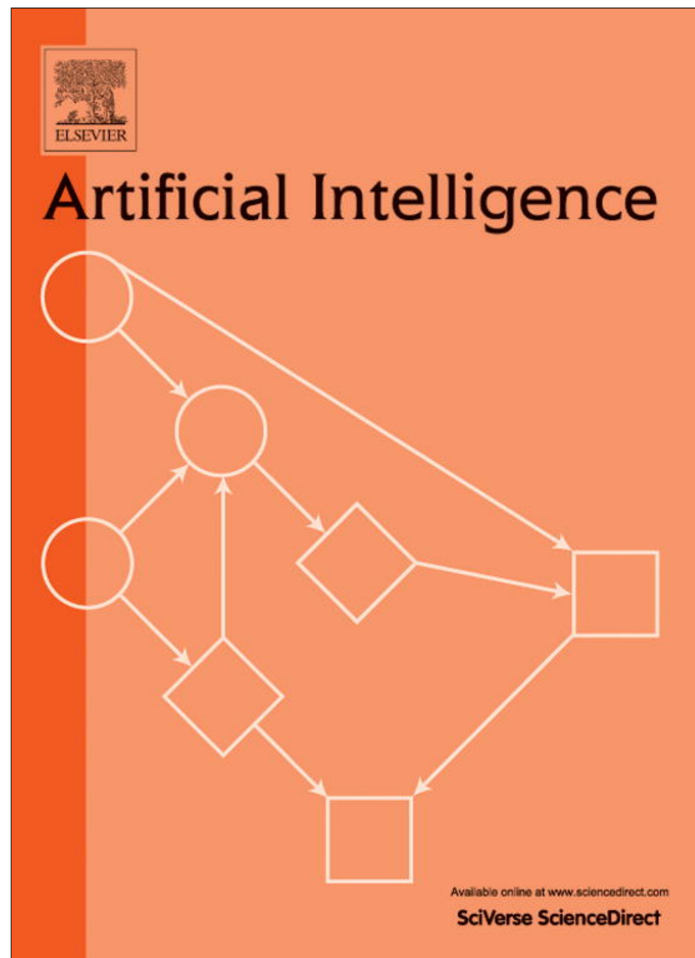
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Artificial Intelligence 196 (2013) 53–76

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Predicting the size of IDA*'s search tree

Levi H.S. Lelis [a], Sandra Zilles [b,*], Robert C. Holte [a]

[a] Computing Science Department, University of Alberta, Edmonton, AB, T6G 2E8, Canada
[b] Department of Computer Science, University of Regina, Regina, Saskatchewan, S4S 0A2, Canada

ARTICLE INFO

Article history:
Received 24 November 2011
Received in revised form 13 December 2012
Accepted 12 January 2013
Available online 16 January 2013

Keywords:
Heuristic search
Predicting search performance

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. Independently, Chen developed a method (SS) that can also be used to predict the number of nodes expanded by IDA*. In this paper we improve both of these prediction methods. First, we present ϵ-truncation, a method that acts as a preprocessing step and improves CDP's prediction accuracy. Second and orthogonally to ϵ-truncation, we present a variant of CDP that can be orders of magnitude faster than CDP while producing exactly the same predictions. Third, we show how ideas developed in the KRE line of research can be used to improve the predictions produced by SS. Finally, we make an empirical comparison between our new enhanced versions of CDP and SS. Our experimental results suggest that CDP is suitable for applications that require less accurate but fast predictions, while SS is suitable for applications that require more accurate predictions but can afford more computation time.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Tree search is a popular technique for solving combinatorial problems [1]. A frequent impediment of the application of tree searching algorithms is the inability to quickly predict the running time of an algorithm on a particular problem instance. While one instance of a problem might be solved in a blink of an eye, another instance of the same problem might take centuries.

Korf, Reid, and Edelkamp [2] launched a line of research aimed at creating a method to predict exactly how many nodes the search algorithm Iterative-Deepening A* (IDA*) [1] would expand on an iteration with cost bound d given a particular heuristic function. This was in contrast with the traditional approach to search complexity analysis, which focused on "big-O" complexity typically parameterized by the accuracy of the heuristic [3–6]. Korf, Reid, and Edelkamp developed a prediction formula, known as KRE as a reference to the author's names, for the special case of consistent heuristics,[1] proved that it was exact asymptotically (in the limit of large d), and experimentally showed that it was extremely accurate even at depths of practical interest. Zahavi et al. [7] created Conditional Distribution Prediction (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 Section 2.

* Corresponding author.
E-mail address: zilles@cs.uregina.ca (S. Zilles).

[1] Heuristic h is consistent iff $h(s) \leqslant c(s,t) + h(t)$ for all states s and t, where $c(s,t)$ is the cost of the cheapest path from s to t and $h(s)$ is the estimated cost-to-go from s to a goal state.

0004-3702/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.artint.2013.01.001

Author's personal copy

In the present paper we identify a source of prediction error that has hitherto been overlooked in the CDP system. We call it the “discretization effect”. We also disprove the intuitively appealing idea, specifically asserted by Zahavi et al., that a “more informed” system cannot make worse predictions than a “less informed” system.[2] The possibility of this statement being false follows directly from the discretization effect, because a more informed system is more susceptible to the discretization effect than a less informed one. We will show several cases of this statement being false and use the phrase “informativeness pathology” to refer to this situation. One of our contributions is a method for counteracting the discretization effect, which we call “ϵ-truncation”. One way to view ϵ-truncation is that it makes a prediction system less informed, in a carefully chosen way, so as to improve its prediction accuracy by reducing the discretization effect. In our experiments ϵ-truncation rarely degraded predictions; in the vast majority of cases it improved the prediction accuracy, often substantially. Our second contribution to the CDP system is an algorithmic improvement to CDP that reduces its running time. Our CDP variant, named Lookup CDP (or L-CDP for short), decomposes a CDP prediction into independent subproblems. The solutions to 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) [8], 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. ϵ-truncation and L-CDP are orthogonal to each other as the former improves the prediction accuracy and the latter improves the prediction runtime of CDP.

Independent of the KRE line of research, Knuth [9] created a method to efficiently predict the size of a search tree by making random walks from the root node. Knuth's assumption was that branches not visited would have the same structure as the single branch visited by the random walk. Despite its simplicity, Knuth proved his method to be efficient in the domains tested. However, as pointed out by Knuth himself, his method does not produce accurate predictions when the tree being sampled is unbalanced. Chen [10] extended Knuth's method to use a *stratification* of the state space to reduce the variance of sampling. Like CDP's type systems, the stratification 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 in Section 8. In addition to the contributions to the CDP prediction method, in this paper we connect the KRE line of research to that of SS. We do so by showing 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.

The final contribution of the present paper is an empirical comparison of our enhanced versions of CDP and SS. In our empirical comparison we consider 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 estimating 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 suggest 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.

We start by reviewing the CDP prediction method and type systems in the next two sections, before introducing the discretization effect (Section 4) and ϵ-truncation (Section 5). We show empirically that ϵ-truncation can substantially improve CDP's prediction accuracy in Section 6. In Section 7 we present L-CDP and show empirically that it can be orders of magnitude faster than CDP while producing exactly the same predictions. In Section 8 we review SS and in Section 9 we show how the type systems developed for CDP can substantially improve SS's predictions. Finally, in Section 10 we present an empirical comparison of our enhanced versions of CDP and SS.

This article extends earlier conference publications [11,12].

2. The CDP prediction framework

The notation introduced in this and the next section is summarized in Table 1. We now review the CDP system. CDP predicts the number of nodes expanded on an iteration of IDA* for a given cost bound assuming that no goal is found during the iteration. In CDP, predictions 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 $s \in t$.

For the sliding-tile puzzle (defined in Section 6), for example, one could define a type system based on the position of the blank tile. In this case, two nodes s and s' would be of the same type if s has the blank in the same position as s', regardless of the configuration of the other tiles in the two nodes. As another example, s and s' could be of the same type

[2] “More informed” is defined formally in Definition 4 below.

Author's personal copy

Table 1
Notation used in the CDP prediction framework.

Notation	Meaning
T	type system
$T(s)$	type of a node s
$p(t'\|t)$	probability of type t generating type t'
b_t	average branching factor of nodes of type t
$\pi(t'\|t)$	approximation of $p(t'\|t)$
β_t	approximation of b_t
$P(t,i,d)$	pruning function
$N(i,t,s^*,d)$	number of nodes of type t at level i
$T_h(s)$	type defined as $(h(parent(s)), h(s))$
$T_c(s)$	type defined by T_h and the h-values of s's children
$T_{gc}(s)$	type defined by T_c and the h-values of s's grandchildren
$T_1 \preceq T_2$	T_1 is a refinement of T_2

if s and s' have the same heuristic value. In some cases we use not only the heuristic value of a node when computing its type, but also the heuristic value of the nodes in its neighborhood, as we explain in Section 3 below.

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. Thus, the subtree below a node s depends on the parent from which s was generated. Zahavi et al. [7] use the information of the parent of a node s when computing s's type so that CDP is able to make accurate predictions 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 $T(s) = T(s')$, where h is the underlying heuristic function. 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 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 when computing the type of the node. 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, as though s and its children were part of a search tree. 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.

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

Here the outermost summation iterates over the children of the start state s^*. Assuming unit-cost edges, in the middle summation we account for g-costs from 1 to the cost bound d (the g-cost of node s in a search tree is the lowest cost path in the tree from the start state to s); 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 $T(n) = t$ occurring at level i of the search tree rooted at s. A value of one is added to the summation 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 $h(t) + i \leqslant d$, and is 0 otherwise.

Author's personal copy

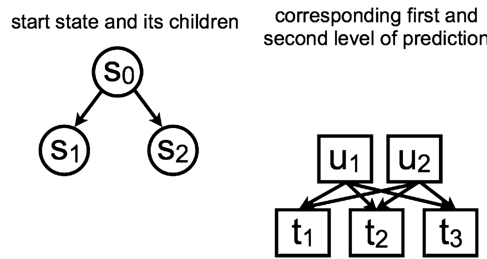


Fig. 1. The first step of a CDP prediction for start state s_0.

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 or 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 number of nodes of each type 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 $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$. Note that $N(1, u_i, s_i, d) = 1$ for $i \in \{1, 2\}$, $N(1, u_1, s_2, d) = N(1, u_2, s_1, d) = 0$, and $N(1, u, s, d) = 0$ for all other pairs of values of u and s. Thus $\pi(t_1|u_1)\beta_{u_1} + \pi(t_1|u_2)\beta_{u_2}$ equals $\sum_{u \in T} N(1, u, s_1, d)\pi(t_1|u)\beta_u + \sum_{u \in T} N(1, u, s_2, d)\pi(t_1|u)\beta_u$.

If $h(t_1) + 2$ (heuristic value of type t_1 plus the cost of reaching t_1) 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 at every level is the estimated number of nodes expanded by IDA* with cost bound d for start state s_0.

