Predicting Solution Cost with Conditional Probabilities

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

Roni Stern
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel 85104
(roni.stern@gmail.com)

Shahab Jabbari Arfaee
Computing Science Department
University of Alberta
Edmonton, AB, Canada T6G 2E8
(jabbaria@cs.ualberta.ca)

Abstract

Classical heuristic search algorithms find the *solution cost* of a problem while finding the path from the start state to a goal state. However, there are applications in which finding the path is not needed. In this paper we propose an algorithm that accurately and efficiently predicts the *solution cost* of a problem without finding the actual solution. We show empirically that our predictor makes more accurate predictions when compared to the bootstrapped heuristic, which is known to be a very accurate inadmissible heuristic. In addition, we show how our prediction algorithm can be used to enhance heuristic search algorithms. Namely, we use our predictor to calculate a bound for a bounded best-first search algorithm and to tune the w-value of Weighted IDA*. In both cases major search speedups were observed.

Introduction

Heuristic search algorithms such as A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) are guided by the cost function $f(s) = g(s) + h(s)$, where $g(s)$ is the lowest cost path from the start state to s and $h(s)$ is an estimate of the cost of the lowest cost path from s to a goal. This estimate, also known as a heuristic function, is called *admissible* if it never overestimates the cost of the lowest cost path from state s to the goal, and it is called *inadmissible* otherwise. Heuristic search algorithms guided by the cost function $f = g + h$ where h is an admissible heuristic are guaranteed to find optimal solutions (Hart, Nilsson, and Raphael 1968). A considerable amount of effort has been devoted to creating admissible (Culberson and Schaeffer 1996; Helmert, Haslum, and Hoffmann 2007; Yang et al. 2008; Sturtevant et al. 2009) and inadmissible (Ernandes and Gori 2004; Samadi, Felner, and Schaeffer 2008; Jabbari Arfaee, Zilles, and Holte 2010; Thayer, Dionne, and Ruml 2011) heuristics. Regardless of admissibility, these heuristics share a property: the heuristic evaluation must be fast enough to be computed for every state in the search tree, or at least for a large number of states.

A* and IDA* find the *solution cost* of a problem while finding the path from a start state to a goal state. However, there are applications in which finding the path is not needed. Every decision problem, such as *"is there a vertex cover of size k ?"*, is of this type. As an example, consider an owner of a construction company, which 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, after bidding. Thus, an important question to be answered is the following. Can we efficiently predict the *solution cost* of a problem without actually solving the problem (*i.e.*, without finding the sequence of actions from the start state to a goal)?

A heuristic function is in fact an estimate of the solution cost. However, it is clear that admissible heuristics will provide inaccurate estimations of the solution cost as they are biased to never overestimate the actual value. In some cases even inadmissible heuristics are biased towards admissibility so that search algorithms using them find solutions that are "close" to optimal (Ernandes and Gori 2004; Samadi, Felner, and Schaeffer 2008). The algorithm presented in this paper can be viewed as an *inadmissible* heuristic function that aims at accurately predicting the solution cost of a given start state. Moreover, in contrast to other heuristics, we do not intend to use our heuristic to guide search algorithms. Thus, we expect to gain accuracy by allowing our method to be slower than "traditional heuristics".

Our solution is based on the `CDP` (Conditional Distribution Prediction) formula described by Zahavi *et al.* (2010). For this reason we prefer to call the method presented in this paper a *predictor* instead of a *heuristic*. The `CDP` prediction formula requires the state space to be sampled with respect to a "type system", and the information gathered during sampling is used to efficiently predict the number of nodes expanded on an iteration of IDA* with cost bound d. Recently, Lelis *et al.* (2011) identified a source of error in the `CDP` formula that had been previously overlooked. They observed that "rare events" could reduce `CDP`'s prediction accuracy, and in addition, they presented a method that systematically disregards these harmful rare events. Lelis *et al.*'s method, named ϵ-truncated `CDP`, substantially improved prediction accuracy. We extend the ideas of Zahavi *et al.* and Lelis *et al.* to predict the solution cost of a problem instead of predicting the number of nodes expanded by IDA*.

We show empirically that our method accurately predicts

(1) the average solution cost of a set of start states and (2) the solution cost of single start states. Moreover, our empirical results show that our method is consistently more accurate than the bootstrapped heuristic (Jabbari Arfaee, Zilles, and Holte 2010), which is known to be a very accurate inadmissible heuristic for a range of domains (Jabbari Arfaee, Zilles, and Holte 2010; Thayer, Dionne, and Ruml 2011).

In the second part of this paper we show novel ways of using inadmissible heuristics to enhance search algorithms. Different from the traditional approach, we do not use the heuristic to calculate $f(s) = g(s) + h(s)$. Instead, we show how our prediction method can be used to (1) tune the w-value of Weighted A* (Pohl 1970) and Weighted IDA*, and (2) guide search in a bounded cost search algorithm. Our experiments show major speedups while using the predictor presented in this paper for these purposes.

