurate predictions in several domains. In addition, we show that BiSS scales to state spaces much larger than can be solved optimally. In particular, we estimate the average solution cost for the 6×6, 7×7, and 8×8 Sliding-Tile puzzle and provide indirect evidence that these estimates are accurate. As a practical application of BiSS, we show how to use its predictions to reduce the time required by another system to learn strong heuristic functions from days to minutes in the domains tested.

1. Introduction

Many real-world problems can be cast as state-space search problems. For instance, state-space search algorithms have been used in a number of applications: robotics [37], domain-independent planning [2], chemical compounds discovery [14], bin packing [23], sequence alignment [22], automating layouts of sewers [4], and network routing [36], among others.

Heuristic search algorithms such as A* [13] and Iterative-Deepening-A* (IDA*) [24] are guided by the cost function $f(s) = g(s) + h(s)$ while finding solutions for state-space problems. Here, $g(s)$ is the cost of reaching state s from the root of the underlying search tree representing the state-space problem and $h(s)$ is a heuristic function providing an estimate of the cost-to-go of a solution going through state s in the underlying search tree.

Such algorithms are designed to find a least-cost path from a start state to a goal state in state-space search problems. The solution cost of such a path is found as a byproduct. In this paper we are interested in applications for which one only needs to know the optimal solution cost or an accurate estimate of the optimal solution cost – the solution path is not

* Corresponding author.
E-mail address: levilelis@gmail.com (L.H.S. Lelis).

http://dx.doi.org/10.1016/j.artint.2015.09.012
0004-3702/

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 this 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 quickly predict the optimal solution cost of a problem without finding an optimal sequence of actions from the start to a goal?

1.1. Heuristic functions

The heuristic function $h(\cdot)$ used by heuristic search algorithms is in fact an estimate of the optimal solution cost. This estimate is called *admissible* if it never overestimates the cost of the lowest-cost path from state s to the goal. Heuristic search algorithms, such as A* and IDA* guided by the f function are guaranteed to find an optimal solution when h is admissible [13,24]. A considerable amount of effort has been devoted to creating admissible heuristics [8,16,48,51] and inadmissible heuristics [11,20,44,49]. Admissible heuristics frequently provide inaccurate predictions of the optimal solution cost as they are biased to never overestimate the actual cost [12]. In some cases, even inadmissible heuristics are biased towards admissibility [11,44].

Regardless of admissibility, heuristics share a property: the heuristic evaluation must be fast enough to be computed for every node generated during search (in some settings it is more efficient to perform lazy heuristic computation during node expansion [42,50]), while a solution cost predictor is run only on the start state. In fact, often, heuristic functions sacrifice accuracy for speed. By contrast, solution cost predictors 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; 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.

1.2. Contributions

In this paper we present an algorithm for quickly and accurately predicting the optimal solution cost of state-space search problems. Our solution cost predictor, Bidirectional Stratified Sampling (BiSS), overcomes the two drawbacks of the Solution Cost Predictor (SCP) [33], another algorithm we introduced for predicting the optimal solution cost. Namely, in contrast to SCP, BiSS scales to very large state spaces and it has the guarantee of eventually converging to the correct answer. We describe SCP in Section 2.4 below.

A preliminary version of this paper appeared in the Proceedings of the International Conference on Automated Planning and Scheduling (2012) [28] and as a short paper in the Proceedings of the Symposium on Combinatorial Search [30]. The current paper substantially extends its preliminary versions. In addition to a comprehensive explanation of the algorithm, we include new experimental results. To be specific, in this paper we make the following contributions.

- We introduce BiSS, a prediction algorithm that 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.
- 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 8×8 configuration, which has more than 10^{88} reachable states, and provide indirect evidence that BiSS's predictions for these huge state spaces are quite accurate.
- As an application of BiSS we show how to quickly learn strong heuristics from predictions. 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.

Although BiSS overcomes two of the main limitations of SCP, it has two disadvantages SCP does not have. First, BiSS only produces predictions for domains with single goal states. Second, BiSS is applicable only to domains for which it is possible to "reason" backwards from the goal state. We discuss BiSS's weaknesses in Section 6.

This paper is organized as follows. In the next section we review Chen's Stratified Sampling (SS) [7], an algorithm for efficiently estimating the size of search trees. In Section 3 we describe BiSS, which is a bidirectional variation of SS for predicting the optimal solution cost, and show that BiSS is guaranteed to produce perfect predictions as the sample size tends to infinity. In Section 4 we show empirically that BiSS produces accurate predictions of the optimal solution cost, scaling to state spaces much larger than can be solved optimally. In Section 5 we show how to use BiSS to generate a training set to learn strong heuristic functions. In Section 6 we list BiSS's limitations and finally, in Section 7, we conclude the paper.

1.3. Problem formulation

Given a directed and implicitly defined search tree representing a state-space search problem rooted at start state s^* [38], called the underlying search tree (*UST*), we are interested in estimating the optimal solution cost of a path from s^* to the

goal node without necessarily finding an actual path from s^* to the goal. We assume state-space problems with unit-edge costs and a single goal state.

2. Background

BiSS uses a partition of the states in the problem's underlying state space which we call a *type system*. Type systems have been used to guide the sampling of prediction algorithms for estimating the number of nodes expanded by state-space search algorithms [6,27,10,53,5,31,32]. BiSS also uses a type system to guide its sampling, but instead of producing predictions of the number of nodes expanded, BiSS produces predictions of the optimal solution cost of a given state-space search problem.

2.1. Type systems for state-space problems

For convenience, we define the type system as a partition of the nodes in the problem's *UST*.

Definition 1 *(Type system).* Let $S(s^*) = (N, E)$ be a *UST* rooted at s^*, where N is its set of nodes and for each $s \in N$, $\{s' \mid (s, s') \in E\}$ is s's set of child nodes. $T = \{t_1, \ldots, t_k\}$ is a type system for $S(s^*)$ 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 1. For the Sliding-Tile puzzle (described in A.2), for example, one could define a type system based on the position of the blank tile. In this case, two nodes s and s' would be of the same type if s has the blank in the same position as s', regardless of the configuration of the other tiles in the two nodes.

We call a type system *heuristic-based* if all nodes of the same type have the same heuristic value. For a heuristic-based type system we define $h(t)$ to be the heuristic value of the states of type t. We assume in this work that the type system used is heuristic-based.

The simplest heuristic-based type system we use is the one introduced by Zahavi et al. [53] in which two nodes are of the same type if they have the same heuristic value. We also use variations of Zahavi et al.'s type system introduced by Lelis et al. [34]. To be specific, in our type systems, in addition to accounting for the heuristic value of the node when computing its type, we also account for the heuristic distribution of the children and sometimes even the grandchildren of the node in the *UST* when computing its type. The exact type systems we use are defined in the experimental section of this paper (Section 4.2).

2.1.1. Type systems are not abstractions

A common misconception is to think of type systems as state-space abstractions. Although type systems and abstractions are similar, they are conceptually different, as we now explain. Prieditis [41] 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*, and a type system does not necessarily define the relation between the types. Indeed, in some cases it is possible to represent a type system as a graph by defining the relation between pairs of types. This is what it is done with CDP and SCP (explained below) by sampling the state space and learning a set of conditional probabilities. By contrast, abstractions necessarily offer a partition of the nodes in the *UST*. Therefore, although type systems cannot necessarily be used as abstractions, abstractions can always be used as type systems.

2.2. Predicting the size of the search tree

Both SCP and BiSS, algorithms for estimating the optimal solution cost, are based on methods developed for estimating the search tree size. Knuth [21] developed a method for estimating the size of the search tree expanded by search algorithms such as chronological backtracking. Knuth's method works by sampling a small portion of the search tree and from there inferring the total search tree size. Under the mild assumption that the time required for expanding nodes is constant throughout the search tree, an estimate of the size of the search tree provides an estimate of the search algorithm's running time. Knuth noted that users of search algorithms usually do not know a priori how long the search will take. Knuth's method was later improved by Chen [6] through the use of a type system to reduce the variance of sampling. We call Chen's method Stratified Sampling (SS). BiSS is based on SS, which is explained in detail in Section 2.5.

Independently of Knuth and Chen, Korf et al. [27] developed a method for estimating the size of the search tree expanded by Iterative-Deepening A* (IDA*) [24]. Korf et al.'s method makes accurate predictions of the IDA* search tree size for the special case of consistent heuristics[1] and for sets of start states. Zahavi et al. [53] generalized Korf et al.'s method by

[1] A heuristic h is said to be consistent iff $h(s) \leq c(s, t) + h(t)$ for all states s and t, where $c(s, t)$ is the cost of the cheapest path from s to t.

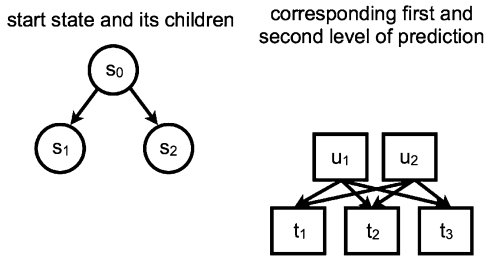

Fig. 1. The first step of a CDP prediction for start state s_0.

presenting CDP, a method that also produces accurate estimates of the IDA* search tree size when inconsistent heuristics are employed.

SS and CDP have in common the use of a type system to guide their sampling. Lelis et al. [34] discovered that the type systems developed to be used with CDP could substantially improve SS's prediction power. The main difference between SS and CDP is that while the former samples the search tree one wants to predict the size of, the latter samples the entire state space. Due to this difference in sampling strategy, as the number of samples grow large, SS has the guarantee of producing perfect predictions, while CDP does not. This distinction of sampling strategies is important in the context of solution cost prediction because the method we present in this paper, BiSS, is based on SS and it also has the guarantee of producing perfect predictions, while SCP, based on CDP, does not have such a guarantee.

