Ann Math Artif Intell
DOI 10.1007/s10472-014-9432-8

Predicting optimal solution cost with conditional probabilities

Predicting optimal solution cost

Levi H. S. Lelis · Roni Stern · Ariel Felner · Sandra Zilles · Robert C. Holte

Abstract Heuristic search algorithms are designed to return an optimal path from a start state to a goal state. They find the optimal solution cost as a side effect. However, there are applications in which all one wants to know is an estimate of the optimal solution cost. The actual path from start to goal is not initially needed. For instance, one might be interested in quickly assessing the monetary cost of a project for bidding purposes. In such cases only the cost of executing the project is required. The actual construction plan could be formulated later, after bidding. In this paper we propose an algorithm, named Solution Cost Predictor (SCP), that accurately and efficiently predicts the optimal solution cost of a problem instance without finding the actual solution. While SCP can be viewed as a heuristic function, it differs from a heuristic conceptually in that: 1) SCP is not required to be fast enough to guide search algorithms; 2) SCP is not required to be admissible; 3) our measure of effectiveness is the prediction accuracy, which is in contrast to the solution quality and number of nodes expanded used to measure the effectiveness of heuristic functions. We show empirically that SCP makes accurate predictions on several heuristic search benchmarks.

Keywords Optimal solution cost prediction · Type systems · Heuristic search

Mathematics Subject Classifications (2010) 68T20 · 68W20 · 68W25

This work was carried out while L. H. S. Lelis was at the University of Alberta.

L. H. S. Lelis (✉)
Departamento de Informática, Universidade Federal de Viçosa, Viçosa, Minas Gerais, Brazil
e-mail: levi.lelis@ufv.br

R. Stern · A. Felner
Information Systems Engineering, Ben Gurion University, Beer-Sheva, Israel

S. Zilles
Department of Computer Science, University of Regina, Regina, SK, Canada

R. C. Holte
Computing Science Department, University of Alberta, Edmonton, AB, Canada

Published online: 03 October 2014

1 Introduction

Heuristic search algorithms such as A* [11] and IDA* [17] are guided by the cost function $f(s) = g(s) + h(s)$, where $g(s)$ is the lowest cost path known from the start state to s and $h(s)$, known as the heuristic function, is an estimate of the lowest cost path from s to a goal. A* and IDA* are designed to find a path from a start state to a goal state. They find the solution cost as a side effect. However, there are applications in which all one wants is to know the optimal solution cost or an accurate estimate of the optimal solution cost – the actual path is not needed. For example, consider an owner of a construction company that is required to quickly assess the monetary cost of a project for bidding purposes. In such a case, only the cost of executing the project is required. The actual construction plan could be formulated later, if the bid is won.

Thus, an important question to be answered is the following. Can we accurately and efficiently predict the optimal solution cost of a problem without finding a solution path from the start to a goal? Korf et al. [22] stated: "Predicting the optimal solution cost for a given problem instance (...) is an open problem".

In this paper we present an efficient solution for this problem. We show empirically that our method accurately predicts the optimal solution cost of problem instances in different heuristic search benchmark domains. Our solution, named Solution Cost Predictor (SCP), is based on the prediction formula by Zahavi et al. [39], named Conditional Distribution Prediction (CDP), which predicts the number of nodes expanded on an iteration of IDA* for a given cost bound. We extend the ideas of Zahavi et al. to predict the optimal solution cost of a problem instead of predicting the number of nodes expanded by IDA*.

The heuristic function $h(s)$ 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. This distinction is important, since heuristic search algorithms, such as A* and IDA*, guided by the cost function $f = g + h$ are guaranteed to find an optimal solution when h is admissible [11, 17]. A considerable amount of effort has been devoted to creating admissible heuristics [6, 13, 35, 38] and inadmissible heuristics [8, 14, 32, 36]. As shown in our experimental results, admissible heuristics usually provide inaccurate predictions of the optimal solution cost as they are biased to never overestimate the actual value. In some cases even inadmissible heuristics are biased towards admissibility [8, 32].

Regardless of admissibility, heuristics share a property: the heuristic evaluation must be fast enough to be computed for every state generated during the search,[1] while the solution cost predictor is run only on the start state. In fact, often heuristic functions sacrifice accuracy for speed. By contrast, the algorithm presented in this paper aims at accurately predicting the optimal solution cost of a problem instance. In summary, while SCP can be viewed as a heuristic, it differs from a heuristic conceptually in that: 1) SCP is not required to be fast enough to guide search; 2) SCP does not favor admissibility; 3) SCP aims at making accurate predictions and thus our measure of effectiveness is the prediction accuracy, in contrast to the solution quality and number of nodes expanded used to measure the effectiveness of heuristic functions.

[1] In some settings it is more efficient to perform heuristic calculation lazily during node expansion [31].

The paper is structured as follows. In Section 2, we describe in detail the CDP formula for predicting the number of nodes expanded by IDA*, which is the basis of our prediction algorithm. Section 3 presents SCP. Section 4 presents experimental results showing the accuracy of the predictions made by SCP. In Section 5 we study empirically the parameters required by SCP. In Section 6 we mention additional applications of SCP. In Section 7 we discuss the related work. Finally, in Section 8 we draw the concluding remarks of the work presented in this paper.

This paper substantially extends a conference paper [24], which is to the best of our knowledge the seminal work on efficiently predicting the optimal solution cost of individual problem instances. In addition to a comprehensive explanation of the algorithm, we include new experimental and theoretical results. We also include an empirical study of SCP's input parameters.

2 The CDP prediction framework

In this paper we use the term "search tree" of a search algorithm to refer to the nodes generated by that search algorithm until it halts. A node represents a unique state in the state-space, and a state s might be represented by more than one node in a search tree, since there might exist more than one path from the start state to s. In addition, in contrast with states, nodes have a g-cost and, if not the root of a search tree, nodes also have a parent node. When clear from the context, we use the terms nodes and states interchangeably. We assume unit-cost edges throughout.

The prediction algorithm presented in this paper is based on the CDP formula by Zahavi et al. [39]. In this section we introduce relevant notation and review CDP. Note that SCP predicts the optimal solution cost, while CDP predicts the number of nodes expanded on an iteration of IDA* for a given cost bound.

2.1 Type systems

The CDP formula, as well as the SCP prediction algorithm presented in this paper, are based on a partition of the nodes in a search tree named *type system*.

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

The accuracy of the CDP formula is based on the assumption that two nodes of the same type root subtrees of the same size. IDA* with parent pruning will not generate a node $\hat{s}$ from s if $\hat{s}$ is the parent of s. Therefore, because of parent pruning, the subtree below a node s differs depending on the parent from which s was generated. Zahavi et al. use the information about the parent of s when computing s's type so that CDP is able to make accurate predictions of the number of nodes expanded by IDA* when parent pruning is used.

Definition 2 (**Heuristic-preserving type system**) A type system T is said to be heuristic-preserving if for every type, all nodes of that type have the same heuristic value. We then write $h(t)$ for any type $t \in T$ to denote the heuristic value of the nodes of type t.

As in Zahavi et al.'s work, all type systems considered in this paper are heuristic-preserving.

Definition 3 Let $t, t' \in T$. $p(t'|t)$ denotes the average fraction of the children generated by a node of type t that are of type t'. b_t is the average number of children generated by a node of type t.

For example, if a node of type t generates 5 children on average ($b_t = 5$) and 2 of them are of type t', then $p(t'|t) = 0.4$. CDP samples the state space in order to estimate $p(t'|t)$ and b_t for all $t, t' \in T$. CDP does its sampling as a preprocessing step and although type systems are defined for nodes in 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 the parent of nodes in a search tree. As explained above, due to parent-pruning, CDP uses the information about the parent of a node n when computing n's type. Therefore, when estimating the values of $p(t'|t)$ and b_t the sampling is done based on the children of the state s drawn randomly from the state space. We denote by $\pi(t'|t)$ and β_t the respective estimates thus obtained. The values of $\pi(t'|t)$ and β_t are used to estimate the number of nodes expanded on an iteration of IDA*. The following example illustrates the prediction process.

Example 1 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. We generate the children of s_0, depicted in the figure by s_1 and s_2, so that the types that will seed the prediction formula can be calculated. 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

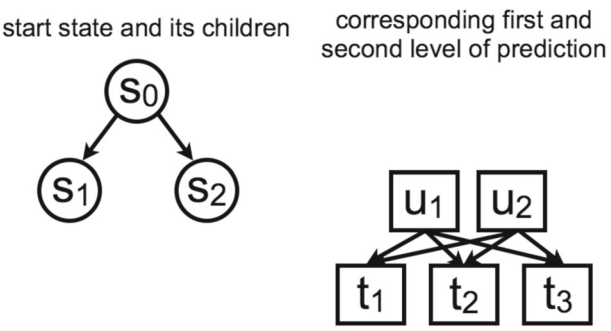

Fig. 1 The first step of a CDP prediction for start state s_0

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.

2.2 The CDP formula

Next, we describe the CDP prediction formula formally. Let $child(n)$ be the set of children of node n in the search tree. The predicted number of nodes expanded by IDA* according to the CDP formula, for start state s^*, cost bound d, heuristic h, and type system T is as follows.