Related Work

Even though we use the ideas of Zahavi *et al.* (2010) and Lelis *et al.* (2011), we are solving a different problem. They aimed at accurately predicting the number of nodes expanded on an iteration of IDA*, whereas we are aiming at accurately predicting the solution cost of a problem instance. In fact, one of the chief difficulties of using CDP in practice is to determine a cost bound to provide to the prediction formula. The solution cost predictor sheds some light on this problem as it suggests an accurate cost bound.

Inadmissible heuristics, such as the bootstrapped heuristic (Jabbari Arfaee, Zilles, and Holte 2010) are designed to guide search algorithms, and their effectiveness is measured by the number of nodes expanded and by the quality of the solution found by search algorithms using them. Our predictor is designed to make accurate predictions of the solution cost, and our measure of effectiveness is the prediction accuracy.

The CDP Prediction Framework

Here we briefly sketch the CDP system using our own notation. The reader is referred to the original paper (Zahavi et al. 2010) for full explanations and illustrative examples.

Let S be the set of states, $E \subseteq S \times S$ the set of directed edges over S representing the parent-child relation in the underlying state space, and $h : S \to \mathbb{N}$ the heuristic function.

Definition 1 *$T = \{t_1, \ldots, t_n\}$ is a type system for (S, E) if it is a disjoint partitioning of E. For every $(\hat{s}, s) \in E$, $T(\hat{s}, s)$ denotes the unique $t \in T$ with $(\hat{s}, s) \in t$.*

Definition 2 *Let $t, t' \in T$. $p(t'|t)$ denotes the probability that a node s with parent $\hat{s}$ and $T(\hat{s}, s) = t$ generates a node c with $T(s, c) = t'$. b_t denotes the average number of children generated by a node s with parent $\hat{s}$ with $T(\hat{s}, s) = t$.*

Parent-pruning is an easy-to-implement enhancement of IDA* that avoids re-expanding the parent of a node. IDA* with parent-pruning will not generate a node $\hat{s}$ from s if $\hat{s}$ is the parent of s. In making the prediction of the number of nodes expanded on an iteration of IDA* we are interested in estimating the number of nodes in the subtree below a given node. Because of parent-pruning the subtree below a node differs depending on the node from which it was generated. Like Zahavi *et al.*, to account for this effect of parent-pruning we define types over node pairs instead of just nodes.[1]

All type systems considered in this paper have the property that $h(s) = h(s')$ if $T(\hat{s}, s) = T(\hat{s}', s')$. We assume this property in the formulae below, and denote by $h(t)$ the value $h(s)$ for any $s, \hat{s}$ such that $T(\hat{s}, s) = t$.

CDP samples the state space in order to estimate $p(t'|t)$ and b_t for all $t, t' \in T$. We denote by $\pi(t'|t)$ and $\beta(t)$ the respective estimates thus obtained. The predicted number of nodes expanded by IDA* (with parent pruning) for start state $\hat{s}^*$, cost bound d, heuristic h, and type system T is

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

Here $N(i, t, (\hat{s}^*, s^*), d)$ is the number of pairs $(\hat{s}, s)$ with $T(\hat{s}, s) = t$ and s at level i of the search tree rooted at s^*. It is computed recursively as follows.

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

and, for $i > 1$, the value $N(i, t, (\hat{s}^*, s^*), d)$ is given by

$$\sum_{u \in T} N(i-1, u, (\hat{s}^*, s^*), d)\pi(t|u)\beta_t P(t, i, d) \qquad (1)$$

where $P(t, i, d)$ = 1 if $h(t) + i \leq d$, and is 0 otherwise.

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

As our basic type system, T_h, we use Zahavi *et al.*'s basic "two-step" model, defined (in our notation) as $T_h(\hat{s}, s) = (h(\hat{s}), h(s))$. Two new domain-independent type systems we will also use, which are "more informed" than T_h, are:

$T_c(\hat{s}, s) = (T_h(\hat{s}, s), c((\hat{s}, s), 0), \ldots, c((\hat{s}, s), H))$, where $c((\hat{s}, s), k)$ 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;

[1] Zahavi *et al.* call the type definition over pairs a "two-step" model.

[2] That is why we ensure all nodes mapped to a type have the same heuristic value, as mentioned above.

$T_{gc}(\hat{s}, s) = (T_c(\hat{s}, s), gc((\hat{s}, s), 0), \ldots, gc((\hat{s}, s), H))$, where $gc((\hat{s}, s), k)$ is the number of grandchildren of s, considering parent pruning, whose h-value is k.

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

Definition 3 *Let T_1, T_2 be type systems. T_1 is a refinement of T_2, denoted $T_1 \prec T_2$, if $|T_1| > |T_2|$ and for all $t_1 \in T_1$ there is a $t_2 \in T_2$ with $\{(\hat{s}, s)|T_1(\hat{s}, s) = t_1\} \subseteq \{(\hat{s}, s)|T_2(\hat{s}, s) = t_2\}$. If $t_1 \in T_1$ and $t_2 \in T_2$ are related in this way, we write $T_2(t_1) = t_2$.*

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

ϵ-truncation

Intuitively, if $T_1 \prec T_2$ one would expect CDP's predictions using T_1 to be at least as accurate as the predictions using T_2, since all the information that is being used by T_2 to condition its predictions is also being used by T_1 (Zahavi et al. 2010, p. 59). However, we have shown empirically in a separate paper (Lelis, Zilles, and Holte 2011) that this intuition is often wrong. Refined type systems are more susceptible to a harmful phenomenon (the *discretization effect*) that can reduce prediction accuracy considerably.

