Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

Fast and Accurate Predictions of IDA*'s Performance

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

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

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

Abstract

Korf, Reid and Edelkamp initiated a line of research for developing methods (KRE and later CDP) that predict the number of nodes expanded by IDA* for a given start state and cost bound. Independent of that, Chen developed a method (SS) that can also be used to predict the number of nodes expanded by IDA*. In this paper we advance these prediction methods. First, we develop a variant of CDP that can be orders of magnitude faster than CDP while producing exactly the same predictions. Second, we show how ideas developed in the KRE line of research can be used to substantially improve the predictions produced by SS. Third, we make an empirical comparison between our new enhanced versions of CDP and SS. Our experimental results point out that CDP is suitable for applications that require less accurate but very fast predictions, while SS is suitable for applications that require more accurate predictions but allow more computation time.

Introduction

Usually the user of tree search algorithms does not know *a priori* how long search will take to solve a particular problem instance. One problem instance might be solved within a fraction of a second, while another instance of the same domain might take years to be solved. Korf, Reid, and Edelkamp (2001) started a line of research aimed at efficiently predicting the number of nodes expanded on an iteration of IDA* (Korf 1985) for a given cost bound. They presented an efficient algorithm, known as KRE, that makes predictions when a consistent heuristic is employed. Zahavi et al. (2010) created CDP, an extension of KRE that also makes accurate predictions when the heuristic employed is inconsistent. CDP works by sampling the state space as a preprocessing step with respect to a *type system*, i.e., a partition of the state space. The information learned during sampling is used to efficiently emulate the IDA* search tree and thus to approximate the number of nodes expanded on an iteration of the algorithm. CDP is reviewed in detail below.

Independent of the KRE line of research, Knuth (1975) created a method to efficiently predict the size of a search tree by making random walks from the root node. Chen (1989) extended Knuth's method to use a *stratification* of the state space to reduce the variance of sampling. Like CDP's type systems, the stratifier used by Chen is a partition of the state space. The method developed by Chen, Stratified Sampling, or SS, relies on the assumption that nodes in the same *part of the partition* will root subtrees of the same size. SS is reviewed in detail below.

In this paper we connect and advance these prediction methods. We make the following contributions. First, we develop a variant of CDP, named Lookup CDP (or L−CDP for short), that decomposes the CDP predictions into independent subproblems. The solutions for these subproblems are computed in a preprocessing step and the results stored in a lookup table to be reused later during prediction. L−CDP can be orders of magnitude faster than CDP while it is guaranteed to produce the same predictions as CDP. Similar to a pattern database (PDB) (Culberson and Schaeffer 1996), L−CDP's lookup table is computed only once, and the cost of computing it can be amortized by making predictions for a large number of instances. Second, we show that the type systems employed by CDP can also be used as stratifiers for SS. Our empirical results show that SS employing CDP's type systems substantially improves upon the predictions produced by SS using the type system proposed by Chen. Third, we make an empirical comparison between our enhanced versions of CDP and SS considering scenarios that require (1) fast, and (2) accurate predictions. The first scenario represents applications that require less accurate but almost instantaneous predictions. For instance, quickly knowing the size of search subtrees would allow one to fairly divide the search workload among different processors in a parallel processing setting. The second scenario represents applications that require more accurate predictions but allow more computation time. Our experimental results point out that if L−CDP's preprocessing time is acceptable or can be amortized, it is suitable for applications that require less accurate but very fast predictions, while SS is suitable for applications that require more accurate predictions but allow more computation time.

This paper is organized as follows. After reviewing CDP and presenting L−CDP, we show empirically that L−CDP can be orders of magnitude faster than CDP. Then, we present SS and show empirically that the type systems developed to be used with CDP can substantially improve upon

the predictions produced by SS. Finally, we compare SS employing CDP's type systems with L−CDP.

The CDP Prediction Framework

We now review the CDP system. In CDP, predictions of the number of nodes expanded by IDA* for a given cost bound are based on a partition of the nodes in an IDA* search tree. We call this partition a *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 $S(s^*)$. For every $s \in S(s^*)$, $T(s)$ denotes the unique $t \in T$ with $s \in t$.*

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

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. (2010) use the information of the parent of a node s when computing s's type so that CDP is able to make accurate prediction of the number of nodes expanded on an iteration of IDA* when parent pruning is used.

Note that, as in Zahavi et al.'s work, all type systems considered in this paper have the property that $h(s) = h(s')$ if $T(s) = T(s')$. We assume this property in the formulae below, and denote by $h(t)$ the value $h(s)$ for any s such that $T(s) = t$.