According to the formulae above and the example in Fig. 1, in order to predict the number of nodes IDA* expands with a cost bound d, for every level $i \leqslant d$, CDP predicts how many instances *of each type* will be generated; i.e., it predicts a vector $(N[1], \ldots, N[|T|])$ of numbers of instances of each type on a level.[3] We will call such a vector a *type allocation vector*. The type allocation vector for the first level of prediction is computed from the types of the children of the start state (the $i = 1$ base case of the recursive calculation shown above). Once the allocation vector is calculated for the first level, the vector for the next level is estimated according to Eq. (2). At level i, for each type t such that $h(t) + i$ exceeds the cost bound d, the corresponding entry in the type allocation vector, $N[t]$, is set to zero to indicate that IDA* will prune nodes of this type from its search. The prediction continues to deeper and deeper levels as long as at least one entry in the type allocation vector is greater than zero.

CDP is seeded with the types of the children of the start state s^*, as shown in Eq. (1). Zahavi et al. [7] 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. In our experiments we also used this improved version of CDP.

3. Improved CDP predictions

Zahavi et al. derived a set of conditions for which CDP is guaranteed to make perfect predictions [7, Section 4.5.1, p. 60]. This set of conditions can be generalized with the definition of the *purity* of a type system, i.e., if a type system is *pure*, then CDP predictions are guaranteed to be perfect.

Definition 3. A type system T is said to be pure if node n has exactly $p(t'|t) \times b_t$ children of type t' for all $t, t' \in T$ and all $n \in t$.

Intuitively, there is no uncertainty in the prediction model when a pure type system is employed, and as stated by Zahavi et al., a simple proof by induction shows that a pure type system results in perfect predictions.

A trivial example of a pure type system is a one-to-one mapping from the original state space S to the type system space T. In this case, every node in the search tree is of a different type. Such a type system is not of interest for large state spaces and d-values because the prediction calculations would be too costly. Unfortunately, type systems that are pure and compact are hard to obtain in practice. For instance, Zahavi et al.'s basic "two-step" model defined (in our notation)

[3] We use $N[t]$ or $N(i, t)$ to denote $N(i, t, s, d)$ when i, s and d are clear from the context.

Author's personal copy

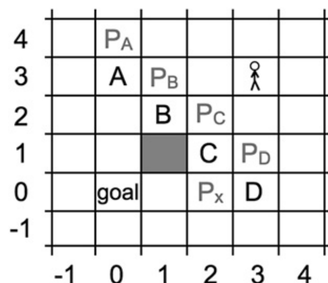


Fig. 2. Grid example.

as $T_h(s) = (h(parent(s)), h(s))$, where $parent(s)$ returns the parent of s in the search tree, is not pure, as verified in their experiments.

In a first attempt to improve CDP's prediction accuracy we used "more informed" type systems, i.e., type systems that split every type in T_h into a set of types. Two new domain-independent type systems we introduce, which are "more informed" than T_h, are:

$T_c(s) = (T_h(s), c(s, 0), \dots, c(s, H))$, 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;

$T_{gc}(s) = (T_c(s), gc(s, 0), \dots, gc(s, H))$, 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} type (where gc stands for grandchildren) if besides matching on the information required by the T_c type system, s and s' generate the same number of *grandchildren* with same heuristic distribution.

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

Definition 4. Let T_1, T_2 be type systems. T_1 is a refinement of T_2, denoted $T_1 \preceq T_2$, if $|T_1| \geqslant |T_2|$ and for all $t_1 \in T_1$ there is a $t_2 \in T_2$ with $\{s \mid T_1(s) = t_1\} \subseteq \{s \mid T_2(s) = t_2\}$. 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 $T_{gc} \preceq T_c \preceq T_h$, and so, by transitivity, $T_{gc} \preceq T_h$.

Example 2. Consider the grid domain depicted in Fig. 2. In this domain an agent lies on an infinite grid and wants to arrive at the goal position; Fig. 2 shows the agent starting at position $(3, 3)$ and the goal at position $(0, 0)$. The agent can move to one of the four adjacent positions, except when the position is blocked by a wall; in Fig. 2 a gray square represents a wall. If Manhattan Distance is the heuristic function used in this domain, then the heuristic value of the agent's state is 6 (3 from the x-coordinate plus 3 from the y-coordinate).

Consider a type system in which nodes with the same heuristic value are of the same type. In this case states A, B, C, and D would be of the same type. Note, however, that these four states are not necessarily of the same T_h type. Recall that T_h uses both the heuristic value of the node and of the node's parent in the search tree. A, B, C, and D will be of the same T_h type if they are generated by P_A, P_B, P_C and P_D, respectively. In this case, A, B, C, and D have a heuristic value of 3 and are generated by nodes with heuristic value of 4, resulting in the $(4, 3)$ type.

Consider again nodes A, B, C, and D when they are generated by parents P_A, P_B, P_C and P_D, respectively, and are therefore of the same type—$(4, 3)$—according to T_h. The T_c type system, which is a refinement of T_h, further partitions the nodes that are of the same type according to T_h. In our example, according to the T_c type system, nodes A and D are of different type than B and C. A and D are of the same T_c type because, with parent-pruning, they both generate two children with heuristic value of 4 (for A these are grid cells $(-1, 3)$ and $(1, 3)$) and one child with heuristic value of 2 (for A this is grid cell $(0, 2)$). B and C are of another T_c type: with parent-pruning both generate one child with heuristic value of 4 and one child with heuristic value of 2.

Intuitively, if $T_1 \preceq T_2$ one would expect 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 [7, p. 59]. However, our experiments show that this is not always true. The underlying cause of poorer predictions by T_1 when $T_1 \preceq T_2$ is the discretization effect, which we will now describe.

4. The discretization effect

In this section we identify a source of error in CDP's predictions that has previously gone unnoticed, which we call the discretization effect. Understanding this source of error allows us to propose a method to counteract it—the ϵ-truncation method that is fully described in the next section.

Author's personal copy

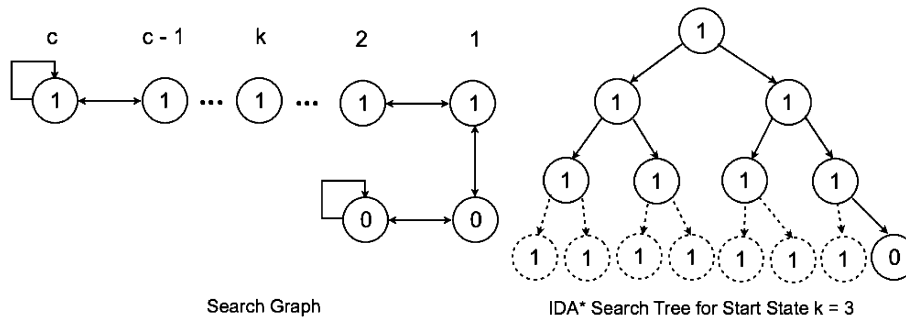


Fig. 3. Example of the discretization effect. Numbers represent the heuristic value of a state, and states with heuristic value of zero are goal states. The search graph has c states with heuristic value of one and two goal states. The right part shows the search tree of the last iteration of IDA* with start state $k=3$ and cost bound $d=3$; dashed lines represent pruned nodes.

Table 2
Exact type-transition probabilities for the graph shown in Fig. 3.

$p(t_0 \mid t_1)$	$\frac{1}{2c}$
$p(t_1 \mid t_1)$	$\frac{2c-1}{2c}$
$p(t_0 \mid t_0)$	$\frac{3}{4}$
$p(t_1 \mid t_0)$	$\frac{1}{4}$

Table 3
Transition probabilities estimated by sampling the state space in Fig. 3 when $c=10$.

$p(t_0 \mid t_1)$	0.05
$p(t_1 \mid t_1)$	0.95
$p(t_0 \mid t_0)$	0.75
$p(t_1 \mid t_0)$	0.25

Consider the state space shown in the left part of Fig. 3. Each circle is a state, and the number inside a circle is its heuristic value. All states in this space have two neighbors. There is a chain containing $c+2$ states, terminating at each end with a self-loop, with two goal states at one end of the chain having a heuristic value of 0 and the remaining c states having a heuristic value of 1. In this example we ignore parent-pruning and we use the following type system: $T(s)=h(s)$. Hence there are only two types: t_0 is the type of the nodes with heuristic value 0, and t_1 is the type of the nodes with heuristic value 1. The key feature of this construction is that the probability of a node of type t_1 generating a node of type t_0 can be made arbitrarily small by making c sufficiently large. Table 2 shows the exact probabilities of each kind of transition between types. CDP's sampling would estimate these; Table 3 shows that the estimates (to two decimal places) based on an exhaustive sample of size 12 for $c=10$ are equal to the theoretical values.

We are interested in the prediction for a single start state, the kth node with heuristic value 1. Note that the solution depth for this state is $d=k$. The IDA* search tree for start state $k=3$ and cost bound $d=3$ is shown in the right part of Fig. 3. Dashed lines represent nodes that are generated but not expanded because their f-value exceeds d. When $d=k$, as in the figure, level i contains 2^i expanded nodes, for $0 \leqslant i < d$, and level d contains one expanded node (the one with heuristic value 0 is counted as being expanded), so the total number of expanded nodes is 2^d.

For start state $k=4$, cost bound $d=4$, and $c=10$, IDA* expands $2^4=16$ nodes but CDP, using the exact transition probabilities (see Table 3), predicts it will expand 17.0264, an error of 1.0264. Table 4 summarizes CDP predictions at each level. We see that CDP's predictions of the total number of nodes expanded at each level (rightmost column) is perfect at all levels except the last. The cause of the error can be seen in the middle two columns, which show the predicted number of nodes expanded at each level of a particular type. At every level CDP is overestimating the number of nodes of type t_0 and correspondingly underestimating the number of nodes of type t_1. This occurs because one of the ten possible start states of type t_1 (namely, $k=1$) would generate one child of type t_0, while all the others would generate none. This is represented in CDP's predictions by saying that every start state of type t_1 generates 0.1 children of type t_0. For start states other than $k=1$, probability mass is being drawn away from the true prediction by an event that happens rarely for that type. Moreover, the error is compounded at each successive level in the tree: already at level 3, the percentage of nodes of type t_0 has risen to more than 10% of the nodes at that level, whereas at level 1 only 5% of the nodes are predicted to be of type t_0. These errors are invisible until IDA*'s pruning starts to treat nodes of type t_0 different than nodes of type t_1. In the example, this happens at level 4, where nodes of type t_1 are pruned but nodes of type t_0 are not.

Note that we would obtain a smaller prediction error if we were to totally ignore the rare event of t_1 generating a child of type t_0 by artificially setting the $p(t_0|t_1)$ entry in Table 3 to 0.0 and renormalizing $p(t_1|t_1)$ (it would become 1.0). With this change CDP would get exactly the right total number of nodes on all levels except level 4, where it would predict that 0 nodes are expanded (since it thinks they are all of type t_1). This is an error of 1.00 compared to the error of 1.0264 obtained using the actual values in Table 3.

Our method is based on this idea of altering the estimated probabilities so that rare events are ignored and common events correspondingly have their probabilities increased. However, we do this in a more sophisticated manner than having a global threshold to identify rare events. We use an optimization method to define a threshold, ϵ_i, to be used at level i. This allows different levels to have different definitions of "rare event". Applied to the example in this section, our method will set $p(t_0|t_1)$ to 0 at levels 0 to 3, but will leave it as 0.05 at level 4. This keeps its predictions perfect at levels 0 to 3 and predicts that there will be 0.8 $(0.05 \cdot 8 \cdot 2)$ nodes expanded at level 4, for an error of 0.2.