In addition to showing that more informed type systems often make worse predictions than less informed ones, Lelis *et al.* (2011) introduced a method called ϵ-truncation that carefully disregards information from a type system with the aim of reducing the discretization effect, and therefore increasing the accuracy of the predictions. The ϵ-truncation method can be summarized as follows.

1. As in normal CDP, sample the state space to obtain $\pi(t|u)$.
2. Compute a cutoff value ϵ_i for each i between 1 and d.
3. Use ϵ_i to define $\pi^i(t|u)$, a version of $\pi(t|u)$ that is specific to level i. In particular, if $\pi(t|u) < \epsilon_i$ then $\pi^i(t|u) = 0$; the other $\pi^i(t|u)$ are set by scaling up the corresponding $\pi(t|u)$ values so that they sum to 1.
4. In computing CDP use $\pi^i(t|u)$ at level i instead of $\pi(t|u)$.

The ϵ_i values are computed in a preprocessing phase that involves running normal CDP on a small number of start states and solving a set of linear programming optimization problems for every level of every one of those CDP runs.

ϵ-truncation substantially improved the prediction accuracy for the domains tested (the same ones used in this paper), especially when refined type systems were employed. The "pathological" behavior of more informed type systems leading to poorer predictions did not occur when ϵ-truncation was applied. In this paper we use ϵ-truncation in all experiments.

Predicting the Solution Cost

Instead of predicting the number of nodes expanded for a given cost bound and start state as CDP does, we are aiming at predicting the solution cost for a given start state. As for that we do the following.

1. Make a CDP prediction initially bounded by the heuristic value of the start state and increment it by a minimal amount in subsequent iterations;

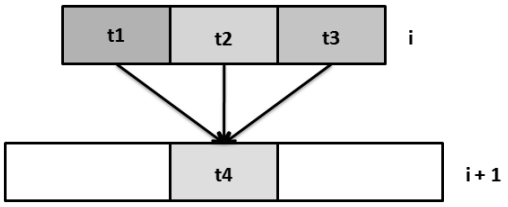

Figure 1: Types at level i are used to calculate the probability of t_4 existing at level $i + 1$.

2. For every level i, during the CDP iterations, estimate the probability of finding a goal at that level;
3. Terminate when a goal is found with probability higher than a confidence value provided by the user.

We use the probability of a type "existing" at a *level of prediction* as an estimation for the probability of finding a state of that type at a *level of the actual search*. Therefore, the probability of finding a goal at a level of the actual search is estimated by the probability of a *goal type* existing at a level of prediction. A *goal type* is a type to which only goal states are mapped. If we allow non-goal states to be mapped to goal types, we will create additional "shortcuts" to the goal type in the "type system state space". These shortcuts can potentially reduce prediction accuracy. A *goal type* is defined as follows.

Definition 4 ***(goal type)*** *- A type $t^g \in T$ is a goal type iff for all pairs $(s, s') \in t^g$, s' is a goal state.*

The probability of a goal type existing at a level i of prediction depends on: (a) the probability of at least one state from a type that exists at level $i-1$ generating the goal type; and (b) the probability of the types at level $i-1$ existing. We define $p(i, t, \hat{s}^*, d)$ as the approximated probability of finding a state of type t, in a search tree rooted at state $\hat{s}^*$, at level i, with cost bound of d. We explain how $p(i, t, \hat{s}^*, d)$ is calculated with an example.

Figure 1 illustrates how the probability of a type existing at a level of prediction is calculated. Type t_4 exists at level $i + 1$ iff *at least* one of the types at the previous level generates one or more t_4's. In addition, we have to consider the probability of the types at the previous level existing in the first place. Let $\phi(M, \pi(t'|t), k)$ be the probability of M nodes of type t generating k nodes of type t'. We define $p_{i+1}(t'|t)$ as the probability of type t existing at level i *and* generating *at least* one t' at level $i + 1$. $p_{i+1}(t'|t)$ is calculated as follows.

$$p_{i+1}(t'|t) = p(i, t, \hat{s}^*, d)\big(1 - \phi(N(i, t, \hat{s}^*, d) \cdot \beta_t, \pi(t'|t), 0)\big) \quad (2)$$

here, for ease of notation we define $N(i, t, \hat{s}^*, d)$ as,[3]

$$N(i, t, \hat{s}^*, d) = \sum_{(\hat{s}^*, s^*) \in E} N(i, t, (\hat{s}^*, s^*), d) \quad (3)$$

[3]We write $N(i, t, d)$ instead of $N(i, t, \hat{s}^*, d)$ whenever $\hat{s}^*$ is clear from context.

