Estimating Search Tree Size with Duplicate Detection

Levi H. S. Lelis
Departamento de Informática
Universidade Federal de Viçosa
Viçosa, Brazil

Roni Stern
Information Systems Engineering
Ben Gurion University
Beer-Sheva, Israel

Nathan R. Sturtevant
Computer Science Department
University of Denver
Denver, USA

Abstract

In this paper we introduce Stratified Sampling with Duplicate Detection (SSDD), an algorithm for estimating the number of state expansions performed by heuristic search algorithms seeking solutions in state spaces represented by undirected graphs. SSDD is general and can be applied to estimate other state-space properties. We test SSDD on two tasks: (i) prediction of the number of A* expansions in a given f-layer when using a consistent heuristic function, and (ii) prediction of the state-space radius. SSDD has the asymptotic guarantee of producing perfect estimates in both tasks. Our empirical results show that in task (i) SSDD produces good estimates in all four domains tested, being in most cases orders of magnitude more accurate than a competing scheme, and in task (ii) SSDD quickly produces accurate estimates of the radii of the 4×4 Sliding-Tile Puzzle and the 3×3×3 Rubik's Cube.

Introduction

State-space search algorithms, such as A* (Hart, Nilsson, and Raphael 1968), are fundamental in many AI applications. A* uses the $f(s) = g(s) + h(s)$ cost function to guide its search. Here, $g(s)$ is the cost of the path from the start state to state s, and $h(s)$ is the estimated cost-to-go from s to a goal; $h(\cdot)$ is known as the heuristic function. Users of search algorithms usually do not know *a priori* how long it will take to solve a problem instance — some instances could be solved quickly while others could take a long time. One way of estimating the running time of a search algorithm is by predicting the number of node expansions the algorithm performs during search (Knuth 1975). Assuming that the time required for expanding a node is constant throughout the state space, an estimate of the number of expansions gives an estimate of the algorithm's running time.

Predictions of the number of node expansions could have other applications. For example, one could use the predictions to decide which heuristic function or search algorithm to use to solve a particular problem instance. Or, one could use the predictions to fairly divide the search workload among different computers in a parallel computing setting.

Many search algorithms, such as IDA* (Korf 1985), expand a tree while searching the state space. By contrast, A* using a consistent heuristic[1] expands a graph while searching the state space. The distinction between trees and graphs is important: multiple nodes in a search tree might represent the same state in a search graph. As a result, the tree might be substantially larger than the graph. For example, in Figure 1 the graph is finite while the tree is not. Several algorithms have been developed for estimating the search tree size (Chen 1989; Knuth 1975; Korf, Reid, and Edelkamp 2001; Kilby et al. 2006; Zahavi et al. 2010; Burns and Ruml 2012; Lelis, Otten, and Dechter 2013; Lelis 2013; Lelis, Otten, and Dechter 2014). In this paper we introduce an algorithm for estimating the size of the expanded graph. We use the word "state" to refer to a vertex in the graph and the word "node" to refer to a vertex in the tree. Throughout this paper we do parent pruning, i.e., the parent of a given node is not generated among its children.

Contributions In this paper we introduce a novel algorithm for estimating the size of undirected search graphs. Our algorithm is based on Stratified Sampling (SS) (Chen 1989). SS samples the search tree and it does not account for multiple nodes representing the same state. As a result, SS usually produces gross overestimates when used to estimate the size of a search graph. We equip SS with a system for detecting nodes in the search tree that represent the same state. We name the resulting algorithm Stratified Sampling with Duplicate Detection (SSDD). We build our system on top of SS because Lelis et al. (2013) have shown that SS produces much more accurate predictions of the search tree size than competing schemes such as CDP (Zahavi et al. 2010).

SSDD is general and can be used to estimate other state-space properties. We apply SSDD to two tasks: (i) prediction of the number of A* expansions in a given f-layer when using consistent heuristics, and (ii) prediction of the state-space radius from the goal, i.e., the maximum shortest distance from any state to a goal. SSDD has the asymptotic guarantee of producing perfect estimates in both tasks. Our empirical results show that in task (i) SSDD produces predictions that are up to several orders of magnitude more accu-

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

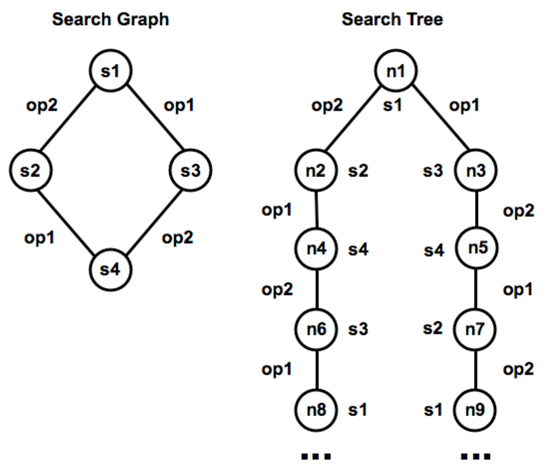

Figure 1: Search graph and the search tree generated when s_1 is the root. The label by the nodes in the search tree shows the state each node represents.

rate than SS in four standard search benchmarks. In task (ii) SSDD produces accurate estimates of the state-space radius from the goal state for the (4×4) Sliding-Tile puzzle and the 3×3×3 Rubik's Cube.

Problem Formulation

Let the *underlying search tree* (*UST*) be the full brute-force tree created from a connected, undirected and implicitly-defined *underlying search graph* (*USG*) describing a state space. Some search algorithms expand a subtree of the *UST* while searching for a solution (e.g., a portion of the *UST* might not be expanded due to heuristic guidance); we call this subtree the *expanded search tree* (*EST*). In this paper we want to estimate the size of the subgraph expanded by an algorithm searching the *USG*; we call this subgraph the *expanded search graph* (*ESG*).

Let $G = (N, E)$ be a graph representing an *ESG* where N is its set of states and for each $n \in N$ $op(n) = \{op_i | (n, n_i) \in E\}$ is its set of operators. We use the words edges and operators interchangeably. The prediction task is to estimate the size of N without fully generating the *ESG*.