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

Here the first summation iterates over the nodes that seed the prediction formula; these are the children of the start state s^* ($child(s^*)$). In the example above, these nodes were s_1 and s_2. In the second summation we account for g-costs from 1 to the cost bound d, assuming nonnegative heuristic estimates; any value of i greater than d would be pruned by IDA*. The innermost summation iterates over the types in T. Finally, $N(i, t, s, d)$ is the number of nodes n of type t at level i of the subtree rooted at s. A value of one is added to 1 as CDP expands the start state so that the type of its children can be computed. $N(i, t, s, d)$ is computed recursively as follows.

$$N(1, t, s, d) = \begin{cases} 0 & \text{if } T(s) \neq t\,, \\ 1 & \text{if } T(s) = t\,. \end{cases} \tag{2}$$

The case $i = 1$ is the base of the recursion and is calculated based on the types of the children of the start state. In the example in Fig. 1 $N(1, t, s, d)$ is 1 only for types u_1 and u_2, and 0 for all other types in T. For $i > 1$,

$$N(i, t, s, d) = \sum_{u \in T} N(i-1, u, s, d)\pi(t|u)\beta_u \cdot Prune(t, i, d)\,. \tag{3}$$

Here $\pi(t|u)\beta_u$ is the estimated number of states of type t a node of type u generates; $Prune()$ is a pruning function that is 1 if the cost to reach a pair of type t plus the heuristic cost estimate of reaching the goal from there is less than or equal to the cost limit d, i.e., $Prune(t, i, d) = 1$ if $h(t) + i \leq d$, and is 0 otherwise.

In summary, the CDP formula given above 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 until the cost bound d. This is done incrementally. First the number of nodes of each type at the first level of the search is given by the types of the children of the start state. The number of instances of each type is then estimated for the second level of the search based on (3). This prediction process continues to deeper and deeper levels until reaching the level of the cost bound d. The sum of the nodes of different types at every level is an estimate of the total number of nodes predicted by CDP.

Zahavi et al. [39] introduced a method for improving CDP predictions for a given problem instance. Instead of directly predicting how many nodes will be expanded for cost bound d and start state s^*, all states at depth $r < d$, denoted by C_r, are generated and one then predicts how many nodes will be expanded for depth bound $d - r$ with C_r as the set of

start states. We call this technique *prediction lookahead*, with r being the parameter that determines the depth of the *prediction lookahead*. SCP also uses the prediction lookahead to improve the accuracy of its predictions.

2.3 Different type systems

As defined above, a type system can be any partition over nodes of a search tree (Definition 1). Generally speaking, the type of node s can be any set of features of s. In this paper we use domain-independent type systems defined using a given heuristic function, as described next.

As our basic type system, T_h, we use Zahavi et al.'s basic "two-step" model, defined (in our notation) as $T_h(s) = (h(parent(s)), h(s))$, where $parent(s)$ returns the node that generated node s.

In addition, we also experimented with the following two type systems, which are extensions of the T_h type system.

- $T_c(s) = (T_h(s), c(s, 0), \ldots, c(s, H))$, where $c(s, k)$ is the number of children of s, considering parent pruning, whose h-value is k, and H is the maximum h-value observed in the sampling process;
- $T_{gc}(s) = (T_c(s), gc(s, 0), \ldots, gc(s, H))$, where $gc(s, k)$ is the number of grandchildren of s, considering parent pruning, whose h-value is k.

For instance, two nodes s and s' will be of the same T_c type (where c stands for children) if $h(parent(s)) = h(parent(s'))$ and $h(s) = h(s')$, and, in addition, s and s' generate the same number of children with the same heuristic distribution. Similarly, two nodes s and s' are of the same T_{gc} (where gc stands for grandchildren) type if besides matching on the information required by the T_c type system, s and s' generate the same number of grandchildren with the same heuristic distribution. We specify and justify the choice of each type system used in our experiments in Section 4 below.

Recently, we identified a source of error in the CDP formula that had previously been overlooked [25]. We observed that "rare events" (i.e., low values of $\pi(t|t')$) could reduce CDP's prediction accuracy. In addition, we presented a method that systematically disregards rare events, named ϵ-truncated CDP, that substantially improves prediction accuracy. In this paper we have also implemented ϵ-truncation as part of the solution cost prediction algorithm described in the next section. We have empirically observed (see Section 5.3) that ϵ-truncation also improves the prediction accuracy of SCP.

2.4 Type systems are not abstractions

A common misconception is to think of type systems as state-space abstractions. Prieditis [30] 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 search tree. It might not even be possible to represent a type system as a graph since there might not be edges connecting types, a type system does not necessarily define the relation between the types.

However, note that abstractions also offer a partition of the nodes in the search tree. Therefore, although type systems may not be used as abstractions in some cases, abstractions can always be used as type systems.

3 Solution cost predictor

We now describe the Solution Cost Predictor, SCP, a method based on CDP for estimating the optimal solution cost of state-space search problems.

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

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

Type space 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 search tree. 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.

3.1 The SCP prediction formula

Like CDP, SCP also samples the state space of a given problem with respect to a type system as a preprocessing step to find the values of $\pi(t|u)$ and β_u (see Section 2). After sampling, SCP predicts the optimal solution cost for a given start state s^* based on the values of $\pi(t|u)$ and β_u.

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

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

Example 2 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 other

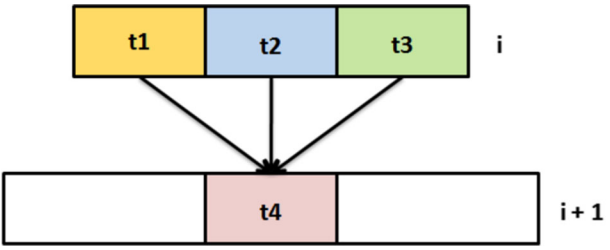

Fig. 2 Types at level i are used to calculate the approximated probability of t_4 existing at level $i+1$

words, for any type $t \notin \{t_1, t_2, t_3\}$, we have that $\pi(t_4|t) = 0$. A node of type t_4 exists at level $i+1$ iff at least one of the nodes of types t_1, t_2 or t_3 at the previous level generates one or more instances of the type t_4.

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

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

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

$$p(i, t \rightarrow t', s^*, d) = p(i, t, s^*, d)\big(1 - \phi(N(i, t, s^*, d) \cdot \beta_t, t, t')\big) \tag{4}$$

Recall that $N(i, t, s^*, d)$ is the number of nodes of type t at level i of an IDA* search tree rooted at s^* with cost bound d. In 4 the term we subtract from 1 is the probability that none of the $N(i, t, s^*, d) \cdot \beta_t$ nodes generated by nodes of type t are of type t'; this subtraction gives us the probability of at least one node of type t' being generated by nodes of type t.

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

$$p(i, t, s^*, d) = 1 - \prod_{u \in T} (1 - p(i-1, u \rightarrow t, s^*, d)) \tag{5}$$

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

Note that (4) and (5) are recursive as $p(i, t \rightarrow t', s^*, d)$ depends on $p(i, t, s^*, d)$, which in turn depends on $p(i-1, u \rightarrow t, s^*, d)$. The base of the recursion is defined for $i = 1$ where we know exactly the probability of finding a node of a given type as this corresponds to the type of the children of the start state.

$$p(1, t, s^*, d) = \begin{cases} 1 & \text{if there exists a child } s \text{ of } s^* \text{ such that } T(s) = t\,, \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

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

Next, we describe the SCP algorithm, which uses (5) presented above for estimating the probability of an instance of type t existing at level i of a search tree bounded by cost d.

3.2 The SCP Algorithm

SCP predicts the optimal solution cost as follows:

1. First, set the cost bound d to the heuristic value of the start state.
2. For every level i, estimate the probability of finding a goal type at that level in a search tree bounded by cost d, using (5).
3. Terminate returning d as the predicted optimal solution cost when a goal type exists at level i with probability higher than a threshold value c provided by the user. Otherwise, increase the cost bound d by one and go to step 2.

Algorithm 1 SCP

Require: start state s^*, threshold value c, type system T
Ensure: predicted optimal solution cost
1: $d \leftarrow h(s^*)$
2: initialize $N(1,t)$ and $p(1,t)$
3: **loop**
4: **for** $i = 2$ to d **do**
5: **for all** $t \in T$ **do**
6: $p(i,t) \leftarrow$ compute-probability(i,t,d) // see Algorithm 2
7: **if** t is a goal type AND $p(i,t) > c$ **then**
8: return d
9: **end if**
10: **end for**
11: **end for**
12: $d \leftarrow d + 1$
13: **end loop**

Algorithm 2 compute-probability

Require: level i, type $t \in T$, cost bound d
Ensure: the estimated probability of existing a node of type t at level i of the search tree
1: **if** $h(t) + i > d$ **then**
2: $p(i,t) = 0, N(i,t) = 0$
3: **end if**
4: $n \leftarrow 0$
5: $p \leftarrow 1$
6: **for all** $u \in T$ **do**
7: $n \leftarrow n + N(i-1,u) \cdot \pi(t|u) \cdot \beta_u$
8: $p(i-1, u \rightarrow t) \leftarrow p(i-1,u)\big(1 - \phi(N(i-1,u) \cdot \beta_u, t, u)\big)$
9: $p \leftarrow p \cdot (1 - p(i-1, u \rightarrow t))$
10: **end for**
11: $p(i,t) = 1 - p$
12: $N(i,t) = n$
13: return $p(i,t)$

SCP searches iteratively in the type space, incrementing the cost bound by one in case a goal type is not found with probability higher than c. This process is similar to how IDA* searches in the original state space. Algorithms 1 and 2 provide the pseudocode for SCP. The SCP algorithm starts in Algorithm 1. Initially, SCP uses the heuristic value of the start state s^* to initialize the cost bound d (line 1 of Algorithm 1). Initializing $N(1,t)$ and $p(1,t)$ (line 2) is done according to the children of the start state as shown in (2) and 6 for $N(1,t)$ and $p(1,t)$, respectively. $N(1,t)$ is the number of nodes of type t among the children of s^*. $p(1,t)$ is one if there is an instance of type t among the children of s^*, or zero otherwise.

