Published in *Artificial Intelligence* **129** (2001), no. 1–2, pp. 199–218.

Time Complexity of Iterative-Deepening-A*

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Michael Reid
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003
mreid@math.umass.edu

Stefan Edelkamp
Institute für Informatik
Georges-Köhler-Allee D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

March 2, 2001

Abstract

We analyze the time complexity of iterative-deepening-A* (IDA*). We first show how to calculate the exact number of nodes at a given depth of a regular search tree, and the asymptotic brute-force branching factor. We then use this result to analyze IDA* with a consistent, admissible heuristic function. Previous analyses relied on an abstract analytic model, and characterized the heuristic function in terms of its accuracy, but do not apply to any concrete problems. In contrast, our analysis allows us to accurately predict the performance of IDA* on actual problems such as the sliding-tile puzzles and Rubik's Cube. The heuristic function is characterized by the distribution of heuristic values over the problem space. Contrary to conventional wisdom, our analysis shows that the asymptotic heuristic branching factor is the same as the brute-force branching factor, and that the effect of a heuristic function is to reduce the effective depth of search, rather than the effective branching factor, relative to a brute-force search.

1 Introduction and Overview

Our goal is to predict the running time of iterative-deepening-A* (IDA*)[5], a linear-space version of the A* algorithm[4]. Both these algorithms rely on a heuristic evaluation function $h(n)$ that estimates the cost of reaching a goal from node n. If the $h(n)$ is *admissible*, or never overestimates actual cost from node n to a goal, then both algorithms return optimal solutions.

The running time of IDA* is usually proportional to the number of nodes expanded. This depends on the cost of an optimal solution, the number of nodes in the brute-force search tree, and the heuristic function. In Section 2, we show how to compute the size of a brute-force search tree, and its asymptotic branching factor. In Section 3, we use this result to predict the number of nodes expanded by IDA* using a consistent heuristic function. The key to this analysis is characterizing the heuristic function.

Previous work on this problem characterized the heuristic by its accuracy as an estimate of actual solution cost. The accuracy of a heuristic is very difficult to obtain, and the corresponding asymptotic results, based on an abstract model, don't predict performance on any concrete problems. In contrast, we characterize a heuristic by its distribution of values, a characterization that is easy to determine. As a result, we can predict the performance of IDA* on the sliding-tile puzzles and Rubik's Cube to within 1% of experimental results. In contrast to previous work, our analysis shows that the asymptotic heuristic branching factor is the same as the brute-force branching factor, and that the effect of a heuristic function is to reduce the effective depth of search, rather than the effective branching factor, relative to a brute-force search.

Most of this work originally appeared in two AAAI-98 papers, one on the brute-force branching factor [2], and the other on the analysis of IDA* [7]. We begin with brute-force search trees.

2 Branching Factor of Regular Search Trees

2.1 Graph vs. Tree-Structured Problem Spaces

Most problem spaces are graphs with cycles. Given a root node of any graph, however, it can be expanded into a tree. For example, Figure 1 shows a search graph, and the top part of its tree expansion, rooted at node A. In a tree expansion of a graph, each distinct path to a node of the graph generates a different node of the tree. The tree expansion of a graph can be much larger than the original graph, and in fact can be infinite even for a finite graph.

In this paper, we focus on problem-space trees. The reason is that IDA* uses depth-first search to save memory, and hence cannot detect most duplicate nodes. Thus, it potentially explores every path to a given node, and searches the tree-expansion of the problem-space graph.

We can characterize the size of a brute-force search tree by its asymptotic branching factor. The branching factor of a node is the number of children it has. In most trees, however, different nodes have different numbers of children. In that case, we define the *asymptotic branching factor* as the number of nodes at a given depth, divided by the number of nodes at the next shallower depth, in the limit as the depth goes to infinity.

We present examples of problem-space trees, and compute their asymptotic branching factors. We formalize the problem as the solution of a set of simultaneous equations. We present both analytic and numerical techniques for computing the number of nodes at a given depth, and determining the asymptotic branching factor. We give the branching factors of Rubik's Cube and sliding-tile puzzles from the Five Puzzle to the Ninety-Nine Puzzle.

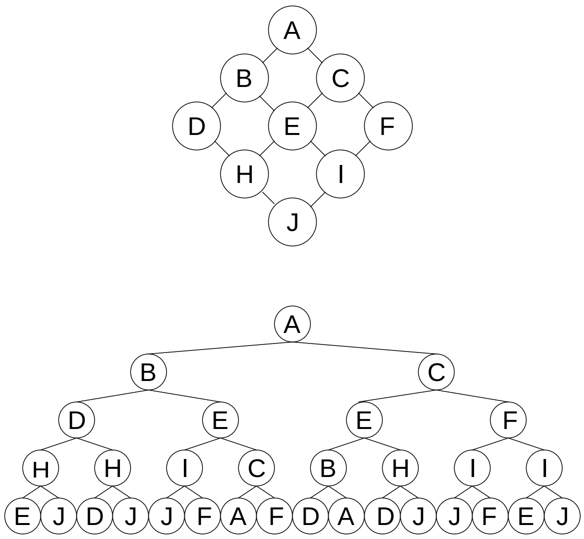

Figure 1: Graph and Part of its Tree Expansion

2.2 Example: Rubik's Cube

Consider Rubik's Cube, shown in Figure 2. We define any 90, 180, or 270 degree twist of a face as one move. Since there are six faces, this gives an initial branching factor of $6 \cdot 3 = 18$. We never twist the same face twice in a row, however, since the same result can be obtained with a single twist of that face. This reduces the branching factor to $5 \cdot 3 = 15$ after the first move.