Stratified Sampling

Knuth (1975) presented a method to estimate the size of the *EST* of search algorithms such as IDA*. His method worked by repeatedly performing a random walk from the root of the tree. Under the assumption that all branches have the same structure, performing a random walk down one branch is enough to estimate the size of the entire tree. Knuth observed that his method was not effective when the tree is unbalanced. Chen (1992) addressed this problem with a stratification of the search tree through a *type system* to reduce the variance of the sampling process. We call Chen's method Stratified Sampling (SS).

Algorithm 1 SS, a single probe

```
Input: root $n^*$ of a tree and a type system $T$
Output: an array of sets $A$, where $A[i]$ is the set of pairs
        $\langle n, w\rangle$ for the nodes $n$ expanded at level $i$.
 1: $A[0] \leftarrow \{\langle n^*, 1\rangle\}$
 2: $i \leftarrow 0$
 3: while stopping condition is false do
 4:   for each element $\langle n, w\rangle$ in $A[i]$ do
 5:     for each child $\hat{n}$ of $n$ do
 6:       if $A[i+1]$ contains an element $\langle n', w'\rangle$ with
          $T(n') = T(\hat{n})$ then
 7:         $w' \leftarrow w' + w$
 8:         with probability $w/w'$, replace $\langle n', w'\rangle$ in
            $A[i+1]$ by $\langle \hat{n}, w'\rangle$
 9:       else
10:         insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$
11:   $i \leftarrow i + 1$
```

Definition 1 (Type System). *Let $S = (N, E)$ be a* UST. *$T = \{t_1, \ldots, t_n\}$ is a type system for S if it is a disjoint partitioning of N. If $n \in N$ and $t \in T$ with $n \in t$, we write $T(n) = t$.*

SS is a general method for approximating any function of the form

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

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

Instead of traversing the entire tree and summing all z-values, SS assumes subtrees rooted at nodes of the same type 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 search trees of practical interest have far too many nodes to be examined exhaustively.

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

$$\hat{\varphi}(n^*, T) = \sum_{\langle n, w\rangle \in A} w \cdot z(n) , \tag{1}$$

Algorithm 1 shows SS in detail. The set A is divided into subsets, one for every level in the search tree; $A[i]$ is the set of representative-weight pairs for the types encountered at level i. In SS the types are required to be partially ordered such that a node's type must be strictly greater than the type of its parent. Chen suggests that this can always be guaranteed by adding the depth of a node to the type system and then sorting the types lexicographically. In our implementation of SS, types at one level are treated separately from types at another level by the division of A into the $A[i]$. If the same type occurs on different levels, the occurrences will be treated as though they were different types — the depth of search is implicitly added to the type system.

$A[0]$ is initialized to contain only the root of the tree to be probed, with weight 1 (line 1). In each iteration (lines 4 through 10), all the nodes from $A[i]$ are expanded to get representative nodes for $A[i+1]$ as follows. Every node in $A[i]$ is expanded and its children are considered for inclusion in $A[i+1]$. If a child $\hat{n}$ has a type t that is already represented in $A[i+1]$ by another node n', then a *merge* action on $\hat{n}$ and n' is performed. In a merge action we increase the weight in the corresponding representative-weight pair of type t by the weight $w(n)$ of $\hat{n}$'s parent n (from level i) since there were $w(n)$ nodes at level i that are assumed to have children of type t at level $i+1$. $\hat{n}$ will replace the n' according to the probability shown in line 8. Chen (1992) proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, we expand the nodes in $A[i+1]$. This process continues until reaching a level i^* where $A[i^*]$ is empty.

One run of the SS algorithm is called a *probe*, and we denote as $\hat{\varphi}^{(p)}(n^*, T)$ the p-th probing result of SS. SS is unbiased, i.e., the average of the $\hat{\varphi}(n^*, T)$-values converges to $\varphi(n^*)$ in the limit as the number of probes goes to infinity.

Theorem 1. *(Chen 1992) Given a partially-ordered type system T and a set of p independent probes $\hat{\varphi}^{(1)}(n^*, T), \cdots, \hat{\varphi}^{(p)}(n^*, T)$ from a search tree $S(n^*)$, $\frac{1}{p}\sum_{j=1}^{p}\hat{\varphi}^{(j)}(n^*, T)$ converges to $\varphi(S)$.*

While SS is unbiased when estimating the size of the search tree, it is not unbiased for estimating the size of the search graph. During sampling SS accounts for multiple nodes representing the same state in the graph. As a result, SS's estimation converges to the size of the tree, not to the size of the graph. As shown in Figure 1, the size of the tree can be a gross overestimate of the size of the graph.

Predicting the Size of the Search Graph

We now introduce *Stratified Sampling with Duplicate Detection* (SSDD), an algorithm for estimating properties of the *USG* such as the *ESG* size of heuristic search algorithms.

A path π in the *UST* is a sequence of operators applied to the root node n^*. Applying an operator incurs a cost, and the cost of a path is the sum of the costs of its constituent operators.

Definition 2 (Cost-lexicographic order). *A cost-lexicographic order over paths is based on a total order of the operators $o_1 < o_2 < \cdots < o_n$. For arbitrary paths A and B, a cost-lexicographic order would order A before B iff either the cost of A is lower than the cost of B, or their costs are equal and $o_a < o_b$ where o_a and o_b are the left-most operators where A and B differ (o_a is in A and o_b is in the corresponding position in B).*

Cost-lexicographic order is similar to the length-lexicographic operator sequence order defined by Burch and Holte (2012). Let $\mathcal{O}$ be a cost-lexicographic order over paths in the *UST*. Since the *UST* is a tree rooted at n^*, every node n in it corresponds to a path, and thus $\mathcal{O}$ also defines an order over nodes. For ease of notation, when a node n is smaller than n' according to $\mathcal{O}$ we write $n < n'$.