For every level i (ranging from 2 to d), SCP calculates the values of $N(i,t)$ and $p(i,t)$ of every type t by calling Algorithm 2 (line 6 of Algorithm 1). Algorithm 2 computes the values of $N(i,t)$ as shown in line 7 and $p(i,t)$ as shown in line 9. Algorithm 2 returns $p(i,t)$. Every node of type t at level i for a given cost bound d will be pruned by the search if $i+h(t) > d$. Thus, in such cases we set $N(i,t)$ and $p(i,t)$ to zero (line 2 in Algorithm 2).

After every call to Algorithm 2 for a type t, if t is a goal type and $p(i,t)$ is above the user-defined threshold parameter c, the prediction halts and d is returned (line 8) as SCP's optimal solution cost prediction. The reason SCP returns d and not i is explained in the next section.

3.3 Conditions for producing perfect predictions

In this section we show certain conditions in which SCP is guaranteed to produce perfect estimates of the optimal solution cost. First, we define a *perfect type system* and then we show that SCP using such type system produces perfect predictions as long as each type is sampled at least once.

Definition 5 (**Perfect type system**) A type system T is perfect if (i) every goal node has a goal type and (ii) for every $t \in T$, all nodes $n \in t$ generate the same number of nodes and with the same distribution of types.

Let T^* be a perfect type system, and C^* the optimal solution cost.

Lemma 1 *In SCP, for any type t and level i, $N(i,t) > 0 \Leftrightarrow p(i,t) > 0$*

Proof We prove Lemma 1 by induction on i. In the base level (i=1), $N(1,t)$ and $p(1,t)$ are set according to the children of the initial state, i.e., $N(1,t)$ and $p(1,t)$ are both zero for all types except the types of the children of the initial state. Next, assume that Lemma 1 is true for all $i < n$, and prove it for $i = n$.

We first prove that $N(n,t) > 0 \Rightarrow p(n,t) > 0$. Let t be a type for which $N(n,t) > 0$. Following Algorithm 2, we observe that $N(n,t)$ is initialized to zero, and can only gain values larger than zero if there exists a type t' for which $N(n-1,t') \cdot \pi(t|t')\beta_{t'} > 0$ (see line 7). Both $\beta_{t'}$ and $\pi(t|t')$ are non-negative, and thus $N(n-1,t') > 0$. According to the induction assumption, this entails that $p(n-1,t') > 0$. Therefore $p(n-1, t' \rightarrow t) > 0$ and $p(n,t) > 0$ as required.

Next, we proof the other direction, i.e., $N(n,t) > 0 \Leftarrow p(n,t) > 0$. If t is a type for which $p(n,t) > 0$, then there exists a type t' for which

$$\begin{aligned} p(n-1,t') \cdot (1 - \phi(N(n-1,t') \cdot \beta_{t'}, t, t')) &> 0 \quad \text{(see line 9 of Algorithm 2)} \\ 1 - \phi(N(n-1,t') \cdot \beta_{t'}, t, t') &> 0 \\ (1 - \pi(t'|t))^{N(n-1,t') \cdot \beta_{t'}} &< 1 \end{aligned}$$

Since $\pi(t'|t)$, $N(n-1,t')$, and $\beta_{t'}$ cannot zero, then

$$\begin{aligned} 1 - \pi(t'|t) < 1 \quad &\wedge \quad N(n-1,t') \cdot \beta_{t'} > 0 \\ \pi(t'|t) > 0 \quad &\wedge \quad N(n-1,t') > 0 \quad \wedge \quad \beta_{t'} > 0 \end{aligned}$$

According to Algorithm 2, $N(n,t) \geq \pi(t'|t) \cdot \beta_{t'} \cdot N(n-1,t')$ (see line 7). Therefore, $N(n,t) > 0$ as required. □

The computation of $N(i,t)$, the number of states of type t predicted for level i, is exactly the same as in CDP. Zahavi et al. [39] (Section 4.5.1) showed that with a perfect type system, and assuming that all types have been sampled (when constructing π and β), then the predictions of CDP are exactly correct.

Lemma 2 *If SCP uses a perfect type system and every type has been sampled at least once, then there exists a goal type t_g such that $N(C^*,t_g) > 0$ and $p(C^*,t_g) > 0$. In addition, $N(i,t) = 0$ and $p(i,t) = 0$ for any goal type t and every level $i < C^*$.*

Proof Since C^* is the optimal solution, at least one goal state must occur at depth C^*. Let t_g be the type of that goal state. As the type system is perfect and all types have been sampled, CDP is exact, and thus $N(C^*,t_g) > 0$ and following Lemma 1 $p(C^*,t_g) > 0$. Similarly, there is no goal state at a depth lower than C^*. Thus, $N(i,t) = 0$ for any goal type t and level $i < C^*$, and correspondingly $p(i,t) = 0$. □

Theorem 1 (**Perfect predictions**) *SCP using a perfect type system T^* produces perfect predictions as long as each type in T^* is sampled at least once and the threshold value c is set to be zero.*

Proof SCP halts when $p(i,t_g) > c$, where t_g is a goal type. From Lemma 2 we have that $p(i,t_g) = 0$ for any goal type if $i < C^*$. Thus, SCP would never underestimate. Also from Lemma 2 we have that there exists a goal type t_g for which $p(C^*,t_g) > 0$. Thus, SCP would also not overestimate. Therefore, SCP produces perfect predictions. □

3.4 The cost bound equality property of SCP

Next, we show that if a heuristic-preserving type system is based on a consistent heuristic, the value of i when SCP reaches line 8 of Algorithm 1 will always be equal to the current cost bound d — we call this property the *cost bound equality property*.

Definition 6 Let $u \in T$ and $i, q \geq 1$. We say that type $t \in T$ at level q cannot modify the value of $p(i,u)$ if the computation of $p(i,u)$ is independent of the value of $N(q,t)$, i.e., changes in $N(q,t)$ do not affect the value of $p(i,u)$.

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

Lemma 3 *Let $t_g \in T$ be a goal type. Let $i, q \geq 1$, $i \geq q$. If $f(q,u) > i$ for type $u \in T$ at level q of the search tree, then u cannot modify the value of $p(i,t_g)$.*

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

$$\pi(t'|t) > 0 \text{ implies } h(t') \geq h(t) - 1\,. \tag{7}$$

Let $f(q,u) > i$. Suppose u can modify the value of $p(i,t_g)$. Thus, type u can generate a goal type at level i. In particular, there must exist a type sequence of length $i-q+1$, say $(t_{i-q}, t_{i-q-1}, t_{i-q-2}, \cdots, t_1, t_g)$, such that $t_{i-q} = u$ and $\pi(t_{i-q-1}|t_{i-q}) > 0, \cdots, \pi(t_g|t_1) > 0$. Since $h(t_g) = 0$, (7) yields

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

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

Theorem 2 *If* SCP *uses a heuristic-preserving type system T built from a consistent heuristic h, then, for any value of c, the value of i in line 8 of Algorithm 1 is always equal to the current cost bound d.*

Proof Let us first consider the case of the first iteration of the algorithm, when $d = h(s^*)$. By analogy with the proof of Lemma 3, when replacing q by 0, there exists a sequence of types $(t_i, t_{i-1}, \cdots, t_1, t_g)$ such that $\pi(t_{i-1}|t_i) > 0, \cdots, \pi(t_g|t_1) > 0$. Since h is consistent, (7) yields $h(s) \geq h(parent(s)) - 1$ for every node s. Since $h(t_g) = 0$, we obtain $i \geq h(s^*) = d$. Line 4 of Algorithm 1 implies $i \leq d = h(s^*)$, and thus $i = d$.

Next, consider the case $d > h(s^*)$. During an iteration with $d = U$, SCP will expand all the types that were expanded during the iteration with $d = U - 1$, and also the types u with $f(u) = d$ that were not expanded in the previous iteration. Lemma 3 states that no type t with $f(q,t) = U$ can modify the probability $p(i,t_g)$ for goal type t_g at a level $i < U$. Therefore, for $i = U-1$, if SCP does not find a goal type t_g with $p(i,t_g) \geq c$ in the iteration with $d = U - 1$, then SCP will not find a goal type t_g with $p(i,t_g) \geq c$ in the iteration with $d = U$ either. So, for the test in line 7 of Algorithm 1 to be successful, i has to equal d. □

3.4.1 Type systems based on inconsistent heuristics

If the type system is based on an inconsistent heuristic, Theorem 2 does not necessarily hold. That is, type $u \in T$ at level q of the search tree with $f(q,u) > i$ can potentially modify the value of $p(i,t_g)$ for goal type t_g. Thus, when using a type system built from an inconsistent heuristic, if returning i instead of d, Algorithm 1 could return a value smaller than the current cost bound d. In fact, SCP could even return a value of i that is lower than the heuristic value of the start state.