As a practical application of SS, Paudel et al. [40] showed how the prediction algorithm can be used to partition the workload of large work-list based applications on distributed/shared-memory systems.

In the next section we explain CDP and in Section 2.4 we explain SCP – for a detailed explanation of CDP and SCP the reader should refer to Zahavi et al. [53] and Lelis et al. [33]. In Section 2.5 we turn our attention to SS.

2.3. Conditional distribution prediction (CDP)

Let $t, u \in T$. $p(t|u)$ denotes the average fraction of the children generated by a node of type u that are of type t. b_u is the average number of children generated by a node of type u. For example, if nodes of type u generate 5 children on average ($b_u = 5$) and 2 of them are of type t, then $p(t|u) = 0.4$. CDP samples the state space in order to estimate $p(t|u)$ and b_u for all $t, u \in T$. CDP does its sampling as a preprocessing step and, although type systems are defined for nodes in a search tree rooted at s^*, sampling is done before knowing the start state s^*. This is achieved by considering a state s drawn randomly from the state space as a node in a search tree. We denote by $\pi(t|u)$ and β_u the respective estimates of $p(t|u)$ and b_u obtained through sampling. In CDP, the values of $\pi(t|u)$ and β_u are used to estimate the number of nodes expanded on an iteration of IDA*.

The following example illustrates CDP's prediction process.

Example 2. Consider the example in Fig. 1. Here, after sampling the state space to calculate the values of $\pi(t|u)$ and β_u, we want to predict the number of nodes expanded on an iteration of IDA* with cost bound d for start state s_0. CDP generates all nodes at distance r from s_0 to seed its prediction. In this example we generate all nodes at distance 1 from s_0, depicted in the figure by s_1 and s_2. Given that $T(s_1) = u_1$ and $T(s_2) = u_2$ and that IDA* does not prune s_1 and 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 Fig. 1. We now use the values of π and β to estimate the types of the nodes 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 $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$. If $h(t_1) + 2$ (heuristic value of type t_1 plus its g-cost) 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 is the estimated number of nodes expanded by IDA* with cost bound d for start state s_0.

In summary, CDP predicts the number of nodes expanded by IDA* with cost bound d, by predicting the number of nodes of each type generated by IDA* at every level of the search up to and including level d. This is done incrementally. First the prediction is seeded with the types of the nodes at distance r from start. The number of nodes of each type is then estimated for level $r+1$ of the search. This prediction process continues to deeper and deeper levels until reaching the level of the cost bound d. The total number of nodes predicted by CDP is the sum of the estimated number of nodes of different types at every level, added to the number of nodes expanded in the initial search to level r.

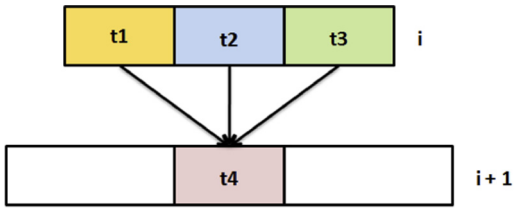

Fig. 2. Types at level i are used to calculate the approximated probability of t_4 existing at level $i+1$.

2.4. Solution cost predictor (SCP)

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.

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 2 *(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.

The SCP sampling procedure is the same as CDP's, but with the following difference: we perform a goal test on each node s sampled. If s is a goal, then its type becomes a goal type with only goal nodes in it. There can be as many goal types as different goal nodes in a search tree. 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.

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. The probability that a node of a goal type exists at the ith 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. SCP assumes the variables $\pi(\cdot|\cdot)$ to be independent.

Example 3. We now illustrate how $p(i, t, s^*, d)$ is calculated with the example shown in Fig. 2. Assume that only nodes of types t_1, t_2 and t_3 can generate a node of type t_4 in the type space. 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.

SCP predicts i to be the optimal solution cost of start state s^* with i being the first level in which the probability of a goal type existing exceeded a threshold value c set by the user.

2.4.1. Sampling drawbacks

The sampling strategy used by both CDP and SCP has two drawbacks. First, it can be prohibitively time-consuming. Second, the prediction algorithms using such a strategy are not guaranteed to produce perfect estimates as the number of samples grow large (CDP of the search tree size and SCP of the optimal solution cost). This is because the values of $\pi(t|u)$ and β_u are dependent on the state space as opposed to being dependent on the problem instance [34]. Thus, due to SCP's sampling strategy, the parameter c does not necessarily approximate the actual probability of finding a goal type at a given level of the search from the start state.

2.5. Stratified sampling for tree size prediction (SS)

The prediction algorithm introduced in this paper, BiSS, is based on a method by Knuth [21] that was later improved by Chen [7]. In this section, we describe their method; in Subsection 2.5.1 we describe how it can be adapted to do optimal solution cost prediction.

Knuth [21] 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 such as the longest uncrossed knight's path. However, Knuth himself pointed out that his method was not effective when the tree being sampled is unbalanced. Chen [7] addressed this problem with a stratification of the search tree through a type system (see Definition 1) to reduce the variance of the probing process.

Algorithm 1 Stratified sampling.

Input: root s^* of a tree and a type system T
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.

```
 1: A[0] ← {⟨s*, 1⟩}
 2: i ← 0
 3: while stopping condition is false do
 4:    for each element ⟨s, w⟩ in A[i] do
 5:       for each child ŝ of s do
 6:          if A[i+1] contains an element ⟨s', w'⟩ with T(s') = T(ŝ) then
 7:             w' ← w' + w
 8:             with probability w/w', replace ⟨s', w'⟩ in A[i+1] by ⟨ŝ, w'⟩
 9:          else
10:             insert new element ⟨ŝ, w⟩ in A[i+1].
11:          end if
12:       end for
13:    end for
14:    i ← i + 1
15: end while
```

Chen's 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.

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 included into all of our type systems.

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 weight 1 (Line 1). In each iteration (Lines 4 through 13), 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 s' according to the probability shown in Line 8. Chen [7] 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 $\hat{\varphi}(s^*, T)$ converges to $\varphi(s^*)$ in the limit as the number of probes goes to infinity.

2.5.1. Using SS for optimal solution cost prediction

A possible approach for using SS for predicting the optimal solution cost is as follows. 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.

In a preliminary experiment we ran on the (5×5)-Sliding-Tile puzzle (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. However, after we published the preliminary version of this

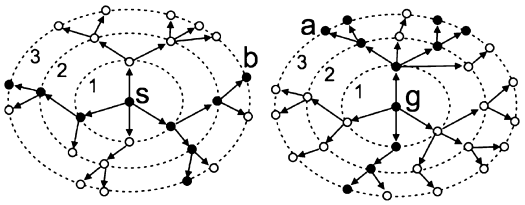

Fig. 3. Hypothetical example of bidirectional sampling.

paper [28], we tried the same strategy on domains such as the Blocks World and the Pancake puzzle and found that SS could find near-optimal solution paths on those spaces. This finding evolved into an algorithm called Stratified Tree Search (STS) [35]. STS differs from SS in that the former stops when it encounters the goal and the latter when reaching leaf nodes. Also, STS outputs the best solution encountered across multiple probes, while SS averages the results of multiple probes. Although BiSS does not find solution paths, it produces accurate estimates in all our experiments; STS fails to find near-optimal solutions on the Sliding-Tile puzzle. For a reference, see Table 3 of Lelis et al. [35], where we present the average suboptimality of 27.6 for STS on the (5×5)-Sliding-Tile puzzle. This suboptimality value means that if used as a prediction algorithm, STS would produce predictions with errors of 27.6% on average. By contrast, as we show in Section 4.5 below, BiSS produces predictions with an error of approximately 3% on average. Since the two algorithms have different purposes – STS finds near-optimal solution paths and BiSS produces solution cost predictions – we did not perform further experiments comparing the two algorithms.

3. Bidirectional stratified sampling (BiSS)

In this section we describe our new algorithm BiSS, a bidirectional version of SS. First BiSS samples the state space for the given start and goal states, then it uses the information gathered during sampling to estimate the optimal solution cost from start to goal. In Section 3.1 we describe how BiSS samples the state space. In Section 3.2 we describe how to combine multiple samples to produce an estimate of the optimal solution cost and we present the pseudocode of the complete BiSS algorithm. BiSS is designed in a way that is guaranteed to produce perfect estimates of the optimal solution cost in the limit, as the number of samples goes to infinity; in Section 3.3 we prove this.

3.1. BiSS's sampling procedure

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.

Fig. 3 illustrates the situation after three steps in each direction. Nodes around both the start state s and goal state g are shown. The black nodes are those that BiSS expands in its first three steps from s and its first three steps from g.

3.1.1. Sampling 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.[2] Since A^F and A^B contain individual states that have been generated by SS in each direction, testing if a state has been generated in both directions 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. We therefore base our stopping condition on the set of types that have occurred at each level of the searches and define $\boldsymbol{\tau}^F[n] = \{T(s) \mid \langle s, w\rangle \in A^F[n]\}$, the set of types of nodes expanded at level n by the copy of the SS algorithm searching forward from the start state, and $\boldsymbol{\tau}^B[m] = \{T(s) \mid \langle s, w\rangle \in A^B[m]\}$, the set of types of nodes expanded at level m by the copy of the SS algorithm searching backward from the goal state.