Author's personal copy

Table 4
CDP prediction level by level when $c = 10$, $k = 4$, and $d = 4$. The dash represents a type and level that was pruned off.

Level	t_0	t_1	Total by level
0	0	1	1
1	0.1	1.9	2
2	0.34	3.66	4
3	0.876	7.124	8
4	2.0264	–	2.0264
Total	3.3424	13.684	**17.0264**

5. The ϵ-truncation prediction method

Our ultimate goal is to minimize the error in predicting the number of nodes expanded by IDA*. Section 4 suggests that this requires avoiding the discretization effect—by ignoring small probabilities of generating nodes of a certain type. The CDP system does not ignore small fractional numbers. By contrast, there is a chance that minimizing the absolute error for type allocation vectors might force the system to ignore some of these harmful small numbers.

It is hence natural to consider modifying the π-values used as estimates in CDP according to the following procedure (P1).

Procedure (P1). For each type $u \in T$ do:

1. Compute a redistribution of $\pi(t|u)$ values, by solving the following optimization problem for each level i.

   Find a type allocation vector $(a_1, \ldots, a_{|T|})$ *that minimizes*:

$$\sum_{t \in T} \sum_{j=0}^{\lceil N(i,u,s^*,d) \cdot \beta_u \rceil} pr\big(j, t, u, \lceil N\big(i, u, s^*, d\big) \cdot \beta_u \rceil\big) \cdot \big|(a_t - j)\big| \tag{3}$$

   subject to the following constraints:

$$\sum_{t \in T} a_t = \lceil N\big(i, u, s^*, d\big) \cdot \beta_u \rceil \quad \text{and} \quad a_t \geqslant 0 \quad \text{for } t \in T.$$

   Here $pr(j, t, u, N)$ is short for the estimated probability of generating exactly j children of type t from N many parents of type u, i.e.,

$$pr(j, t, u, N) = \pi(t|u)^j \big(1 - \pi(t|u)\big)^{N-j}.$$

2. For each $t' \in T$, replace $\pi(t'|u)$ by $a_{t'} / \sum_{t \in T} a_t$.

However, Procedure (P1) is flawed. For any type t, it ignores the possibility that a large number of states of distinct types occurring at one level can all generate a state of type t with low probability, summing up to a large probability of having a state of type t at the next level.

Example 3. Suppose at prediction level i one node each of 100 different types $t_1, \ldots, t_{100}$ occurs. Suppose further that each of these 100 types generates one node of type t with probability 0.01. Procedure (P1) would correspond to solving at least 100 optimization problems (one for each type), potentially replacing $\pi(t|t_i)$ by 0 for all $i \in \{1, \ldots, 100\}$. Consequently, the prediction system might suggest that no node of type t occurs at level $i + 1$. However, it would be better to consider the interaction of the 100 types $t_1, \ldots, t_{100}$ at level i and to predict that one node of type t will occur at level $i + 1$.

In order to take into account that, for some type t, nodes of different types t_1 and t_2 may both generate nodes of type t at any level of prediction, we need to reformulate Procedure (P1) using a system of "supertypes".

5.1. Supertypes

Intuitively, a supertype at level i is a set of pairs (t, u) of types where type t occurs at level i and type u is generated at level $i + 1$.

Example 4. We now illustrate the concept of supertypes with the example of the grid domain shown in Fig. 2. We adopt the T_h type system in this example. Consider that at a given level of search we see node B of type $(4, 3)$, with B being

Author's personal copy

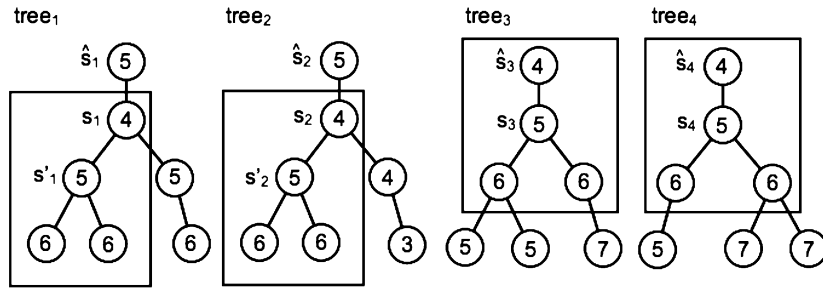


Fig. 4. Four potential search subtrees for a search problem. Numbers inside the nodes denote h-values. For $j \in \{1,2,3,4\}$, $tree_j$ determines the type $T_{gc}(s_j)$. The subtrees framed in boxes determine the type $T_c(s'_1) = T_c(s'_2) = T_c(s_3) = T_c(s_4)$.

generated by moving the agent from P_B to B, and node D of type $(2,3)$, with D being generated by moving the agent from P_X to D. The pairs of types $((4,3),(3,4))$ and $((2,3),(3,4))$ will be in the same supertype as both B of type $(4,3)$ and D of type $(2,3)$ can generate a node of type $(3,4)$—from B the agent could move to P_C and from D the agent could move to P_D.

The following conditions should be met by the supertypes:

- If the set of types that can be generated by a type t_1 occurring at level i overlaps with the set of types that can be generated by a type t_2 occurring at level i—say both sets contain type u—then (t_1,u) and (t_2,u) will be in the same supertype.
  Consider for instance the four trees in Fig. 4. Suppose nodes s_1 and s_2 of types $t_1 = T_{gc}(s_1)$ and $t_2 = T_{gc}(s_2)$, respectively, occur at level i of the prediction. The types t_1 and t_2 can potentially generate pairs of type $t_3 = T_{gc}(s_3)$ and pairs of type $t_4 = T_{gc}(s_4)$ at level $i+1$ (following the left branches of $tree_1$ and $tree_2$, framed in boxes in the figure). Hence we would like to put (t_1,t_3) and (t_2,t_3) in one supertype for level i; similarly we would put (t_1,t_4), and (t_2,t_4) in one supertype for level i.
- Any type u at level $i+1$ will be generated by a single supertype at level i. This is achieved by taking information from a coarser type system into account. If t_1 and t_2 at level i can generate type u_1 and u_2, respectively, at level $i+1$, and u_1 and u_2 are indistinguishable in a fixed *coarser* type system, then (t_1,u_1) and (t_2,u_2) will be in the same supertype for level i.
  In the example in Fig. 4, both s_1 (of type t_1) and s_2 (of type t_2) generate nodes of type $T_c(s'_1) = T_c(s'_2)$ in the coarser type system T_c (see the framed boxes in $tree_1$ and $tree_2$). These state pairs at level $i+1$ (see the framed boxes in $tree_3$ and $tree_4$) could be of type t_3 or t_4 in the (refined) type system T_{gc}. Thus (t_1,t_3), (t_2,t_3), (t_1,t_4), and (t_2,t_4) will all be in a single supertype for level i.

Formally, we define supertype systems as follows.

Definition 5. Let T, T' be type systems, $T \preceq T'$, and $t' \in T'$. For all i, the supertype $st(t',i,s)$ over T contains exactly the pairs $(t_1,t_2) \in T \times T$ for which $T'(t_2) = t'$ and t_1 occurs at level i starting the prediction from s. The supertype system $ST(i,s)$ over T with respect to T' is defined by $ST(i,s) = (st(t'_1,i,s),\ldots,st(t'_z,i,s))$ where $T' = \{t'_1,\ldots,t'_z\}$. We write st instead of $st(t',i,s)$ whenever t', i, s are clear from context.

Let T, T' be type systems such that $T \preceq T'$ that induce the supertype $ST(i,s^*)$. In order to adapt CDP and Procedure (P1) to supertypes, we estimate, for each node s^*, level i, cost bound d, type t, and supertype $st \in ST(i,s^*)$, the probability of generating a node of type t from a node of supertype st at level i. We denote this estimate by $\pi^{i,d}_{s^*}(t|st)$, defined by

$$\pi^{i,d}_{s^*}(t|st) = \frac{\sum_{\{t_p|(t_p,t)\in st\}} \pi(t|t_p)\beta_{t_p} N(i,t_p,s^*,d)}{\sum_{\{t_p|(t_p,t)\in st\}} \beta_{t_p} N(i,t_p,s^*,d)}.$$

We write $\pi(t|st)$ instead of $\pi^{i,d}_{s^*}(t|st)$, whenever i, d, and s^* are clear from the context.
The number of nodes $N_{ST}(i,st,s^*,d)$ of a supertype $st \in ST$ at a level i of prediction is given by

$$N_{ST}\big(i,st,s^*,d\big) = \sum_{(t_p,t_c)\in st} N\big(i,t_p,s^*,d\big)\beta_{t_p}. \tag{4}$$

We then reformulate the CDP formula equivalently, computing $N(i,t,s^*,d)$ by

$$N\big(i,t,s^*,d\big) = \sum_{st\in ST} N_{ST}\big(i-1,st,s^*,d\big)\pi^{i-1,d}_{s^*}(t|st)P(t,i,d), \tag{5}$$

Author's personal copy

instead of Eq. (2). Note that, in contrast with Eq. (2), Eq. (5) does not include any β-values as factors, due to the fact that those are already incorporated in the calculation in the N_{ST} values in Eq. (4).

Procedure (P1) would then be adapted to the following procedure (P2).

Procedure (P2). For each supertype $st \in ST$ at any level i do:

1. Compute a redistribution of $\pi(t|st)$ values, by solving the following optimization problem.

   Find a type allocation vector $(a_1, \ldots, a_{|T|})$ *that minimizes*:

$$\sum_{t \in T} \sum_{j=0}^{\lceil N_{ST}(i,st,s^*,d) \rceil} pr\big(j, t, st, \lceil N_{ST}\big(i, st, s^*, d\big)\rceil\big) \cdot \big|(a_t - j)\big| \tag{6}$$

   subject to the following constraints:

$$\sum_{t \in T} a_t = \lceil N_{ST}\big(i, st, s^*, d\big)\rceil \quad \text{and} \quad a_t \geqslant 0 \quad \text{for } t \in T.$$

   Here $pr(j, t, st, N)$ is short for the probability of generating exactly j children of type t from N many parents of supertype st, i.e.,

$$pr(j, t, st, N) = \pi(t|st)^j \big(1 - \pi(t|st)\big)^{N-j}.$$

2. For each $t' \in T$, replace $\pi(t'|st)$ by $a_{t'} / \sum_{t \in T} a_t$.

5.2. ϵ-Truncation as a preprocessing step

Ideally, one would now follow Procedure (P2) at every step of the prediction. However, although the optimization problem can be solved in polynomial type, solving distinct instances at every step of prediction is computationally prohibitive. For example, in an experiment we ran on the (4×4) 15 sliding-tile puzzle, following Procedure (P2) at every step of the prediction was almost three orders of magnitude slower than CDP without the optimization. We hence developed a method that sacrifices the optimality of the optimization problem in Procedure (P2) for feasibility, by redistributing $\pi(t|st)$ values and $\pi(t|u)$ values only in a preprocessing step. The goal of this preprocessing step is to find, for each level i, a cutoff value ϵ_i, below which $\pi(t|u)$ values will be set to zero. Our approach, called ϵ-truncation, can be summarized as follows.