Example 3 Consider the example shown in Fig. 3. Here we use the T_h type system. We represent a type of a node n as $t_{x,y}$, where x is the heuristic value of the parent of n, and y is the heuristic value of n. Figure 3a shows states and edges of two paths to the goal in the original state space. The value in brackets in each circle is the heuristic value of the corresponding state – note the heuristic inconsistency between nodes A and B, when the heuristic value changes by more than one (edge cost). Figure 3b shows the corresponding types for the states and edges shown in Fig. 3a. Now, assume that SCP is run on state A from Fig. 3a. State A has a heuristic value of 6, and thus the optimal solution cost is at least 6. However, SCP might return 3, since as shown in Fig. 3c there is a path of cost 3 in the type space from type $t_{7,6}$ – during sampling, SCP has seen a node of type $t_{7,6}$ that generates a node of type $t_{6,2}$, which generates a node of type $t_{2,1}$, and in a different part of the state space SCP has sampled a node of type $t_{2,1}$ that generates a goal node.

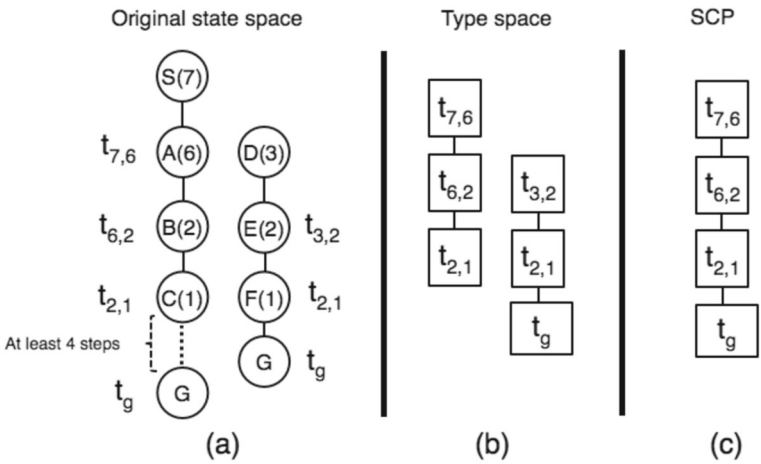

Fig. 3 States and edges in the original state space, their corresponding types in the type space, and a possible "shortcut" path in the type space

We have observed in our experiments that returning d instead of i in line 8 of Algorithm 1 yields slightly more accurate predictions when the type system is built from an inconsistent heuristic.

3.5 Reducing the size of type systems built from inconsistent heuristics

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

When using a type system that is very large, it may be difficult to sample properly all types. For example, a T_c type system for the 15-pancake puzzle was built based on the inconsistent heuristic that takes the maximum of the regular and the dual lookup of a PDB [10, 40]. Even after sampling 100 million random states and using biased sampling, there were types that were not sampled.

We use a process similar to bidirectional pathmax (BPMX) [10] to shrink the size of type systems based on inconsistent heuristics. This is done as follows. For every sampled state s, the heuristic values of its children are increased by propagating the heuristic values from other nodes while computing the type of s.

Example 4 Figure 4 illustrates this process. The children of the highlighted state in (a) with heuristic values of 2 and 1 can be raised to 4 as there is a child with heuristic value of 6. This is computed by subtracting the cost of the shortest path between the two siblings from the highest heuristic value among the children ($6 - 2 = 4$). The same process is applied to the highlighted state in (b): the states with heuristic value of 1 can be raised to 4. Therefore,

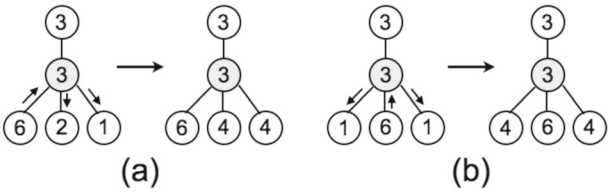

Fig. 4 Example of how we readjust the heuristic values of the children of the highlighted state while computing that state's type. This process reduces the size of a type system. The highlighted states in (**a**) and (**b**) end up being of the same type after propagating the heuristic values among the children. The heuristic propagation could be extended to the highlighted states and their parents, but by propagating only among the children we get the desired reduction of the type system size

two states that would belong to different types in T_c after applying the heuristic propagation now belong to the same type. Consequently, this process reduces the size of the type system.

Although not shown in the example, we also use the heuristic value of the sampled node (the nodes highlighted in Fig. 4) and of the parent of the sampled node to adjust the heuristic value of the children of the sampled node. In this case we propagate the heuristic value of the parent or of the grandparent downward to the children. Note that we could use a similar strategy to increase the heuristic value of the highlighted nodes in Fig. 4 and further shrink the size of the type system. However, we get type systems of good sizes by adjusting only the value of the children of the highlighted nodes. We use this technique only with the T_c type system.

3.6 SCP running time behavior

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

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

4 Experiments

We test SCP with different type systems based on different heuristic functions. We compare the accuracy of SCP with the accuracy of the heuristic functions used to define the type systems used by SCP. Finally, in one of the tested domains we compare SCP to a Bootstrap heuristic [14], a machine-learned estimator found in the literature.

4.1 Problem domains

Our experiments are run on three domains: the sliding-tile puzzle (15-puzzle), pancake puzzle (15-pancake puzzle), and Towers of Hanoi (12-disk, 4-peg).

– **Sliding-tile puzzle** [33] – The sliding-tile puzzle consists of $n^2 - 1$ numbered tiles that can be moved in an $n \times n$ grid. A state is a vector of length n^2 in which component k names what is located in the k^{th} puzzle position (either a number $1, \ldots, n^2 - 1$ for a tile

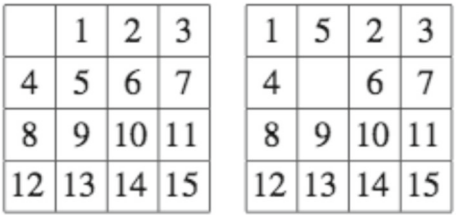

Fig. 5 The goal state for the 15-puzzle (*left*) and a state two moves from the goal (*right*)

or a symbol representing the blank). Every operator swaps the blank with a tile adjacent to it. The left part of Fig. 5 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 mean branching factor of the 15-puzzle is 2.1304 [22], and the average solution cost is 53 [19]. The number of states reachable from any given state is $(n^2)!/2$ [1].

– **Pancake puzzle** [7] – In the n-pancake puzzle, a state is a permutation of n numbered tiles and has $n-1$ successors, with the l^{th} successor formed by reversing the order of the first $l+1$ positions of the permutation ($1 \leq l \leq n-1$). All $n!$ permutations are reachable from any given state. We report results for $n = 15$ which contains 15! reachable states. The upper part of Fig. 6 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. The average solution cost of the 15-pancake puzzle is approximately 14.
– **Towers of Hanoi** [21] – The goal of this puzzle is to move all the disks from the original position onto a single peg. Only one disk can be moved at a time from the top of a peg onto another peg. A larger disk cannot be placed on top of a smaller disk. See Fig. 7 for an example of the goal state of the Towers of Hanoi with 5 disks and 4 pegs. We ran experiments with the 12-disk and 4-peg Towers of Hanoi, which has 4^{12} reachable states from the goal state [21]. The average branching factor of the 4-peg Towers of Hanoi is about 3.766 [19].

4.2 Experimental setup

Parameter setting In all the experiments in this section we set the user-defined threshold parameter c to 0.99. In Section 5.1 we empirically study the influence of different c-values on the prediction accuracy. Analogously, in Section 5.2 we show empirically the effect of different r-values and in Section 5.3 the effect of ϵ-truncation in SCP's prediction accuracy.

The depth of the prediction lookahead (i.e, the r-value shown in Section 2) was set to 1 ($r = 1$) in most of the experiments in this section; on the 15-puzzle we also experiment with $r = 25$. When experimenting with $r > 1$ we also make a prediction lookahead for the heuristic estimates, i.e., use the lowest heuristic value among the nodes at distance r from the start state. In Section 5.2 we show experiments with different values of r and we analyze how this parameter affects the runtime and accuracy of SCP's predictions.

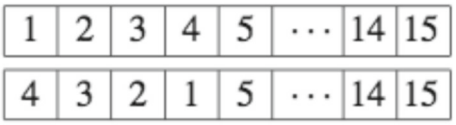

Fig. 6 The goal state for the 15-pancake puzzle (*above*) and a state one move from the goal (*below*)

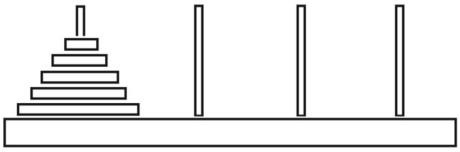

Fig. 7 The 5-disk 4-peg Towers of Hanoi

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

Results are presented in plots such as Fig. 8. The x-axis groups start states by their optimal solution costs and the y-axis represents the relative unsigned error of the predictions. The error bars represent the 95 % confidence interval based on the assumption that the prediction error for a given optimal solution cost follows a normal distribution.