Algorithm 1 SCP($\hat{s}^*, c$)

1: $d \leftarrow h(\hat{s}^*)$
2: $\mathbb{D} \leftarrow \{\langle N(1,t,\hat{s}^*,d), 1.0\rangle | t \in T\}$
3: $i \leftarrow -1$
4: **while** $i < 0$ **do**
5: $\quad [d,i] \leftarrow$ SCP-CDP($\mathbb{D}, d, 1, -1, c$)
6: **end while**
7: return i

Now $p(i,t,\hat{s}^*,d)$ can be formally defined as follows.[4]

$$p(i,t,\hat{s}^*,d) = 1 - \prod_{u \in T} (1 - p_i(t|u)) \tag{4}$$

Note that the Equations 2 and 4 are recursive as $p_{i+1}(t'|t)$ depends on $p(i,t,\hat{s}^*,d)$ that in turn depends on $p_i(t|u)$. The base of the recursion is defined for $i = 1$ as we use the set of types of the children of a start state s^* to seed our prediction formula. The base of recursion is defined as follows.

$$p(1,t,s^*,d) = \begin{cases} 1 & \text{if } \exists(s^*,\hat{s}^*) \in E \text{ s.t. } T(s^*,\hat{s}^*) = t\,, \\ 0 & \text{otherwise}\,, \end{cases} \tag{5}$$

The pseudocode depicted in Algorithms 1 and 2 shows our algorithm, named Solution Cost Predictor (SCP). The first iteration of the algorithm uses the heuristic value of the start state $\hat{s}^*$ as cost bound (line 1 of Algorithm 1). In line 2 of Algorithm 1 the set $\mathbb{D}$ is initialized with tuples of the form $\langle N(i,t,\hat{s}^*,d), x\rangle$. The first element of the tuple is the number of nodes of type t, at level i, in a search tree rooted at state $\hat{s}^*$, and with a cost bound d, as defined previously. The second element of the tuple is the estimated probability of a state of type t existing at level i of the actual search. In line 2 of Algorithm 1 we are initializing $\mathbb{D}$ to seed our algorithm according to the children of the start state $\hat{s}^*$, thus they exist with probability 1.0.

In Algorithm 2 we iterate over all tuples in set $\mathbb{D}$ to generate the set of tuples $\hat{\mathbb{D}}$ for the next level of prediction (line 4). Pruning is implemented in line 6 of the same algorithm. A type t' is pruned from level $l+1$ of prediction if $h(t')+l+1$ is greater than the bound d, where the value of l is the current level of prediction. In line 7 of Algorithm 2, $N(l+1,t',d)$ and $p(l+1,t',d)$ are updated according to Equations 1 and 4, respectively. Between lines 9 and 12 we keep track of the minimum amount for which the cost bound will be increased for a potential next iteration of the algorithm. Finally, in line 18, the level corresponding to $\hat{\mathbb{D}}$ $(l+1)$ is returned if a goal type is found with probability greater than the confidence parameter c. The function will call itself recursively for the next level of prediction otherwise.

Time Complexity The time complexity of SCP is exactly the same of CDP and it depends on the size of the type system used. The worst case time complexity of one iteration of SCP is calculated as follows. The maximum number of types at a level of prediction is $|T|$. Since each type at a level i of the prediction can generate at most $|T|$ types on the next level of prediction, the worst case time complexity for one "step" of SCP is $O(|T|^2)$. If q is the number of steps the SCP algorithm makes before all types exceed the cost bound d, the worst case time complexity of one iteration of the algorithm is $O(q \cdot |T|^2)$. Even though the worst case analysis is unrealistic, it gives us a good idea of how much faster the prediction can be when compared to the actual search.

[4] We write $p(t,i,d)$ instead of $p(t,\hat{s}^*,i,d)$ whenever $\hat{s}^*$ is clear from context.

Algorithm 2 SCP-CDP($\mathbb{D}, d, l, limit, c$)

1: **if** $\mathbb{D}$ is empty **then**
2: $\quad$ return $[limit, -1]$
3: **end if**
4: **for all** $\langle N(l,t,d), p(l,t,d)\rangle \in \mathbb{D}$ **do**
5: $\quad$ **for all** $t' \in T$ **do**
6: $\quad\quad$ **if** $h(t')+l+1 \leq d$ **then**
7: $\quad\quad\quad$ update tuple $\langle N(l+1,t',d), p(l+1,t',d)\rangle \in \hat{\mathbb{D}}$ according to Equations 1 and 4, respectively.
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ $newlimit \leftarrow h(t')+l+1$
10: $\quad\quad\quad$ **if** $newlimit < limit$ or $limit < 0$ **then**
11: $\quad\quad\quad\quad$ $limit \leftarrow newlimit$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: **end for**
16: **for all** $\langle N(l+1,u,d), p_u\rangle \in \hat{\mathbb{D}}$ **do**
17: $\quad$ **if** u is a goal type and $p_u \geq c$ **then**
18: $\quad\quad$ return $[limit, l+1]$
19: $\quad$ **end if**
20: **end for**
21: SCP-CDP($\hat{\mathbb{D}}, bound, l+1, limit$)

Accuracy Experiments

In the first set of experiments we are interested in evaluating the prediction accuracy of the algorithm presented in the previous section. We report results for predictions of single start states and sets of start states We ran experiments on the 15-puzzle and the 10-pancake puzzle. Together the two domains offer a good challenge for our prediction formula as they have different properties. The former has a small branching factor and deep solutions, while the latter has a large branching factor and shallow solutions.