1. As before, sample the state space to obtain $\pi(t|u)$ and β_u for each $t, u \in T$.
2. For each level $i \in \{1, \ldots, d\}$ compute a cutoff value ϵ_i. (This step will be explained in detail below.)
3. For each level i and each $t, u \in T$, replace the estimate $\pi(t|u)$ by an estimate $\pi_i(t|u)$ that is specific to level i. $\pi_i(t|u)$ is determined as follows.
   (a) If $\pi(t|u) < \epsilon_i$ then $\pi_i(t|u) = 0$.
   (b) If $\pi(t|u) \geqslant \epsilon_i$ then

$$\pi_i(t|u) = \frac{\pi(t|u)}{\sum_{v \in T, \pi(v|u) \geqslant \epsilon_i} \pi(v|u)}.$$

   Thus the $\pi(t|u)$ values not smaller than ϵ_i are scaled so that they sum up to 1.
4. In computing CDP use $\pi_i(t|u)$ at level i instead of $\pi(t|u)$.

The key step in this process is Step 2, the calculation of the ϵ_i values.

2. For each level $i \in \{1, \ldots, d\}$ compute a cutoff value ϵ_i as follows.
   (a) Solve a (small) specified number z of instances of Eq. (6) for level i.
   (b) For every previously estimated value $\pi(t|st)$, compute the fraction of times that this $\pi(\cdot|\cdot)$-value was set to zero in the z solutions to Eq. (6) for level i.
   In Fig. 5 the set of values that $\pi(t|st)$ can assume corresponds to the x-axis. The fraction of times a $\pi(\cdot|\cdot)$-value was set to zero is the corresponding value on the y-axis.
   (c) Compute a candidate cutoff value $\hat{\epsilon}_i$ as follows. $\hat{\epsilon}_i$ is the largest $\pi(t|st)$ for which all values $\pi \leqslant \pi(t|st)$ were set to zero in at least 50% of the z instances at level i.
   In Fig. 5 this is the smallest x-value at which the curve intersects the horizontal $y = 0.5$ line. We thus suggest to ignore (i.e., set to zero) only probabilities $\pi(\cdot|\cdot)$ that were set to zero in the *majority* of the z solutions to the optimization problem.
   (d) Compute the actual cutoff value ϵ_i as follows:

Author's personal copy

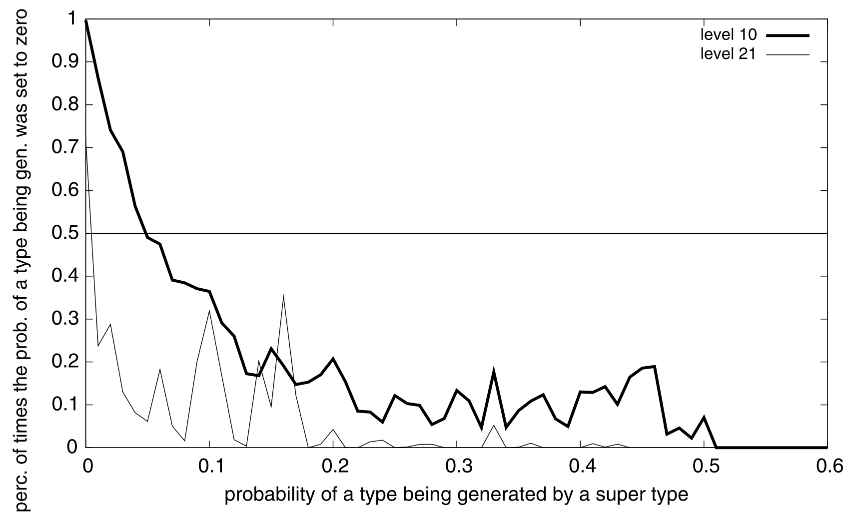


Fig. 5. $\hat{\epsilon}_i$ calculation for $i = 10$ and $i = 21$ (8-puzzle with Manhattan Distance).

Table 5
ϵ_i values for the 8-puzzle with Manhattan Distance.

Level (i)	10	11–12	13	14	15	16–17	18–19	20–23	24
ϵ_i	0.05	0.07	0.08	0.07	0.05	0.04	0.03	0.01	0.00

i. If for each type $u \in T$ there is at least one type $t \in T$ such that $\pi(t|u) \geqslant \hat{\epsilon}_i$, let $\epsilon_i = \hat{\epsilon}_i$.
In this case, it is safe to use $\hat{\epsilon}_i$ as a cutoff value, since for each $u \in T$ there will be some $\pi(\cdot|u)$-value that is not set to zero.

ii. If for some type $u \in T$ there is no type $t \in T$ such that $\pi(t|u) \geqslant \hat{\epsilon}_i$, let ϵ_i be the largest value $\delta_i < \hat{\epsilon}_i$ such that, for all $u \in T$ there is some $t \in T$ such that $\pi(t|u) \geqslant \delta_i$.
In this case, we cannot use $\hat{\epsilon}_i$ as a cutoff value, since this would imply setting all $\pi(\cdot|u)$ values to zero, for type u.

For illustration, Table 5 shows the ϵ_i values calculated using 10,000 randomly generated start states for the 8-puzzle with Manhattan Distance. The value of 0.05 for level 15 in Table 5 means that, out of the 10,000 searches, the majority of types that were generated with probability of 0.05 or lower at level 15 had their values of $p(\cdot|\cdot)$ set to zero by the optimization algorithm. In this table, as in all the experiments in this paper, the ϵ_i values approach zero as i gets larger.

CDP is applicable only in situations in which one is interested in making a large number of predictions, so that the time required for sampling is amortized. We show in the next section that the ϵ-truncation procedure can substantially increase the accuracy of the CDP predictions. However, this improvement in accuracy comes at the cost of an increased preprocessing time. For instance, it takes approximately 10 hours to sample one billion random states to approximate the values of $p(t|u)$ and b_u for the 15-puzzle. The ϵ-truncation procedure adds another 15 hours of preprocessing for finding the value of ϵ_i. In the experiments described in the next section we assume one is interested in making a sufficiently large number of predictions, so that the preprocessing time required by ϵ-truncation is amortized.

6. Experimental results on ϵ-truncation

This section presents the results of experiments showing that: (a) refining a type system often reduces prediction accuracy; (b) ϵ-truncation often substantially improves predictions; (c) ϵ-truncation of a refinement of a type system usually gives greater improvements than ϵ-truncation of the basic type system, and (d) ϵ-truncation rarely reduces the prediction accuracy. Each experiment will use two type systems, a basic one and a refinement of the basic one, and will compare the predictions made by CDP with each type system and with ϵ-truncation applied to both type systems.

Domains. Our experiments are run on three domains: the sliding-tile puzzle, the pancake puzzle, and Rubik's Cube.

- **Sliding-tile puzzle** [13]—The sliding-tile puzzle with parameters n and m consists of $n \times m - 1$ numbered tiles that can be moved in an $n \times n$ grid. A state is a vector of length $n \times m$ in which component k names what is located in the kth puzzle position (either a number in $\{1, \ldots, n \times m - 1\}$ representing a tile or a special symbol representing the blank). Every operator swaps the blank with a tile adjacent to it. The left part of Fig. 6 shows the goal state that we used for the (4×4)-puzzle, also called the 15-puzzle, while the right part shows a state created from the goal state by applying

Author's personal copy

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

1	5	2	3
4		6	7
8	9	10	11
12	13	14	15

Fig. 6. The goal state for the 15-puzzle (left) and a state two moves from the goal (right).

1	2	3	4	5	...	14	15

4	3	2	1	5	...	14	15

Fig. 7. The goal state for the 15-pancake puzzle (above) and a state one move from the goal (below).

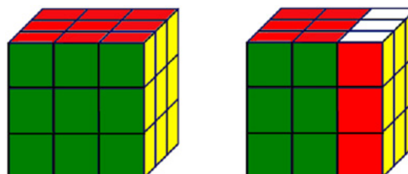


Fig. 8. Rubik's Cube (modified from Zahavi et al. [7]).

two operators, namely swapping the blank with tile 1 and then swapping it with tile 5. The number of states reachable from any given state is $(n \times m)!/2$, cf. [14].

We use three sizes of the sliding-tile puzzle: two that are small enough that the entire reachable portion of the state space can be enumerated and used in lieu of "sampling"—the (3×3)-puzzle, also called the 8-puzzle, and the (3×4)-puzzle—and one that is large enough to be of practical interest—the 15-puzzle. The small domains are an important element of the experiments because phenomena witnessed in them cannot be attributed to sampling effects.

- **Pancake puzzle** [15]—In the pancake puzzle with parameter n, a state is a permutation of n numbered tiles and has $n - 1$ successors, with the lth successor formed by reversing the order of the first $l + 1$ positions of the permutation $(1 \leqslant l \leqslant n - 1)$. The upper part of Fig. 7 shows the goal state of the 15-pancake puzzle, while the lower part shows a state in which the first four positions have been reversed.
  All $n!$ permutations are reachable from any given state. We report results for $n = 15$ which contains 15! reachable states. One may think of each tile as a pancake and each permutation as a pile of pancakes that have to be sorted into the goal permutation. To move a pancake from position 1 into position p in the pile, all the pancakes stacked from position 1 to position p have to be flipped together.
- **Rubik's Cube** [16]—Rubik's Cube is a $3 \times 3 \times 3$ cube made up of 20 moveable $1 \times 1 \times 1$ "cubies" with colored stickers on each exposed face. Each face of the cube can be independently rotated by 90 degrees clockwise or counterclockwise, or by 180 degrees. The left part of Fig. 8 shows the goal state for Rubik's Cube while the right part shows the state produced by rotating the right face 90 degrees counterclockwise.

Experimental setup. The choice of the set of start states will be described in the specific sections below, but we always applied the same principle as Zahavi et al. [7]: start state s is included in the experiment with cost bound d only if IDA* would actually have used d as a cost bound in its search with s as the start state; Zahavi et al. called this selection of s and d the restricted selection and showed that if the selection of s and d is not restricted, the number of nodes expanded by the IDA* in the experiments would be substantially different than the number of nodes expanded by the algorithm in real situations (cf. Table 5 of Zahavi et al. [7]). Like Zahavi et al., we are interested in verifying the accuracy of the predictions in real situations, thus we also adopt the restricted selection. As mentioned in Section 2, unlike an actual IDA* run, we count the number of nodes expanded in the entire iteration for a start state even if the goal is encountered during the iteration.

The number of start states used to determine the ϵ_i values is closely related to the value of r that will be used in the experiment—recall that the value of r determines the level at which CDP collects states to seed the prediction. For example, the number of states at level 10 of the 8-puzzle is expected to be much lower than the number of states at level 25 of the 15-puzzle. Therefore, in order to find suitable ϵ-values for the 8-puzzle we have to use more start states than are required to determine ϵ-values for the 15-puzzle. The number of states used to determine the ϵ_i values is stated below for each experiment.

Error measures. We report the prediction results using three different measures: *Relative Signed Error*, *Relative Unsigned Error*, and *Root Mean Squared Relative Error* (RMSRE).