Algorithm 2 SSDD, a single probe

```
Input: root $n^*$ of a tree, a type system $T$, number of random
    walks for duplicate detection $k$
Output: an array of sets $A$, where $A[i]$ is the set of pairs
    $\langle n, w\rangle$ for the nodes $n$ expanded at level $i$.
 1: $A[0] \leftarrow \{\langle n^*, 1\rangle\}$
 2: $i \leftarrow 0$
 3: while stopping condition is false do
 4:   for each element $\langle n, w\rangle$ in $A[i]$ do
 5:     if SDD$(n, k)$ then
 6:       remove $n$ from $A[i]$
 7:       continue to the next pair in $A[i]$
 8:     for each child $\hat{n}$ of $n$ do
 9:       if $A[i+1]$ contains an element $\langle n', w'\rangle$ with
          $T(n') = T(\hat{n})$ then
10:         $w' \leftarrow w' + w$
11:         with probability $w/w'$, replace $\langle n', w'\rangle$ in
            $A[i+1]$ by $\langle \hat{n}, w'\rangle$
12:       else
13:         insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$
14:   $i \leftarrow i + 1$
```

Let $state(n)$ denote the state in the *USG* represented by node n in the *UST*, and let $\pi(n)$ denote a path from the root n^* to n in the *UST*. A *cycle* is a non-empty path π in the *UST* from n_1 to node n_2 such that $state(n_1) = state(n_2)$. Because the *USG* is undirected, for any node n in a cycle π, there are two paths $\pi_1 \subseteq \pi$ from n to n_1 and $\pi_2 \subseteq \pi$ from n to n_2 in the *USG* with $\pi_1 \cup \pi_2 = \pi$. Since we do parent pruning, then $\pi_1 \neq \pi_2$.

Definition 3 (Canonical and Duplicate Nodes). *A node n in the* UST *is called a duplicate node if there exists another node n' in the* UST *such that $state(n) = state(n')$ and $n' < n$ according to $\mathcal{O}$. A node that is not a duplicate is called a canonical node.*

Clearly, there is exactly one canonical node in the *UST* for each state in the *USG*.

Stratified Sampling with Duplicate Detection

SSDD combines SS, which samples the *UST*, with a stochastic duplicate detection algorithm for pruning duplicate nodes. If the duplicate detection algorithm correctly identifies all duplicate nodes, then SS considers only canonical nodes, de facto sampling the *USG*, as required.

Algorithm 2 shows SSDD's pseudocode. As can be seen, it is very similar to SS (Algorithm 1). In fact, the only difference between SS and SSDD is that SSDD uses our duplicate detection algorithm (called SDD and described next) for pruning duplicate nodes (lines 5 to 7 in Algorithm 2).

Sampling-based Duplicate Detection (SDD) Let n be a duplicate node. This means that there exists an alternative

Algorithm 3 SDD

```
Input: node $n$, number of random walks $k$
Output: boolean indicating whether $n$ is a duplicate node
 1: Put all states in $\pi(n)$ in HashTable
 2: for 1 to $k$ do
 3:     $m \leftarrow n$, $cost \leftarrow 0$
 4:     while $cost \leq g(n)$ do
 5:         $c \leftarrow$ random child of $m$
 6:         if $c$ is the same state as $c'$ in HashTable then
 7:             if path from $c$ to $n$ is cost-lexicographically
                smaller than path from $c'$ to $n$ then
 8:                 return true
 9:         $m \leftarrow c$ ; $cost \leftarrow cost + cost(m, c)$
10: return false
```

path from n^* to $state(n)$ that is smaller (according to $\mathcal{O}$) than the original path $\pi(n)$ from n^* to n. Since the *USG* is undirected, all alternative paths between n^* and $state(n)$ are reachable from n. SDD exploits this, and checks if n is a duplicate by searching the subtree below n for a path back to n^* that is smaller than $\pi(n)$.

Full exploration of the subtree below n is too costly to be practical. As an alternative, SDD performs k random walks from n. If one of these random walks finds a path from n to n^* that is smaller than $\pi(n)$, then n is declared a duplicate. Otherwise, n is assumed to be canonical.

Each of these random walks is cost bounded by $g(n)$, due to the following lemma.

Lemma 1. *If n is a duplicate node, then there exists a path π' from n to the root n^* with cost of $g(n)$ or less.*

Proof. Since n is a duplicate, there exists a node n' such that $n' < n$ and $state(n') = state(n)$. As $\mathcal{O}$ is a cost-lexicographic order, $n' < n$ entails that $g(n') \leq g(n)$. Since the *USG* is undirected, this means that there is a path from n to the root n^* with cost $g(n)$ or lower, as required. □

The pseudocode for SDD is given in Algorithm 3. SDD accepts two parameters: node n and number of random walks k. First, all states in $\pi(n)$ are added to a hash table (line 1). Then, k random walks are performed starting from n. After every step in the random walk, SDD checks if the state reached, denoted c, is in $\pi(n)$ (line 6). If c is not in $\pi(n)$, then the random walk continues to perform another step, unless the cost bound $g(n)$ is reached. If c is in $\pi(n)$, then the alternative path found to c by the random walk is compared against the original path to c in $\pi(n)$. If the alternative path is smaller (according to $\mathcal{O}$), then n is declared a duplicate and SDD returns true (line 8). If after using k or fewer random walks no better path to n^* or any other state in $\pi(n)$ from n is found, then n is assumed to be a canonical node and SDD returns false (line 10).

To demonstrate how SSDD and SDD work, consider the following example.