Note that twists of opposite faces are independent of each other and hence commute. For example, twisting the left face followed by the right face gives the same result as twisting the right face followed by the left face. Thus, if two opposite faces are twisted consecutively, we require them to be twisted in a particular order, to eliminate the same state resulting from twisting them in the opposite order. For each pair of opposite faces, we arbitrarily label one a "first" face, and the other a "second" face. Thus, if Left, Up and Front were the first faces, then Right, Down, and Back would be the second faces. After a first face is twisted, there are three possible twists of each of the remaining five faces, for a branching factor of 15. After a second face is twisted, however, we can only twist four remaining faces, excluding the face just twisted and its corresponding first face, for a branching factor of 12. Thus, the asymptotic branching factor is between 12 and 15.

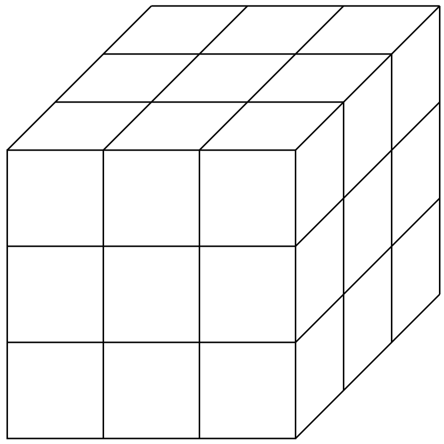

Figure 2: Rubik's Cube

The exact asymptotic branching factor depends on the relative fraction of nodes where the last move was a twist of a first face (type-1 nodes), or a twist of a second face (type-2 nodes). Define the *equilibrium fraction* of type-1 nodes as the number of type-1 nodes at a given depth, divided by the total number of nodes at that depth, in the limit of large depth. The equilibrium fraction is not 1/2, because a twist of a first face can be followed by a twist of any second face, but a twist of a second face cannot be followed immediately by a twist of the corresponding first face. To determine the asymptotic branching factor, we need the equilibrium fraction of type-1 nodes. The fraction of type-2 nodes is one minus the fraction of type-1 nodes.

Each type-1 node generates $2 \cdot 3 = 6$ type-1 nodes and $3 \cdot 3 = 9$ type-2 nodes as children, the difference being that you can't twist the same first face again. Each type-2 node generates $2 \cdot 3 = 6$ type-1 nodes and $2 \cdot 3 = 6$ type-2 nodes, since you can't twist the corresponding first face next, or the same second face again. Thus, the number of type-1 nodes at a given depth is 6 times the number of type-1 nodes at the previous depth, plus 6 times the number of type-2 nodes at the previous depth. The number of type-2 nodes at a given depth is 9 times the number of type-1 nodes at the previous depth, plus 6 times the number of type-2 nodes at the previous depth.

Let f_1 be the fraction of type-1 nodes, and $f_2 = 1 - f_1$ the fraction of type-2 nodes at a given depth. If n is the total number of nodes at that depth, then there will be nf_1 type-1 nodes and nf_2 type-2 nodes at that depth. In the limit of large depth, the fraction of type-1 nodes will converge to the equilibrium fraction, and remain constant. Thus, at large depth,

$$f_1 = \frac{\text{type 1 nodes at next level}}{\text{total nodes at next level}} = \frac{6nf_1 + 6nf_2}{6nf_1 + 6nf_2 + 9nf_1 + 6nf_2} =$$

$$\frac{6f_1 + 6f_2}{15f_1 + 12f_2} = \frac{6f_1 + 6(1 - f_1)}{15f_1 + 12(1 - f_1)} = \frac{6}{3f_1 + 12} = \frac{2}{f_1 + 4} = f_1$$

Cross multiplying gives us the quadratic equation $f_1^2 + 4f_1 = 2$, which has a positive root at $f_1 = \sqrt{6} - 2 \approx .44949$. This gives us an asymptotic branching factor of $15f_1 + 12(1 - f_1) = 3\sqrt{6} + 6 \approx 13.34847$.

2.3 A System of Simultaneous Equations

In general, this analysis produces a system of simultaneous equations. For example, consider the Five Puzzle, the 2×3 version of the well-known sliding-tile puzzles (see Figure 3A).