The results of the experiments show that (1) the average cost predicted by our algorithm is almost exactly equal to the average optimal cost for a set of start states, and (2) our algorithm makes very accurate predictions for single start states. Moreover, we show that SCP produces more accurate predictions than the bootstrapped heuristic (Jabbari Arfaee, Zilles, and Holte 2010) for the 15-puzzle, which is known to be a very accurate inadmissible heuristic. We evaluated the accuracy of the predictions for a set of start states with the *ratio of the average prediction*. The *ratio of the average prediction* is the sum of the predicted cost for each instance in the set of start states, divided by the sum of the actual cost for each instance in the set of start states. A perfect score ac-

cording to this measure is 1.00. The ratio of the average prediction is not a good measure when we want to evaluate the accuracy of our system for single start states as overestimations and underestimations of the predictions might cancel out. The correct measure in this case is the absolute error. In addition, our absolute error will be an average over start states with a given solution cost. Thus, for each instance with optimal solution cost of C one computes the absolute difference of the predicted solution cost and C, adds these up, and divides by the number of start states with optimal solution cost C multiplied by C. A perfect score according to this measure is 0.00. The measures will be rounded up to two decimal places. Thus a ratio of 1.00 and an error of 0.00 do not imply perfect predictions.

The upper part of Tables 1 and 2 presents the results for predictions of single start states. The first column on the left, named "Cost", shows the actual solution cost to solve a problem. The "Cost" column is followed by estimation results. The bottom part of the tables presents the results for predictions of a set of start states.

In our experiments SCP uses the $\pi(t|u)$-values given by ϵ-truncation. It is important that the goal depth prediction is used in conjunction with ϵ-truncation as the rare events reduce the accuracy of the solution cost prediction. When we set the low $\pi(t|u)$-values to zero, we are eliminating the "rare paths" to the goal type in the type system state space. Preliminary experiments had shown that if rare events are not disregarded, the goal depth predictions will often be underestimations of the actual costs. The number of states used to derived the ϵ_i-values for each experiment is defined below. For both domains we used the T_{gc} type system. The heuristic function used in the type system is also defined below for each experiment.

The Pancake Puzzle

The 10-pancake puzzle has $10!$ states and a maximum optimal solution depth of 11. We used 100 random start states to determine the ϵ_i and 5,000 to measure prediction accuracy. The heuristic used to define T_{gc} was a pattern database (PDB (Culberson and Schaeffer 1996)) based on the smallest four pancakes. The confidence parameter was set to 0.9 in this experiment.

Table 1 shows the results for the pancake puzzle. We used the same heuristic used to define T_{gc} as a baseline. SCP makes fairly accurate predictions for single start states and almost perfect predictions for the set of 5,000 random start states. It is important to note that missing by one the actual solution cost already makes the absolute error large for the pancake puzzle. For instance, if the optimal solution cost for one start state is 7 and the predictor predicts 8, then the absolute error according to our measure would be of $(|8 - 7|)/1 \cdot 7 = 0.14$.

Absolute Error		
Cost	h	SCP
7	0.24	0.17
8	0.26	0.07
9	0.28	0.03
10	0.29	0.07
Ratio of Average Prediction		
	h	SCP
	0.72	1.01

Table 1: 10-pancake. SCP and PDB (0-4).

The Sliding-Tile Puzzle

For the 15-puzzle we used 5 random start states to determine the ϵ_i and 1,000 to measure prediction accuracy. The 15-puzzle has $16!/2$ states reachable from the goal state. Thus, unlike the 10-pancake puzzle, we could not enumerate all states to define $\pi(t|u)$ and β_t. Therefore, to define them, one billion random states were sampled and, in addition, we used the process described by Zahavi *et al.* (2010) to non-randomly extend the sampling: we sampled the child of a sampled state if the type of that child had not yet been sampled. The heuristic used to define T_{gc} was the sum of the Manhattan Distance of the tiles in the puzzle. The confidence parameter was set to 1.0 in this experiment. In theory, the probability of a type existing at a level of search can only be 1.0 if all types in the path of that type in the "prediction search tree" are generated with probability 1.0. However, in a computer implementation with fixed-precision floating-point numbers the value of $\phi(N(i-1,t)\beta_t, \pi(u|t), 0)$ (probability of generating zero u out of $N(i-1,t)\beta_t$ nodes of type t) goes to zero for large values of $N(i-1,t)\beta_t$, even if the value of $\pi(u|t)$ is relatively small. In all the experiments we observed that eventually the probability of finding a goal type goes to one due to limited precision.