Definition 2. *Let $t, t' \in T$. $p(t'|t)$ denotes the average fraction of the children generated by a node of type t that are 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 predicted number of nodes expanded by IDA* with parent pruning for start state s^*, cost bound d, heuristic h, and type system T is formalized as follows.

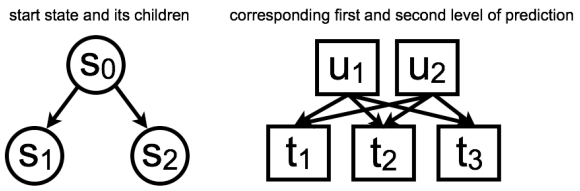

Figure 1: The first step of a CDP prediction for start state s_0.

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

Here the first summation iterates over the children of the start state s^*. Assuming unit-cost edges, in the second summation we account for g-costs from 1 to the cost bound d; any value of i greater than d would be pruned by IDA*. The innermost summation iterates over the types in T. Finally, $N(i, t, s, d)$ is the number of nodes n with $T(n) = t$ and n at level i of the search tree rooted at s. A value of one is added to Equation 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.

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

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

$$\sum_{u \in T} N(i-1, u, s, d)\pi(t|u)\beta_u P(t, i, d)\,. \tag{2}$$

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

Example 1. *Consider the example in Figure 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 Figure 1. We now use the values of π and β to estimate the types of the nodes on the next level of search. For instance, to estimate how many nodes of type t_1 there will be on the next level of search we sum up the number of nodes of type t_1 that are generated by nodes of type u_1 and u_2. Thus, the estimated number of nodes of type t_1 at the second level of search is given by $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$. If $h(t_1) + 2$ (heuristic value of type t_1 plus its g-cost) exceeds the cost*

bound d, then the number of nodes of type t_1 is set to zero, because IDA would have pruned those nodes. This process is repeated for all types at the second level of prediction. Similarly, we get estimates for the third level of the search tree. Prediction goes on until all types are pruned. The sum of the estimated number of nodes of every type is the estimated number of nodes expanded by IDA* with cost bound d for start state s_0.*

CDP is seeded with the types of the children of the start state s^*, as shown in Equation 1. Zahavi et al. (2010) showed that seeding the prediction formula with nodes deeper in the search tree improves the prediction accuracy at the cost of increasing the prediction runtime. In this improved version of CDP one collects C_r, the set of nodes s such that s is at a distance $r < d$ from s^*. Then the prediction is made for a cost bound of $d - r$ when nodes in C_r seed CDP.

Lelis et al. (2011) identified a source of error in the CDP predictions known as the *discretization effect*. They also presented a method, ϵ-truncation, that counteracts the discretization effect as a preprocessing step. ϵ-truncation was used in all CDP experiments in this paper.

Lookup CDP

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

The following procedure summarizes Lookup CDP.

1. As in CDP, we sample the state space to approximate the values of $p(t'|t)$ and b_t and to compute the ϵ-values needed for ϵ-truncation (Lelis, Zilles, and Holte 2011).
2. We compute the predicted number of nodes expanded for each pair (t, d) and store the results in a lookup table. This is done with dynamic programming: pairs (t, d) with smaller values of d are computed first. This way, when computing the (t, k)-values for a fixed k, we can use the (t, k')-values with $k' < k$ that were already computed.
3. For start state s^* and cost bound d we collect the set of nodes C_r. Then, for each node in C_r with type t, we sum the entries of the $(t, d-r)$-values from our lookup table. This sum added to the number of nodes expanded while collecting the nodes in C_r is the predicted number of nodes expanded by IDA* for s^* and d.

The worst-case time complexity of a CDP prediction is $O(|T|^2 \cdot (d-r) + Q_r)$ as there can be $|T|$ types at a level of prediction that generate $|T|$ types on the next level. $d - r$ is the largest number of prediction levels in a CDP run. Finally, Q_r is the number of nodes generated while collecting C_r. The time complexity of an L−CDP prediction (Step 3 above) is $O(Q_r)$ as the preprocessing step has reduced the L−CDP computation for a given type to a constant-time table lookup. The preprocessing L−CDP does is not significantly more costly than the preprocessing CDP does because the runtime of the additional preprocessing step of L−CDP (Step 2 above) is negligible compared to the runtime of Step 1 above. Both CDP and L−CDP are only applicable when one is interested in making a large number of predictions so that their preprocessing time is amortized.