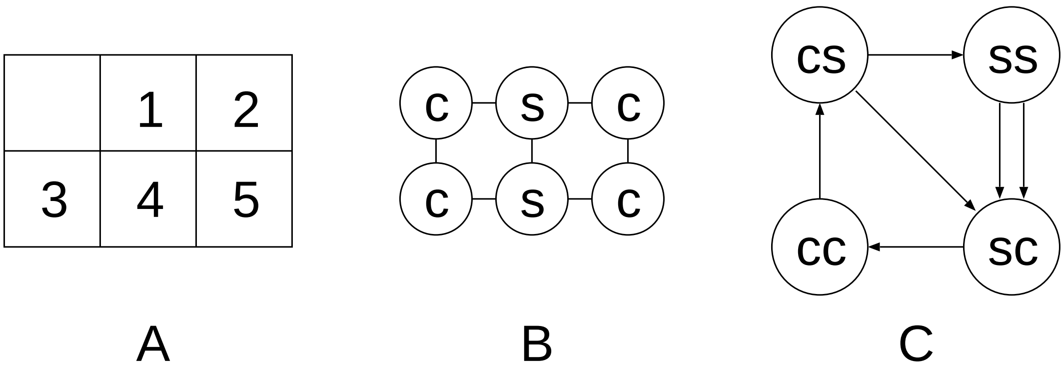

Figure 3: The Five Puzzle

In this problem, the branching factor of a node depends on the blank position. In Figure 3B, the positions are labelled s and c, representing side and corner positions, respectively. We don't generate the parent of a node as one of it's children, to avoid duplicate nodes representing the same state. This requires keeping track of both the current and previous blank positions. Let cs denote a node where the blank is currently in a side position, and the last blank position was a corner position. Define ss, sc and cc nodes analogously. Since cs and ss nodes have two children each, and sc and cc nodes have only one child each, we have to know the equilibrium fractions of these different types of nodes to determine the asymptotic branching factor. Figure 3C shows the different types of states, with arrows indicating the type of children they generate. For example, the double arrow from ss to sc indicates that each ss node generates two sc nodes at the next level.

Let $N(t,d)$ be the number of nodes of type t at depth d in the search tree. Then, we can write the following recurrence relations directly from the graph in figure 3C. For example, the last equation comes from the fact that there are two arrows from ss to sc, and one arrow from cs to sc.

$$\begin{aligned} N(cc,d+1) &= N(sc,d) \\ N(cs,d+1) &= N(cc,d) \\ N(ss,d+1) &= N(cs,d) \\ N(sc,d+1) &= 2N(ss,d)+N(cs,d) \end{aligned}$$

The initial conditions are that the first move either generates an ss node and two sc nodes, or a cs node and a cc node, depending on whether the blank starts in a side or corner position, respectively.

2.3.1 Numerical Solution

The simplest way is to compute the branching factor is to numerically compute the values of successive terms of these recurrences, until the relative frequencies of different state types converge. Let f_{cc}, f_{cs},f_{ss} and f_{sc} be the number of nodes of each type at a given depth, divided by the total number of nodes at that depth. After a hundred iterations, we get the equilibrium fractions $f_{cc} = .274854$, $f_{cs} = .203113$, $f_{ss} = .150097$, and $f_{sc} = .371936$. Since cs and ss states generate two children each, and the others generate one child each, the asymptotic branching factor is $f_{cc} + 2f_{cs} + 2f_{ss} + f_{sc} = 1.35321$. Alternatively, we can simply compute the ratio between the total nodes at two successive depths to get the branching factor. The running time of this algorithm is the product of the number of different types of states, e.g. four in this case, and the search depth. In contrast, searching the actual tree to depth 100 would generate over 10^{13} states in this case.

2.3.2 Analytical Solution

To compute the exact branching factor, we assume that the fractions eventually converge to constant values. This generates a set of equations, one from each recurrence. Let b represent the asymptotic branching factor. If we view f_{cc} as the number of cc nodes at depth d, for example, then the number of cc nodes at depth $d+1$ will be bf_{cc}. This allows us to rewrite the above recurrences as the following set of equations. The last one constrains the fractions to sum to one.

$$\begin{aligned} bf_{cc} &= f_{sc} \\ bf_{cs} &= f_{cc} \\ bf_{ss} &= f_{cs} \\ bf_{sc} &= 2f_{ss} + f_{cs} \\ 1 &= f_{cc} + f_{cs} + f_{ss} + f_{sc} \end{aligned}$$

Repeated substitution to eliminate variables reduces this system of five equations in five unknowns to the single equation, $b^4 - b - 2 = 0$, with a solution of $b \approx 1.35321$. In general, the degree of the polynomial will be the number of different types of states. The Fifteen Puzzle, for example, has three types of positions, and six types of states.

If we make the naive and incorrect assumption that each blank position is equally likely in the Five Puzzle, we get an incorrect branching factor of $(2 \cdot 2 + 1 \cdot 4)/6 = 1.33333$. Another natural but erroneous approach is to include the parent of a node as one of its children, compute the resulting branching factor, and then subtract one from the result to eliminate the inverse of the last move. This gives an incorrect branching factor of 1.4142 for the Five Puzzle. The error here is that eliminating the inverse of the last move changes the equilibrium fractions of the different types of states.

2.4 Results

We computed the asymptotic branching factors of square sliding-tile puzzles up to 10×10. Table 1 gives the even- and odd-depth branching factors for each puzzle. The last column is their geometric mean, or the square root of their product, which is the best estimate of the overall branching factor. Most of these values were computed by numerical iteration of the recurrence relations. As n goes to infinity, all the values converge to three, the branching factor of an infinite sliding-tile puzzle, since most positions have four neighbors, one of which was the previous blank position.

n	n^2-1	even depth	odd depth	mean
3	8	1.5	2	$\sqrt{3}$
4	15	2.1304	2.1304	2.1304
5	24	2.30278	2.43426	2.36761
6	35	2.51964	2.51964	2.51964
7	48	2.59927	2.64649	2.62277
8	63	2.69590	2.69590	2.69590
9	80	2.73922	2.76008	2.74963
10	99	2.79026	2.79026	2.79026

Table 1: The Asymptotic Branching Factor for the (n^2-1)-Puzzle

To see why the even and odd branching factors are different, color the positions of a puzzle in a checkerboard pattern, and note that the blank always moves between squares of different colors. If the sets of different-colored squares are equivalent to each other, as in the Five and Fifteen puzzles, there is one branching factor. If the sets of different-colored squares are different however, as in the Eight Puzzle, there will be different even and odd branching factors. In general, an $n \times m$ sliding-tile puzzle will have different branching factors if and only if both n and m are odd.