In this experiment we compare the accuracy of the SCP predictions with the estimates of the bootstrapped heuristic (Jabbari Arfaee, Zilles, and Holte 2010). The bootstrap algorithm presented in Jabbari Arfaee *et al.* (2010) iteratively improves an initial heuristic function by solving a set of successively more difficult training problems in each iteration. It starts with a time limited search on the set of training instances using an initial (weak) heuristic function. A heuristic is learned from the training instances solved at the end of an iteration. Failing to solve enough training instances in an iteration, the bootstrap algorithm increases the time limit to fill the gap between the difficulty of the training instances and the weakness of the current heuristic function. This process is then repeated on the unsolved training instances using the heuristic learned during the previous iteration. This algorithm was shown to create very effective heuristics for a variety of classic heuristic search domains.

In addition, we show the accuracy of two admissible heuristics: Manhattan Distance and 7-8 additive pattern databases. Manhattan Distance heuristic (shown as MD in the table) is a popular and easy to implement heuristic function. The 7-8 additive pattern database (shown as APDB on the table) is a known effective heuristic for the 15 puzzle (Felner, Korf, and Hanan 2004; Korf and Felner 2002). It consists of the sum of two heuristics based on disjoint pattern databases, one based on the first 7 tiles and the other based on the 8 remaining tiles.

Table 2 shows the results. Even though the additive PDBs

represent a major advance in the creation of admissible heuristics, they present poor predictions when compared to the bootstrapped heuristic and to the SCP predictions. This result illustrates the intuitive idea presented earlier in the paper that admissible heuristics tend to make poor estimates as they are biased to never overestimate the optimal solution cost. The SCP predictions are more accurate than the bootstrapped heuristic for all the solution costs reported, with the exception of solution costs of 44, 45 and 46. The highlighted values on the table represent statistically significant results with 95% confidence[5]. The SCP predictions are particularly accurate around the value of 52, which is known to be the median solution cost value for the 15-puzzle. The prediction of the bootstrapped heuristic for the set of start states has a ratio of 1.06, while the SCP prediction is nearly perfect yielding a ratio of 1.00.

Absolute Error				
Cost	MD	APDB	Bootstrapped	SCP
44	0.30	0.08	0.08	0.08
45	0.29	0.09	0.07	0.09
46	0.33	0.08	0.04	0.05
47	0.32	0.07	0.07	0.05
48	0.29	0.10	0.08	**0.05**
49	0.29	0.09	0.07	**0.04**
50	0.29	0.12	0.07	**0.04**
51	0.29	0.13	0.07	**0.02**
52	0.30	0.14	0.07	**0.02**
53	0.30	0.15	0.06	**0.03**
54	0.30	0.16	0.07	**0.02**
55	0.29	0.18	0.07	**0.03**
56	0.30	0.19	0.06	**0.03**
57	0.29	0.22	0.06	**0.04**
58	0.28	0.24	0.07	**0.04**
59	0.27	0.22	0.08	**0.04**
60	0.28	0.23	0.06	0.05
61	0.27	0.27	0.07	0.05
62	0.27	0.31	0.09	**0.06**
63	0.27	0.26	0.08	0.06

Ratio of Average Prediction				
	MD	APDB	Bootstrapped	SCP
	0.70	0.85	1.06	**1.00**

Table 2: 15-Puzzle. A Bootstrapped Heuristic and SCP.

Applications of the Solution Cost Prediction

For scenarios, such as decision problems (*e.g.*, is there a vertex cover of size k), where knowing the solution cost is enough, the merit of predicting it is clear. However, in some other cases one would like to find a path to the goal. Next we describe two ways that the accurate inadmissible heuristics provided by SCP can be used to enhance search algorithms: 1) as a bound for a bounded cost search algorithm, and 2) as a tool for tuning a parametric search algorithm.[6]

Using Cost Prediction as a Bound

Assuming that the predictions are perfect, a search algorithm could be biased to optimally solve problems as follows. First, every state s with its $g(s) + h(s)$ value larger than the predicted solution cost could be pruned off, so that fewer nodes are expanded. Second, a goal test should be performed only for nodes with g-value equal to the predicted cost, reducing the time per node of these nodes.

Recently, Stern *et al.* (2011) presented the Potential Search algorithm (denoted hereafter as PTS), a cost bounded search algorithm that efficiently searches for a solution with cost less than or equal to a given cost bound (Stern, Puzis, and Felner 2011). This is done by focusing search on nodes that are more likely to lead to a goal with cost less than or equal to the desired bound. Stern *et al.* showed theoretically that PTS can be implemented for the 15-puzzle by using a simple Best-First Search with a cost function of $\frac{h}{Y-g}$, where Y is the desired bound.

Algorithm 3 PTS+Prediction

```
P ← RunCostPrediction()
Run PTS with bound P
if PTS did not find goal with cost ≤ P then
   Run optimal solver
end if
```

The PTS algorithm requires a bound on the desired solution cost. We propose a variant of PTS (summarized in Algorithm 3) that does not require such a bound, by setting the bound to be the prediction returned by SCP. When the prediction is equal to or greater than the optimal solution cost, PTS finds a goal with the predicted cost (or less) quickly. When the prediction underestimates the optimal cost, PTS will exhaustively search all states s with $g(s) + h(s) \leq P$, and then run an optimal solver (*e.g.*, A* or IDA*). In our experiments we used A* in such cases. Note that any optimal solver has to expand all the states s with $g(s) + h(s) \leq P$ before finding the optimal cost. Therefore, the overhead of running PTS in cases where the predicted solution cost underestimates the actual value is relatively small.