Example 1. *Consider the graph and the tree shown in Figure 1. Here all operators have unit cost, the type system accounts only for the depth of the node in the tree (i.e., nodes at the same depth have the same type), and the cost-lexicographic order $\mathcal{O}$ uses the trivial operator ordering $op_1 < op_2$. Both n_4 and n_5 in the tree represent state s_4 in the* USG*. According to $\mathcal{O}$, n_4 is a duplicate as the paths $\pi(n_4) = \{op_2, op_1\}$ and $\pi(n_5) = \{op_1, op_2\}$ have the same cost, but the former is lexicographically larger.*

Assume that in a given probe SS samples the left branch of the search tree (i.e., n_2 and n_3 have the same type and SS randomly chooses to expand n_2). SS accounts for n_4 in its prediction because it does not know that n_4 is a duplicate. When facing the same scenario, before expanding n_4, SSDD applies SDD to search in the subtree rooted at n_4 to check if n_4 is a duplicate. During this search SDD encounters the cycle $\pi(n_8)$ where both n_1 and n_8 represent state s_1. Cycle $\pi(n_8)$ contains two paths in the graph from s_1 to s_4: $\pi(n_4) = \{op_2, op_1\}$ and $\pi' = \pi(n_5) = \{op_1, op_2\}$. SDD then finds n_4 to be a duplicate because the path π' SDD encountered during its sampling is cost-lexicographically smaller than $\pi(n_4)$. Thus, SSDD correctly prunes n_4.

Theoretical Guarantee For sufficiently large k, SSDD's estimation eventually converges to the correct size of the graph. This is because SSDD correctly identifies and prunes duplicate nodes in the search tree and the prediction converges to the size of the tree without the duplicates, which is the size of the graph.

Lemma 2. *Let $P(n, k)$ be the probability that SDD correctly identifies a node n as a duplicate or a canonical node using k or less random walks. We have that $\lim_{k \to \infty} P(n, k) = 1$.*

Proof. A given node n is either canonical or a duplicate. If n is canonical, then there is no smaller path (according to $\mathcal{O}$) from n^* to $state(n)$ than $\pi(n)$ and thus SDD will always return that n is canonical. If n is a duplicate, then there exists a path which is smaller than $\pi(n)$ in the subtree bounded by $g(n)$ rooted at $state(n)$. Thus, there exists a non-zero probability of finding this path by one of SDD's random walks. Therefore, the probability of at least one of these k random walks finding this alternative path, proving n to be a duplicate, converges to one as $k \to \infty$. □

Let $\hat{\varphi}_k^{(m)}$ be the result of the m^{th} SSDD probe, where k is the parameter given to SDD.

Theorem 2. *For a given partially-ordered type system T and a set of m SSDD independent probes $\hat{\varphi}_k^{(1)}(n^*, T), \cdots, \hat{\varphi}_k^{(m)}(n^*, T)$ from a tree $S(n^*)$, the probability of $\frac{1}{m}\sum_{j=1}^{m} \hat{\varphi}_k^{(j)}(n^*, T)$ converging to the size of $S(n^*)$'s underlying graph approaches one as both $k \to \infty$ and $m \to \infty$.*

Proof. SSDD detects and prunes duplicates in the *UST* during sampling with probability one as $k \to \infty$ (Lemma 2). From Theorem 1 we have that $\frac{1}{m}\sum_{j=1}^{m} \hat{\varphi}_k^{(j)}(n^*, T)$ converges to the size of the tree without the duplicates, which is the size of the graph, as $m \to \infty$. □

Applications of SSDD

SSDD is a general framework for estimating state-space properties. Next, we describe two applications of SSDD to predict concrete properties.

A* Prediction

We use SSDD to estimate the number of states expanded by A* using a consistent heuristic for a given f-value. This is the number of states s expanded by A* with $f(s) \leq x$, for some x. We call this set of states the f-layer of x. Korf et al. (2001) performed similar experiments to evaluate the number of nodes expanded by IDA*.

A* has the following properties when using a consistent heuristic: (i) it never expands a state more than once; (ii) it expands all states with f-value of x before expanding states with f-value of y for $x < y$. Property (i) emphasizes the importance of detecting duplicates while sampling the *UST* for estimating the number A* expansions with f-value of x. Property (ii) allows one to account only for states with f-value no greater than x when estimating the size of the f-layer of x. SSDD is already equipped with a system for detecting duplicate nodes. However, we have to make a modification in SSDD to accommodate (ii). Namely, SSDD only expands node n if $f(n) \leq x$.

We do not expect SSDD to produce good predictions of the number of nodes expanded by A* when using an inconsistent heuristic. This is because A* might re-expand the same state multiple times when using inconsistent heuristics (Martelli 1977; Felner et al. 2011) and SSDD assumes that each state in the state-space is expanded at most once.

State-Space Radius Prediction

SSDD can also be used to estimate the *radius* of a state s. The radius of s is the maximum shortest distance of any state from s (Korf 2008). We use SSDD to predict the state space radius from s in two different ways, which we call the MAX (maximum) scheme and the AVG (average) scheme.

The MAX Scheme In the MAX scheme we run m probes of SSDD starting from s, and, in contrast with the A* predictions, we do not bound the paths expanded during sampling (for A* predictions we bound the paths by the f-value). Every SSDD probe finishes when the structure $A[i]$ (line 4 in Algorithm 2) is empty, which happens when all expanded paths reach a duplicate or a leaf node. Let $r_1, \cdots, r_m$ be the largest g-value observed in Algorithm 2 in each of the m probes from s. The MAX estimate of the radius is given by $r_{max} = \max(r_1, \cdots, r_m)$.

The MAX scheme has the asymptotic guarantee of producing perfect estimates of the state-space radius as the number of probes m and the number of random walks k grow large. For very large k SSDD correctly detects duplicates during sampling (Lemma 2). As the number of probes m grows large, the probability of one of the probes reaching the states with the maximum shortest distance from s approaches one. A practical implication of this is that one will tend to get more accurate estimates by simultaneously increasing the values of m and k.

The AVG Scheme When applied to radius prediction, SSDD has two sources of error. In the first type of error, the longest shortest path may not be found by a given probe, resulting in an underestimate. In the second type of error, a duplicate node may not be identified as such, which might result in an overestimate. By taking the maximum value of multiple probes the MAX scheme tries to address the errors of the first type, but it ignores errors of the second type.