2.5 Generality of This Technique

In some problem spaces, every node has the same branching factor. In other spaces, every node may have a different branching factor, requiring exhaustive search to compute the average branching factor. The technique described above determines the size of a brute-force search tree in intermediate cases, where there are a small number of different types of states, whose generation follows a regular pattern. Computing the size of the brute-force search tree is the first step in determining the time complexity of IDA*, our next topic.

3 Time Complexity of IDA*

IDA* uses the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the sum of the edge costs from the initial state to node n, and $h(n)$ is an estimate of the cost of reaching a goal from node n. Each iteration is a depth-first search where a branch is pruned when it reaches a node whose total cost exceeds the cost threshold of that iteration. The cost threshold for the first iteration is the heuristic value of the initial state, and increases in each iteration to the lowest cost of all nodes pruned on the previous iteration. It continues until a goal node is found whose cost does not exceed the current cost threshold.

3.1 Previous Work

Most previous analyses of heuristic search focussed on A* [10, 3, 9], and used an abstract problem-space tree where every node has b children, every edge has unit cost, and there is a single goal node at depth d. The heuristic is characterized by its error in estimating actual solution cost. This model predicts that a heuristic with constant absolute error results in linear time complexity, while constant relative error results in exponential time complexity [10, 3].

There are several limitations of this model. The first is that it assumes there is only one path from the start to the goal state, whereas problem spaces contain multiple paths to each state. The second limitation is that in order to determine the accuracy of the heuristic on even a single state, we have to determine the optimal solution cost from that state, which is expensive to compute. Doing this for a significant number of states is impractical for large problems. Finally, the results are only asymptotic, and don't predict actual numbers of node generations. Because of these limitations, the previous work cannot be used to accurately predict the performance of A* or IDA* on any concrete problems with real heuristics. That requires a different approach.

3.2 Overview

We begin with the consistency property of heuristics, and the conditions for node expansion by A* or IDA*. Next, we characterize a heuristic by the distribution of heuristic values over the problem space. Our main result is a formula for the number of node expansions as a function of the heuristic distribution, the cost threshold of an iteration, and the number of nodes of each cost in a brute-force search. Finally, we compare our analytic predictions with experimental data on Rubik's Cube and the Eight and Fifteen Puzzles. One implication of our analysis is that the effect of a heuristic function is to decrease the effective depth of search, rather than the branching factor.

3.3 Consistent Heuristics

One property of the heuristic required by our analysis is that it be *consistent*. A heuristic function $h(n)$ is consistent if for any node n and any neighbor n', $h(n) \leq k(n, n') + h(n')$, where $k(n, n')$ is the cost of the edge from n to n' [9]. An equivalent definition of consistency is that for any pair of nodes n and m, $h(n) \leq k(n, m) + h(m)$, where $k(n, m)$ is the cost of an optimal path from n to m. Consistency is similar to the triangle inequality of metrics, and implies admissibility, but not vice versa. However, most naturally occurring admissible heuristic functions are consistent as well [9].

3.4 Conditions for Node Expansion

We measure the time complexity of IDA* by the number of node expansions. If a node can be expanded and its children evaluated in constant time, the asymptotic time complexity of IDA* is simply the number of node expansions. Otherwise, it's the product of the number of node expansions and the time to expand a node. Given a consistent heuristic function, both A* and IDA* must expand all nodes whose total cost, $f(n) = g(n) + h(n)$, is less than c, the cost of an optimal solution [9]. Some nodes with the optimal solution cost may be expanded as well, until a goal node is chosen for expansion, and the algorithms terminate. In other words, $f(n) < c$ is a sufficient condition for A* or IDA* to expand node n, and $f(n) \leq c$ is a necessary condition. For a worst-case analysis, we adopt the weaker necessary condition.

An easy way to understand the node expansion condition is that any search algorithm that guarantees optimal solutions must continue to expand every possible solution path, until its cost is guaranteed to exceed the cost of an optimal solution, lest it lead to a better solution.

On the final iteration of IDA*, the cost threshold will equal c, the cost of an optimal solution. In the worst case, IDA* will expand all nodes n whose cost $f(n) = g(n) + h(n) \leq c$. We will see below that this final iteration determines the overall asymptotic time complexity of IDA*.

3.5 Characterization of the Heuristic

Previous analyses characterized the heuristic function by its accuracy as an estimator of optimal costs. As explained above, this is difficult to determine for a real heuristic, since obtaining optimal solutions is extremely expensive. In contrast, we characterize a heuristic function by the distribution of heuristic values over the nodes in the problem space. In other words, we need to know the number of states with heuristic value 0, how many states have heuristic value 1, the number with heuristic value 2, etc. Equivalently, we can specify this distribution by a set of parameters $D(h)$, which is the fraction of total states of the problem whose heuristic value is less than or equal to h. We refer to this set of values as the *overall distribution* of the heuristic. $D(h)$ can also be defined as the probability that a state chosen randomly and uniformly from all states in the problem has heuristic value less than or equal to h. h can range from zero to infinity, but for all values of h greater than or equal to the maximum value of the heuristic, $D(h) = 1$.