- **Relative signed error**—For each prediction system we will report the ratio of the predicted number of nodes expanded, averaged over all the start states, to the actual number of nodes expanded, on average, by IDA*. Let *PI* be the set of problem instances used in an experiment; $\mathrm{CDP}(s, d, h, T)$ is the predicted number of nodes expanded by IDA* for start

Author's personal copy

state s, cost bound d, heuristic function h, and type system T; $A(s, d)$ is the actual number of nodes expanded by IDA* when s is the start state with cost bound d. The relative signed error for experiment with PI is calculated as follows.

$$\frac{\sum_{s \in PI} \text{CDP}(s, d, h, T)}{\sum_{s \in PI} A(s, d)}.$$

This ratio will be rounded to two decimal places. Thus a ratio of 1.00 does not necessarily mean the prediction is perfect, it just means the ratio is closer to 1.00 than it is to 0.99 or 1.01. This ratio we call the relative signed error. It is the same as the "Ratio" reported by Zahavi et al. [7] and is appropriate when one is interested in predicting the total number of nodes that will be expanded in solving a set of start states. It is not appropriate for measuring the accuracy of the predictions on individual start states because errors with a positive sign cancel errors with a negative sign. If these exactly balance out, a system will appear to have no error (a ratio of 1.00) even though there might be substantial error in every single prediction.

- **Relative unsigned error**—To evaluate the accuracy of individual predictions, an appropriate measure is relative unsigned error, calculated as follows.

$$\frac{\sum_{s \in PI} \frac{|\text{CDP}(s,d,h,T) - A(s,d)|}{A(s,d)}}{|PI|}.$$

A perfect score according to this measure is 0.00.

- **RMSRE**—Another error measure we use to compare predictions for individual start states is the *root mean squared relative error* (RMSRE), which is calculated as follows.

$$\sqrt{\frac{\sum_{s \in PI} (\frac{\text{CDP}(s,d,h,T) - A(s,d)}{A(s,d)})^2}{|PI|}}.$$

A perfect score according to this measure is 0.00.

All our experiments were run on an Intel Xeon CPU X5650, 2.67 GHz.

6.1. Sliding-tile puzzles

We used the same type system as Zahavi et al. [7], which is a refinement of T_h we call $T_{h,b}$. $T_{h,b}$ is defined by $T_{h,b}(s) = (T_h, blank(parent(s)), blank(s))$ where $blank(s)$ returns the kind of location (corner, edge, or middle) the blank occupies in state s. For instance, if we assume that the right-hand state in Fig. 6 (let us call it s) was generated by moving tile 4 to the left, then we would have the following $T_{h,b}$ type for s: $(3, 2, E, M)$. The 3 in the tuple stands for the heuristic value of the parent of s (we assume the heuristic being used is Manhattan Distance); the 2 stands for the heuristic value of s; the E tells us that the blank position of the parent of s was on an edge; finally, the M means that the blank is on a middle position in s.

For the (3×4)-puzzle there are two kinds of edge locations that $blank(s)$ needs to distinguish—edge locations on the short side (length 3) and edge locations on the long side (length 4). $T_{gc,b}$ is defined analogously. For square versions of the puzzle, T_{gc} is exactly the same as $T_{gc,b}$ and therefore $T_{gc} \preceq T_{h,b}$. However, for the (3×4)-puzzle, T_{gc} and $T_{gc,b}$ are not equal. We used the following coarse type systems to define the supertypes for T_{gc}, $T_{gc,b}$, and $T_{h,b}$: T_c, T_c augmented with the kind of blank location of the parent of the node, and $T_{p,b}(s) = (h(s), blank(s))$, respectively.

For the 8-puzzle we used 10,000 random start states to determine the ϵ_i values and every solvable state in the space to measure prediction accuracy. The upper part of Table 6 shows the results for the Manhattan Distance heuristic, which is admissible and consistent, with $r = 10$. The bold entries in this and all other tables of results indicate the best predictions for a given error measure. Here we see a few cases of the informativeness pathology: T_{gc}'s predictions are worse than $T_{h,b}$'s, despite its being a refinement of $T_{h,b}$. Applying ϵ-truncation substantially reduces T_{gc}'s prediction error for all three error measures.

We also ran experiments on the 8-puzzle using the inconsistent heuristic defined by Zahavi et al. [7]. Two pattern databases (PDBs) [8] were built, one based on the identities of the blank and tiles 1–4 (tiles 5–8 were indistinguishable), and another based on the identities of the blank and tiles 5–8 (tiles 1–4 were indistinguishable). The locations in the puzzle are numbered in increasing order left-to-right and top-to-bottom and the first PDB is consulted for states having the blank in an even location; the second PDB is consulted otherwise. Since the blank's location changes parity every time it moves, we are guaranteed that the heuristic value of a child node will be taken from a different PDB than that of its parent. Heuristics defined by PDBs are admissible and consistent, but as we are alternating the lookup between two different PDBs the resulting heuristic is admissible but inconsistent. Again we used 10,000 random start states to determine the ϵ_i values and every solvable state in the space to measure prediction accuracy. The results of this experiment, with $r = 1$, are shown in the middle part of Table 6. They exhibit the informativeness pathology and demonstrate that ϵ-truncation can substantially reduce prediction error. ϵ-truncation produces slightly worse predictions when using the coarser type

Author's personal copy

Table 6
8-puzzle.

d	IDA*	Signed error				Unsigned error				RMSRE			
		$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}
Manhattan Distance, $r = 10$													
18	134.4	**1.00**	**1.00**	**1.00**	**1.00**	0.03	0.03	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
19	238.4	**1.00**	**1.00**	**1.00**	**1.00**	0.04	0.04	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
20	360.1	1.01	1.01	1.01	**0.99**	0.05	0.05	0.03	**0.02**	0.07	0.07	0.04	**0.03**
21	630.7	1.00	1.00	1.02	**0.99**	0.06	0.06	**0.03**	**0.03**	0.07	0.07	**0.04**	**0.04**
22	950.6	1.01	1.01	1.03	**0.98**	0.07	0.07	0.05	**0.04**	0.09	0.09	0.06	**0.05**
23	1649.5	**1.00**	**1.00**	1.04	0.98	0.07	0.07	0.05	**0.04**	0.09	0.09	0.06	**0.05**
24	2457.5	**1.01**	**1.01**	1.06	0.97	0.08	0.08	0.08	**0.05**	0.11	0.11	0.09	**0.07**
25	4245.5	**1.00**	**1.00**	1.07	0.97	0.09	0.09	0.08	**0.05**	0.11	0.11	0.09	**0.06**
26	6294.4	**1.00**	**1.00**	1.10	0.96	0.10	0.10	0.10	**0.06**	0.12	0.12	0.12	**0.08**
27	10,994.9	**0.99**	**0.99**	1.11	0.97	0.11	0.11	0.11	**0.06**	0.14	0.14	0.13	**0.08**
Inconsistent heuristic, $r = 1$													
18	14.5	0.71	0.78	1.16	**1.02**	0.37	0.42	0.43	**0.36**	**0.55**	0.67	0.69	0.57
19	22.2	0.72	0.73	1.19	**1.01**	0.45	0.45	0.49	**0.39**	0.59	0.59	0.69	**0.54**
20	27.4	0.73	0.82	1.23	**0.98**	0.45	0.50	0.54	**0.40**	0.62	0.75	0.78	**0.57**
21	43.3	0.74	0.77	1.27	**0.99**	0.48	0.49	0.56	**0.40**	0.63	0.65	0.77	**0.53**
22	58.5	0.75	0.83	1.33	**0.95**	0.47	0.51	0.62	**0.40**	0.63	0.73	0.86	**0.54**
23	95.4	0.75	0.81	1.39	**0.97**	0.48	0.49	0.62	**0.38**	0.63	0.68	0.84	**0.50**
24	135.7	0.76	0.84	1.45	**0.92**	0.45	0.47	0.65	**0.37**	0.58	0.66	0.90	**0.49**
25	226.7	0.76	0.83	1.51	**0.97**	0.43	0.42	0.65	**0.33**	0.55	0.57	0.88	**0.44**
26	327.8	0.76	0.85	1.57	**0.91**	0.41	0.39	0.67	**0.30**	0.49	0.51	0.90	**0.40**
27	562.0	0.76	0.85	1.63	**0.98**	0.39	0.36	0.66	**0.26**	0.46	0.44	0.86	**0.34**
Same inconsistent heuristic, $r = 10$													
18	14.5	0.88	0.90	**1.00**	**1.00**	0.04	0.04	**0.01**	**0.01**	0.11	0.10	**0.03**	**0.03**
19	22.2	0.87	0.88	**1.00**	**1.00**	0.06	0.06	**0.01**	**0.01**	0.13	0.12	**0.05**	**0.05**
20	27.4	0.86	0.87	**1.01**	**1.01**	0.07	0.07	**0.03**	**0.03**	0.15	0.14	**0.07**	**0.07**
21	43.3	0.86	0.87	**1.02**	**1.02**	0.09	0.09	**0.04**	**0.04**	0.16	0.16	**0.09**	**0.09**
22	58.5	0.85	0.85	**1.03**	**1.03**	0.11	0.10	**0.07**	**0.07**	0.18	0.17	**0.12**	**0.12**
23	95.4	0.84	0.85	1.05	**1.04**	0.13	0.12	**0.09**	**0.09**	0.19	0.18	0.14	**0.13**
24	135.7	0.83	0.83	1.07	**1.05**	0.15	0.15	0.11	**0.10**	0.21	0.20	0.15	**0.14**
25	226.7	0.82	0.82	1.10	**1.06**	0.17	0.17	0.13	**0.12**	0.22	0.22	0.17	**0.15**
26	327.8	0.81	0.82	1.13	**1.07**	0.19	0.19	0.16	**0.13**	0.23	0.23	0.19	**0.16**
27	562.0	0.81	0.80	1.17	**1.08**	0.21	0.21	0.18	**0.13**	0.24	0.24	0.22	**0.16**

system for some of the cost bounds when measuring unsigned error or RMSRE (see $d = 20, 21, 22, 23, 24, 25, 26$ in the middle part of Table 6). Note, however, that this decrease in accuracy is not observed when signed error is measured. Large overestimations and underestimations of the actual number of nodes expanded might cancel each other out when the signed error is measured, giving the impression that the predictions are accurate even if they are not. The decrease in performance caused by ϵ-truncation disappears for larger values of r as shown in the bottom part of Table 6. The improvements in accuracy by ϵ-truncation are still observed in the bottom part of Table 6.

For the (3×4)-puzzle we used 10 random start states to determine the ϵ_i values and 10,000 to measure prediction accuracy. The upper part of Table 7 shows the results for Manhattan Distance. Both the unsigned error and the RMSRE for $T_{gc,b}$ are very close to those for $T_{h,b}$'s, suggesting that being more informed provides no advantage. ϵ-truncation substantially improves $T_{gc,b}$'s predictions in all three error measures. The lower part of the table is for Manhattan Distance multiplied by 1.5, which is inadmissible and inconsistent. Here $T_{gc,b}$'s predictions are considerably more accurate than $T_{h,b}$'s and are substantially improved by ϵ-truncation. In both cases ϵ-truncation did not modify the predictions for T_h.

For the 15-puzzle, we used 5 random start states to determine the ϵ_i values and 1000 to measure prediction accuracy. To define $\pi(t|u)$ and β_t, one billion random states were sampled and, in addition, we used the process described by Zahavi et al. [7] 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. Table 8 shows the results when Manhattan Distance is the heuristic and $T_{h,b}$ and T_{gc} are the type systems. Here again we see the informativeness pathology ($T_{h,b}$'s predictions are better than T_{gc}'s) which is eliminated by ϵ-truncation. Like for the (3×4)-puzzle, ϵ-truncation does not modify the predictions when using the coarser type system.