PTS Experiments The experiment with the PTS algorithm bounded by the solution cost prediction was run on the 15-puzzle. The heuristic function used to guide PTS and A* was Manhattan Distance. We chose to use Manhattan Distance as our goal was not to solve instances of the 15-puzzle faster, but to demonstrate how the solution cost prediction can be used to speed up search. In our experiment we add a slack value to the predicted solution cost in order to increase the likelihood of finding a goal within the bound. We report

[5]The results tend not be significant on the tails as start states with solution costs lower than 48 and greater than 59 are less common to be encountered in a uniform sample of the state space.

[6]In the experiments of this section we ignore the "number of nodes expanded" by the prediction algorithm as the time complexity analysis suggests SCP expands considerably fewer nodes than the actual search.

Algorithm	Suboptimality	Exp. A*	Solved
A*	0.00	1	90%
PTS+P	0.01	28	90%
PTS+P+1	0.01	22	90%
PTS+P+2	0.02	34	90%
PTS+P+3	0.02	43	90%
PTS+P+4	0.03	56	93%
PTS+P+5	0.05	68	93%
PTS+P+6	0.06	79	93%
PTS+P+7	0.07	93	96%
PTS+P+8	0.08	145	97%
PTS+P+9	0.09	145	99%

Table 3: Reduction in expanded nodes with PTS+Prediction.

results for different slack values (denoted by PTS+P+S in Table 3, where S is the slack value). The memory allowed to solve a problem instance was limited to 1 gigabyte.

The results are shown in Table 3 and they are an average over 1,000 random start states. The column "Algorithm" denotes the algorithm used to generate the result of that row. The next column shows how suboptimal the solutions are. The measure for suboptimality, also referred in this paper as solution quality, is the following. For each start state one sums up the difference of the solution cost found by the algorithm with the optimal solution cost, and divides this sum by the sum of the optimal solution costs. If all the solutions are optimal this measure will be 0.00. The column "Exp. A*" tells us the reduction factor of the number of nodes expanded when compared to A*. The reduction factor is calculated by dividing the number of nodes expanded by A* to solve the set of problems by the number of nodes expanded by the row's algorithm to solve the same set of problems. Finally, the last column shows the number of problems solved with a memory limit of 1 gigabyte.

The first row of Table 3 shows the results for A* with Manhattan Distance. As all the problems are solved optimally, the suboptimality value is of 0.00. The reduction factor is obviously 1, and the percentage of problems is 90%. By using the solution cost prediction as a bound for PTS we solve the same percentage of problems as A*, but expanding 28 times less nodes, also at a cost of finding solutions of slightly lower quality. As we start to add the slack value, we start to solve the problems even faster, at a cost of finding solutions of lower quality. For example, adding a slack value of 9 to the predicted cost (denoted by PTS+P+9) results in PTS expanding 145 times less nodes than A* and solving 99% of the instances.

Tuning Search Parameters

Many search algorithms have tunable parameters. A prominent example is the w-value of Weighted A* (Pohl 1970) and Weighted IDA* (Korf 1985). These algorithms use the cost function $f(s) = g(s) + w \times h(s)$, where w is a parameter determining the weight given to the heuristic. For a w-value of 1, Weighted A* and Weighted IDA* reduce to A* and IDA*, respectively. The value of w is known to have a major impact on the runtime and on the solution quality of these algorithms. For example, Weighted A* is often part of planning systems participating in the satisficing track of the International Planning Competition (IPC)[7]. In the IPC setting a planning system is required to solve problems having no prior knowledge about them within a time constraint. In this case it is challenging to determine w-values that give a good trade-off between solution quality and runtime.

One possible approach for finding an effective w-value is as follows. One solves optimally a sufficiently large number of problem instances, calculates for every solved instance the ratio between its heuristic value and its optimal solution cost, and set the w as the average of these ratios. The idea is to "learn" a global error ratio of the heuristic function. This approach has two shortcomings. First, solving a set of instances optimally is often prohibitively time-consuming. Second, heuristic functions are usually more accurate in some parts of the state space than in others. As suggested by Valenzano *et al.* (2010), ideally we would have a possibly different w-value for each problem instance. We describe next how we use `SCP` to overcome these two shortcomings.

Weighted IDA* Experiments Once again our experiment is run on the 15-puzzle with MD as the heuristic function. The results are averaged over 1,000 random start states. With this experiment we aim at comparing three different approaches to set the w-value. (i) As explained above, we solve the entire set of start states and use the average of the ratios of the optimal solution cost to solve the instances by their heuristic value as the w-value. (ii) We do exactly the same as the previous approach, except that instead of the optimal solution costs we use the predicted solution costs. (iii) For each start state we predict its solution cost and use the ratio between the predicted value and its heuristic value as the w-value for that particular state. The approaches (i), (ii) and (iii) appear under the headings of "AvgOpt", "AvgPred" and "Pred", respectively, in the table of results.