Table 2 shows the overall distribution for the Manhattan distance heuristic on the Five Puzzle. Manhattan distance is computed by counting the number of grid units that each tile is displaced from its goal position, and summing these values for all tiles. The first column of Table 2 gives the heuristic value. The second column gives the number of states of the Five Puzzle with each heuristic value. The third column gives the total number of states with a given or smaller heuristic value, which is simply the cumulative sum of the values from the second column. The fourth column gives the overall heuristic distribution $D(h)$. These values are computed by dividing the value in the third column by 360, the total number of states in the problem space. The remaining columns will be explained below.

The overall distribution is easily obtained for any heuristic. For heuristics implemented by table-lookup, or *pattern databases* [1, 6, 8], the distribution can be determined exactly by scanning the table, as shown above. Alternatively, for a heuristic computed by a function, such as Manhattan distance on large sliding-tile puzzles, we can randomly sample the problem space to estimate the overall distribution to any desired degree of accuracy. For heuristics that are the maximum of several different heuristics, we can approximate the distribution of the combined heuristic from the distributions of the individual heuristics by assuming that the individual heuristic values are independent.

The distribution of a heuristic function is not a measure of its accuracy,

h	States	Sum	D(h)	Corner	Side	Csum	Ssum	P(h)
0	1	1	.002778	1	0	1	0	.002695
1	2	3	.008333	1	1	2	1	.011028
2	3	6	.016667	1	2	3	3	.016915
3	6	12	.033333	5	1	8	4	.033333
4	30	42	.116667	25	5	33	9	.115424
5	58	100	.277778	38	20	71	29	.276701
6	61	161	.447222	38	23	109	52	.446808
7	58	219	.608333	41	17	150	69	.607340
8	60	279	.775000	44	16	194	85	.775707
9	48	327	.908333	31	17	225	102	.906594
10	24	351	.975000	11	13	236	115	.974503
11	8	359	.997222	4	4	240	119	.997057
12	1	360	1.000000	0	1	240	120	1.000000

Table 2: Heuristic Distributions for Manhattan Distance on the Five Puzzle

and says little about the correlation of heuristic values with actual costs. The only connection between the accuracy of a heuristic and its distribution is that given two admissible heuristics, the one with higher values will be more accurate than the one with lower values on average.

3.5.1 The Equilibrium Distribution

While the overall distribution is the easiest to understand, the complexity of IDA* depends on a potentially different distribution. The *equilibrium distribution* $P(h)$ is defined as the probability that a node chosen randomly and uniformly among all nodes at a given depth of the brute-force search tree has heuristic value less than or equal to h, in the limit of large depth.

If all states of the problem occur with equal frequency at large depths in the search tree, then the equilibrium distribution is the same as the overall distribution. For example, this is the case with the Rubik's Cube search tree described in Section 2.2. In general, however, the equilibrium distribution may not equal the overall distribution. In the Five Puzzle, for example, the overall distribution assumes that all states, and hence all blank positions, are equally likely. As we saw in section 2.3, however, at deep levels in the tree, the blank is in a side position in more than 1/3 of the nodes, and in a corner position in less than 2/3 of the nodes. In the limit of large depth, the equilibrium frequency of side positions is $f_s = f_{cs} + f_{ss} = .203113 + .150097 = .35321$. Similarly, the frequency of corner positions is $f_c = f_{cc} + f_{sc} = .274854 + .371936 = .64679 = 1 - f_s$. Thus, to compute the equilibrium distribution, we have to take these equilibrium fractions into account.

The fifth and sixth columns of Table 2, labelled "Corner" and "Side", give the number of states with the blank in a corner or side position, respectively, for each heuristic value. The seventh and eighth columns, labelled "Csum" and "Ssum", give the cumulative numbers of corner and side states with heuristic values less than or equal to each particular heuristic value. The last column gives the equilibrium distribution $P(h)$. The probability $P(h)$ that the heuristic value of a node is less than or equal to h is the probability that it is a corner node, .64679, times the probability that its heuristic value is less than or equal to h, given that it is a corner node, plus the probability that it is a side node, .35321, times the probability that its heuristic value is less than or equal to h, given that it is a side node. For example, $P(1) = .64679 \cdot (2/240) + .35321 \cdot (1/120) = .011028$. This differs from the overall distribution $D(1) = .008333$.

The equilibrium heuristic distribution is not a property of a problem, but of a problem space. For example, including the parent of a node as one of its children can affect the equilibrium distribution, by changing the equilibrium fractions of different types of states. When the equilibrium distribution differs from the overall distribution, it can still be computed from a pattern database, or by random sampling of the problem space, combined with the equilibrium fractions of different types of states, as illustrated above.