Lookup CDP Experimental Results

We now compare the prediction runtime of CDP with L−CDP. Note that the accuracy of both methods is the same as they make exactly the same predictions. Thus, here we only report prediction runtime. All our experiments were run on an Intel Xeon CPU X5650, 2.67GHz. Unless stated otherwise, all our experiments are run on the following domains and heuristics: 4x4 sliding-tile puzzle (15-puzzle) with manhattan distance (MD); 15 pancake puzzle with a PDB heuristic that keeps the identities of the smallest eight pancakes; Rubik's Cube with a PDB based on the corner "cubies" (Korf 1997) with the additional abstraction that the six possible colors of the puzzle are mapped to only three colors. Both L−CDP and CDP used the same type system and the same values of $\pi(t'|t)$ and β_t. We used a set of 1,000 random start states to measure the runtime, but, like Zahavi et al., we only predict for a cost bound d and start state s^* if IDA* would actually search with the cost bound d for s^*.

Table 1 presents the average prediction runtime in seconds for L−CDP and CDP for different values of r and d. The bold values highlight the faster predictions made by L−CDP. For lower values of r, L−CDP is orders of magnitude faster than CDP. However, as we increase the value of r the two prediction systems have similar runtime. For instance, with the r-value of 25 on the 15-puzzle L−CDP is only slightly faster than CDP as, in this case, collecting C_r dominates the prediction runtime.

The Knuth-Chen Method

Knuth (1975) presents a method to predict the size of a search tree by repeatedly performing a random walk from the start state. Each random walk is called a *probe*. Knuth's method assumes that all branches have a structure similar

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

Table 1: L-CDP and CDP runtime (seconds).

to that of the path visited by the random walk. Thus, walking on one path is enough to predict the structure of the entire tree. Knuth noticed that his method was not effective when the tree being sampled is unbalanced. Chen (1992) addressed this problem with a stratification of the search tree through a type system (or stratifier) to reduce the variance of the probing process. We call Chen's method SS.

We are interested in using SS to predict the number of nodes expanded by IDA* with parent pruning. Like CDP, when IDA* uses parent pruning, SS makes more accurate predictions if using type systems that account for the information of the parent of a node. Thus, here we also use type systems that account for the information about the parent of node n when computing n's type.

SS can be used to approximate any function of the form

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

where z is any function assigning a numerical value to a node, and, as above, that $S(s^*)$ is the set of nodes of a search tree rooted at s^*. $\varphi(s^*)$ represents a numerical property of the search tree rooted at s^*. For instance, if $z(s)$ is the cost of processing node s, then $\varphi(s^*)$ is the cost of traversing the tree. If $z(s) = 1$ for all $s \in S(s^*)$, then $\varphi(s^*)$ is the size of the tree.

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

Algorithm 1 Stratified Sampling

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

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

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

One run of SS is called a *probe*. Each probe generates a possibly different value of $\hat{\varphi}(s^*, T)$; averaging the $\hat{\varphi}(s^*, T)$ value of different probes improves prediction accuracy. In fact, Chen proved that the expected value of $\hat{\varphi}(s^*, T)$ converges to $\varphi(s^*)$ in the limit as the number of probes goes to infinity.

Algorithm 1 describes SS in detail. For convenience, the set A is divided into subsets, one for every layer in the search tree; hence $A[i]$ is the set of types encountered at level i. In SS the types are required to be partially ordered: a node's type must be strictly greater than the type of its parent. Chen suggests that this can be guaranteed by adding the depth of a node to the type system and then sorting the types lexicographically. In our implementation of SS, due to the division of A into the $A[i]$, if the same type occurs on different levels the occurrences will be treated as though they were different types – the depth of search is implicitly added to any type system used in our SS implementation.

$A[1]$ is initialized to contain the children of s^* (Line 3). $A[1]$ contains only one child s for each type. We initialize the weight in a representative-weight pair to be equal to the

number of children of s^* of the same type. For example, if s^* generates children s_1, s_2, and s_3, with $T(s_1) = T(s_2) \neq T(s_3)$, then $A[1]$ will contain either s_1 or s_2 (chosen at random) with a weight of 2, and s_3 with a weight of 1.