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

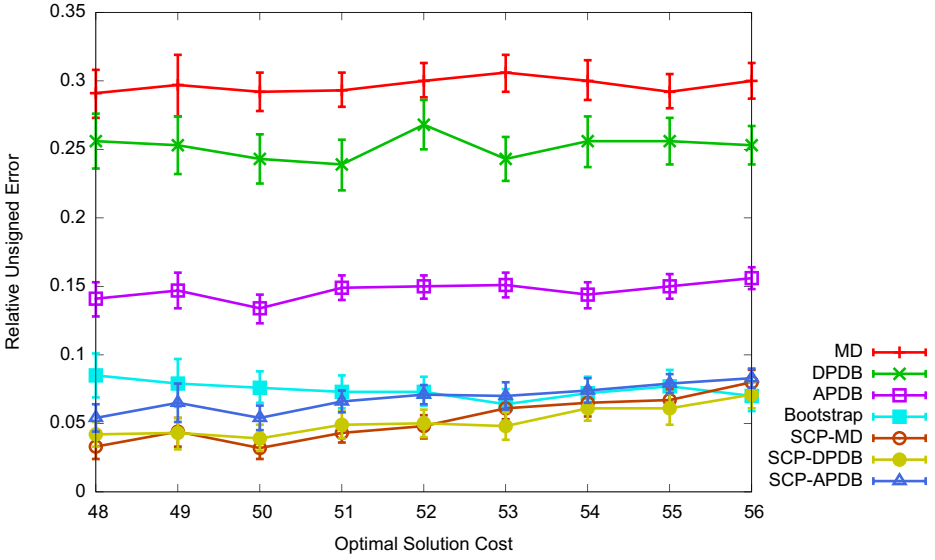

Fig. 8 15-puzzle

and T_{gc} type systems as there is a larger variety of heuristic values among the children and grandchildren of a node. The branching factor of a domain also influences the size of a type system. For instance, the 15-pancake puzzle, which has a much larger branching factor than the 15-puzzle, will likely have larger T_c and T_{gc} type systems due to the larger number of heuristic values considered. The type system used in each experiment is specified below.

4.3 The sliding-tile puzzle

For the 15-puzzle we solved optimally and performed the SCP prediction for 1,000 random solvable states to measure prediction accuracy. To define $\pi(t|u)$ and β_t, one billion random states were sampled and, in addition, we use the biased sampling process introduced by Zahavi et al. [39], in which we sample the child of a sampled state if the type of that child had not yet been sampled.

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

- Manhattan Distance (MD) – This is a popular and easy-to-implement heuristic function for the sliding-tile puzzles. It sums the Manhattan Distance of the individual tiles to their goal position (excluding the blank tile). This heuristic is consistent and we use the T_{gc} type system with it. The predictions using this type system will be referred to by SCP-MD.
- 7-8 Additive PDBs (APDB) – The 7-8 additive pattern database is an effective heuristic for the 15 puzzle [9, 20]. It consists of the sum of two disjoint pattern databases, one based on tiles 1-7 and the other based on tiles 8-15. The APDB is inconsistent[2] and the type system we use with it is the T_c type system. Predictions using this type system will be referred to by SCP-APDB.
- Double Inconsistent PDB (DPDB) – This is the same heuristic used by [39]. Two PDBs were created, one based on tiles 1-7 and another one based on tiles 9-15. For states with the blank in a location with an even number according to the goal state (see Fig. 5) the first PDB is consulted, the other PDB is consulted otherwise. As the blank always changes from even to odd or odd to even from one state to a neighbor, the PDB that is consulted alternates from parent to child. This generates inconsistency. The type system we use is T_c. Predictions using this type system will be referred to by SCP-DPDB.

In addition to comparing SCP predictions with the heuristics used to build the type systems, we compare SCP predictions with the inadmissible Bootstrap heuristic [14]. The Bootstrap algorithm iteratively improves an initial heuristic function by solving a set of successively more difficult training problems in each iteration. This process was shown to create effective heuristics for a variety of domains [14, 36].

Figure 8 presents the results. First, the results show the well-known fact that the Manhattan Distance (MD) heuristic is less accurate than the Additive PDB (APDB) heuristic. Second, SCP is able to produce accurate predictions, with errors less than 10 % of the optimal solution cost. Note that by comparison, all the tested admissible heuristics, MD, DPDB and APDB, tend to make inaccurate estimates, having an error of approximately 30 %, 25 % and 15 %, respectively. This corresponds to the intuitive observation mentioned earlier in the paper that admissible heuristics tend to make poor estimates as they are biased

[2] See footnote 1 in [9]. We built the same PDBs.

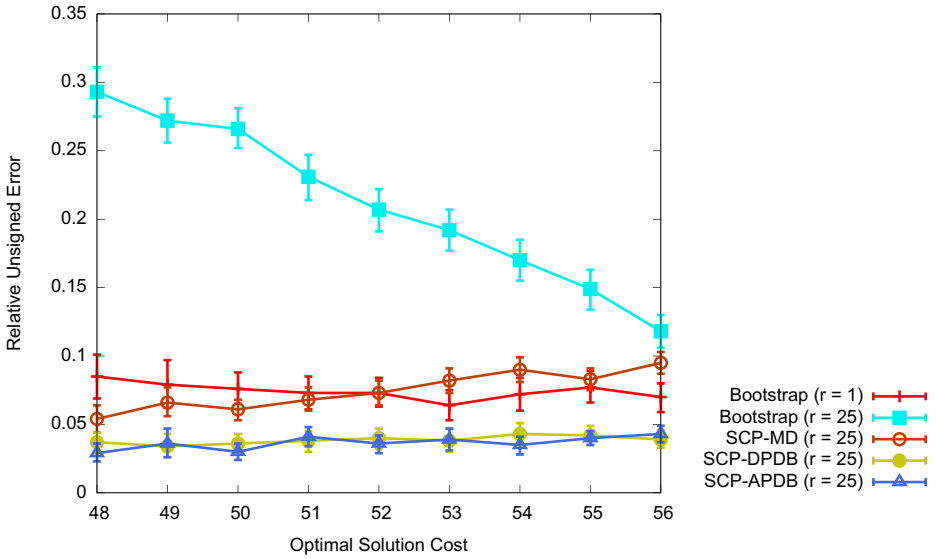

Fig. 9 15-puzzle (r=25)

to never overestimate the optimal cost. It is clear in this experiment that SCP using a type system based on a heuristic function produces more accurate predictions than the heuristic itself. We also observe in Fig. 8 that SCP's prediction error is about half of Bootstrap's for problems with optimal solution cost of 51 or less.

According to the results shown in Fig. 8 the accuracy of the heuristic used to build a type system does not seem to affect SCP's prediction accuracy much. SCP-MD, SCP-DPDB, and SCP-APDB are of similar accuracy, even though APDB is more accurate than MD and DPDB. While there is no definite explanation for this phenomenon, the results indicate that a type system does not necessarily have to be built from an accurate heuristic to allow SCP to make accurate predictions.

We also compare SCP's prediction accuracy with the accuracy of Bootstrap when using the prediction lookahead. Figure 9 presents the results. For convenience we repeat the results of Bootstrap with no prediction lookahead (r=1) shown in Fig. 8. It is interesting to see that the prediction lookahead substantially reduces Bootstrap's prediction accuracy. SCP, on the other hand, tends to benefit from the prediction lookahead as it is seeded with the exact distribution of types at distance r from the start state. SCP-DPDB and SCP-APDB are more accurate than Bootstrap using either $r = 1$ or $r = 25$. In Section 5.2 we study the tradeoff between prediction accuracy and prediction runtime of the prediction lookahead.

4.4 The pancake puzzle

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

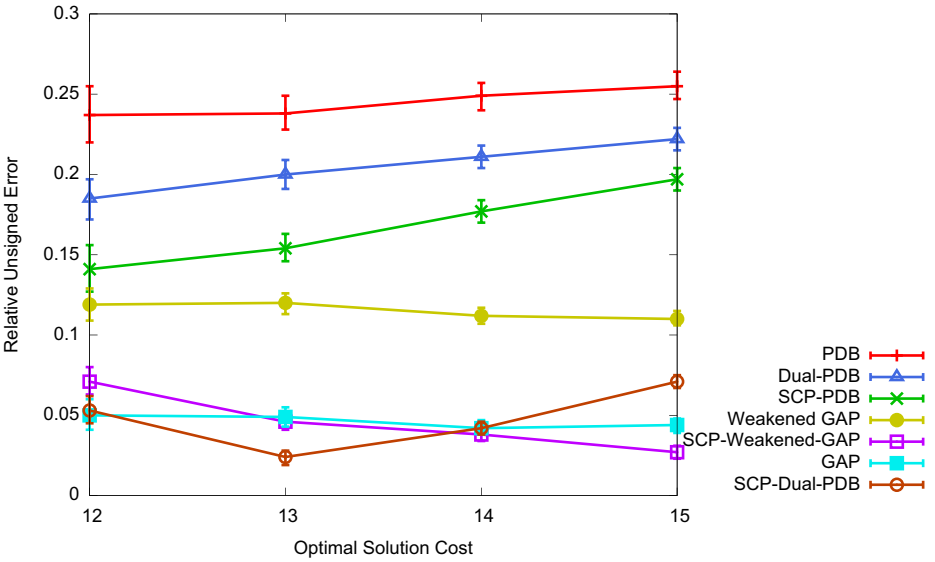

Fig. 10 15-pancake puzzle

Type systems As with the 15-puzzle, we use three different heuristic functions to define the type systems used in this experiment. The type systems used were based on the T_c type system.