The AVG scheme attempts to remedy this by averaging multiple runs of the MAX scheme. Namely, for given values of m and k, we execute the MAX scheme d times producing estimates $r_{max,1}, \cdots, r_{max,d}$. Then, the AVG estimate of the radius is given by $\frac{1}{d}(r_{max,1} + r_{max,2} + \cdots + r_{max,d})$. Clearly, the AVG scheme reduces to the MAX scheme when $d = 1$. The reasoning behind the AVG scheme is that it allows SSDD errors of the first and second types to cancel out. Intuitively, due to the cancellation of under and overestimates, the AVG scheme will tend to produce more accurate predictions than the MAX scheme.

Because the MAX scheme has the asymptotic guarantee of producing perfect estimates, the AVG scheme also has the same guarantee when the values of m and k grow large. We note, however, that such guarantee of perfect estimates of the AVG scheme does not relate to the increase of the d-value.

Empirical Results

We first present experiments on estimating the number of A* expansions, and then present experiments on estimating the state-space radius.

Experiments on Estimating A* Expansions

We test SSDD in four domains: (17,4)-Topspin, 12-disks 4-pegs Towers of Hanoi, 15-Pancake, and 20-Gripper. These domains were chosen to ensure experimentation in domains both with and without a significant number of duplicates.

For comparison, we also experiment with SS. We implemented SSDD and SS in PSVN (Hernadvolgyi and Holte 1999). Pattern databases (PDBs) (Culberson and Schaeffer 1998) are used to generate consistent heuristics. The PDBs are produced by using the standard PSVN system for creating abstractions of the original state spaces. We chose the abstractions used to create our PDBs arbitrarily. In the type system we use in this experiment nodes n_1 and n_2 have the same type if $h(n_1) = h(n_2)$ (Zahavi et al. 2010).

We run experiments on a set of f-layer *prediction tasks*. An f-layer prediction task is a pair (s, x), where s is a start state and x is an f-layer A* searches starting from s. The prediction tasks are collected by having A* solving 1,000 different start states in each domain with a memory limit of 8GB. We use two error measures in this experiment: the *signed error* and the *absolute error*. The signed error is computed as $\frac{PRED}{SEARCH}$, where $PRED$ is the total number of predicted expansions in all tasks, and $SEARCH$ is the total number of actual expansions in all tasks. The signed error measures a system's accuracy in a set of tasks. A perfect score according to this error measure is 1.0. The absolute error is computed as $\frac{|pred-search|}{search}$, where $pred$ is the number of predicted expansions and $search$ is the number

(17,4)-Topspin														
SS					**SSDD** ($m = 1$)					**Parallel SSDD** ($m = 100$)				
p	mean.	median	sign.	%	k	mean.	median	sign.	%	k	mean.	median	sign.	%
3,000	873.97	1,329.21	1,065.07	8.55	3,000	2.62	0.95	2.71	1.76	3,000	1.52	1.03	2.58	3.68
5,000	873.58	1,406.36	1,065.54	14.07	7,000	2.72	0.96	2.35	3.72	7,000	1.00	0.50	1.97	8.08
6,000	873.75	1,461.14	1,065.42	17.18	10,000	2.07	0.96	1.89	5.15	10,000	0.84	0.41	1.77	11.39
7,000	873.61	1,398.75	1,065.4	19.68	11,000	1.90	0.95	1.59	5.59	11,000	0.86	0.44	1.77	12.45
12-disks 4-pegs Towers of Hanoi														
SS					**SSDD** ($m = 1$)					**Parallel SSDD** ($m = 100$)				
p	mean.	median	sign.	%	k	mean.	median	sign.	%	k	mean.	median	sign.	%
300	4.00e+36	1.13e+40	8.37e+36	15.50	3,000	6,146.39	0.99	7,817.65	4.32	3,000	2.25	0.95	1.54	6.03
400	4.05e+36	1.16e+40	8.48e+36	20.65	7,000	185.45	0.99	234.70	9.01	7,000	1.07	0.97	0.26	12.35
500	3.83e+36	9.61e+39	8.03e+36	25.76	10,000	98.63	0.99	65.74	12.55	10,000	1.57	0.98	0.64	17.28
1,000	3.97e+36	1.08e+40	8.31e+36	51.89	11,000	52.91	0.99	41.55	13.74	11,000	1.74	0.98	0.80	18.59
15-Pancake														
SS					**SSDD** ($m = 1$)					**Parallel SSDD** ($m = 100$)				
p	mean.	median	sign.	%	k	mean.	median	sign.	%	k	mean.	median	sign.	%
3,000	7.74	7.80	10.27	12.09	3,000	7.92	2.94	12.36	4.70	3,000	6.07	5.44	8.35	4.89
5,000	7.76	7.54	10.27	19.79	7,000	6.18	2.35	8.63	10.80	7,000	5.41	4.90	7.55	11.30
6,000	7.75	7.64	10.29	23.22	10,000	5.22	2.20	6.80	15.50	10,000	5.31	4.73	7.33	16.12
7,000	7.79	7.43	10.34	23.09	11,000	5.33	2.02	7.09	16.98	11,000	5.05	4.54	7.03	17.75
20-Gripper														
SS					**SSDD** ($m = 1$)					**Parallel SSDD** ($m = 100$)				
p	mean.	median	sign.	%	k	mean.	median	sign.	%	k	mean.	median	sign.	%
3,000	4.88e+12	5.50e+12	1.62e+13	12.80	3,000	654,236.54	1.00	672,015.38	0.86	3,000	308,132.55	93,446.62	315,641.81	2.17
5,000	4.91e+12	5.37e+12	1.63e+13	21.20	5,000	184,995.55	0.99	199,937.87	1.08	5,000	71,951.11	13,132.61	74,350.97	3.09
7,000	5.06e+12	5.64e+12	1.68e+13	29.58	100,000	1,036.63	0.99	1,168.51	8.06	100,000	1.53	0.90	1.11	20.33
8,000	4.93e+12	5.26e+12	1.63e+13	33.76	140,000	6.95	0.99	5.80	10.35	140,000	1.24	0.93	0.64	26.98

Table 1: SS, SSDD ($m = 1$), and Parallel SSDD ($m = 100$) prediction errors for the number of A* expansions for different number of SS probes p, and SSDD random walks k. % shows the average percentage of the A* runtime covered by the prediction algorithm.

of actual expansions in a specific task. The absolute error measures a system's accuracy in individual tasks. A perfect score according to this measure is 0.0. We report the mean and median values of the absolute error. In addition to the error measures, we show the average percentage of the A* search time covered by the prediction algorithm (denoted as %). For instance, a %-value of 30 means that the prediction was produced in 30% of the A* running time.