The nodes in $A[i]$ are expanded to get the nodes of $A[i+1]$ as follows. In each iteration (Lines 6 through 17), all nodes in $A[i]$ are expanded. The children of each node in $A[i]$ are considered for inclusion in $A[i+1]$. If a child c of node s has a type t that is already represented in $A[i+1]$ by another node s', then a *merge* action on c and s' is performed. In a merge action we increase the weight in the corresponding representative-weight pair of type t by the weight $w(c)$. c will replace s' according to the probability shown in Line 11. Chen (1992) proved that this probability reduces the variance of the estimation. Once all the nodes in $A[i]$ are expanded, we move to the next iteration. In the original SS, the process continued until $A[i]$ was empty; Chen was assuming the tree was naturally bounded.

Chen used SS's approximation of the number of nodes in a search tree whose f-value did not exceed the cost bound d as an approximation of the number of nodes expanded by IDA* with cost bound d. However, when an inconsistent heuristic[1] is used, there can be nodes in the search tree whose f-values do not exceed the cost bound d but are never expanded by IDA* as one of their ancestors had an f-value that exceeded d. Predictions made by SS as described by Chen (1992) will overestimate the number of nodes expanded by IDA* when an inconsistent heuristic is used. We modify SS to produce more accurate predictions when an inconsistent heuristic is employed by adding Line 8 in Algorithm 1. Now a node is considered by SS only if all its ancestors are expanded. Another positive effect of Line 8 in Algorithm 1 is that the tree becomes bounded by d.

Better Type Systems for SS

The prediction accuracy of SS, like that of CDP, depends on the type system used to guide its sampling (Chen 1989). Chen suggests a type system that counts the number of children a node generates as a general type system to be used with SS. We now extend Chen's general type system to include information about the parent of the node so it makes more accurate predictions when parent pruning is considered. We define it as $T_{nc}(s) = nc(s)$, where $nc(s)$ is the number of children a node s generates accounting for parent-pruning. Recall that in our implementation of SS the depth of search is implicitly considered in any type system.

We define a type system to be *pure* if it groups together nodes that root subtrees of the same size. It is easy to see that SS using a *pure* type system makes perfect predictions. A trivial example of a pure type system is the one that maps every node s to a unique type t. However, prediction computations using this type system would be costly. Pure type systems that substantially compress the original state space are often hard to design. Thus, we must employ type systems that reduce the variance (not necessarily to zero as a pure type system does) of the size of subtrees rooted at nodes of the same type, but that at the same time substantially compress the state space.

In order to reduce the variance of the size of subtrees rooted at nodes of the same type it is useful to include the *heuristic value* of the node in the type system. Intuitively, search trees rooted at nodes with higher heuristic value are expected to have fewer nodes when compared to trees rooted at nodes with lower heuristic value as IDA* prunes "more quickly" nodes with higher heuristic value. The following type systems were defined by Zahavi et al. (2010) and Lelis et al. (2011).

$T_h(s) = (h(parent(s)), h(s))$, where $parent(s)$ returns the parent of s in the search tree;

$T_c(s) = (T_h(s), c(s,0), \ldots, c(s,H))$, where $c(s,k)$ is the number of children of s whose heuristic value is k, and H is the maximum heuristic value a node can assume. Clearly T_c is larger than T_h;

$T_{gc}(s) = (T_c(s), gc(s,0), \ldots, gc(s,H))$, where $gc(s,k)$ is the number of grandchildren of s whose heuristic value is k. Again, clearly T_{gc} is larger than T_c.

We now show empirically that using these type systems instead of Chen's substantially improves SS's predictions.

Comparison of SS with Different Type Systems

We say that a prediction system V *dominates* another prediction system V' if V is able to produce more accurate predictions in less time than V'. In our tables of results we highlight the runtime and error of a prediction system if it dominates its competitor. The results presented in this section experimentally show that SS employing type systems that account for the heuristic value dominates SS employing the general type system introduced by Chen on the domains tested.

In our experiments, prediction accuracy is measured in terms of the *Relative Unsigned Error*, which is calculated as,

$$\frac{\sum_{s \in PI} \frac{|Pred(s,d) - R(s,d)|}{R(s,d)}}{|PI|}$$

where PI is the set of problem instances, $Pred(s,d)$ and $R(s,d)$ are the predicted and actual number of nodes expanded by IDA* for start state s and cost bound d. A perfect score according to this measure is 0.00.