- PDB – We created a PDB based on the location of the eight leftmost pancakes. The resulting PDB is consistent. The predictions using this type system will be referred to by SCP-PDB.
- Dual-PDB – For this type system we use a heuristic that makes a regular and a dual lookup on a PDB based on the leftmost eight pancakes and returns the maximum of them [10, 40]. The resulting heuristic is inconsistent. For the Dual-PDB there were types that were not sampled even after sampling 100 million random states. Thus, we used the process described in Section 3.5 to shrink the size of the type system so that all types could be sampled at least once with 100 million random states.
- Weakened GAP – GAP is a very accurate hand-crafted consistent heuristic for the pancake puzzle [12].[3] As the GAP heuristic already provides accurate cost estimates, it is not interesting to build a type system for SCP with such an accurate heuristic. Thus, we use a weakened version of it to build a type system. In our weakened version of GAP we use the number of adjacent pancakes whose number differs by more than one except for the rightmost pancake. We show the accuracy of GAP and its weakened version.

Figure 10 shows the results for the 15-pancake puzzle. As seen in the previous experiment, using SCP with a type system based on a given heuristic produces substantially more accurate predictions than using the heuristic itself as a predictor. This can be seen by the difference between PDB and SCP-PDB, Dual-PDB and SCP-Dual-PDB, and Weakened GAP and SCP-Weakened-GAP. SCP using a type system built from Dual-PDB is accurate and competitive with GAP.

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

The results in Fig. 10 also show that the PDB based on the eight leftmost pancakes is the least accurate estimator, giving estimates with errors of about 25 %. The type system built with the consistent PDB clearly fails to offer a good partition of the state space. As a consequence, SCP-PDB is the least accurate of the SCP predictions. SCP-Dual-PDB and SCP-Weakened-GAP on the other hand produce very accurate predictions – the errors indicate that the predictions are less than one move longer than the average optimal number of moves. Interestingly, SCP with Dual-PDB as well as SCP with Weakened GAP produce predictions of similar accuracy even though the estimates of Dual-PDB are much less accurate than the estimates of Weakened GAP.

4.5 Towers of hanoi

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

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

- PDB1 – We created a PDB built by projecting out 20 of the 48 state variables. Namely, we project out the even disks from pegs 1 and 2, disks 2, 8, and 10 from peg 3, and disks 2, 4, 6, 8, and 10 from peg 4. The resulting PDB is consistent. Predictions using this type system will be referred to by SCP-PDB1.
- PDB2 – We created another PDB by projecting out 19 of the 48 state variables. Namely, we projected out from pegs 1 and 2 the same disks we did for PDB1; from peg 3 we projected out the same disks as in PDB1 with the exception of projecting out disk 8 instead of disk 12; from peg 4 we also projected out the same disks as in PDB1, with the exception of disk 10, which was not projected out.

Figure 11 shows the results for the Towers of Hanoi. The least accurate estimations of the solution cost are given by PDB1, followed by PDB2. A major improvement is observed with the SCP predictions. For instance, the error drops from around 50 % to about 5 % when comparing PDB2 with SCP-PDB2.

4.6 Discussion

There are several trends that are observed in all three domains. First, for every domain and every heuristic, using SCP with a given heuristic always produces predictions that are substantially more accurate than the heuristic used to build the type system. This shows the

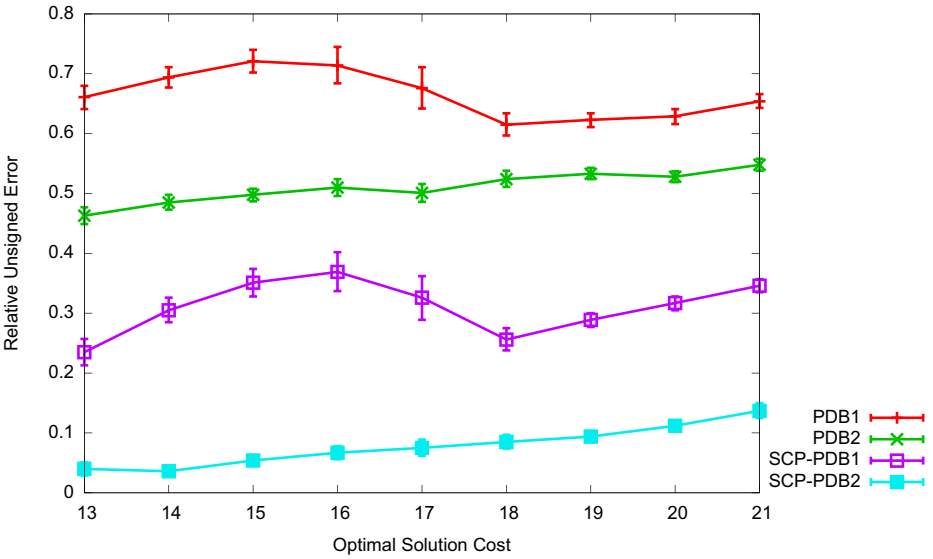

Fig. 11 12-disk 4-peg towers of hanoi

benefit of using SCP to predict the optimal cost over using heuristics. Furthermore, we have observed that SCP is able to make accurate predictions of the optimal solution cost even when the type system being employed is built from an "inaccurate" heuristic function. We have also observed in some cases that SCP might produce inaccurate predictions, such as SCP-PDB1 in Fig. 11, where the error is about 30 % of the optimal solution cost. However, even in that case SCP is substantially more accurate than the heuristic function used to build the type system, which in that case produces estimates with errors of about 70 % of the optimal solution cost.

4.7 SCP's empirical running time

In this section we compare SCP's and IDA*'s running time on the 15-puzzle. On the lefthand side of Table 1 we show the running time in seconds of both algorithms, as well as the ratio between IDA*'s and SCP's running time for problems with different solution costs (column "Cost" on the table). Ratio values larger than one mean that SCP is faster than IDA* (e.g., a ratio of 27 means that SCP is 27 times faster than IDA*). We also present, on the righthand side of the table, the number of nodes expanded by each algorithm as well as their ratio. Again, ratio values larger than one mean that SCP expands fewer nodes than IDA*. It is important to show both running time and number of nodes expanded because the latter is an implementation-independent measure.

The results on Table 1 show that IDA*'s running time and nodes expanded grow quickly with the optimal solution cost. By contrast, SCP's running time and number of nodes expanded remain almost constant with the increase of the optimal solution cost. This difference in the algorithms' behavior is noted in the ratios, which increase with the optimal solution cost. Finally, we remark that SCP can be substantially faster than IDA*, specially for larger costs.

Table 1 SCP's and IDA*'s running time in seconds and number of node expansions; average over 1,000 problem instances of the 15-puzzle

Cost	Running time			Node expansions		
	IDA*	SCP	Ratio	IDA*	SCP	Ratio
48	1.05	0.34	3.04	18,519,586	163,991	113
49	0.83	0.36	2.30	14,940,091	166,664	90
50	1.01	0.38	2.69	18,126,211	168,436	108
51	1.58	0.39	4.02	28,088,059	173,322	162
52	2.27	0.40	5.65	40,638,275	172,300	236
53	3.86	0.40	9.54	69,577,952	173,491	401
54	4.69	0.42	11.26	83,975,666	173,743	483
55	12.00	0.41	29.13	216,134,233	169,253	1,277
56	11.55	0.42	27.34	210,066,690	172,159	1,220

5 Empirical study of SCP's parameters

In this section we make an empirical study of the parameters required by SCP. Namely, we study the threshold parameter c, the prediction lookahead r, and the effects of ϵ-truncation on SCP's prediction accuracy.

5.1 Empirical study of the threshold parameter

In Section 4 SCP was tested for a fixed set of parameters. Namely, we used a threshold parameter c of 0.99. In this section we discuss the effect of this c-value on the accuracy of