The naive stopping condition would be to stop as soon as $\boldsymbol{\tau}^F[n]$ and $\boldsymbol{\tau}^B[m]$ 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, states of the same type might occur close to the start and the goal even if the start and goal are far apart. In Fig. 3, 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).

[2] The correct stopping condition is more complex [18].

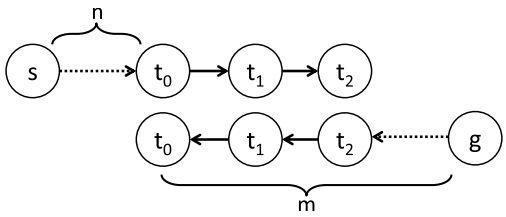

Fig. 4. Illustration of a match for $K = 2$.

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 3 *(Match).* For any n and m we say that $\boldsymbol{\tau}^F[n]$ and $\boldsymbol{\tau}^B[m]$ match if $\boldsymbol{\tau}^F[n+v] \cap \boldsymbol{\tau}^B[m-v] \neq \emptyset$ for all $v \in \{0, 1, \cdots, K\}$ where K is defined so it grows linearly with m as follows: $K = \max\{\lfloor \gamma \cdot m \rfloor, 1\}$. Here $\gamma \in [0, 1]$ is an input parameter.

After each step in each direction we test if the same type occurs in both $\boldsymbol{\tau}^F[n]$ and $\boldsymbol{\tau}^B[m]$, 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 3, can be fully tested. This concept of match is illustrated in Fig. 4 for $K = 2$. Each circle in the figure represents a set of types at a level of search ($\boldsymbol{\tau}^F[\cdot]$ or $\boldsymbol{\tau}^B[\cdot]$); 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 also has a state of type t_0 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 $n + m$ 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. Note that in terms of implementation, the layers of the forward search that are expanded while checking for a match can be kept in memory and reused in future layer expansions by the algorithm in case a match does not occur.

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.

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.

We note that our matching criterion is asymmetric in the sense that it is the forward search that is extended K extra levels. An alternative matching criterion is to extend each frontier $K/2$ extra levels. We have experimented with this alternative approach while designing the algorithm and observed that we would need larger values of K for obtaining results similar to those obtained by only extending the forward search K extra steps. We conjecture that by extending the forward search we are able to reach a region of the state space where the heuristic function used to define the type system is more accurate. For example, intuitively, heuristic functions such as pattern databases can be more accurate around the goal state, and by extending the forward search we might reach this "discriminative" region of the state space. By contrast, when extending the backward search we tend to reach a less discriminative, and thus less informative, region of the state space. Since the running time of the algorithm increases as we increase the value of K (a larger value of K results in more type comparisons), we decided to only extend the forward search while looking for a match.

3.2. Combining multiple probes to produce the overall prediction

The procedure just described represents one probe of `BiSS`. 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 $\boldsymbol{\tau}_i^F[n_i]$ and $\boldsymbol{\tau}_i^B[m_i]$, 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 $\boldsymbol{\tau}_*^F[n]$ denote the union of all the $\boldsymbol{\tau}_i^F[n]$, for $0 \leq n \leq \max_i\{n_i\}$ and let $\boldsymbol{\tau}_*^B[m]$ denote the union of all the $\boldsymbol{\tau}_i^B[m]$, for $0 \leq m \leq \max_i\{m_i\}$. We treat

Algorithm 2 Bidirectional stratified sampling, overall algorithm.

Input: start state s^*, goal state g, number of probes p, match parameter γ, and type system T.
Output: estimated optimal cost c of a path from s^* to g.
1: initialize $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ with empty sets
2: **for** $i = 1$ to p **do**
3: $(\boldsymbol{\tau}^F_i, \boldsymbol{\tau}^B_i) \leftarrow$ BidirectionalProbe(s^*, g, γ, T) // see Algorithm 3
4: $\boldsymbol{\tau}^F_* \leftarrow \boldsymbol{\tau}^F_* \cup \boldsymbol{\tau}^F_i$
5: $\boldsymbol{\tau}^B_* \leftarrow \boldsymbol{\tau}^B_* \cup \boldsymbol{\tau}^B_i$
6: **end for**
7: $n_F \leftarrow 0$; $n_B \leftarrow 0$
8: **while** true **do**
9: **if** $\boldsymbol{\tau}^F_*[n_F]$ and $\boldsymbol{\tau}^B_*[n_B]$ match according to γ **then**
10: return $n_F + n_B$
11: **end if**
12: $n_F \leftarrow n_F + 1$
13: **if** $\boldsymbol{\tau}^F_*[n_F]$ and $\boldsymbol{\tau}^B_*[n_B]$ match according to γ **then**
14: return $n_F + n_B$
15: **end if**
16: $n_B \leftarrow n_B + 1$
17: **end while**

Algorithm 3 BidirectionalProbe.

Input: start state s^*, goal state g, match parameter γ, and type system T.
Output: Two arrays of sets of types $\boldsymbol{\tau}^F$ and $\boldsymbol{\tau}^B$ for the forward and the backward searches, where $\boldsymbol{\tau}^F[i]$ is the set of types encountered at level i of the forward search, and $\boldsymbol{\tau}^B[j]$ is the set of types encountered at level j of the backward search.
1: $\boldsymbol{\tau}^F[0] \leftarrow \{T(s^*)\}$; $\boldsymbol{\tau}^B[0] \leftarrow \{T(g)\}$
2: $A^F[0] \leftarrow \langle s^*, 1\rangle$; $A^B[0] \leftarrow \langle g, 1\rangle$
3: $n_F \leftarrow 0$; $n_B \leftarrow 0$
4: **while** true **do**
5: **if** $A^F[n_F]$ and $A^B[n_B]$ match according to γ **then**
6: return $\boldsymbol{\tau}^F$ and $\boldsymbol{\tau}^B$
7: **end if**
8: MoveOneStep(A^F, $\boldsymbol{\tau}^F$, n_F) // see Algorithm 4
9: $n_F \leftarrow n_F + 1$
10: **if** $A^F[n_F]$ and $A^B[n_B]$ match according to γ **then**
11: return $\boldsymbol{\tau}^F$ and $\boldsymbol{\tau}^B$
12: **end if**
13: MoveOneStep(A^B, $\boldsymbol{\tau}^B$, n_B) // see Algorithm 4
14: $n_B \leftarrow n_B + 1$
15: **end while**

$\boldsymbol{\tau}^F_i[n] = \emptyset$ if BiSS did not reach level n during the i-th probe of the forward search. Likewise, we treat $\boldsymbol{\tau}^B_i[m] = \emptyset$ 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 once again use our matching scheme (Definition 3) as follows. We set m and n to zero and gradually increment them, checking for a match between $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$ after each increment; $n + m$ is returned as the predicted optimal solution cost when the first match occurs. In Section 3.3 we show that such an approach allows BiSS to guarantee asymptotic exact predictions. In summary, the complete BiSS algorithm works as follows: we use our matching scheme as the stopping criterion for different independent probes to obtain the sets $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$; then we use the same matching scheme to produce the final prediction given the sets $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$.

Algorithm 2 describes the overall BiSS algorithm, which receives a start state s^*, a goal state g, a number of probes p, a γ value for determining when the forward and the backward frontiers match, and a type system T. BiSS returns an estimated cost c of the optimal path from s^* to g.

In lines 2 through 6 of Algorithm 2 BiSS performs p bidirectional probes. The process of performing a probe is described in Algorithm 3. As we described in Section 3.1, during a probe BiSS interleaves the execution of two copies of SS, one proceeding forwards from s^* and another backwards from g. Algorithm 4 describes the process of moving one step in each direction. The reader will notice that Algorithm 4 is very similar to one iteration of SS, with the difference that in Algorithm 4 we keep track of all types encountered during the expansion of the n-th layer in the $\boldsymbol{\tau}$ structure. Note that if Algorithm 4 is expanding one layer of the forward search, then we use the forward operators to generate the children $\hat{s}$ of s, and we use the backward operators otherwise. The set of all types encountered during the p probes in the forward search is stored in the structure $\boldsymbol{\tau}^F_*$ (line 4 of Algorithm 2). Similarly, the set of all types encountered during the p probes in the backward search is stored in the structure $\boldsymbol{\tau}^B_*$ (line 5 of Algorithm 2).

Once BiSS finishes the p probes it uses the information stored in $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ to produce an estimate of the optimal solution cost (from line 7 to line 17 of Algorithm 2). BiSS looks for a match of the forward and the backward frontiers

Algorithm 4 MoveOneStep.

Input: 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; an array of sets of types $\boldsymbol{\tau}$, where $\boldsymbol{\tau}[i]$ is the set of types encountered at level i of a given search; number of steps n.
Output: an array of sets A, and an array of sets of types $\boldsymbol{\tau}$.

```
1: for each pair ⟨s, w⟩ in A[n] do
2:    for each child ŝ of s do
3:       if τ[n+1] does not contain T(ŝ) then
4:          τ[n+1] ← τ[n+1] ∪ {T(ŝ)}
5:       end if
6:       if A[n+1] contains an element ⟨s', w'⟩ with T(s') = T(ŝ) then
7:          w' ← w' + w
8:          with probability w/w', replace ⟨s', w'⟩ by ⟨ŝ, w'⟩
9:       else
10:         insert new element ⟨ŝ, w⟩ in A[n+1].
11:      end if
12:   end for
13: end for
```

while accounting for the set of types in $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$. That is, once a match occurs between $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ BiSS, returns $n_F + n_B$ as the estimated optimal solution cost of a path between s^* and g.