Like for the 8-puzzle, an inconsistent heuristic for the 15-puzzle was created with one PDB based on the identities of the blank and tiles 1–7, and another that kept the identities of the blank and tiles 9–15, exactly as used by Zahavi et al. (see their Table 11). We alternate the PDB that is used for the heuristic lookup depending on the position of the blank as described for the 8-puzzle. The results with $T_{h,b}$ and T_c as type systems are shown in Table 9. Here we see that even though $T_{h,b}$ presents a reasonable signed error, it has in fact a very large unsigned error and RMSRE, and once again ϵ-truncation produced substantial improvement in prediction accuracy—in this case for both coarse and refined type systems. These prediction results could be improved by increasing the r-value used. However, we wanted our results to be comparable to those in Zahavi et al.'s Table 11.

Author's personal copy

Table 7
(3 × 4)-puzzle, $r = 20$.

d	IDA*	Signed error				Unsigned error				RMSRE			
		$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$	$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$	$T_{h,b}$	ϵ-$T_{h,b}$	$T_{gc,b}$	ϵ-$T_{gc,b}$
Manhattan Distance.													
33	30,461.9	1.02	1.02	1.01	**1.00**	0.02	0.02	**0.01**	**0.01**	0.02	0.02	**0.01**	**0.01**
34	49,576.8	1.02	1.02	1.02	**1.01**	0.02	0.02	0.02	**0.01**	0.03	0.03	0.02	**0.01**
35	80,688.2	1.04	1.04	1.04	**1.01**	0.03	0.03	0.03	**0.02**	0.04	0.04	0.03	**0.02**
36	127,733.4	1.05	1.05	1.05	**1.02**	0.05	0.05	0.04	**0.02**	0.06	0.06	0.05	**0.03**
37	201,822.7	1.07	1.07	1.08	**1.03**	0.06	0.06	0.06	**0.03**	0.08	0.08	0.07	**0.04**
38	327,835.3	1.09	1.09	1.11	**1.04**	0.08	0.08	0.09	**0.04**	0.10	0.10	0.10	**0.05**
39	478,092.5	1.12	1.12	1.15	**1.05**	0.11	0.11	0.13	**0.06**	0.13	0.13	0.14	**0.07**
40	822,055.4	1.16	1.16	1.20	**1.07**	0.14	0.14	0.17	**0.08**	0.17	0.17	0.19	**0.09**
41	1,163,312.1	1.20	1.20	1.26	**1.10**	0.17	0.17	0.23	**0.10**	0.21	0.21	0.25	**0.12**
42	1,843,732.2	1.27	1.27	1.34	**1.13**	0.23	0.23	0.30	**0.13**	0.27	0.27	0.32	**0.15**
Manhattan Distance multiplied by 1.5.													
33	926.2	1.05	1.04	1.01	**1.00**	0.02	0.02	**0.00**	**0.00**	0.04	0.03	**0.01**	**0.01**
34	1286.9	1.06	1.06	1.02	**1.00**	0.03	0.02	**0.01**	**0.01**	0.05	0.05	**0.02**	**0.02**
35	2225.6	1.09	1.08	1.03	**1.00**	0.05	0.05	0.02	**0.01**	0.07	0.07	0.03	**0.02**
36	2670.7	1.11	1.10	1.04	**0.99**	0.06	0.05	0.02	**0.01**	0.09	0.09	0.04	**0.03**
37	3519.5	1.14	1.13	1.06	**0.99**	0.08	0.08	0.04	**0.02**	0.12	0.11	0.06	**0.04**
38	5570.8	1.19	1.18	1.09	**0.98**	0.12	0.12	0.06	**0.03**	0.16	0.16	0.08	**0.05**
39	6983.8	1.23	1.22	1.12	**0.97**	0.15	0.15	0.08	**0.03**	0.20	0.20	0.11	**0.06**
40	9103.3	1.29	1.28	1.18	**0.97**	0.19	0.19	0.12	**0.05**	0.25	0.25	0.16	**0.08**
41	13,635.3	1.36	1.36	1.24	**0.96**	0.27	0.27	0.18	**0.06**	0.33	0.33	0.21	**0.09**
42	16,634.2	1.43	1.43	1.30	**0.95**	0.32	0.32	0.22	**0.07**	0.39	0.39	0.27	**0.10**

Table 8
15-puzzle. Manhattan Distance, $r = 25$.

d	IDA*	Signed error				Unsigned error				RMSRE			
		$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}	$T_{h,b}$	ϵ-$T_{h,b}$	T_{gc}	ϵ-T_{gc}
50	8,909,564.5	1.16	1.16	1.18	**1.08**	0.09	0.09	0.10	**0.05**	0.12	0.12	0.12	**0.07**
51	15,427,786.9	1.15	1.15	1.19	**1.07**	0.11	0.11	0.12	**0.07**	0.13	0.13	0.14	**0.08**
52	28,308,808.8	1.25	1.25	1.28	**1.14**	0.14	0.14	0.17	**0.09**	0.18	0.18	0.20	**0.11**
53	45,086,452.6	1.23	1.23	1.29	**1.13**	0.16	0.16	0.20	**0.11**	0.20	0.20	0.23	**0.13**
54	85,024,463.5	1.36	1.36	1.41	**1.22**	0.21	0.21	0.27	**0.15**	0.26	0.26	0.30	**0.17**
55	123,478,361.5	1.36	1.36	1.45	**1.24**	0.24	0.24	0.31	**0.17**	0.29	0.29	0.34	**0.20**
56	261,945,964.0	1.44	1.44	1.54	**1.30**	0.28	0.28	0.39	**0.21**	0.35	0.35	0.43	**0.25**
57	218,593,372.3	1.43	1.43	1.57	**1.32**	0.33	0.33	0.45	**0.26**	0.40	0.40	0.49	**0.30**

Table 9
15-puzzle. Inconsistent heuristic, $r = 1$.

d	IDA*	Signed error				Unsigned error				RMSRE			
		$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c	$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c	$T_{h,b}$	ϵ-$T_{h,b}$	T_c	ϵ-T_c
50	562,708.5	0.55	0.24	1.77	**1.20**	537.97	124.62	1.29	**1.17**	3157.70	733.36	2.14	**2.08**
51	965,792.6	0.70	0.31	1.39	**1.04**	812.37	157.73	1.32	**1.12**	6449.94	1236.62	2.30	**1.92**
52	1,438,694.0	**0.96**	0.43	1.68	1.23	513.99	151.99	1.52	**1.35**	2807.23	696.84	2.51	**2.34**
53	2,368,940.3	**1.29**	0.58	1.75	1.32	694.34	216.27	1.56	**1.26**	5665.10	1696.83	2.56	**2.05**
54	3,749,519.9	1.64	**0.73**	2.03	1.54	647.24	226.79	1.77	**1.53**	3309.93	1054.35	2.75	**2.46**
55	7,360,297.6	1.90	**0.86**	2.07	1.59	650.59	246.84	1.72	**1.35**	5080.16	1900.50	2.68	**2.12**
56	12,267,171.0	2.30	**1.03**	2.19	1.61	927.71	367.99	2.16	**1.86**	6380.99	2454.03	3.53	**3.41**
57	23,517,650.8	2.69	**1.21**	2.29	1.78	600.13	243.38	2.02	**1.55**	3819.40	1522.75	3.08	**2.40**

6.2. Pancake puzzle

For the 15-pancake puzzle, we used 10 random start states to determine the ϵ_i values and 1000 to measure prediction accuracy. We used T_h and T_c as the type systems. The coarser type systems used to define the supertypes for T_h and T_c were T_h and $T_p(s) = (h(s))$, respectively. To define $\pi(t|u)$ and β_t, 100 million random states were sampled and, in addition, we used the extended sampling process described for the 15-puzzle. The results with $r = 4$ and a PDB heuristic that keeps the identities of the smallest eight pancakes are shown in the upper part of Table 10. In both cases T_c outperforms T_h but is also substantially improved by ϵ-truncation. As in the previous experiment, here ϵ-truncation does not modify the predictions for the coarser type system.

Author's personal copy

Table 10
15-pancake puzzle, $r = 4$.

d	IDA*	Signed error				Unsigned error				RMSRE			
		T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c
Admissible and consistent heuristic													
11	44,771.2	1.12	1.12	1.06	**1.00**	0.51	0.51	0.19	**0.13**	0.59	0.59	0.22	**0.15**
12	346,324.5	1.15	1.15	1.07	**0.98**	0.59	0.59	0.23	**0.14**	0.70	0.70	0.27	**0.18**
13	2,408,281.6	1.27	1.27	1.14	**1.01**	0.63	0.63	0.25	**0.15**	0.74	0.74	0.30	**0.19**
14	20,168,716.0	1.37	1.37	1.19	**1.05**	0.67	0.67	0.28	**0.17**	0.78	0.78	0.33	**0.21**
15	127,411,357.4	1.60	1.60	1.30	**1.15**	0.76	0.76	0.32	**0.20**	0.84	0.84	0.37	**0.25**
The heuristic above multiplied by 1.5													
12	188,177.1	1.99	1.99	1.25	**1.13**	1.50	1.50	0.50	**0.37**	1.78	1.78	0.62	**0.47**
13	398,418.8	2.12	2.12	1.31	**1.12**	1.61	1.61	0.52	**0.39**	2.08	2.08	0.74	**0.54**
14	3,390,387.6	2.31	2.31	1.37	**1.11**	1.62	1.62	0.50	**0.32**	1.96	1.96	0.68	**0.44**
15	6,477,150.7	2.23	2.23	1.27	**0.98**	1.73	1.73	0.54	**0.36**	2.23	2.23	0.75	**0.49**
16	16,848,215.1	2.79	2.79	1.49	**1.12**	1.97	1.97	0.55	**0.37**	2.46	2.46	0.75	**0.47**

Table 11
$3 \times 3 \times 3$ Rubik's Cube, $r = 3$.

d	IDA*	Signed error				Unsigned error				RMSRE			
		T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c	T_h	ϵ-T_h	T_c	ϵ-T_c
Admissible and consistent heuristic													
9	119,506.2	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
10	1,626,583.9	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
11	21,985,207.8	0.94	0.94	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
12	295,893,415.9	0.93	0.93	**0.99**	**0.99**	0.06	0.06	**0.02**	**0.02**	0.07	0.07	**0.03**	**0.03**
The heuristic above multiplied by 1.5													
9	7515.5	0.91	0.91	**0.98**	**0.98**	0.08	0.08	**0.02**	**0.02**	0.09	0.09	**0.03**	**0.03**
10	51,616.2	0.91	0.91	**0.98**	**0.98**	0.08	0.08	**0.02**	**0.02**	0.10	0.10	**0.03**	**0.03**
11	685,630.9	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.03**	**0.03**	0.09	0.09	**0.04**	**0.04**
12	8,674,465.2	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.02**	**0.02**	0.09	0.09	**0.03**	**0.03**
13	116,376,337.0	0.91	0.91	**0.98**	**0.98**	0.07	0.07	**0.02**	**0.02**	0.08	0.08	**0.03**	**0.03**

6.3. Rubik's Cube

For the $3 \times 3 \times 3$ Rubik's Cube we used 10 random start states to determine the ϵ_i values and 1000 to measure prediction accuracy. We sampled 100 million random states. The random states were generated by random walks from the goal state, whose length was randomly selected between 0 and 180 steps. We prune redundant moves in the main search as described by Korf [16], which reduces the branching factor from 18 to approximately 13.35. Korf considered two kinds of redundant move pruning. First, he noted that twisting the same face twice in a row leads to redundant states; second, twisting the front face and then the back face leads to the same state as twisting the faces in opposite order. We used the same procedure described by Zahavi et al. [7] to implement redundant move pruning during sampling: the last operator in the random walk is used as a basis to prune redundant moves. We used T_h and T_c as the type systems. The coarser type systems used to define the supertypes for T_h and T_c were T_p and T_h, respectively. The heuristic we used was a PDB of the 8 corner cubies [16] over an abstraction on the sides of the puzzle. The corner cubies are those with three sides exposed in the puzzle, see Fig. 8. The abstraction was built by mapping three colors to one color and the other three colors to a second color, in such a way that, in the abstract goal state, two opposite sides of the cube always differ in color.