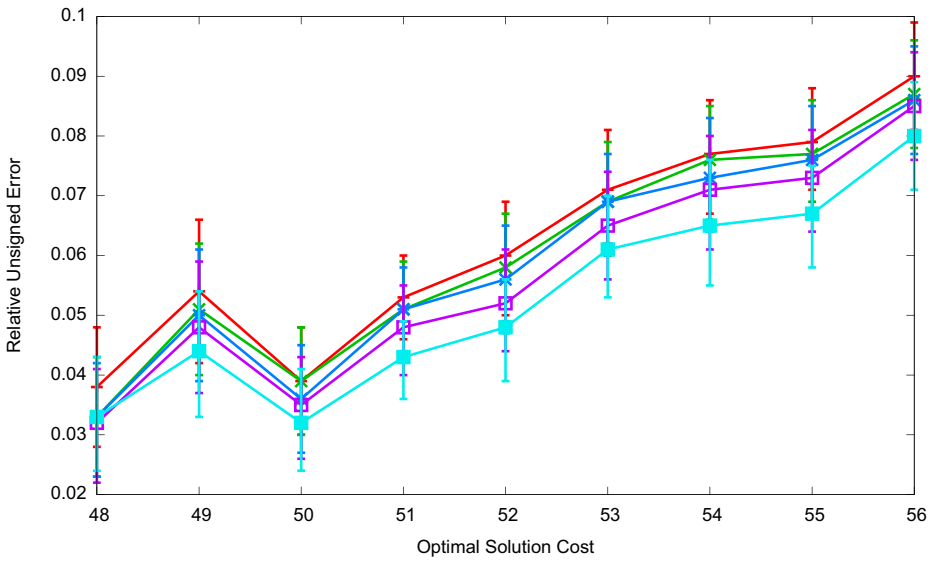

Fig. 12 Robustness to the parameter c for the 15-puzzle. SCP-MD

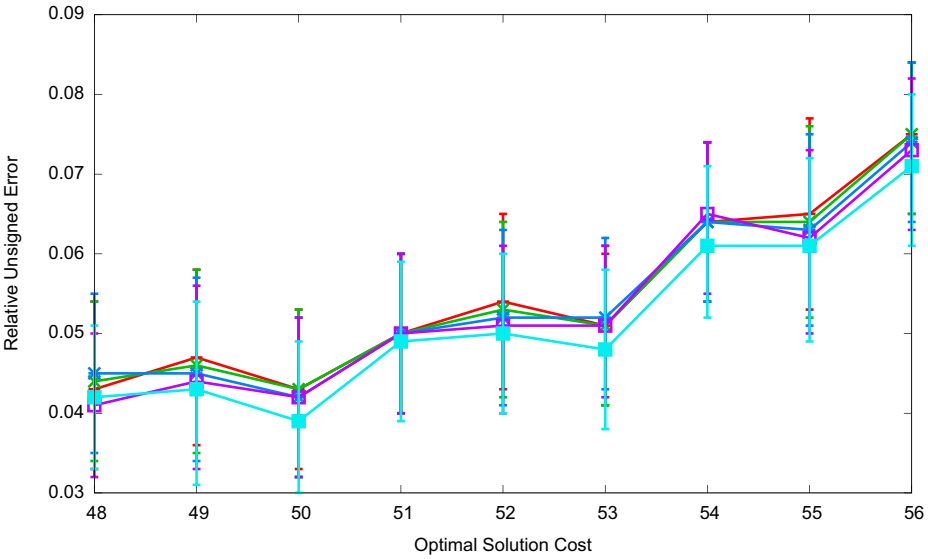

Fig. 13 Robustness to the parameter c for the 15-puzzle. SCP-DPDB

SCP. We experimentally show the algorithm's accuracy with a c-value of 0.80, 0.85, 0.90, 0.95, and 0.99.

Figures 12, 13, and 14 present SCP's prediction errors for different c-values for the 15-puzzle. Different curves correspond to different values of c. As can be observed, the effect of different c values is minor, and the curves are clustered together; there is no substantial difference in prediction accuracy for the different c-values used. The accuracy of SCP is

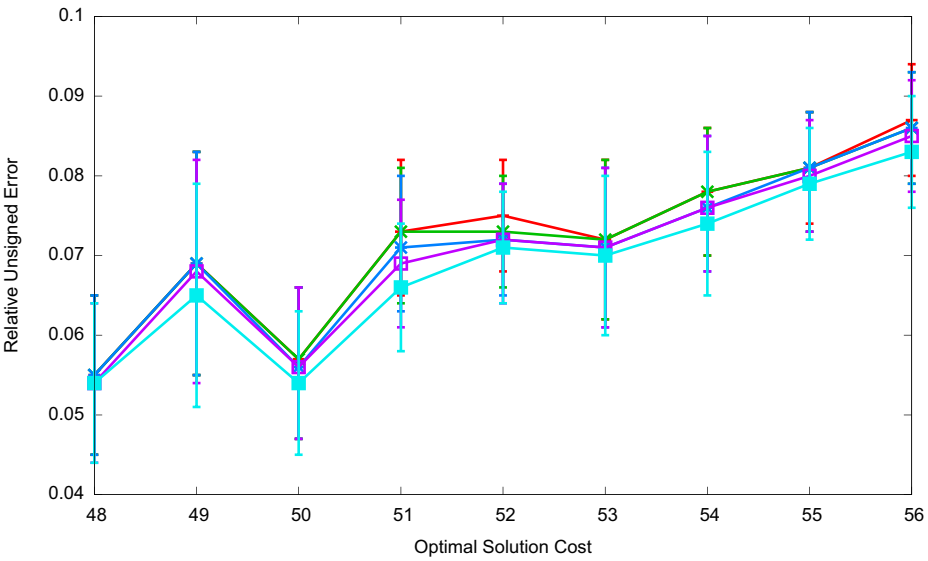

Fig. 14 Robustness to the parameter c for the 15-puzzle. SCP-APDB

relatively robust to the choice of c. Similar results were observed on the 15-pancake-puzzle and on the 12-disk 4-peg Towers of Hanoi.

5.2 Empirical study of the prediction lookahead

CDP is known to improve its accuracy by increasing the r-value [39]. Here we observe a similar effect for SCP. To focus only on the effect of the r-values, we did not use ϵ-truncation in the following experiment.

Table 2 shows prediction results of SCP using the T_{gc} type system with MD as the heuristic function for five different values of r. Each row in Table 2 presents results for start states with a given optimal solution cost, shown in the first column of the table. Besides the relative unsigned error (Error), we also show the maximum (t_{max}) and the minimum (t_{min}) of the prediction runtime in seconds of start states in a set of 1,000 random problem instances.

The first observation in Table 2 is that the error decreases as the value of r increases. Larger values of r result in SCP being seeded with types of nodes farther from the start state. Like CDP, SCP emulates a search tree in the type space where states in the actual state space are compressed into types. While this compression into types makes the process of predicting the optimal solution cost faster, it makes the prediction subject to error. For instance, SCP's predicted distribution of types at distance x from the start state (i.e., the values of $N(x,t)$ and $p(x,t)$) is only an approximation of the actual distribution. When using $r = x$, SCP uses the actual distribution of types at that level. Therefore, larger values of r reduce the error of the type distribution at a given distance from the start state, which, as suggested by Table 2, also reduce the prediction error.

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

Table 2 Effect of different r-values on SCP's prediction accuracy and runtime for the 15-puzzle. We show the relative unsigned error and the minimum and maximum runtimes in seconds for a set of 1,000 random start states

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

that the maximum runtime when $r = 21$ is substantially higher than the other maximum runtimes.

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

5.3 Empirical study of ϵ-truncation with SCP

Next, we check the effect of ϵ-truncation on the accuracy of SCP predictions. We isolate the effect of ϵ-truncation, in the subsequent experiment by setting the r-value to one. Table 3 shows the relative unsigned error of SCP with and without ϵ-truncation, on 1,000 random 15-puzzle instances. Results are shown for each of the type systems described for the 15-puzzle in Section 4. For convenience we drop the prefix "SCP" from the name of the type systems, and, in addition, we add an ϵ to the name of the type system if SCP uses ϵ-truncation. We highlight an ϵ-truncation entry in the table if it is at least as accurate as its counterpart.

As can be observed in Table 3, ϵ-truncation substantially improves the prediction accuracy of SCP in the 15-puzzle. ϵ-truncation was designed to carefully ignore rare and harmful events observed during the CDP sampling. For instance, if a type t rarely generates a type t', then CDP improves its prediction accuracy by completely ignoring this rare event. The rare events also seem to be harmful to SCP as carefully ignoring them improves SCP's prediction accuracy. We conjecture that rare events create "shortcuts" to the goal type in the type space, making SCP find the goal type prematurely (see [28] for details on ϵ-truncation).

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

Table 3 Effect of ϵ-truncation on SCP's prediction accuracy for the 15-puzzle. The results with lower prediction error are highlighted in bold

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

6 Possible applications of SCP

SCP could have other practical applications in addition to the merit of predicting the optimal solution cost. Here are two possibilities.

Chenoweth and Davis [5] showed that by multiplying the heuristic estimates $h(\cdot)$ by a suitable constant $w > 0$ one could provably reduce the time complexity of a heuristic search algorithm using $h(\cdot)$ from exponential to polynomial in some problem domains. They also suggested a method for selecting a suitable value of w to "correct" the heuristic error and quickly find near-optimal solutions. The drawback of their method is that it requires problem instances to be solved optimally. SCP predictions could be used to efficiently predict the optimal solution cost of problem instances and thus find a weight that also "corrects" the heuristic error. Such w-value could be used with algorithms such as WIDA* [18] and WA* so that they quickly find near-optimal solutions.

Several search algorithms require an upper bound on the solution cost, e.g., Branch and Bound [2] and Potential Search (PTS) [34]. PTS is a bounded-cost search algorithm that efficiently searches for a solution with cost less than or equal to a given upper bound on the solution cost. This is done by focusing the search on nodes that are more likely to lead to a goal with cost less than or equal to the desired bound. SCP can be used to find an accurate prediction of the optimal solution cost which is then multiplied by a weight $w > 1$ to provide an upper bound for bounded cost search algorithms. In some cases the SCP prediction itself represents an upper bound to the optimal solution cost. However, because SCP does not guarantee the predicted value to be an upper or lower bound to the optimal solution cost, multiplying the predicted value by $w > 1$ increases the chances of having an upper bound.

7 Related work

7.1 Search tree size predictors

SCP is based on CDP, a method developed for estimating the search tree size. The first known method for estimating the search tree size is due to Knuth [16]. 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 does not know a priori how long the search will take. Knuth's method was later improved by Chen [4] through the usage of a type system to reduce the variance of sampling. We call Chen's method Stratified Sampling (SSS). Lelis [26] showed how to incorporate active sampling to SSS, further reducing the variance of sampling.

Independently of Knuth and Chen, Korf et al. [22] developed a method for estimating the size of the search tree expanded by IDA* [17]. Korf et al.'s method makes accurate predictions of the IDA* search tree size for the special case of consistent heuristics. Zahavi et al.'s CDP generalized Korf et al.'s method to also produce accurate estimates of the IDA* search tree size when inconsistent heuristics are employed. Burns and Ruml extended CDP to work on domains with real-valued edge costs [3]. Lelis et al. presented a method called ϵ-truncation that mitigates a source of error that had been overlooked in CDP [25].