BiSS's pseudocode can be optimized in two different ways. First, there is no need to store sets $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ in memory when using a single probe. That is, the value of $n_F + n_B$ when a match is found in Algorithm 3 is already the output of the overall algorithm. Second, when using multiple probes, $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ can be aggregated to A^F and A^B while searching for a match in Algorithm 3. This will possibly allow BiSS to find a match more quickly in Algorithm 3. This is because sets A^F and A^B will then contain not only the types encountered during the current probe, but also all the types encountered before that. Intuitively, by enlarging the sizes of A^F and A^B one increases the chances of finding a match. This optimization is possible because BiSS's predicted value depends on $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$, then there is no need to search to a depth in Algorithm 3 that will not be reached while searching for a match between $\boldsymbol{\tau}^F_*$ and $\boldsymbol{\tau}^B_*$ in Algorithm 2.

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 4. Let $BFSF[n]$ be the set of types of all nodes at distance n from the start state as if enumerated using a Breadth-First Search to depth n, and $BFSB[m]$ be the set of types of all nodes at distance m in the backwards direction.

Lemma 1. *For any level n in the forward search and any level m in the backward search, the probability of $\boldsymbol{\tau}^F_*[n]$ being equal to $BFSF[n]$ and $\boldsymbol{\tau}^B_*[m]$ being equal to $BSFB[m]$ approaches one as $p \to \infty$.*

Proof. Every node in the *UST* has a nonzero probability of being the representative node of its type during a BiSS probe (line 8 of Algorithm 4). Therefore, every node in the *UST* has a nonzero probability of being expanded by BiSS. As the number of probes $p \to \infty$, the probability of at least one node of each type at each level of the tree rooted at s being expanded approaches one. The same argument is also true for the tree rooted at g. Therefore, the probability of $\boldsymbol{\tau}^F_*[n]$ being equal to $BFSF[n]$ and $\boldsymbol{\tau}^B_*[m]$ being equal to $BSFB[m]$ approaches one as $p \to \infty$. □

Lemma 2. *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 $p \to \infty$.*

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 $BFSF[v+y]$ and $BFSB[v'-y]$ have at least one type in common for all $y \in \{0, 1, \cdots, K\}$ with $K = \max\{\lfloor \gamma \cdot v \rfloor, 1\}$ and $\gamma \in [0, 1]$. Hence, for any $\gamma \in [0, 1]$, as $p \to \infty$, it follows from Lemma 1 that BiSS finds the level n and the level m for which $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$ 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. □

Lemma 2 guarantees that in the limit, as the number of probes grows large, BiSS will always underestimate the actual optimal solution cost. Note, however, that the number of probes used in practical scenarios will likely not be large enough to meet this asymptotic guarantee. Thus, in practice, BiSS can also overestimate the actual optimal solution cost.

By mapping the goal state g to a special unique *goal type* [33] (Definition 2) and setting $\gamma = 1.0$, we prove that, as $p \to \infty$, the probability of BiSS producing a perfect prediction approaches one.

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, the probability of* BiSS *producing an estimate $\hat{c}^*$ with $\hat{c}^* = c^*$ approaches one as $p \to \infty$.*

Proof. Let $\hat{c}^* = n + m$ where $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$ is the first match found by BiSS for $\gamma = 1.0$. That the probability of having $n + m \leq c^*$ approaches one as $p \to \infty$ follows from Lemma 2.

We now prove that the probability of having $n + m \geq c^*$ also approaches one as $p \to \infty$. Note that $\boldsymbol{\tau}^B_*[0]$ contains only the goal type t_g. Thus, with $\gamma = 1.0$, a match between $\boldsymbol{\tau}^F_*[n]$ and $\boldsymbol{\tau}^B_*[m]$ occurs only if $t_g \in \boldsymbol{\tau}^F_*[m + n]$. Since t_g contains only the goal state g, g must be on a path of cost $m + n$ from s. Such a path is encountered with probability approaching one as $p \to \infty$ (Lemma 1). 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 $p \to \infty$. □

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 $\boldsymbol{\tau}^F_*[\cdot]$ and $\boldsymbol{\tau}^B_*[\cdot]$. 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.

When predicting the search tree size SS assumes that nodes of the same type root subtrees of the same size [7]. BiSS assumes that subtrees R_s and $R_{s'}$ rooted at nodes s and s' of the same type have the same set of types at any level i. Let R^i_s and $R^i_{s'}$ be the sets of nodes encountered at level i of subtrees R_s and $R_{s'}$, respectively. Given a type system T, BiSS assumes that

$$\{T(a) \mid a \in R^i_s\} = \{T(a) \mid a \in R^i_{s'}\}. \tag{1}$$

Intuitively, a single probe of BiSS is able to produce a perfect prediction of the optimal solution cost when using a type system that has goal types and obeys Equation (1) for all nodes s and s' of the same type, and when $\gamma = 1.0$. This is because a single probe of BiSS's forward and backward searches would encounter exactly the set of types in *BFSF* and *BFSB*.

3.4. Running time analysis

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 $\boldsymbol{\tau}^F[.]$ and $\boldsymbol{\tau}^B[.]$ for all values of v (as in Definition 3) 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's running time is on the order of $p \cdot (|T| \cdot b \cdot C + |T| \cdot C^2)$.

3.5. Memory requirement analysis

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 $\boldsymbol{\tau}^F[\cdot]$ and $\boldsymbol{\tau}^B[\cdot]$ 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's worst-case memory requirement is on the order of $C \cdot |T|$.

4. Experimental results

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

4.1. Heuristic functions as baselines

As stated earlier, 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 [33], admissible heuristic functions are known to be poor estimators of the optimal solution cost compared to inadmissible heuristics. For

examples of inadmissible heuristics see, e.g., Bonet and Geffner [2], Hoffmann and Nebel [17], and Richter, Helmert, and Westphal [43]. Like we did in the SCP experiments [33], we choose the Bootstrap heuristic [20] 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. [20]). Third, the bootstrap heuristic was shown to be more accurate than a simple linear regression that tries to learn the heuristic error [20,49], and thus provides also a comparison against methods that estimate solution cost by learning the error of the heuristic. We explain the Bootstrap system in detail in Section 5.1.

4.2. Type systems

We use the following type systems in our experiments.

$$T_c(s) = (h(s), c(s,0), \ldots, c(s,H)), \tag{2}$$

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;

$$T_{gc}(s) = (T_c(s), gc(s,0), \ldots, gc(s,H)), \tag{3}$$

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.

4.3. Experimental setup

In this section 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 3), BiSS is able to make accurate predictions in domains with different average solution costs while using the same γ-value. In Section 4.8 we empirically study how BiSS is affected by the choice of the input parameters and present a procedure that can help choosing BiSS's parameters.

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 a 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 in Appendix A. We used the same type systems with both BiSS and SCP.

Predictions are compared using relative unsigned error (Lelis et al. [33]) 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 unsigned error of 0.00. We also present the standard deviation of the relative unsigned error in our table of results below. All our experiments are run on 2.67 GHz Intel Xeon CPUs.

In this paper we use four different problem domains: the Blocks World, the Sliding-Tile puzzle, the Pancake puzzle, and Rubik's Cube. Taken together these domains offer a good challenge for the prediction systems evaluated as they have very distinct properties. For instance, the Blocks World, the Pancake puzzle and Rubik's Cube have shallow solutions; the Sliding-Tile puzzle has deeper solutions. The Pancake puzzle and the Blocks World have larger branching factors; the Sliding-Tile puzzle has a much lower branching factor. For a detailed description of each of these domains see Appendix A.

In the Blocks World, the Sliding-Tile puzzle, and the Pancake puzzle we use two state space sizes: 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. First we present the results on the smaller versions of the problem domains (Section 4.4) and then we present the results on the larger versions of the problem domains (Section 4.5). Initially we present only the accuracy results, and in Section 4.6 we show the empirical runtime of BiSS. Finally, in Section 4.7 we present prediction results on very large versions of the Sliding-Tile puzzle (up to the 8×8 configuration).

4.4. Accuracy comparison of BiSS and SCP

In this section we present prediction results on the 15 Blocks World, the (4×4)-Sliding-Tile puzzle (15-puzzle), and the 10 Pancake puzzle. Accuracy was measured over 1000 randomly generated instances of each domain. The results are shown in Table 1. The first column shows the optimal solution cost, followed by the relative unsigned error of different predictors. The optimal solution cost for the problem instances used in this experiment were obtained with IDA*. 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

Table 1
Prediction results on smaller domains in terms of relative unsigned error ± standard deviation as well as the percentage of problem instances for which the prediction was perfect.