3.6 An Example Search Tree

To provide some intuition behind our main result, Figure 4 shows a schematic representation of a search tree generated by an iteration of IDA* on an abstract problem instance, where all edges have unit cost. The vertical axis represents the depth of a node, which is also its g value, and the horizontal axis represents the heuristic value of a node. Each box represents a set of nodes at the same depth with the same heuristic value, labelled with the number of such nodes. The arrows represent the relationship between parent and child node sets. These particular numbers were generated by assuming that each node generates one child each with heuristic value one less, equal

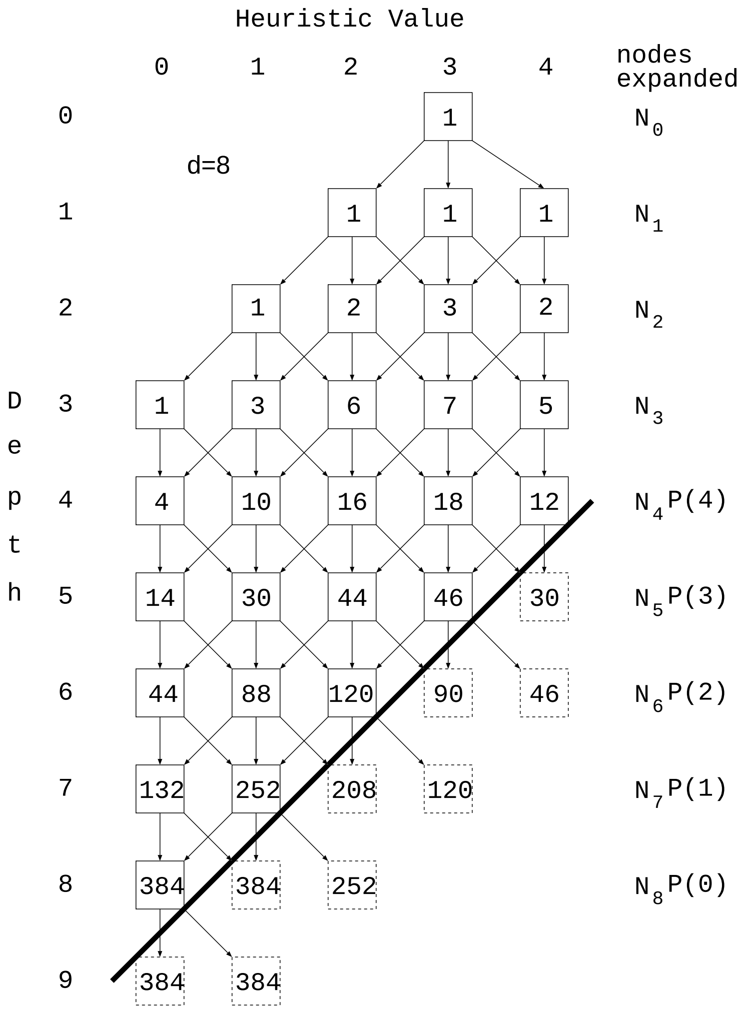

Figure 4: Sample Graph for Analysis of IDA*

to, and one greater than the heuristic value of the parent. For example, there are 6 nodes at depth 3 with heuristic value 1, 1 whose parents has heuristic value 1, 2 whose parents have heuristic value 2, and 3 whose parents have heuristic value 3. In this example, the maximum value of the heuristic is 4, and the heuristic value of the initial state is 3.

One assumption of our analysis is that the heuristic is consistent. Because of this, and since all edges have unit cost in this example, the heuristic value of a child must be at least the heuristic value of its parent, minus one. We assume a cutoff threshold of eight moves for this iteration of IDA*. Solid boxes represent sets of "fertile" nodes that will be expanded, while dotted boxes represent sets of "sterile" nodes that will not be expanded, because their total cost, $f(n) = g(n) + h(n)$ exceeds the cutoff threshold of 8. The thick diagonal line separates the fertile node sets from the sterile node sets.

3.6.1 Nodes Expanded as a Function of Depth

The values at the far right of Figure 4 show the number of nodes expanded at each depth, which is the number of fertile nodes at that depth. N_i is the number of nodes in the brute-force search tree at depth i, and $P(h)$ is the equilibrium heuristic distribution. The number of nodes generated is the branching factor times the number expanded.

Consider the graph from top to bottom. There is a root node at depth 0, which generates N_1 children. These nodes collectively generate N_2 child nodes at depth 2. Since the cutoff threshold is 8 moves, in the worst-case, all nodes n whose total cost $f(n) = g(n) + h(n) \leq 8$ will be expanded. Since 4 is the maximum heuristic value, all nodes down to depth $8 - 4 = 4$ will be expanded, Thus, for $d \leq 4$, the number of nodes expanded at depth d will be N_d, the same as in a brute-force search. Since 4 is the maximum heuristic value, $P(4) = 1$, and hence $N_4 P(4) = N_4$.

The nodes expanded at depth 5 are the fertile nodes, or those for which $f(n) = g(n) + h(n) = 5 + h(n) \leq 8$, or $h(n) \leq 3$. At sufficiently large depths, the distribution of heuristic values converges to the equilibrium distribution. Assuming that the heuristic distribution at depth 5 approximates the equilibrium distribution, the fraction of nodes at depth 5 with $h(n) \leq 3$ is approximately $P(3)$. Since all nodes at depth 4 are expanded, the total number of nodes at depth 5 is N_5, and the number of fertile nodes is $N_5 P(3)$.

There exist nodes at depth 6 with heuristic values from 0 to 4, but their distribution does not equal the equilibrium distribution. In particular, nodes with heuristic values 3 and 4, shown in dotted boxes, are underrepresented relative to the equilibrium distribution, because these nodes are generated by parents with heuristic values from 2 to 4. At depth 5, however, the nodes with heuristic value 4 are sterile, producing no offspring at depth 6, hence reducing the number of nodes at depth 6 with heuristic values 3 and 4.