SS and CDP have in common the usage of a type system to guide their sampling. Lelis et al. [28] discovered that the type systems developed to be used with CDP could substantially improve SSS's prediction power. The main difference between SSS 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, independently of the type system being used, SSS has the guarantee of producing perfect predictions, while CDP does not.

Knuth also noted in his seminal work that his method would not produce accurate predictions of the size of the search tree expanded by branch and bound methods. Kilby et al. [15] developed methods that use the information seen during search to infer how many nodes branch and bound algorithm would expand. Similar approach was taken by Thayer et al. [37] to build a "progress bar" of best-first search variants. Lelis et al. [27] extended SSS into a prediction algorithm they called Two-Step Stratified Sampling (TSS) and showed empirically on optimization problems over probabilistic graphical models [29] that TSS is able to produce good estimates of the size of the Depth-First Branch and Bound search tree.

7.2 Another solution cost predictor

Since SCP was first published, we developed another algorithm named Bidirectional Stratified Sampling (BiSS) [23]. BiSS predicts the optimal solution cost of individual problem instances by running a bidirectional search on the state space. It samples the state space for each problem instance separately and it searches simultaneously from the start and the goal. By contrast, in SCP the sampling of the original state space is performed only in a preprocessing phase, and the search is activated on the type space. The instance-specific sampling BiSS does allows it to scale to very large state spaces. However, BiSS also has a few disadvantages. BiSS requires there to be a single goal state and is therefore not suitable for domains in which a set of goal conditions is given instead of an actual goal state. Another limitation is that BiSS is only applicable in domains in which it is possible to reason backwards from the goal. SCP is the algorithm of choice when only goal conditions are given and also when it is not possible to reason backwards from the goal state.

8 Conclusions

In many real world scenarios it is sufficient to know the solution cost of a problem. Classical search algorithms find the solution cost by finding an optimal path from the start to goal. Heuristic functions estimate the length of such a path, but are required to be fast enough to be calculated for many nodes during the search.

In this paper we proposed SCP, an algorithm designed to accurately and efficiently predict the optimal solution cost of a problem. While SCP can be viewed as a heuristic, it differs from a heuristic conceptually in that: 1) SCP is not required to be fast enough to guide search algorithms; 2) SCP does not favor admissibility; 3) SCP aims at making accurate predictions and thus our measure of effectiveness is the prediction accuracy, in contrast to the solution quality and number of nodes expanded used to measure the effectiveness of other heuristic functions.

We showed empirically that SCP makes predictions with errors of less than 15 % of the optimal solution cost in all three domains tested. Our experiments also show that SCP was always substantially more accurate than the heuristic functions used to build its type systems. Moreover, SCP was consistently more accurate than the Bootstrap heuristic, a machine-learned inadmissible heuristic.

We also studied the impact of the parameters required by SCP on the prediction accuracy. Namely, we empirically studied the impact of the threshold parameter c, of the ϵ-cuts, and

of the prediction lookahead r on the prediction accuracy. Our results suggested that (1) for any value of c between 0.8 and 0.99 SCP makes accurate predictions; (2) ϵ-truncation can substantially improve the prediction accuracy; and (3) the prediction lookahead can improve the prediction accuracy at the cost of increasing the runtime.

Acknowledgments This work was supported by the Laboratory for Computational Discovery at the University of Regina. The authors gratefully acknowledge the research support provided by Alberta Innovates – Technology Futures, AICML, and NSERC.

References

1. Archer, A.F.: A modern treatment of the 15-puzzle. Am. Math. Mon. **106**, 793–799 (1999)
2. Balas, E., Toth, P.: Branch and bound methods. In: Lawler, E.L., Lenstra, J.K., Rinnooy Kart, A.H.G., Shmoys, D.B. (eds.) The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, New York (1985)
3. Burns, E., Ruml, W.: Iterative-deepening search with on-line tree size prediction. In: Proceedings of the International Conference on Learning and Intelligent Optimization, pp. 1–15 (2012)
4. Chen, P.-C.: Heuristic Sampling on Backtrack Trees. PhD thesis, Stanford University (1989)
5. Chenoweth, S.V., Davis, H.W.: High performance A* search using rapidly growing heuristics. In: International Joint Conference on Artificial Intelligence (1991)
6. Culberson, J.C., Schaeffer, J.: Searching with pattern databases. In: Proceedings of the Canadian Conference on Artificial Intelligence, volume 1081 of Lecture Notes in Computer Science, pp. 402–416. Springer (1996)
7. Dweighter, H.: Problem E2569. Am. Math. Mon. **82**, 1010 (1975)
8. Ernandes, M., Gori, M.: Likely-admissible and sub-symbolic heuristics. In: Proceedings of the European Conference on Artificial Intelligence, pp. 613–617 (2004)
9. Felner, A., Korf, R.E., Hanan, S.: Additive pattern database heuristics. J. Artif. Intell. Res. **22**, 279–318 (2004)
10. Felner, A., Zahavi, U., Schaeffer, J., Holte, R.C.: Dual lookups in pattern databases. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 103–108 (2005)
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **SSC-4**(2), 100–107 (1968)
12. Helmert, M.: Landmark heuristics for the pancake problem. In: Proceedings of the Symposium on Combinatorial Search, pp. 109–110. AAAI Press (2010)
13. Helmert, M., Haslum, P., Hoffmann, J.: Flexible abstraction heuristics for optimal sequential planning. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 176–183 (2007)
14. Jabbari Arfaee, S., Zilles, S., Holte, R.C.: Learning heuristic functions for large state spaces. Artif. Intell. **175**(16–17), 2075–2098 (2011)
15. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Estimating search tree size. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1014–1019. AAAI Press (2006)
16. Knuth, D.E.: Estimating the efficiency of backtrack programs. Math. Comp. **29**, 121–136 (1975)

17. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artif. Intell. **27**(1), 97–109 (1985)
18. Korf, R.E.: Linear-space best-first search. Artif. Intell. **62**(1), 41–78 (1993)
19. Korf, R.E.: Linear-time disk-based implicit graph search. J. ACM **55**(6), 26:1–26:40 (2008)
20. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artif. Intell. **134**(1–2), 9–22 (2002)
21. Korf, R.E., Felner, A.: Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 2324–2329 (2007)
22. Korf, R.E., Reid, M., Edelkamp, S.: Time complexity of iterative-deepening-A*. Artif. Intell. **129**(1–2), 199–218 (2001)
23. Lelis, L., Stern, R., Felner, A., Zilles, S., Holte, R.C.: Predicting optimal solution cost with bidirectional stratified sampling. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 155–163. AAAI Press (2012)
24. Lelis, L., Stern, R., Jabbari Arfaee, S.: Predicting solution cost with condiditional probabilities. In: Proceedings of the Symposium on Combinatorial Search, pp. 100–107. AAAI Press (2011)
25. Lelis, L., Zilles, S., Holte, R.C.: Improved prediction of IDA*'s performance via ϵ-truncation. In: Proceedings of the Symposium on Combinatorial Search, pp. 108–116. AAAI Press (2011)
26. Lelis, L.H.S.: Active stratified sampling with clustering-based type systems for predicting the search tree size of problems with real-valued heuristics. In: Proceedings of the Symposium on Combinatorial Search, pp. 123–131. AAAI Press (2013)
27. Lelis, L.H.S., Otten, L., Dechter, R.: Predicting the size of depth-first branch and bound search trees. In: International Joint Conference on Artificial Intelligence, pp. 594–600 (2013)
28. Lelis, L.H.S., Zilles, S., Holte, R.C.: Predicting the size of IDA*'s search tree. Artif. Intell., 53–76 (2013)
29. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann (1988)
30. Prieditis, A.E.: Machine discovery of effective admissible heuristics. Mach. Learn. **12**(1–3), 117–141 (1993)
31. Richter, S., Helmert, M.: Preferred operators and deferred evaluation in satisficing planning. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 273–280 (2009)
32. Samadi, M., Felner, A., Schaeffer, J.: Learning from multiple heuristics. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 357–362. AAAI Press (2008)
33. Slocum, J., Sonneveld, D.: The 15 Puzzle. Slocum Puzzle Foundation (2006)
34. Stern, R., Puzis, R., Felner, A.: Potential search: a bounded-cost search algorithm. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 234–241 (2011)
35. Sturtevant, N.R., Felner, A., Barrer, M., Schaeffer, J., Burch, N.: Memory-based heuristics for explicit state spaces. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 609–614 (2009)
36. Thayer, J., Dionne, A., Ruml, W.: Learning inadmissible heuristics during search. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 250–257 (2011)
37. Thayer, J.T., Stern, R., Lelis, L.H.S.: Are we there yet? - estimating search progress. In: Proceedings of the 5th Annual Symposium on Combinatorial Search, pp. 129–136. AAAI Press (2012)
38. Yang, F., Culberson, J.C., Holte, R.C., Zahavi, U., Felner, A.: A general theory of additive state space abstractions. J. Artif. Intell. Res. **32**, 631–662 (2008)
39. Zahavi, U., Felner, A., Burch, N., Holte, R.C.: Predicting the performance of IDA* using conditional distributions. J. Artif. Intell. Res. **37**, 41–83 (2010)
40. Zahavi, U., Felner, A., Holte, R.C., Schaeffer, J.: Duality in permutation state spaces and the dual search algorithm. Artif. Intell. **172**(4–5), 514–540 (2008)