Comparison of SSDD with SS The prediction results for SS and SSDD are shown in Table 1 (the results under "Parallel SSDD" will be discussed later and should be ignored for now). The columns "mean", "median", "sign.", and "%" correspond to the mean absolute error, median absolute error, signed error, and percentage of the A* running time covered by the prediction.

SSDD is not able to correctly detect duplicate nodes if using a small value of k. In this case, increasing the number of probes m alone will not improve its accuracy. In this experiment we fix m to 1 to allow larger values of k while still producing predictions quickly (relatively to A*). The SS value of p and the SSDD value of k were chosen so we would have results with different %-values. We allow SS more computation time (larger %-values) in each row of Table 1, thereby giving it some advantage.

As shown in Table 1, SSDD tends to be orders of magnitude more accurate than SS in almost all domains and k values. For instance, on Gripper, SSDD ($k = 140,000$) produces predictions with mean absolute error of 6.95 in 10.35% of the A* running time, while SS ($p = 8,000$) produces predictions with mean absolute error of 4.93×10^{12} in 33.76% of the A* running time. The only domain where SS have reasonable performance compared to SSDD is the 15-pancake. This is because the *UST*s in this domain have fewer duplicates. For all other domains, even when increasing the number of probes p, SS does not produce better predictions. This is reasonable as SS's estimate converges to the size of the search tree and not to the size of the search graph. By contrast, if allowed more computation time by setting larger values of k, SSDD tends to produce more accurate predictions since it has better chances of detecting duplicates. The only domain in which we do not see SSDD improving its prediction by increasing the value of k is the 15-pancake. We conjecture that on the 15-pancake the cycles that prove the duplicate nodes to be duplicates are hard to be found by the SDD random walks. The 15-pancake's relatively large branching factor (=14) could be one explanation for SDD not being able to accurately detect the duplicates. Intuitively, it is more likely that SDD's random walks will find cycles in a *UST* with small branching factor than cycles in a *UST* with large branching factor.

Parallelizing SSDD We now explain how we improve our system's prediction accuracy by taking advantage of the fact that SSDD is easily parallelizable.

The SSDD median absolute error tends to be much smaller than its mean absolute error (see Table 1). This is because in very few cases SSDD fails to detect duplicates and, as a result, it samples a larger portion of the state space thus producing substantial overestimates of the actual number of states expanded. These cases overly skew the SSDD mean absolute error. We note that, when SSDD fails to detect duplicates, its running time tends to be larger than normal, which is explained by the fact that it samples a larger portion of the state space. With the goal of producing more accurate predictions by avoiding the cases in which SSDD produces substantial overestimates, we run multiple SSDD probes in parallel, and we disregard the slowest ones. Namely, we run m SSDD independent probes in parallel and we account for the probing result of only the first 95% probes to finish. As explained, the slowest probes tend to produce large overestimates. (An orthogonal way of parallelizing SSDD is to run the SDD's random walks in parallel, a direction we have not explored in this paper.) The value of 95% was chosen experimentally; values between 95% and 97% yielded good results. We call this SSDD variant Parallel SSDD.[2]

The results for Parallel SSDD are also shown in Table 1 (on the righthand side). For Parallel SSDD we run 100 probes ($m = 100$) in parallel and compute the prediction result by averaging the probing result of the first 95 probes to finish. We observe that Parallel SSDD tends to produce more accurate predictions than SSDD at the cost of increasing the prediction running time. The improvement in accuracy mainly occurs because we avoid the cases in which SSDD substantially overestimates the actual number of nodes expanded. The increase in running time occurs because it is likely that the slowest of the 95 probes is slower than the single probe used with SSDD. Once again the 15-pancake is the exception: we observe a consistent increase in the median absolute error when using Parallel SSDD. We believe this is due to the fact that SDD is not able to correctly detect duplicates on the 15-pancake. This result suggests that increasing the number of probes m might actually degrade the prediction accuracy when SDD is not able to correctly detect the duplicates.

Experiments on Estimating the State-Space Radius

We use the MAX and AVG schemes to estimate the radius of the $3\times3\times3$ Rubik's Cube and the (4×4) Sliding-Tile Puzzle (15-puzzle) from their goal states. The Rubik's Cube's state space has approximately 4.3×10^{19} states and the 15-puzzle approximately 10^{13}. Although both problems have fairly large state spaces, their radius from the goal state is known: 20 for the Rubik's Cube (Rokicki et al. 2013) and 80 for the 15-puzzle (Korf 2008). Thus, when using these two domains we can measure the prediction errors exactly. SS cannot be used in this experiment because the problems' *UST*s are not finite and an SS probe would never finish.

The task in this set of experiments is to predict the radius of a specific state — the goal state. We experiment with m values of 5, 10, 20, 30, and 40, and k values of 50, 1000, 2000, 3000, 4000, and 5000. Results for both schemes are averages of 10 runs. For the AVG scheme we set d to 20, i.e., 20 runs of the MAX scheme. In Table 2, in addition to the average and standard deviation of the predicted values, we present the running time in minutes for both schemes. The type system we use in this experiment partitions the nodes according to their level in the *UST*: two nodes have the same type if they are at the same level of the *UST*.

Discussion Consider first the results of the MAX scheme at the top of Table 2. We observe that increasing the m-value does not necessarily improve the prediction accuracy when k is small (e.g., k equals 50). This is because SSDD does not correctly detect duplicates when k is small. However, we improve SSDD's accuracy by increasing m and k simultaneously. For $m = 40$ and $k = 5,000$ SSDD produces an average estimate of the Rubik's Cube radius of 18.7 ± 5.9 in 0.1 minute (6 seconds). Similar trend is observed on the 15-puzzle: as we simultaneously increase the values of m and k, SSDD tends to produce better estimates. For $m = 40$ and $k = 5,000$ SSDD produces an average estimate of the 15-puzzle radius of 86.1 ± 18.6 in 5 minutes.