The number of nodes at depth 6 with $h(n) \leq 2$ is completely unaffected by any pruning however, since their parents are nodes at depth 5 with $h(n) \leq 3$, all of which are fertile. In other words, the number of nodes at depth 6 with $h(n) \leq 2$, which are the fertile nodes, is exactly the same as in the brute-force search tree, or $N_6 P(2)$.

Due to consistency of the heuristic function, all possible parents of fertile nodes are themselves fertile. Thus, the number of nodes to the left of the diagonal line in Figure 4 is exactly the same as in the brute-force search tree. In other words, heuristic pruning of the tree has no effect on the number of fertile nodes, although it does effect the sterile nodes. If the heuristic were inconsistent, then the distribution of fertile nodes would change at every level where pruning occurred, making the analysis far more complex.

When all edges have unit cost, the number of fertile nodes at depth i is $N_i P(d-i)$, where N_i is the number of nodes in the brute-force search tree at depth i, d is the cutoff depth, and P is the equilibrium heuristic distribution. The total number of nodes expanded by an iteration of IDA* to depth d is

$$\sum_{i=0}^{d} N_i P(d-i)$$

.

3.7 General Result

Here we state and prove our main theoretical result. First, we assume a minimum edge cost, and divide all costs by this value, normalizing it to one. We express all costs as multiples of the minimum edge cost. We allow operators with different costs, and replace the depth of a node by $g(n)$, the sum of the edge costs from the root to the node. Let N_i be the number of nodes n in the brute-force search tree with $g(n) = i$.

Next, we assume the heuristic returns an integer multiple of the minimum edge cost. Given an admissible non-integer valued heuristic, we round up to the next larger integer, preserving admissibility. We also assume that the heuristic is consistent, meaning that for any two nodes n and m, $h(n) \leq k(n,m) + h(m)$, where $k(n,m)$ is the cost of an optimal path from n to m.

Given these assumptions, our task is to determine $E(N, c, P)$, the number of nodes n for which $f(n) = g(n) + h(n) \leq c$, given a problem-space tree with N_i nodes of cost i, with a heuristic characterized by the equilibrium distribution $P(x)$. This is the number of nodes expanded by an iteration of IDA* with cost threshold c, in the worst case.

Theorem 1 *In the limit of large c,*

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i)$$

Proof: $E(N, c, P)$ is the number of nodes n for which $f(n) = g(n) + h(n) \leq c$. Consider the nodes n for which $g(n) = i$, which is the set of nodes of cost i in the brute-force search tree. There are N_i such nodes. The nodes of cost i that will be expanded by IDA* in an iteration with cost threshold c are those for which $f(n) = g(n) + h(n) = i + h(n) \leq c$, or $h(n) \leq c - i$. By definition of P, in the limit of large i, the number of such nodes *in the brute-force search tree* is $N_i P(c - i)$. It remains to show that all these nodes in the brute-force search tree are also in the tree generated by IDA*.

Consider an ancestor node m of such a node n. Since m is an ancestor of n, there is only one path between them in the tree, and $g(n) = i = g(m) + K(m, n)$, where $K(m, n)$ is the cost of the path from node m to node n. Since $f(m) = g(m) + h(m)$, and $g(m) = i - K(m, n)$, $f(m) = i - K(m, n) + h(m)$. Since the heuristic is consistent, $h(m) \leq k(m, n) + h(n)$, where $k(m, n)$ is the cost of an optimal path from m to n in the problem graph. Since $K(m, n) \geq k(m, n)$, $h(m) \leq K(m, n) + h(n)$. Thus, $f(m) \leq i - K(m, n) + K(m, n) + h(n)$, or $f(m) \leq i + h(n)$. Since $h(n) \leq c - i$, $f(m) \leq i + c - i$, or $f(m) \leq c$. This implies that node m is fertile and will be expanded during the search. Since all ancestors of node n are fertile and will be expanded, node n must eventually be generated. Therefore, all nodes n in the brute-force search tree for which $f(n) = g(n) + h(n) \leq c$ are also in the tree generated by IDA*. Since there can't be any nodes in the IDA* tree that are not in the brute-force search tree , the number of such nodes at level i in the IDA* tree is $N_i P(c - i)$. Therefore, the total number of nodes expanded by IDA* in an iteration with cost threshold c, which is the number in the last iteration, is

$$E(N, c, P) = \sum_{i=0}^{c} N_i P(c - i)$$

.

3.8 The Heuristic Branching Factor

The effect of earlier iterations on the time complexity of IDA* depends on the rate of growth of node expansions in successive iterations. The *heuristic branching factor* is the ratio of the number of nodes expanded in a search to cost threshold c, divided by the nodes expanded in a search to cost $c-1$, or $E(N,c,P)/E(N,c-1,P)$, where 1 is the normalized minimum edge cost.