Cost	Relative unsigned error				Correct (%)			n
	BiSS	SCP	BS	h	BiSS	SCP	BS	
15 Blocks World								
16	**0.00** ± 0.00	0.06 ± 0.00	0.44 ± 0.00	0.06 ± 0.00	**100**	0	0	1
17	**0.00** ± 0.00	0.04 ± 0.03	0.33 ± 0.05	0.18 ± 0.00	**100**	40	0	5
18	**0.02** ± 0.03	0.03 ± 0.03	0.28 ± 0.07	0.19 ± 0.03	**75**	43	0	16
19	**0.01** ± 0.03	0.07 ± 0.03	0.22 ± 0.05	0.24 ± 0.03	**73**	2	0	46
20	**0.01** ± 0.03	0.09 ± 0.03	0.20 ± 0.05	0.26 ± 0.02	**77**	1	0	77
21	**0.02** ± 0.03	0.12 ± 0.03	0.15 ± 0.05	0.30 ± 0.02	**70**	0	0	130
22	**0.02** ± 0.03	0.14 ± 0.04	0.12 ± 0.05	0.33 ± 0.02	**70**	0	1	177
23	**0.01** ± 0.02	0.17 ± 0.04	0.08 ± 0.05	0.36 ± 0.02	**74**	0	18	175
24	**0.01** ± 0.02	0.18 ± 0.03	0.06 ± 0.03	0.39 ± 0.02	**81**	0	11	168
25	**0.01** ± 0.02	0.20 ± 0.03	0.04 ± 0.02	0.41 ± 0.02	**78**	0	10	119
26	**0.01** ± 0.01	0.22 ± 0.03	0.02 ± 0.03	0.43 ± 0.02	**84**	0	70	50
27	**0.00** ± 0.00	0.21 ± 0.02	0.03 ± 0.03	0.45 ± 0.01	**100**	0	30	26
28	**0.00** ± 0.00	0.20 ± 0.02	0.05 ± 0.02	0.48 ± 0.02	**100**	0	0	10
15-puzzle								
48	0.08 ± 0.05	**0.06** ± 0.04	0.09 ± 0.06	0.29 ± 0.06	13	**15**	5	51
49	0.07 ± 0.05	**0.05** ± 0.03	0.08 ± 0.06	0.30 ± 0.08	17	**20**	11	45
50	0.07 ± 0.04	**0.04** ± 0.03	0.08 ± 0.05	0.29 ± 0.06	14	**21**	2	74
51	0.07 ± 0.04	**0.03** ± 0.03	0.07 ± 0.05	0.29 ± 0.05	8	**43**	7	71
52	0.07 ± 0.05	**0.03** ± 0.02	0.07 ± 0.05	0.30 ± 0.05	23	**42**	6	73
53	0.07 ± 0.05	**0.03** ± 0.03	0.06 ± 0.04	0.31 ± 0.05	19	**32**	4	62
54	0.07 ± 0.05	**0.03** ± 0.03	0.07 ± 0.05	0.30 ± 0.06	6	**37**	9	64
55	0.06 ± 0.04	**0.03** ± 0.03	0.08 ± 0.05	0.29 ± 0.06	10	**35**	4	82
56	0.07 ± 0.05	**0.04** ± 0.03	0.07 ± 0.05	0.30 ± 0.06	9	**23**	6	88
57	0.06 ± 0.04	**0.04** ± 0.03	0.06 ± 0.04	0.30 ± 0.05	10	**23**	13	46
58	0.08 ± 0.04	**0.04** ± 0.03	0.07 ± 0.05	0.29 ± 0.06	2	**23**	6	46
59	0.05 ± 0.03	**0.04** ± 0.03	0.08 ± 0.04	0.27 ± 0.04	7	**10**	0	39
60	**0.06** ± 0.04	**0.06** ± 0.03	0.07 ± 0.05	0.28 ± 0.05	8	**16**	8	36
61	**0.05** ± 0.04	0.06 ± 0.03	0.08 ± 0.04	0.28 ± 0.05	**23**	11	0	17
62	**0.06** ± 0.03	0.06 ± 0.04	0.09 ± 0.04	0.28 ± 0.05	0	**7**	3	26
63	**0.04** ± 0.04	0.06 ± 0.03	0.08 ± 0.05	0.27 ± 0.04	**17**	5	0	17
10 Pancake puzzle								
7	**0.04** ± 0.07	0.12 ± 0.08	0.08 ± 0.09	0.18 ± 0.13	**73**	25	53	530
8	**0.02** ± 0.05	0.05 ± 0.06	0.07 ± 0.08	0.21 ± 0.11	**84**	60	48	1253
9	0.05 ± 0.06	**0.04** ± 0.05	0.07 ± 0.07	0.23 ± 0.09	58	**67**	46	1796
10	**0.06** ± 0.06	0.07 ± 0.05	0.06 ± 0.06	0.25 ± 0.07	**49**	34	46	1146

percentage of problem instances for which a predictor makes perfect predictions ($\hat{c}^* = c^*$), and the number of problem instances used to compute each entry of the tables (column "n"). The best value in each row is in bold.

BiSS is very accurate for the 15 Blocks World (upper part of Table 1); 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. The base heuristic function h used in this experiment was the very weak "Out of Place" heuristic, which counts the number of blocks not in their goal position, cf. Jabbari Arfaee et al. [20]. We observe that the accuracy of the Out of Place heuristic decreases as we increase the optimal solution cost.

Manhattan Distance (MD) was the heuristic function h used in the experiments on the 15-puzzle. As observed in Table 1, 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 for small solution 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 heuristic h used in the 10 Pancake puzzle experiment was the maximum of the regular and the dual lookups [54] of a pattern database built by keeping the identity of the four smallest pancakes and turning the other pancakes into "don't cares". For the pancake puzzle we also compare to the "GAP" heuristic [15], a highly accurate hand-crafted admissible heuristic for this domain. Our table of results show that even GAP is not as accurate as BiSS for the 10 Pancake puzzle.

4.5. Larger domains

In this section we present the prediction results on the 20 Blocks World, (5×5)-Sliding-Tile puzzle (24-puzzle), 35 Pancake puzzle, and the $3 \times 3 \times 3$ Rubik's Cube (again, for a description of the domains see Appendix A). Unless stated otherwise, in this experiment we used the same heuristic functions used in the experiment on smaller domains. Also, like

Table 2
Prediction results on the 20 Blocks World in terms of relative unsigned error ± standard deviation as well as the percentage of problem instances for which the prediction was perfect.

20 Blocks World						
Cost	Relative unsigned error			Percentage correct		n
	BiSS	BS	h	BiSS	BS	
22	**0.00** ± 0.00	0.50 ± 0.00	0.09 ± 0.00	**100**	0	1
24	**0.02** ± 0.02	0.29 ± 0.08	0.19 ± 0.02	**50**	0	2
25	**0.02** ± 0.02	0.26 ± 0.04	0.22 ± 0.02	**50**	0	6
26	**0.03** ± 0.03	0.26 ± 0.07	0.25 ± 0.03	**38**	0	13
27	**0.01** ± 0.02	0.22 ± 0.05	0.27 ± 0.02	**72**	0	44
28	**0.02** ± 0.03	0.18 ± 0.04	0.29 ± 0.01	**59**	0	54
29	**0.01** ± 0.02	0.15 ± 0.04	0.32 ± 0.02	**66**	0	103
30	**0.02** ± 0.02	0.13 ± 0.04	0.34 ± 0.02	**57**	0	138
31	**0.01** ± 0.02	0.10 ± 0.04	0.36 ± 0.01	**63**	0	148
32	**0.01** ± 0.02	0.08 ± 0.04	0.38 ± 0.01	**68**	6	160
33	**0.01** ± 0.02	0.06 ± 0.03	0.40 ± 0.01	**68**	7	138
34	**0.01** ± 0.02	0.04 ± 0.02	0.42 ± 0.01	**70**	12	110
35	**0.01** ± 0.01	0.03 ± 0.01	0.43 ± 0.01	**79**	11	54
36	0.01 ± 0.01	**0.00** ± 0.01	0.45 ± 0.01	80	**95**	20
37	**0.00** ± 0.00	0.03 ± 0.02	0.47 ± 0.01	**100**	28	7

Table 3
Prediction results on the 24-puzzle in terms of relative unsigned error ± standard deviation as well as the percentage of problem instances for which the prediction was perfect.

24-puzzle						
Cost	Relative unsigned error			Percentage correct		n
	BiSS	BS	h	BiSS	BS	
92	**0.05** ± 0.04	0.07 ± 0.04	0.25 ± 0.04	**19**	3	26
94	**0.04** ± 0.03	0.07 ± 0.04	0.26 ± 0.04	**18**	0	27
96	**0.05** ± 0.03	0.08 ± 0.04	0.25 ± 0.04	**13**	2	37
98	**0.03** ± 0.03	0.07 ± 0.04	0.26 ± 0.04	**20**	5	39
100	**0.03** ± 0.03	0.08 ± 0.04	0.26 ± 0.04	**26**	2	41
102	**0.04** ± 0.03	0.08 ± 0.03	0.25 ± 0.04	**11**	0	44
104	**0.03** ± 0.02	0.08 ± 0.03	0.25 ± 0.03	**25**	0	40
106	**0.04** ± 0.03	0.08 ± 0.04	0.25 ± 0.03	**10**	0	48
108	**0.03** ± 0.03	0.08 ± 0.03	0.24 ± 0.03	**37**	5	37
110	**0.03** ± 0.02	0.09 ± 0.03	0.24 ± 0.03	**17**	0	23
112	**0.04** ± 0.01	0.07 ± 0.02	0.25 ± 0.02	**0**	**0**	6
114	**0.03** ± 0.01	0.08 ± 0.03	0.24 ± 0.04	**0**	**0**	10
116	**0.04** ± 0.04	0.10 ± 0.02	0.20 ± 0.02	**12**	0	8
118	**0.02** ± 0.01	0.10 ± 0.02	0.23 ± 0.02	**33**	0	3
120	**0.04** ± 0.01	0.09 ± 0.01	0.21 ± 0.02	**0**	**0**	4

Table 4
Prediction results on the 35 Pancake puzzle in terms of relative unsigned error ± standard deviation as well as the percentage of problem instances for which the prediction was perfect.