The results for SSDD using the AVG scheme are shown at the bottom of Table 2. In general we observe the same trend observed in the MAX scheme results: predictions tend to be more accurate for larger values of m and k. We notice, however, that the AVG scheme can produce estimates with lower variance than the MAX scheme. This is because the AVG scheme averages the results of multiple runs of the MAX scheme and under and overestimations may cancel each other out, reducing the variance and potentially yielding more accurate estimates (albeit requiring more time).

One way of computing the radius exactly is by running a complete uniform-cost search from the goal. Korf's disk-based search took 28 days and 8 hours to enumerate the 15-puzzle states (Korf 2008). One of the byproducts of such computation is the 15-puzzle radius. Although not perfect, SSDD using the AVG scheme was able to quickly produce accurate estimates of both the Rubik's Cube and the 15-puzzle radii.

Related Work

Thayer et al. (2012) presented methods that also rely on the accuracy of the heuristic to estimate search progress of best-first search variants. By contrast, SSDD is more general in the sense that it does not depend on a heuristic and is able to measure other state-space properties such as state space radius and number of nodes expanded in a single f-layer. Breyer and Korf (2008) predicted the number of nodes expanded by A* for the 15-puzzle, but they used the information learned from a complete breadth-first search in the state space; see also Breyer's PhD thesis (Breyer 2010). Hernadvolgyi and Holte (2004) also made estimations for the number of nodes expanded by A*, but they ignored the transposition detection the algorithm does. Similarly to SS, their method will produce overestimates of the actual number of A* expansions in problem domains with a large number of duplicates.

SSDD could also be used to enhance other prediction

[2]In our experiments we ran a sequential implementation of SSDD and from different probing results we emulated Parallel SSDD.

MAX Scheme												
3x3x3 Rubik's Cube Radius = 20												
m / k	50		1,000		2,000		3,000		4,000		5,000	
	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time
5	14.8 ± 7.5	0.0	8.7 ± 4.6	0.0	10.0 ± 5.0	0.0	9.0 ± 4.1	0.0	8.2 ± 3.7	0.0	7.8 ± 3.8	0.0
10	18.1 ± 7.5	0.0	10.7 ± 4.3	0.0	11.5 ± 5.1	0.0	11.2 ± 4.6	0.0	11.2 ± 5.1	0.0	11.5 ± 4.8	0.0
20	23.1 ± 8.2	0.0	13.7 ± 4.5	0.0	15.3 ± 5.0	0.0	14.3 ± 4.8	0.0	14.9 ± 5.0	0.0	15.3 ± 5.0	0.0
30	26.3 ± 7.9	0.0	16.4 ± 4.9	0.0	16.8 ± 5.4	0.0	17.4 ± 5.3	0.0	17.4 ± 5.6	0.1	17.2 ± 5.6	0.1
40	28.3 ± 8.3	0.0	18.1 ± 5.3	0.0	18.5 ± 4.9	0.0	18.4 ± 5.2	0.1	18.5 ± 5.2	0.1	18.7 ± 5.9	0.1
4x4 Sliding-Tile Puzzle Radius = 80												
m / k	50		1,000		2,000		3,000		4,000		5,000	
	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time
5	132.9 ± 67.4	0.1	66.8 ± 32.9	0.3	57.1 ± 26.0	0.4	54.4 ± 23.9	0.5	52.9 ± 22.5	0.6	52.6 ± 22.4	0.7
10	165.4 ± 71.9	0.2	82.5 ± 34.6	0.6	70.2 ± 26.2	0.7	65.6 ± 23.8	0.9	68.8 ± 25.6	1.3	63.8 ± 24.4	1.3
20	203.8 ± 68.7	0.4	95.5 ± 31.4	1.0	87.9 ± 27.4	1.5	78.8 ± 23.1	1.8	77.5 ± 22.9	2.2	75.5 ± 24.1	2.7
30	228.6 ± 71.2	0.6	107.9 ± 33.4	1.6	97.7 ± 28.1	2.4	91.9 ± 27.1	3.0	86.6 ± 24.1	3.5	82.9 ± 21.6	3.9
40	243.6 ± 74.2	0.9	112.6 ± 27.5	2.0	101.3 ± 27.1	3.0	93.3 ± 24.3	3.7	88.1 ± 21.2	4.3	86.1 ± 18.6	5.0

AVG Scheme												
3×3×3 Rubik's Cube Radius = 20												
m / k	50		1,000		2,000		3,000		4,000		5,000	
	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time
5	14.8 ± 2.1	0.0	8.7 ± 1.0	0.0	10.0 ± 1.2	0.0	9.1 ± 0.8	0.0	8.0 ± 0.6	0.0	7.7 ± 0.5	0.0
10	18.1 ± 2.4	0.0	10.7 ± 1.1	0.0	11.6 ± 1.1	0.0	11.2 ± 0.7	0.1	11.2 ± 1.1	0.2	11.5 ± 1.4	0.2
20	23.1 ± 2.4	0.0	13.6 ± 0.9	0.0	15.2 ± 1.0	0.2	14.3 ± 0.9	0.3	14.8 ± 1.1	0.6	15.5 ± 0.9	0.8
30	26.3 ± 2.4	0.0	16.5 ± 1.0	0.1	16.9 ± 1.2	0.4	17.4 ± 1.2	0.8	17.4 ± 1.4	1.1	17.4 ± 1.3	1.3
40	28.3 ± 2.3	0.0	18.1 ± 1.1	0.2	18.5 ± 0.7	0.7	18.5 ± 1.2	1.0	18.6 ± 1.3	1.5	18.8 ± 1.1	1.9
4×4 Sliding-Tile Puzzle Radius = 80												
m / k	50		1,000		2,000		3,000		4,000		5,000	
	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time	prediction	time
5	132.9 ± 15.8	2.0	66.8 ± 7.7	5.7	57.1 ± 5.3	7.1	54.4 ± 6.3	9.3	52.9 ± 5.1	11.3	52.6 ± 5.8	13.8
10	165.4 ± 16.6	4.0	82.5 ± 6.1	11.4	70.2 ± 6.2	14.0	65.6 ± 6.6	17.5	68.8 ± 6.1	25.9	63.8 ± 4.7	26.5
20	203.8 ± 12.9	8.2	95.5 ± 6.7	19.9	87.9 ± 6.5	30.8	78.9 ± 6.1	36.2	77.7 ± 5.2	45.2	75.7 ± 5.7	53.9
30	228.6 ± 15.8	12.9	108.0 ± 6.7	31.1	97.9 ± 8.3	48.0	92.2 ± 5.4	60.8	86.4 ± 4.4	68.7	82.6 ± 5.0	76.8
40	243.6 ± 15.2	17.4	112.6 ± 5.7	39.7	101.2 ± 6.5	59.7	93.4 ± 5.8	73.5	88.2 ± 4.9	86.5	86.1 ± 5.5	99.8