The upper part of Table 11 shows the results while using the admissible and consistent heuristic described above. Here, T_h results in fairly accurate predictions, which are further improved when T_c is used. ϵ-truncation does not modify the predictions in this case.

The lower part of Table 11 shows the results when the admissible and consistent heuristic used in the previous experiment is multiplied by 1.5. The resulting heuristic is inadmissible and inconsistent. Similar to the previous experiment, CDP makes very accurate predictions and ϵ-truncation does not modify the results.

7. Lookup CDP

We have demonstrated that ϵ-truncation improves the accuracy of the CDP predictions. We now present Lookup CDP (L-CDP), a variant of CDP that improves its runtime; it 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 Eq. (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 Fig. 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

Author's personal copy

subproblems, one for $(u_1, d-1)$ and another for $(u_2, d-1)$; the sum of the solutions to these subproblems plus one (as the start state was expanded) gives the solution to the initial problem. In L-CDP, the predicted number of nodes expanded by each pair (t, d) is computed in 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 3000 different types, and the number of different cost bounds in this domain is 16, which results in only $3000 \times 16 = 48{,}000$ 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 L-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 [11].
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.

7.1. Experimental results on lookup CDP

We now compare the prediction runtime of CDP with that of 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. We ran experiments on the 15-puzzle, 15-pancake puzzle, and Rubik's Cube using the consistent heuristics described before. We used a set of 1000 random start states to measure the runtime for each of the domains.

Table 12 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.

8. The Knuth–Chen method

We now review a method introduced by Knuth [9] that was later improved by Chen [17] and which can also be used to predict the number of nodes expanded on an iteration of IDA* with a given cost bound. The Knuth–Chen method, Stratified Sampling (or SS for short), also uses type systems (Chen called them stratifiers). We will show empirically that the type systems developed to be used with CDP substantially improve the predictions of SS.

Knuth [9] 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 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 [17] 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 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 s when computing s's type.

SS can be used to approximate any function of the form

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

Author's personal copy

Table 12
L-CDP and CDP runtime (seconds).

15-puzzle						
d	$r = 5$		$r = 10$		$r = 25$	
	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						
d	$r = 1$		$r = 2$		$r = 4$	
	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						
d	$r = 2$		$r = 3$		$r = 4$	
	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

where z is any function assigning a numerical value to a node, and, as above, $S(s^*)$ 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 that subtrees rooted at nodes of the same type will have equal values of φ and so only one node of each type, chosen randomly, is expanded. This is the key to SS's efficiency since the search trees of practical interest have far too many nodes to be examined exhaustively.

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

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

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'