The results of the experiment show (1) that on average the Pred approach produces solutions with better quality and it expands fewer nodes; and (2) that the Pred approach usually presents solutions with better quality and it expands fewer nodes when compared to the other methods across start states with different solution costs.

Table 4 presents the results. The first column shows different solution costs; results on a row are averages over start states with the corresponding solution cost. For suboptimality we used the same measure used in the experiment with the PTS algorithm. For search effort we show the average number of nodes expanded. In the last row of Table 4 we present the average over all the start states, independent of their solution cost. A row is highlighted if the Pred approach produces solutions of better quality and expands fewer nodes, or if it produces solutions of the same quality as the other methods but it expands fewer nodes.

It is interesting to note that the values for AvgPred and AvgOpt are identical. This happens due to the nearly perfect predictions for a set of start states, as shown in Table 2. However, the predictions are much cheaper computationally

[7] *http://ipc.icaps-conference.org*

	Suboptimality			Average N. Nodes Expanded		
Cost	AvgOpt	AvgPred	Pred	AvgOpt	AvgPred	Pred
48	0.05	0.05	0.06	1,276,642	1,276,642	569,526
49	0.05	0.05	0.06	865,834	865,834	593,977
50	0.04	0.04	0.05	812,614	812,614	608,403
51	**0.04**	**0.04**	**0.04**	**1,417,335**	**1,417,335**	**806,905**
52	**0.05**	**0.05**	**0.05**	**2,407,257**	**2,407,257**	**1,975,033**
53	**0.04**	**0.04**	**0.04**	**1,848,260**	**1,848,260**	**1,429,117**
54	**0.04**	**0.04**	**0.03**	**2,540,969**	**2,540,969**	**2,140,678**
55	**0.05**	**0.05**	**0.04**	**4,952,380**	**4,952,380**	**3,707,499**
56	**0.04**	**0.04**	**0.03**	**3,678,749**	**3,678,749**	**3,523,372**
57	0.05	0.05	0.03	3,511,613	3,511,613	3,648,784
58	0.05	0.05	0.03	2,624,178	2,624,178	5,390,799
Total	**0.05**	**0.05**	**0.04**	**2,407,027**	**2,407,027**	**2,149,843**

Table 4: Tile puzzle, wIDA* with different weights.

when compared to optimally solving every instance in the set of start states.

We can also observe a natural trade-off between solution quality and the number of nodes expanded: solutions closer to the optimal value come at the expense of expanding more nodes. Notice, however, that AvgOpt and AvgPred are always either expanding more nodes or producing solutions of poorer quality when compared to Pred. On the other hand, there are several rows (which are highlighted) where Pred expands less nodes and produces solutions of better quality when compared to the two other variants. Moreover, for the average over all solution costs, Pred (displayed in the last row of Table 4) produces solutions with better quality and it expands fewer nodes. This result suggests that the w-value should be tuned for each problem instance individually.

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 the path from the start state to a goal state. In this paper we propose an algorithm based on the `CDP` prediction framework to accurately and efficiently predict the *solution cost* of a problem. Our predictor, which can also be seen as a heuristic function, differs from previous works as we 1) do not require it to be fast enough to be used to guide search algorithms; 2) do not favor admissibility; 3) aim at making accurate predictions 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 our predictor makes better predictions when compared to the bootstrapped heuristic. In addition, we showed novel ways of using accurate inadmissible heuristics to enhance heuristic search algorithms. Namely, we used our predictor to calculate a bound for the PTS algorithm and to tune the w-value of Weighted IDA*. In both cases we achieved major search speedups when compared to the search algorithms without the enhancement.

Acknowledgements

The authors would like to thank Rob Holte, Ariel Felner, Sandra Zilles and the anonymous reviewers for reading earlier versions of this paper and making thoughtful comments. 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, and the Alberta Ingenuity Centre for Machine Learning (AICML).

References

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence (LNAI 1081)* 402–416.

Ernandes, M., and Gori, M. 2004. Likely-admissible and sub-symbolic heuristics. In *ECAI*, 613–617.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *J. Artif. Intell. Res. (JAIR)* 22:279–318.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Jabbari Arfaee, S.; Zilles, S.; and Holte, R. C. 2010. Bootstrap learning of heuristic functions. In *SoCS*, 52–60.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2):9–22.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.* 27(1):97–109.

Lelis, L.; Zilles, S.; and Holte, R. C. 2011. Improved prediction of IDA*s performance via ϵ-truncation. In *SoCS*.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artif. Intell.* 1(3-4):193–204.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from multiple heuristics. In *AAAI*, 357–362.

Stern, R.; Puzis, R.; and Felner, A. 2011. Potential search: a bounded-cost search algorithm. In *ICAPS*, 234–241.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Thayer, J.; Dionne, A.; and Ruml, W. 2011. Learning inadmissible heuristics during search. In *ICAPS*, 250–257.

Valenzano, R. A.; Sturtevant, N. R.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *ICAPS*, 177–184.

Yang, F.; Culberson, J. C.; Holte, R. C.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *J. Artif. Intell. Res. (JAIR)* 32:631–662.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of IDA* using conditional distributions. *J. Artif. Intell. Res. (JAIR)* 37:41–83.