Table 2: The MAX and AVG average results over 10 independent runs of the schemes. We experiment with $m = \{5, 10, 20, 30, 40\}$ and $k = \{50, 1000, 2000, 3000, 4000, 5000\}$ for both schemes and use $d = 20$ for the AVG scheme. We present the average and standard deviation of the predicted values, and the average running time in minutes for both schemes. The actual radius of the 4x4 Sliding-Tile Puzzle is 80 and of the 3×3×3 Rubik's Cube is 20.

methods such as KRE (Korf, Reid, and Edelkamp 2001) and CDP (Zahavi et al. 2010). For example, one could use SSDD to estimate the size of the search graph without a heuristic function (i.e., A* with $h = 0$), and then apply the KRE and CDP formulas to estimate the number of A* expansions. In this paper we chose to build on SS because Lelis et al. (2013) showed that SS is the current state of the art for tree size prediction. Nevertheless, we intent to pursue in the future the application of SSDD to KRE and CDP.

Burch and Holte (2011; 2012) presented a move pruning system for preventing duplicates from being expanded during search. While it is plausible to use this approach for detecting duplicates in SSDD and SS, preliminary results on the 4-pegs Towers of Hanoi showed that the results were not substantially better. We conjecture that the duplicates detected by move pruning are not enough for producing accurate estimates of the *ESG* size.

Conclusion

We presented Stratified Sampling with Duplicate Detection (SSDD), an algorithm for estimating properties of state spaces represented by undirected graphs. We tested SSDD on two tasks: (i) prediction of the number of states expanded by A* when using a consistent heuristic in a given f-layer, and (ii) prediction of the state space radius from the goal state. SSDD has the asymptotic guarantee of producing perfect estimates in both tasks. In our empirical evaluation of task (i) a parallel version of SSDD produced accurate predictions in all four domains tested, being in most cases several orders of magnitude more accurate than SS and consistently more accurate than SSDD itself; in task (ii) SSDD was able to quickly produce accurate predictions of the state space radii of the (4×4) Sliding-Tile Puzzle and the 3×3×3 Rubik's Cube.

References

Breyer, T., and Korf, R. 2008. Recent in analyzing the performance of heuristic search. In *In Proceedings of the First International Workshop on Search in Artificial Intelligence and Robotics*, 24–31.

Breyer, T. M. 2010. *Predicting and Improving the Performance of Heuristic Search*. Ph.D. Dissertation, Los Angeles, CA, USA. AAI3446854.

Burch, N., and Holte, R. C. 2011. Automatic move pruning in general single-player games. In *Proceedings of the Symposium on Combinatorial Search*, 31–38. AAAI Press.

Burch, N., and Holte, R. C. 2012. Automatic move pruning revisited. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*. AAAI Press.

Burns, E., and Ruml, W. 2012. Iterative-deepening search with on-line tree size prediction. In *Proceedings of the International Conference on Learning and Intelligent Optimization*, 1–15.

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

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

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence* 175(9–10):1570–1603.

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

Hernadvolgyi, I. T., and Holte, R. 1999. Psvn: A vector representation for production systems. Technical Report 99-04, University of Ottawa Computer Science.

Hernádvölgyi, I. T., and Holte, R. C. 2004. Steps towards the automatic creation of search heuristics. Technical Report TR-04-02, University of Alberta.

Kilby, P.; Slaney, J. K.; Thiébaux, S.; and Walsh, T. 2006. Estimating search tree size. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1014–1019. AAAI Press.

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

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence* 129(1-2):199–218.

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

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *Journal of the ACM* 55(6):26:1–26:40.

Lelis, L. H. S.; Otten, L.; and Dechter, R. 2013. Predicting the size of depth-first branch and bound search trees. In *International Joint Conference on Artificial Intelligence*, 594–600.

Lelis, L. H. S.; Otten, L.; and Dechter, R. 2014. Memory-efficient tree size prediction for depth-first search in graphical models. In *International Conference on Principles and Practice of Constraint Programming*, to appear.

Lelis, L. H. S.; Zilles, S.; and Holte, R. C. 2013. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence* 53–76.

Lelis, L. H. S. 2013. Active stratified sampling with clustering-based type systems for predicting the search tree size of problems with real-valued heuristics. In *Proceedings of the Symposium on Combinatorial Search*, 123–131. AAAI Press.

Martelli, A. 1977. On the complexity of admissible search algorithms. *Artificial Intelligence* 8(1):1–13.

Rokicki, T.; Kociemba, H.; Davidson, M.; and Dethridge, J. 2013. The diameter of the rubik's cube group is twenty. *SIAM Journal on Discrete Mathematics* 27(2):1082–1105.

Thayer, J. T.; Stern, R.; and Lelis, L. H. S. 2012. Are we there yet? – estimating search progress. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*. AAAI Press.

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