35 Pancake puzzle										
Cost	Relative unsigned error					Percentage correct				n
	BiSS	BiSS+G	BS	GAP	h	BiSS	BiSS+G	BS	GAP	
29	**0.03** ± 0.04	0.10 ± 0.02	**0.03** ± 0.03	0.04 ± 0.02	0.21 ± 0.04	**50**	0	33	16	6
30	**0.03** ± 0.02	0.06 ± 0.06	0.05 ± 0.04	**0.03** ± 0.01	0.18 ± 0.04	**36**	**36**	18	18	11
31	**0.02** ± 0.02	0.07 ± 0.05	0.04 ± 0.04	0.03 ± 0.01	0.19 ± 0.03	**36**	12	27	23	47
32	**0.01** ± 0.02	0.06 ± 0.04	0.03 ± 0.03	0.02 ± 0.01	0.19 ± 0.03	**62**	10	30	23	121
33	0.02 ± 0.02	0.07 ± 0.04	0.03 ± 0.02	**0.02** ± 0.01	0.19 ± 0.02	**38**	8	33	34	218
34	0.02 ± 0.02	0.06 ± 0.04	0.03 ± 0.02	**0.02** ± 0.01	0.19 ± 0.02	**39**	8	32	37	294
35	0.02 ± 0.02	0.06 ± 0.04	0.02 ± 0.02	**0.02** ± 0.01	0.19 ± 0.01	34	6	**42**	36	267
36	0.02 ± 0.02	0.04 ± 0.03	**0.01** ± 0.01	0.03 ± 0.00	0.19 ± 0.00	33	19	**52**	0	36

in the former experiment, with the exception of the 24-puzzle, accuracy is measured over 1000 random instances in each domain. In the 24-puzzle we measure accuracy over 433 random instances, which were solved optimally using IDA* with the 6-6-6-6 disjoint pattern database with reflection along the main diagonal [26].

We present the results on the larger versions of the Blocks World, the Sliding-Tile puzzle, and the Pancake puzzle in Tables 2, 3, and 4, respectively; the results on Rubik's Cube will be shown in Table 5.

Table 5
Prediction results on Rubik's Cube in terms of relative unsigned error ± standard deviation as well as the percentage of problem instances for which the prediction was perfect.

Rubik's Cube						
Cost	Relative unsigned error			Percentage correct		n
	BiSS	BS	h	BiSS	BS	
6	**0.01** ± 0.06	0.92 ± 0.35	0.02 ± 0.07	**94**	2	39
7	**0.03** ± 0.08	0.85 ± 0.31	0.07 ± 0.08	**88**	0	52
8	**0.04** ± 0.11	0.91 ± 0.26	0.10 ± 0.07	**88**	0	34
9	**0.05** ± 0.14	0.81 ± 0.28	0.15 ± 0.09	**77**	0	49
10	**0.04** ± 0.11	0.76 ± 0.21	0.21 ± 0.07	**81**	1	53
11	**0.09** ± 0.14	0.65 ± 0.17	0.25 ± 0.06	**56**	0	44
12	**0.13** ± 0.12	0.57 ± 0.12	0.29 ± 0.06	**34**	0	44
13	**0.13** ± 0.09	0.44 ± 0.15	0.36 ± 0.06	**16**	3	55
14	**0.09** ± 0.05	0.37 ± 0.11	0.39 ± 0.05	**18**	0	49
15	**0.05** ± 0.03	0.30 ± 0.07	0.42 ± 0.05	**29**	0	31
16	**0.01** ± 0.02	0.22 ± 0.05	0.47 ± 0.03	**83**	0	6

Table 6
BiSS runtime (in seconds) for $p = 2$, $\gamma = 0.5$.

Domain	min	mean	max
20 Blocks World	26	41	57
24-puzzle	18	30	48
35 Pancake puzzle	19	24	30
Rubik's Cube	1	15	22

For the 20 Blocks World, 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 h 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%.

BiSS is substantially more accurate than Bootstrap on the 24-puzzle. 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.

For the 35 Pancake puzzle the optimal solutions were obtained by having IDA* with the GAP heuristic solve the problems. For this domain we also present results of BiSS using the GAP heuristic to define its type system (BiSS+G in Table 4). All methods tested produced accurate predictions for this domain. BiSS+G is the prediction method which produces the least accurate predictions, although the predictions are still accurate (largest error observed is 0.10 for cost 29). However, BiSS+G would produce predictions as accurate as the other methods should we use $\gamma = 0.4$ instead of 0.5 (results for $\gamma = 0.4$ are not included for clarity and uniformity). We further discuss parameter selection in Section 4.8.

For Rubik's Cube we compare BiSS's predictions with Bootstrap and with the maximum of the three pattern database heuristics introduced by Korf [25]. We also used Korf's pattern database heuristics with IDA* to solve 1000 problem instances generated with a random walk of length 30 from the goal state. The prediction results we report are only for the instances which IDA* managed to solve with a 30-minute time limit. As shown in Table 5, BiSS is able to produce predictions which are far superior to those produced by Bootstrap and the pattern database heuristic.

4.6. 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 (Section 3.4). Table 6 shows how this time complexity translates into actual runtime ($p = 2$ and $\gamma = 0.5$) by showing the fastest (min), the average (mean), and slowest (max) prediction runtimes for the set of problem instances used in the accuracy experiment above.

BiSS is considerably faster than solving the problem suboptimally; the mean times for Bootstrap to suboptimally solve one single instance were 3 hours and 49 minutes for the 20 Blocks World, 14 minutes and 5 seconds for the 24-puzzle, 2 hours and 29 minutes for the 35 Pancake puzzle, and 10 hours and 54 minutes for Rubik's Cube [20].

4.7. Predictions for very large state spaces

We also used BiSS (again using $p = 2$, $\gamma = 0.5$, and the T_{gc} type system with Manhattan Distance) to predict the optimal solution cost of problem instances for the $(n \times n)$-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

Table 7
BiSS's predicted average optimal solution cost for very large state spaces.

	Configuration		
	6×6	7×7	8×8
# instances	1000	650	50
BiSS time (minutes)	6	18	80
BiSS avg predicted cost	172	280	423
Polynomial predicted cost	171	268	397

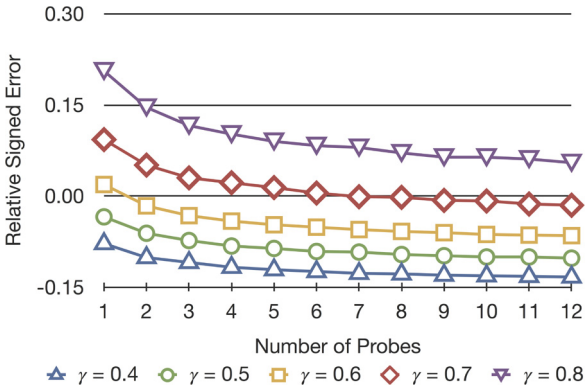

Fig. 5. Each curve represents the relative signed error for different values of γ and number of probes.

were made and the average time (in minutes) taken by BiSS to compute one prediction are shown in the first two rows of Table 7. 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 7.

Parberry [39] 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 cubic polynomial to the known average solution costs and then infer the unknown average solution costs. The average solution cost for the (2×2), (3×3), and (4×4) puzzles are roughly 3, 22, and 53, respectively. The average solution cost obtained from solving the 433 instances of the (5×5) puzzle in Section 4.5 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 7, are very close to BiSS's average predictions.

4.8. 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 and running time 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 2 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 Fig. 5, where each curve shows the average relative signed error of the predictions over 1000 random instances for different values of γ. 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.

Fig. 6 presents the relative unsigned errors of our parameter selection experiment. Increasing the number of probes for small values of γ will increase the error. In particular, the error increases as we increase the number of probes for the smaller values of γ (0.4, 0.5, and 0.6). By contrast, as we increase the number of probes we decrease the error for the larger values of γ (0.7 and 0.8). However, it is likely that by continually increasing the number of probes the error will also increase for the larger values of γ.

We observed trends similar to the ones shown in Figs. 5 and 6 in all other domains.

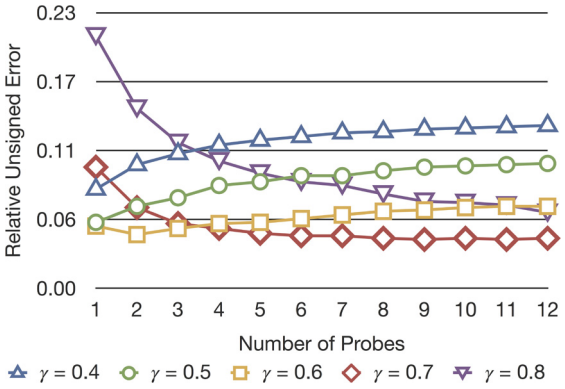

Fig. 6. Each curve represents the relative unsigned error for different values of γ and number of probes.

5. Learning strong heuristic functions from predictions

The use of machine learning to learn a heuristic function is an active research topic, see for instance [3,19,11,20,44,52]. One of the main challenges faced by these learning systems is to collect training instances. We now describe one successful system developed for learning strong heuristic functions for large state spaces called Bootstrap. Then we are going to show how to use BiSS to dramatically reduce the training time required by Bootstrap to learn strong heuristic functions.

5.1. The Bootstrap learning system

Jabbari Arfaee et al. [20] 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 with a heuristic learned by the Bootstrap system.

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 (or BS, as used in Section 4 above).

5.1.1. The Bootstrap drawback

Although the Bootstrap system is able to learn strong heuristic functions for large state spaces, the process of learning can be time-consuming. For example, the Bootstrap system takes two days to learn a strong heuristic for Rubik's Cube. The learning process is time-consuming for two reasons. First, Bootstrap spends a substantial amount of time trying and failing to solve problem instances to form its 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 methods for predicting the IDA* search tree size such as CDP [53] and SS [7]. 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 instances quickly.

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 first generate a set of problem instances and then we use BiSS to label these instances to form our training set.