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

The results for the 15-puzzle when using the inconsistent heuristic defined by Zahavi et al. (2010) are presented on the upper part of Table 2. We chose the number of probes so

[1] A heuristic h is consistent iff $h(s) \leq c(s,t) + h(t)$ for all states s and t, where $c(s,t)$ is the cost of the cheapest path from s to t. A heuristic is called inconsistent if it is not consistent.

that we could show the dominance of T_h over T_{nc}. For T_h we used 50 probes in each prediction, while for T_{nc} we used 5,000. Given the same number of probes as T_h (50), T_{nc} was faster than T_h, but produced predictions with error approximately three times higher than T_h. When the number of probes was increased to improve accuracy, T_{nc} eventually got slower than T_h before its accuracy equalled T_h's. In Table 2 we see that when employing a type system that considers the information provided by a heuristic function SS produces more accurate predictions in less time than when employing T_{nc}. The dominance of SS employing the type systems that account for the heuristic values over T_{nc} is also observed in experiments run on the 15 pancake puzzle and on Rubik's Cube. Improvements over T_{nc} were observed not only when using T_h or T_c, but also when using T_{gc} in all three domains.

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

Table 2: SS employing different type systems.

Comparison Between L−CDP and SS

In this section we make an empirical comparison of our enhanced versions of CDP and SS. We analyze two scenarios. In both scenarios we assume the user is interested in making predictions for a large number of problem instances, so that the preprocessing time of L−CDP is amortized. In the first scenario, after preprocessing, we are interested in making predictions very quickly. In the second scenario, we allow the prediction algorithms more computation time, expecting to get more accurate predictions.

Fast Predictions

We start with fast predictions. The results are shown in Table 3. The value in parenthesis after the algorithm's name indicates the value of r for L−CDP and the number of probes for SS. L−CDP is able to make almost instantaneous predictions even when using a large type system. On the other hand, SS does the sampling for each problem instance separately during prediction. Thus, in order to make fast predictions with SS we must use a smaller type system. We used T_h for SS in all three domains. For L−CDP we used T_{gc} in the experiment on the 15-puzzle, and T_c on the 15 pancake puzzle and Rubik's Cube. Given the same type system as L−CDP, SS was in some cases even more accurate than L−CDP but always about 1,000 times slower; when it was speeded up (by being given the T_h type system) to be within an order of magnitude or two of L−CDP, its predictions were far worse. In all three domains L−CDP dominates SS.

15-puzzle					
		Runtime (s)		Error	
d	IDA*	L−CDP (5)	SS (5)	L−CDP (5)	SS (5)
50	8,909,564.5	**0.0001**	0.0151	**0.62**	0.93
51	15,427,786.9	**0.0002**	0.0167	**0.60**	0.99
52	28,308,808.8	**0.0001**	0.0188	**0.60**	0.84
53	45,086,452.6	**0.0002**	0.0192	**0.57**	0.98
54	85,024,463.5	**0.0002**	0.0215	**0.58**	0.87
55	123,478,361.5	**0.0000**	0.0223	**0.58**	1.11
56	261,945,964.0	**0.0003**	0.0243	**0.56**	0.73
57	218,593,372.3	**0.0001**	0.0241	**0.63**	0.74
15 pancake puzzle					
		Runtime (s)		Error	
d	IDA*	L−CDP (2)	SS (5)	L−CDP (2)	SS (5)
11	44,771.2	**0.0003**	0.0012	**0.22**	0.36
12	346,324.5	**0.0006**	0.0017	**0.22**	0.38
13	2,408,281.6	**0.0005**	0.0029	**0.22**	0.44
14	20,168,716.0	**0.0007**	0.0041	**0.21**	0.34
15	127,411,357.4	**0.0008**	0.0057	**0.22**	0.47
Rubik's Cube					
		Runtime (s)		Error	
d	IDA*	L−CDP (2)	SS (10)	L−CDP (2)	SS (10)
9	119,506.2	**0.0012**	0.0027	**0.05**	0.15
10	1,626,583.9	**0.0014**	0.0032	**0.05**	0.15
11	21,985,207.8	**0.0013**	0.0057	**0.05**	0.16
12	295,893,415.9	**0.0014**	0.0064	**0.04**	0.14

Table 3: Fast predictions. L−CDP and SS.

Accurate Predictions

The results for accurate predictions are shown in Table 4. For these experiments, we used more informed type systems for both CDP and SS, namely T_{gc} for the 15-puzzle and T_c for the 15 pancake puzzle and Rubik's Cube. We also increased the value of r used by L−CDP to increase its prediction accuracy.