Author's personal copy

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 ⟨s, w⟩ for the nodes s expanded at level i.
3: initialize A[1] // see text
4: i ← 1
5: while stopping condition is false do
6:   for each element ⟨s, w⟩ in A[i] do
7:     for each child c of s do
8:       if h(c) + g(c) ⩽ d then
9:         if A[i + 1] contains an element ⟨s′, w′⟩ with T(s′) = T(c) then
10:          w′ ← w′ + w
11:          with probability w/w′, replace ⟨s′, w′⟩ in A[i + 1] by ⟨c, w′⟩
12:        else
13:          insert new element ⟨c, w⟩ in A[i + 1]
14:        end if
15:      end if
16:    end for
17:  end for
18:  i ← i + 1
19: end while
```

according to the probability shown in Line 11. Chen [17] 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 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 [17] 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.

9. Better type systems for SS

The prediction accuracy of SS, like that of CDP, depends on the type system used to guide its sampling [10]. 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.

Like CDP, it is easy to see that SS using a pure type system (i.e., a type system that groups together nodes that root subtrees of the same size) makes perfect predictions. However, as we stated before, 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 nodes with higher heuristic value "more quickly".

We now show empirically that using a type system that accounts for the information provided by a heuristic function instead of Chen's substantially improves SS's predictions.

9.1. 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 equal or less time than V'; we also say that V dominates V' if V is able to produce equally or 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 this section prediction accuracy is measured in terms of the Relative Unsigned Error. 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

Author's personal copy

Table 13
SS employing different type systems.

15-puzzle

d	IDA*	Runtime (s)		Error	
		T_{nc} (5000)	T_h (50)	T_{nc} (5000)	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

d	IDA*	Runtime (s)		Error	
		T_{nc} (1000)	T_c (1)	T_{nc} (1000)	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

d	IDA*	Runtime (s)		Error	
		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**

heuristic value of the node and its parent. The results were averaged over 1000 random start states. The number of probes used in each experiment is shown in parentheses after the name of the type system used.

The results for the 15-puzzle when using the inconsistent heuristic created by Zahavi et al. [7] and defined in Section 6.1 are presented in the upper part of Table 13. We chose the number of probes so 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 5000. 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 became slower than T_h before its accuracy equaled T_h's. In Table 13 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. For both 15-pancake puzzle and Rubik's Cube we used the consistent heuristics defined in Section 6. 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.

10. Comparison between the enhanced versions of CDP and SS

In this section we make an empirical comparison of our enhanced versions of CDP and SS: L-CDP with ϵ-truncation and SS using CDP's type systems. 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 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. Here we run experiments on the 15-puzzle, 15-pancake puzzle and Rubik's Cube with the consistent heuristics described in Section 6.

10.1. Fast predictions

We start with fast predictions. The results are shown in Table 14. The value in parentheses 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

Author's personal copy

Table 14
Fast predictions. L-CDP and SS.

15-puzzle

d	IDA*	Runtime (s)		Error	
		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

d	IDA*	Runtime (s)		Error	
		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

d	IDA*	Runtime (s)		Error	
		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

about 1000 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.

10.2. Accurate predictions

The results for accurate predictions are shown in Table 15. 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.

As observed in the results shown in Section 6, often the error of the CDP predictions increases as we increase the cost bound. For instance, the CDP error shown in Table 15 for the 15-puzzle is 0.05 for $d = 50$, and it grows to 0.26 for $d = 57$. 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.

10.3. Experiments on larger state spaces

In this section we evaluate both CDP with ϵ-truncation and SS on larger state spaces, namely the 24-puzzle and the 60-pancake puzzle.

In the experiments in this section we do not use L-CDP. This is because we use a relatively large value of r in order to produce accurate predictions. Recall that L-CDP and CDP take approximately the same amount of time to produce predictions for larger values of r (see Table 12 in Section 7.1).

10.3.1. 24-puzzle

For the 24-puzzle we used the 6–6–6–6 disjoint PDBs [18]. One single random instance was used to compute the ϵ-values. We used a value of r of 25 and it took about 36 hours to sample one billion states for CDP. SS used 50 probes. Finally, for both CDP and SS we used the T_{gc} type system. Table 16 shows the prediction results for the number of nodes

Author's personal copy

Table 15
Accurate predictions. L-CDP and SS.

15-puzzle					
d	IDA*	Runtime (s)		Error	
		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					
d	IDA*	Runtime (s)		Error	
		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					
d	IDA*	Runtime (s)		Error	
		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 16
CDP and SS on the 24-puzzle using the 6–6–6–6 PDB.

24-puzzle					
d	IDA*	Runtime (s)		Error	
		CDP (25)	SS (50)	CDP (25)	SS (50)
90	164,814,526.6	30.9482	**7.5034**	0.20	**0.03**
92	368,992,103.4	66.6174	**8.9480**	0.34	**0.03**
94	1,985,011,441.3	178.7885	**12.5902**	0.65	**0.04**
96	4,874,007,803.3	277.2403	**16.2077**	1.07	**0.03**
98	11,015,303,521.6	455.4678	**20.2952**	1.68	**0.04**
100	11,976,556,484.1	684.3925	**22.7853**	2.48	**0.04**
102	27,500,453,677.2	1058.3840	**26.0549**	3.76	**0.05**
104	108,902,222,694.8	1643.4890	**30.8277**	5.68	**0.05**
106	204,754,382,723.4	2055.6937	**33.7998**	8.15	**0.06**
108	277,502,287,352.6	2335.7943	**33.6668**	11.47	**0.08**
110	1,954,871,642,630.4	4161.4029	**36.8365**	20.39	**0.10**

generated during IDA* searches on 200 start states. The trend that was observed in the experiment on the 15-puzzle shown in Table 15 is also observed here: as the search gets deeper, CDP's prediction accuracy worsens. SS, on the other hand, makes accurate predictions across different cost bounds. For instance, for a cost bound of 110 SS has an average absolute error of only 0.10.

This experiment also shows that the prediction methods studied in this paper can be substantially faster than IDA* performing the actual search. For instance, IDA* takes approximately 90 hours on average to execute an iteration with cost bound of 110. SS takes only 37 seconds on average to make predictions with the same cost bound—a speedup of more than 8700 times compared to the runtime of the actual IDA* search. CDP is not as fast as SS but it is still substantially faster than the actual IDA* search as CDP takes only little more than one hour on average to make predictions with the cost bound of 110.

10.3.2. 60-pancake puzzle

For the 60-pancake puzzle we used the GAP heuristic [19]. SS tends to perform better on the 60-pancake puzzle when using the T_h type system rather than when using the T_c or the T_{gc} type systems. This is because the 60-pancake puzzle has a relatively large branching factor, namely 59. The larger branching factor slows down the T_c and T_{gc} type computation due

Author's personal copy

Table 17
CDP and SS on the 60-pancake puzzle using the GAP heuristic.

60-pancake puzzle					
d	IDA*	Runtime (s)		Error	
		CDP (8)	SS (400)	CDP (8)	SS (400)
55	4,661,209.3	0.0407	0.1193	0.21	0.20
56	21,878,193.1	0.0970	0.1506	0.21	0.20
57	40,279,688.4	**0.1435**	0.1726	**0.22**	0.22
58	82,790,542.6	**0.1575**	0.1860	**0.21**	0.21
59	242,822,659.9	0.2599	0.2325	0.19	0.21

to the lookahead these type systems perform. The T_c and T_{gc} type computation is also slower for CDP on the 60-pancake puzzle. However, in CDP, most of the expensive type computations are done as a preprocessing step, during sampling. Therefore, we use T_c for CDP and T_h for SS. CDP sampled 10 million start states in 94 hours. As in the other experiments in this paper, we assume that the time required for sampling is amortized over a large number of predictions. For CDP we used 5 problem instances to compute the ϵ-values and an r-value of 8. For SS we used 400 probes.

The results shown in Table 17 are averages over 340 start states. Both CDP and SS are able to quickly make accurate predictions on the 60-pancake puzzle—the absolute error is at most 0.22 and the runtime is at most 0.26 seconds for both algorithms. IDA* using the GAP heuristic takes approximately 30 seconds on average to solve a random instance of the 60-pancake puzzle. CDP and SS take less than a quarter of a second to predict the number of nodes expanded in a given iteration of IDA* in most of the cases. Thus they produce accurate predictions much more quickly than IDA* can solve the problem. The prediction methods are even faster if we compare the time required for IDA* to finish a complete iteration for a given cost bound—recall that IDA* finishes an iteration as soon as a goal is found. For instance, for a cost bound of 59, IDA* takes approximately 37 minutes to finish a complete iteration, ignoring the goal if one is found. Both CDP and SS make predictions for the same cost bound in about a quarter of a second—a speedup of more than 9000 times over the runtime of IDA*.

11. Discussion

We showed empirically that by carefully ignoring rare events ϵ-truncation can substantially improve the accuracy of CDP's predictions. We conjecture that these harmful rare events come from noisy values of $p(t|u)$: the values of $p(t|u)$ represent the type transition probability averaged across the state space, which can be different from the type transition probability averaged across the search tree for a particular start state. Chen [17] was able to prove that the expected value of an SS prediction is the actual number of nodes expanded, i.e., SS is an unbiased estimator, because SS samples the search tree being approximated. The same cannot be said about CDP. We observed empirical evidence that ϵ-truncation minimizes the error inserted by noisy $p(t|u)$-values, but it does not guarantee unbiased predictions. On the other hand, being domain-specific allows CDP to store the prediction results in a lookup table as a preprocessing step and produce predictions much more quickly than SS. 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.

We also showed that both CDP and SS can be used to make predictions on larger state spaces. We observed that CDP and SS can produce predictions much more quickly than IDA* can solve problem instances on the 24-puzzle and on the 60-pancake puzzle. For instance, SS is approximately 8700 times faster than IDA* on the 24-puzzle using the 6–6–6–6 disjoint PDBs as heuristic function. As long as the time required for sampling the state space by CDP can be amortized, both CDP and SS can produce predictions much more quickly than IDA* can solve problem instances.

12. Related work

The approach taken by Chen [17], Korf et al. [2], and Zahavi et al. [7] of predicting the number of nodes expanded on an iteration of IDA* is in contrast with the approach to search complexity analysis, which focused on "big-O" complexity typically parameterized by the accuracy of the heuristic [3–6,20].

Many other algorithms were developed based on Knuth's ideas. For instance, Kilby et al. [21] introduced an online estimator of the size of backtrack search trees of branch-and-bound search algorithms. Later Haim and Walsh [22] used Kilby et al.'s method as a feature for their machine-learned online algorithm for estimating the runtime of SAT solvers. Allen and Minton [23] adapted Knuth's algorithm for constraint satisfaction problems; Lobjois and Lemaître [24] used Knuth's algorithm to select the most promising branch-and-bound algorithm for a given problem; Bresina et al. [25] used Knuth's algorithm to measure the expected solution quality of a scheduling problem. All these algorithms could potentially benefit from the idea of using a heuristic function (or some other source of information) to define type systems to reduce the variance of random probing.

Haslum et al. [26] used KRE to evaluate different PDB heuristics for domain-independent planning. They posed the problem of selecting good abstractions to construct pattern databases as an optimization problem. A hill climbing search

Author's personal copy

algorithm is then employed and KRE is used as an evaluation function. Other prediction methods such as CDP and SS could also be used for this purpose.

Breyer and Korf [27] showed how to use KRE to make accurate predictions of the number of nodes expanded on average for the special case of consistent heuristics by the A* algorithm for the 15-puzzle. In order to make predictions of the number of nodes expanded by A*, due to the transposition detection the algorithm does, one needs to know the number of nodes at a level i in the brute-force search graph [27]. For domains in which the search graph cannot be enumerated, accurately predicting the number of nodes expanded by A* remains an open problem.

Burns and Ruml [28] presented IM, a prediction method that works in domains with real-valued edge costs. IM was developed to make estimations of the number of nodes expanded by IDA* as the algorithm searches. Burns and Ruml's goal was to avoid the poor performance of IDA* in domains with real-valued edge costs by setting a cost bound d that would expand an exponentially larger number of nodes in each iteration. IM works by learning the variation of the f-value (where, for node n, $f(n) = g(n) + h(n)$) between a node and its children. Like CDP, this is done based on a type system, i.e., IM learns the value Δ by which the f-value changes when a node of type u generates a node of type t. In fact, CDP can be seen as a special case of IM, when the edges have unitary cost. The difference between CDP and IM is that IM implicitly incorporates the cost to generate a child in its type system, while CDP assumes in its formulas that the cost is always one. Not surprisingly, Burns and Ruml verified empirically that in domains with unit edge-costs IM and CDP produce predictions with indistinguishable accuracy. Like CDP, IM could also benefit from ϵ-truncation.

13. Conclusion

In this paper we advanced two lines of research, namely, we improved the runtime and prediction accuracy of CDP and SS, two algorithms that were developed independently of each other for predicting the number of nodes expanded on an iteration of a backtrack search algorithm such as IDA*.

As for the CDP algorithm, we have identified a source of prediction error that had previously been overlooked, namely, that low probability events can degrade predictions in certain circumstances. We call this the discretization effect. This insight led us to the ϵ-truncation method for altering the probability distribution used for making predictions at level i of the search tree by setting to zero all probabilities smaller than ϵ_i, an automatically derived threshold for level i. Our experimental results showed that more informed type systems for prediction often suffer more from the discretization effect than less informed ones, sometimes leading to the pathological situation that predictions based on the more informed system are actually worse than those based on the less informed system. In our experiments ϵ-truncation rarely degraded predictions; in the vast majority of cases it improved predictions, often substantially. In addition, 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.

As for the SS algorithm, 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.

Finally, we made an empirical comparison between our enhanced versions of CDP and SS. Our experimental results point out that if 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

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, the Alberta Ingenuity Centre for Machine Learning (AICML), and Canada's Natural Sciences and Engineering Research Council (NSERC).

The authors would like to thank Ariel Felner for his helpful comments on an earlier draft of this paper. We also would like to thank Rong Zhou for providing the number of nodes generated for the instances of the 24-puzzle used in our experiments.

References

[1] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.
[2] R.E. Korf, M. Reid, S. Edelkamp, Time complexity of iterative-deepening-A*, Artificial Intelligence 129 (1–2) (2001) 199–218.
[3] H.T. Dinh, A. Russell, Y. Su, On the value of good advice: The complexity of A* search with accurate heuristics, in: Proceedings of the 22nd Conference on Artificial Intelligence (AAAI 2007), 2007, pp. 1140–1145.
[4] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. thesis, Carnegie-Mellon University, 1979.
[5] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison & Wesley, 1984.
[6] I. Pohl, Practical and theoretical considerations in heuristic search algorithms, Mach. Intelligence 8 (1977) 55–72.
[7] U. Zahavi, A. Felner, N. Burch, R.C. Holte, Predicting the performance of IDA* using conditional distributions, J. Artificial Intelligence Res. 37 (2010) 41–83.
[8] J.C. Culberson, J. Schaeffer, Searching with pattern databases, in: Proceedings of the 11th Canadian Conference on Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 1081, Springer, 1996, pp. 402–416.
[9] D.E. Knuth, Estimating the efficiency of backtrack programs, Math. Comp. 29 (1975) 121–136.

Author's personal copy

[10] P.-C. Chen, Heuristic sampling on backtrack trees, Ph.D. thesis, Stanford University, 1989.
[11] L. Lelis, S. Zilles, R.C. Holte, Improved prediction of IDA*'s performance via ϵ-truncation, in: Proceedings of the 4th Annual Symposium on Combinatorial Search (SoCS 2011), 2011, pp. 108–116.
[12] L.H.S. Lelis, S. Zilles, R.C. Holte, Fast and accurate predictions of IDA*'s performance, in: Proceedings of the 26th Conference on Artificial Intelligence (AAAI 2012), 2012, pp. 514–520.
[13] J. Slocum, D. Sonneveld, The 15 Puzzle, Slocum Puzzle Foundation, 2006.
[14] A.F. Archer, A modern treatment of the 15-puzzle, Amer. Math. Monthly 106 (1999) 793–799.
[15] H. Dweighter, Problem E2569, Amer. Math. Monthly 82 (1975) 1010.
[16] R.E. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proceedings of the 14th Conference on Artificial Intelligence (AAAI 1997) and the 9th Conference on Innovative Applications of Artificial Intelligence (IAAI 1997), 1997, pp. 700–705.
[17] P.-C. Chen, Heuristic sampling: A method for predicting the performance of tree searching programs, SIAM J. Comput. 21 (1992) 295–315.
[18] R.E. Korf, A. Felner, Disjoint pattern database heuristics, Artificial Intelligence 134 (1–2) (2002) 9–22.
[19] M. Helmert, Landmark heuristics for the pancake problem, in: A. Felner, N.R. Sturtevant (Eds.), Proceedings of the Third Annual Symposium on Combinatorial Search, AAAI Press, 2010.
[20] S.V. Chenoweth, H.W. Davis, High-performance A* search using rapidly growing heuristics, in: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI 1991), 1991, pp. 198–203.
[21] P. Kilby, J.K. Slaney, S. Thiébaux, T. Walsh, Estimating search tree size, in: Proceedings of the 21st Conference on Artificial Intelligence (AAAI 2006), 2006, pp. 1014–1019.
[22] S. Haim, T. Walsh, Online estimation of SAT solving runtime, in: Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing, in: Lecture Notes in Computer Science, vol. 4996, Springer, 2008, pp. 133–138.
[23] J.A. Allen, S. Minton, Selecting the right heuristic algorithm: Runtime performance predictors, in: Proceedings of the 11th Canadian Conference on Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 1081, Springer, 1996, pp. 41–53.
[24] L. Lobjois, M. Lemaître, Branch and bound algorithm selection by performance prediction, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI 1998), 1998, pp. 353–358.
[25] J.L. Bresina, M. Drummond, K. Swanson, Expected solution quality, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), 1995, pp. 1583–1591.
[26] P. Haslum, A. Botea, M. Helmert, B. Bonet, S. Koenig, Domain-independent construction of pattern database heuristics for cost-optimal planning, in: Proceedings of the 22nd Conference on Artificial Intelligence (AAAI 2007), 2007, pp. 1007–1012.
[27] T. Breyer, R. Korf, Recent results in analyzing the performance of heuristic search, in: Proceedings of the First International Workshop on Search in Artificial Intelligence and Robotics (held in conjunction with AAAI), 2008, pp. 24–31.
[28] E. Burns, W. Ruml, Iterative-deepening search with on-line tree size prediction, in: Proceedings of the Sixth International Conference on Learning and Intelligent Optimization (LION 2012), 2012, pp. 1–15.