Algorithm 5 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 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. Similarly, lower values of γ will require a smaller overlap of the frontiers, and the predictions will tend to underestimate the optimal solution cost. Aiming at learning

Algorithm 5 BiSS-h.

Input: goal state g, number of probes p, match parameter γ, type system T
Output: heuristic function h
1: Collect a set I of problem instances
2: $\Gamma \leftarrow \emptyset$
3: **for** each $i \in I$ **do**
4: $\quad c_i \leftarrow \texttt{BiSS}(i, g, p, \gamma, T)$
5: $\quad \Gamma \leftarrow \Gamma \cup \{(i, c_i)\}$
6: **end for**
7: learn h from Γ

a heuristic that produces near-optimal estimates of the cost-to-go, we choose a value of γ that will produce near-optimal predictions. Namely, we use $\gamma = 0.5$.

In line 1 BiSS-h collects the set of instances I 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. We describe how the training instances were generated for each domain in Appendix A. 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 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 Γ is used for learning a heuristic function h.

5.2.1. Using neural networks

The learning algorithm we use in our experiments is the same 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 learning features, which we describe in Appendix A.

5.3. Experimental evaluation of BiSS-h

5.3.1. Experimental setup

Again our experiments are on the 20 Blocks World, 24-puzzle, 35 Pancake puzzle, and Rubik's Cube.

We compare BiSS-h solely to Bootstrap because (1) we wanted to show that a solution cost predictor is able to reduce the time required for learning heuristic functions, and (2) in terms of search performance, IDA* using a Bootstrap heuristic outperforms standard suboptimal search algorithms on the domains tested [20]. Both systems are tested on the same test instances.

As we are interested in learning effective heuristics quickly, we use BiSS-h with the smallest training sets reported by Jabbari Arfaee et al. [20] (500 training instances). Smaller training sets result in shorter learning time.

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. All heuristics were tested using the same implementation of IDA*.

5.3.2. Experimental results

Table 8 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 8 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 – he fastest known average optimal solving time for random instances of Rubik's Cube is due to Zahavi et al. [54]: 12 hours, 16 minutes and 41 seconds on average. We note that Zahavi et al. used 3 GHz CPUs in their experiments, while we used 2.67 GHz Intel Xeon CPUs. Thus, our learning and solving times should be expected to be the same or slightly lower than the ones presented in Table 8 if we used the same machines used by Zahavi et al. Moreover, if using 200 training instances instead of 500 we can learn a strong heuristic in only 21 minutes for this domain while keeping the solving time roughly the same.

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 to those presented in Table 8, 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

Table 8
Learning time, solving time, and suboptimality of BS and BiSS-h.

Algorithm	Learning time	Solving time	Suboptimality
24-puzzle			
BS	11 hours 43 minutes	64 seconds	5.7
BiSS-h	17 minutes	77 seconds	5.0
35 Pancake puzzle			
BS	1 day 11 hours	48 seconds	5.5
BiSS-h	33 minutes	33 seconds	5.0
Rubik's Cube			
BS	2 days	2 hours 17 minutes	4.0
BiSS-h	53 minutes	35 minutes	11.4

35 Pancake puzzle in 1 hour and 42 minutes; and an instance of Rubik's Cube in 10 hours and 54 minutes [20]. 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.

5.3.3. The selection of training instances matters

The BiSS-h results shown in Table 8 were generated by using a strategy which selected a mix of easy-to-solve (generated with short random walks from the goal) and harder-to-solve (generated randomly whenever possible) training instances. IDA* solved only 10 out of the 50 test instances with a time limit of one hour per instance when using this strategy with the 20 Blocks World. BiSS-h clearly failed to learn a strong heuristic in this case.

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: learning time of 7 hours (here the learning time represents the time required by BiSS to label the training instances), solving time of 1.4 seconds, and suboptimality of 8.8. By contrast, Bootstrap requires 11 days and 1 hour to complete learning, IDA* using the resulting Bootstrap heuristic finds solutions with average solving time of 23 seconds, and average suboptimality of 9.6% [20].

The failure of BiSS-h on the 20 Blocks World with our standard set of training instances suggests that the selection of the training instances is crucial in the learning process. As shown in Table 2 BiSS is able to accurately assign labels to the set of instances. Investigating how to select the training instances to learn effective heuristic functions is an interesting and challenging direction of future work.

6. Limitations of BiSS

BiSS has a few weaknesses. First, BiSS does not have error bounds that are applicable in practice. Such bounds could be critical in real-world applications in which one needs to know how inaccurate the predictions can be in the worst case. Currently one has to trust the empirical accuracy of the predictions.

Second, BiSS is able to produce predictions only for domains with single goal states. This can potentially preclude the application of BiSS to domain-independent planning, for instance, where a set of goal conditions is provided instead of a goal state. If there are only a small number of goal states, the backward search can be initialized with all of them, so that each BiSS probe will estimate the optimal cost of reaching the nearest goal state.

7. Conclusions

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 the 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. 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.

In this paper we also 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 learn effective heuristics much faster than Bootstrap – a learning method that uses a search algorithm to generate the training set. Our system reduces the time required by Bootstrap to learn effective heuristics from days to minutes. The batch learning process of BiSS-h is even faster than Bootstrap's interleaving process for solving a single instance.

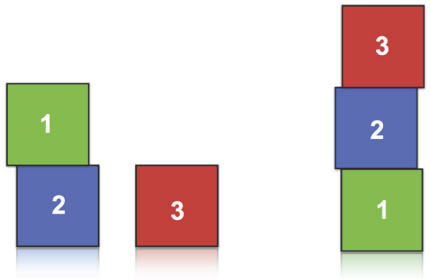

Fig. 7. Two states of the blocks world with three blocks.

	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

Fig. 8. The goal state for the 15-puzzle (left) and a state two moves from the goal (right).

Speeding up the process of learning heuristic functions is just one of the applications of a solution cost predictor. As two other examples, SCP has been applied for setting the w-value of Weighted IDA* and the bound for Potential Search [47]; in both applications we observed search speedups [29]. Exploring other applications of BiSS is an exciting direction of future research.

Acknowledgements

We thank Rong Zhou for providing the optimal solution cost for the instances of the 24-puzzle used in our experiments. This work was supported by the Laboratory for Computational Discovery at the University of Regina, Alberta Innovates – Technology Futures, the Alberta Innovates Centre for Machine Learning, Canada's NSERC, and Brazil's CAPES, FAPEMIG, and Science Without Borders.

Appendix A. Problem domains

A.1. Blocks world

In the blocks world domain one is given a set of n distinct blocks and the goal is to have the blocks stacked up in a specific order. The blocks world is illustrated in Fig. 7, 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.

A solution for the problem shown in Fig. 7 is to unstack block 1 from block 2; stack block 2 on block 1; finally, stack block 3 on block 2.

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.

Optimal solutions on the Blocks World were obtained with Slaney and Thiébaux's [45] 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. [20].

A.1.1. Learning setup

The features used for the NN were seven 2-block pattern databases, the number of out of place blocks, and the number of stacks of blocks. BiSS uses the T_c type system with the Out of Place heuristic to predict the optimal solution cost of the instances in I. We discuss how the training instances in I were selected in Section 5.3.3. We used 50 random instances as the test set in the experiment presented in Section 5.3.

A.2. Sliding-Tile puzzle

The Sliding-Tile puzzle [46] 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 kth 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 Fig. 8 shows the goal state that we used for the 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 $(n \times m)!/2$, cf. [1].

We used T_{gc} with Manhattan Distance (MD) as the type system in our experiments with the Sliding-Tile puzzle.

1	2	3	4	5	···	14	15
4	3	2	1	5	···	14	15

Fig. 9. The goal state for the 15-pancake puzzle (above) and a state one move from the goal (below).

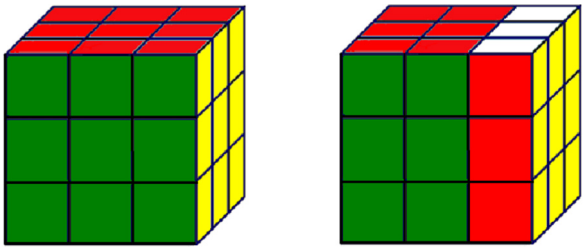

Fig. 10. Rubik's Cube (modified from Zahavi et al. [53]).

A.2.1. Learning setup

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 databases. BiSS used 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; 50 random problem instances constituted the test set.

A.3. Pancake puzzle

In the Pancake puzzle [9] with parameter n, a state is a permutation of n numbered tiles and has $n-1$ successors, with the lth 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 Fig. 9 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.

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

One may think of each tile as a pancake and each permutation as a pile of pancakes that have to be sorted into the goal permutation. To move a pancake from position p into position p' in the pile, all the pancakes stacked from position p to position p' have to be flipped together.

In our experiments on the 35 Pancake puzzle the 5–5–5–5–5–5–5 additive pattern database heuristic [51] was used to construct a "coarser" version of the T_c type system. 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. We also experimented with a coarser T_c type system using the GAP heuristic [15] instead of the additive pattern databases.

A.3.1. Learning setup

The input features for the NN were seven 5-token pattern databases (but instead of using the regular lookup from the pattern databases, we use the maximum of the regular and the dual lookups [54]), a binary value indicating whether the middle pancake is out of place, and the number of the largest out-of-place pancake. BiSS used the T_c type system with the additive pattern databases as heuristic functions [51]. The training instances for the 35 Pancake puzzle were generated exactly in the same way they were generated for the Sliding-Tile puzzle. We used 50 random instances as the test set in the experiment described in Section 5.3.