Similar to the results presented by Lelis et al. (2011), our results show that often the error of the CDP predictions increases as we increase the cost bound. For instance, the CDP error shown in Table 4 for the 15-puzzle is 0.05 for $d = 50$, and it grows to 0.26 for $d = 57$, an increase of 0.21. SS's error increased only by 0.01 for the same cost bounds. Recall that CDP samples the *state space* in a preprocessing step to approximate the values of $p(t|u)$ and b_u, and that these

values might be different from the actual values of $p(t|u)$ and b_u of the *search tree*. CDP is *domain-specific*, instead of *instance-specific*. We conjecture that noisy values of $p(t|u)$ and b_u used by CDP insert errors in early stages of the prediction that compound as the depth increases. SS on the other hand is *instance-specific* and only nodes that are part of the search tree for the given instance are considered for sampling. SS has a similar error when predicting the size of shallow and deep search trees. For the 15-puzzle and 15 pancake puzzle SS dominates CDP for larger cost bounds and it is no worse than CDP for lower cost bounds. Rubik's Cube turned out to be an easy domain in which to make predictions. Both CDP and SS make almost perfect predictions in this domain.

15-puzzle					
		Runtime (s)		Error	
d	IDA*	L-CDP (25)	SS (5)	L-CDP (25)	SS (5)
50	8,909,564.5	3.0207	0.8765	0.05	0.09
51	15,427,786.9	4.3697	0.9715	0.07	0.08
52	28,308,808.8	6.9573	**1.1107**	0.09	**0.09**
53	45,086,452.6	9.1959	**1.1767**	0.11	**0.09**
54	85,024,463.5	14.5368	**1.3577**	0.15	**0.10**
55	123,478,361.5	17.4313	**1.3940**	0.17	**0.10**
56	261,945,964.0	27.6587	**1.6438**	0.21	**0.10**
57	218,593,372.3	23.4482	**1.5258**	0.26	**0.10**
15 pancake puzzle					
		Runtime (s)		Error	
d	IDA*	L-CDP (5)	SS (3)	L-CDP (5)	SS (3)
11	44,771.2	0.0095	0.0180	0.09	0.07
12	346,324.5	0.0341	0.0500	0.10	0.09
13	2,408,281.6	0.1084	0.1176	0.11	0.09
14	20,168,716.0	0.2898	**0.2321**	0.13	**0.10**
15	127,411,357.4	0.6071	**0.3813**	0.16	**0.11**
Rubik's Cube					
		Runtime (s)		Error	
d	IDA*	L-CDP (5)	SS (20)	L-CDP (5)	SS (20)
9	119,506.2	**0.0802**	0.2668	**0.01**	0.02
10	1,626,583.9	0.4217	0.7231	0.01	0.01
11	21,985,207.8	1.6155	1.5098	0.01	0.01
12	295,893,415.9	3.1221	2.5269	0.01	0.01

Table 4: Accurate predictions. L-CDP and SS.

Discussion

Following the discussion above, the key difference between SS and CDP is that the former is instance-specific, while the latter is domain-specific. As we conjectured, being instance-specific allows SS to produce more accurate predictions as only nodes that belong to the search tree are considered for sampling. On the other hand, being domain-specific allows one to store the prediction results in a lookup table as a preprocessing step and produce predictions much more quickly.

To the best of our knowledge there is no general and efficient way of preprocessing SS's predictions without making it a domain-specific method. In fact, any preprocessing done for SS before knowing the start state would make SS quite similar to CDP.

Conclusions

In this paper we connected and advanced two lines of research. First, we presented L-CDP, a variant of CDP that can be orders of magnitude faster than CDP and is guaranteed to make the same predictions as CDP. Second, we showed that type systems employed by CDP can also be used as stratifiers for the SS algorithm. Our empirical results showed that SS employing CDP's type systems substantially improves the predictions produced by SS as presented by Chen. Third, we made an empirical comparison between our new enhanced versions of CDP and SS. Our experimental results point out that if L-CDP's preprocessing time is acceptable or can be amortized, it is suitable for applications that require less accurate but very fast predictions, while SS is suitable for applications that require more accurate predictions but allow more computation time.

Acknowledgements

The first author would like to thank Roni Stern for helpful discussions on an earlier draft of this paper. 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

Chen, P.-C. 1989. *Heuristic Sampling on Backtrack Trees*. Ph.D. Dissertation, Stanford University.

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

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In *Advances in Artificial Intelligence*, volume 1081 of *LNAI*, 402–416. Springer.

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

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

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

Korf, R. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*, 21–26.

Lelis, L.; Zilles, S.; and Holte, R. C. 2011. Improved prediction of IDA*s performance via ϵ-truncation. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search*, 108–116.

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