A.4. 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 [25]. Each face of the cube can be independently rotated by 90 degrees clockwise or counterclockwise, or by 180 degrees. The left part of Fig. 10 shows the goal state for Rubik's Cube while the right part shows the state produced by rotating the right face 90 degrees counterclockwise.

We use the T_c type system with the maximum of three pattern databases [25] as heuristic function. However, when computing $T_c(s)$ (see Equation (2)), instead of using $h(s)$ as the maximum of the three pattern databases, we use the values of each pattern database separately in the type system; we use the maximum of the three pattern databases when computing T_c's $c(\cdot,\cdot)$-values, however.

A.4.1. Learning setup

The features used for Rubik's Cube are the three pattern databases introduced by Korf [25]. The test set used in the experiment described in Section 5.3 consists of Korf's 10 instances [25]. BiSS uses the T_c type system described above in

Appendix A.4. For this domain, we generated 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.

References

[1] A.F. Archer, A modern treatment of the 15-puzzle, Am. Math. Mon. 106 (1999) 793–799.
[2] B. Bonet, H. Geffner, Planning as heuristic search, Artif. Intell. 129 (2001) 5–33.
[3] A. Bramanti-Gregor, H.W. Davis, The statistical learning of accurate heuristics, in: Proceedings of the International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1993, pp. 1079–1087.
[4] N. Burch, R.C. Holte, M. Müller, D. O'Connell, J. Schaeffer, Automating layouts of sewers in subdivisions, in: Proceedings of the European Conference on Artificial Intelligence, IOS Press, 2010, pp. 655–660.
[5] E. Burns, W. Ruml, Iterative-deepening search with on-line tree size prediction, in: Proceedings of the International Conference on Learning and Intelligent Optimization, 2012, pp. 1–15.
[6] P.C. Chen, Heuristic sampling on backtrack trees, Ph.D. thesis, Stanford University, 1989.
[7] P.C. Chen, Heuristic sampling: a method for predicting the performance of tree searching programs, SIAM J. Comput. 21 (1992) 295–315.
[8] J.C. Culberson, J. Schaeffer, Searching with pattern databases, in: Proceedings of the Canadian Conference on Artificial Intelligence, Springer, 1996, pp. 402–416.
[9] H. Dweighter, Problem E2569, Am. Math. Mon. 82 (1975) 1010.
[10] S. Edelkamp, Prediction of regular search tree growth by spectral analysis, in: Proceedings of the Advances in Artificial Intelligence, Joint German/Austrian Conference on AI (KI), 2001, pp. 154–168.
[11] M. Ernandes, M. Gori, Likely-admissible and sub-symbolic heuristics, in: Proceedings of the European Conference on Artificial Intelligence, 2004, pp. 613–617.
[12] L.R. Harris, The heuristic search under conditions of error, Artif. Intell. 5 (1974) 217–234.
[13] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. SSC-4 (2) (1968) 100–107.
[14] A. Heifets, I. Jurisica, Construction of new medicines via game proof search, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 2012, pp. 1564–1570.
[15] M. Helmert, Landmark heuristics for the pancake problem, in: Proceedings of the Symposium on Combinatorial Search, AAAI Press, 2010, pp. 109–110.
[16] M. Helmert, P. Haslum, J. Hoffmann, Flexible abstraction heuristics for optimal sequential planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2007, pp. 176–183.
[17] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[18] R.C. Holte, Common misconceptions concerning heuristic search, in: Proceedings of the Symposium on Combinatorial Search, AAAI Press, 2010, pp. 46–51.
[19] T. Humphrey, A. Bramanti-Gregor, H.W. Davis, Learning while solving problems in single agent search: preliminary results, in: Proceedings of the Italian Association for Artificial Intelligence on Topics in Artificial Intelligence, Springer, 1995, pp. 56–66.
[20] S. Jabbari Arfaee, S. Zilles, R.C. Holte, Learning heuristic functions for large state spaces, Artif. Intell. 175 (2011) 2075–2098.
[21] D.E. Knuth, Estimating the efficiency of backtrack programs, Math. Comput. 29 (1975) 121–136.
[22] R. Korf, Divide-and-conquer frontier search applied to optimal sequence alignment, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 2000, pp. 910–916.
[23] R. Korf, A new algorithm for optimal bin packing, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 2002, pp. 731–736.
[24] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[25] R.E. Korf, Finding optimal solutions to Rubik's cube using pattern databases, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 1997, pp. 700–705.
[26] R.E. Korf, A. Felner, Disjoint pattern database heuristics, Artif. Intell. 134 (2002) 9–22.
[27] R.E. Korf, M. Reid, S. Edelkamp, Time complexity of iterative-deepening-A*, Artif. Intell. 129 (2001) 199–218.
[28] L. Lelis, R. Stern, A. Felner, S. Zilles, R.C. Holte, Predicting optimal solution cost with bidirectional stratified sampling, in: Proceedings of the International Conference on Automated Planning and Scheduling, AAAI Press, 2012, pp. 155–163.
[29] L. Lelis, R. Stern, S. Jabbari Arfaee, Predicting solution cost with conditional probabilities, in: Proceedings of the Symposium on Combinatorial Search, AAAI Press, 2011, pp. 100–107.
[30] L.H.S. Lelis, S. Jabbari Arfaee, S. Zilles, R.C. Holte, Learning heuristic functions faster by using predicted solution costs, in: Proceedings of the Symposium on Combinatorial Search, AAAI Press, 2012, pp. 166–167.
[31] L.H.S. Lelis, L. Otten, R. Dechter, Predicting the size of depth-first branch and bound search trees, in: International Joint Conference on Artificial Intelligence, 2013, pp. 594–600.
[32] L.H.S. Lelis, L. Otten, R. Dechter, Memory-efficient tree size prediction for depth-first search in graphical models, in: Proceedings of the Principles and Practice of Constraint Programming, 2014, pp. 481–496.
[33] L.H.S. Lelis, R. Stern, A. Felner, S. Zilles, R.C. Holte, Predicting optimal solution cost with conditional probabilities, Ann. Math. Artif. Intell. 72 (2014) 267–295.
[34] L.H.S. Lelis, S. Zilles, R.C. Holte, Predicting the size of IDA*'s search tree, Artif. Intell. 196 (2013) 53–76.
[35] L.H.S. Lelis, S. Zilles, R.C. Holte, Stratified tree search: a novel suboptimal heuristic search algorithm, in: Proceedings of the Conference on Autonomous Agents and Multiagent Systems, 2013, pp. 555–562.
[36] Y. Li, J.J. Harms, R.C. Holte, IDA* MCSP: a fast exact MCSP algorithm, in: Proceedings of the International Conference on Communications, IEEE Press, 2005, pp. 93–99.
[37] M. Likhachev, D. Ferguson, Planning long dynamically feasible maneuvers for autonomous vehicles, Int. J. Robot. Res. 28 (2009) 933–945.
[38] N. Nilsson, Principles of Artificial Intelligence, Morgan Kaufmann, 1980.
[39] I. Parberry, A real-time algorithm for the $(n^2 - 1)$-puzzle, Inf. Process. Lett. 56 (1995) 23–28.
[40] J. Paudel, L.H.S. Lelis, J.N. Amaral, Stratified sampling for even workload partitioning applied to IDA* and Delaunay algorithms, in: International Parallel and Distributed Processing Symposium, IEEE Press, 2015, pp. 460–469.
[41] A.E. Prieditis, Machine discovery of effective admissible heuristics, Mach. Learn. 12 (1993) 117–141.
[42] S. Richter, M. Helmert, Preferred operators and deferred evaluation in satisficing planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2009, pp. 273–280.
[43] S. Richter, M. Helmert, M. Westphal, Landmarks revisited, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 2008, pp. 975–982.
[44] M. Samadi, A. Felner, J. Schaeffer, Learning from multiple heuristics, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI Press, 2008, pp. 357–362.

[45] J. Slaney, S. Thiébaux, Blocks world revisited, Artif. Intell. 125 (2001) 119–153.
[46] J. Slocum, D. Sonneveld, The 15 Puzzle, Slocum Puzzle Foundation, 2006.
[47] R. Stern, R. Puzis, A. Felner, Potential search: a bounded-cost search algorithm, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2011, pp. 234–241.
[48] N.R. Sturtevant, A. Felner, M. Barrer, J. Schaeffer, N. Burch, Memory-based heuristics for explicit state spaces, in: Proceedings of the International Joint Conference on Artificial Intelligence, 2009, pp. 609–614.
[49] J. Thayer, A. Dionne, W. Ruml, Learning inadmissible heuristics during search, in: Proceedings of the International Conference on Automated Planning and Scheduling, 2011, pp. 250–257.
[50] D. Tolpin, T. Beja, S.E. Shimony, A. Felner, E. Karpas, Towards rational deployment of multiple heuristics in A*, in: Proceedings of the International Joint Conference on Artificial Intelligence, AAAI Press, 2013, pp. 674–680.
[51] F. Yang, J.C. Culberson, R.C. Holte, U. Zahavi, A. Felner, A general theory of additive state space abstractions, J. Artif. Intell. Res. 32 (2008) 631–662.
[52] S.W. Yoon, A. Fern, R. Givan, Learning control knowledge for forward search planning, J. Mach. Learn. Res. 9 (2008) 683–718.
[53] U. Zahavi, A. Felner, N. Burch, R.C. Holte, Predicting the performance of IDA* using conditional distributions, J. Artif. Intell. Res. 37 (2010) 41–83.
[54] U. Zahavi, A. Felner, R.C. Holte, J. Schaeffer, Duality in permutation state spaces and the dual search algorithm, Artif. Intell. 172 (2008) 514–540.