Assume that the size of the brute-force search tree grows exponentially with cost, or $N_i = b^i$, where b is the brute-force branching factor. In that case, the heuristic branching factor $E(N,c,P)/E(N,c-1,P)$ is

$$\frac{\sum_{i=0}^{c} b^i P(c-i)}{\sum_{i=0}^{c-1} b^i P(c-1-i)} = \frac{b^0 P(c) + b^1 P(c-1) + b^2 P(c-2) + \cdots + b^c P(0)}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)}$$

The first term of the numerator, $b^0 P(c)$, is less than or equal to one, and can be dropped without significantly affecting the ratio. Factoring b out of the remaining numerator gives

$$\frac{b(b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0))}{b^0 P(c-1) + b^1 P(c-2) + \cdots + b^{c-1} P(0)} = b$$

Thus, if the brute-force tree grows exponentially with branching factor b, then the running time of successive iterations of IDA* also grows by a factor of b. In other words, the heuristic branching factor is the same as the brute-force branching factor. In that case, it is easy to show that the overall time complexity of IDA* is $b/(b-1)$ times the complexity of the last iteration[5].

Previous analyses, based on different assumptions, predicted that the effect of a heuristic function is to reduce search complexity from $O(b^c)$ to $O(a^c)$, where $a < b$, reducing the effective branching factor [9]. Our analysis, however, shows that on an exponential tree, the effect of a heuristic is to reduce search complexity from $O(b^c)$ to $O(b^{c-k})$, for some constant k. If we define the *effective depth* of a search as the log base b of the number of nodes expanded, where b is the brute-force branching factor, then a heuristic reduces the effective depth from c to $c-k$ for some constant k, which depends on the heuristic distribution. In other words, a heuristic search to cost c generates the same number of nodes as a brute-force search to cost $c-k$.

3.9 Experimental Results

We tested our analysis experimentally by predicting the performance of IDA* on Rubik's Cube and sliding-tile puzzles, using well-known heuristics. Since all operators have unit cost in these problems, the $g(n)$ cost of a node n is its depth. For N_i, we used the exact numbers of nodes at depth i, which were computed from the recurrence relations described in section 2.3.

3.9.1 Rubik's Cube

We first predicted existing data on Rubik's Cube [6]. The problem space, described in section 2.2, allows 180-degree twists as single moves, disallows two consecutive twists of the same face, and only allows consecutive twists of opposite faces in one order. This search tree has a brute-force branching factor of about 13.34847. The median optimal solution depth is 18 moves.

The heuristic was the maximum of three different pattern databases [1, 6]. It is admissible and consistent, with a maximum value of 11 moves, and an average value of 8.898 moves. The distribution of the individual heuristics was calculated exactly by scanning the databases, and the three heuristics were assumed to be independent to calculate the distribution of the combined heuristic. In this case, the equilibrium distribution is the same as the overall distribution. We ignored goal states, completing each search iteration.

In Table 3, the first column shows the cutoff depth, the next column gives the node generations predicted by our theory, the next column indicates the number of problem instances run, the next column displays the average number of nodes generated by IDA* in a single iteration, and the last column shows the error between the theoretical prediction and experimental results.

The theory predicts the data to within 1% accuracy in every case. Sources of error include the limited number of problem instances, the assumption of independence of the heuristics, and the fact that the heuristic distribution at a finite depth does not equal the equilibrium distribution. The ratio between

Depth	Theoretical	Problems	Experimental	Error
10	1,510	1000	1,501	.596%
11	20,169	1000	20,151	.089%
12	269,229	1000	270,396	.433%
13	3,593,800	100	3,564,495	.815%
14	47,971,732	100	47,916,699	.115%
15	640,349,193	100	642,403,155	.321%
16	8,547,681,506	100	8,599,849,255	.610%
17	114,098,463,567	25	114,773,120,996	.591%

Table 3: Nodes Generated by IDA* on Rubik's Cube

the node generations in the last two levels, which is the experimental heuristic branching factor, is 13.34595, compared to the theoretical value of 13.34847.

3.9.2 Eight Puzzle

We also experimented on the Eight Puzzle, using the Manhattan distance heuristic. It has a maximum value of 22 moves, and a mean value of 14 moves. The optimal solution length averages 22 moves, with a maximum of 31 moves, assuming the blank is in a corner in the goal state. Since the Eight Puzzle has only 181,440 solvable states, the heuristic distributions were computed exactly. Three distributions were used, depending on whether the blank is in a center, corner, or side position. The number of nodes of each type at each depth of the brute-force tree was also computed exactly.

Table 4 shows a comparison of the number of node expansions predicted by our theoretical analysis, to the number of nodes expanded by a single iteration of IDA* to various depths, ignoring goal states. Each data point is the average of all 181,440 problem instances. Since the average numbers of node expansions, the size of the brute-force tree, and the heuristic distributions are all exact, the model predicts the experimental data exactly, to multiple decimal places, verifying that we have accounted for all the relevant factors.

The Eight Puzzle has even and odd-depth brute-force branching factors of 1.5 and 2. The corresponding heuristic branching factors are 1.667 and 1.8, but the product of the two branching factors is 3 in both cases.

Depth	Theoretical	Problems	Experimental	Error
20	393	181,440	393	0.0%
21	657	181,440	657	0.0%
22	1,185	181,440	1,185	0.0%
23	1,977	181,440	1,977	0.0%
24	3,561	181,440	3,561	0.0%
25	5,936	181,440	5,936	0.0%
26	10,686	181,440	10,686	0.0%
27	17,815	181,440	17,815	0.0%
28	32,072	181,440	32,072	0.0%
29	53,450	181,440	53,450	0.0%
30	96,207	181,440	96,207	0.0%
31	160,167	181,440	160,167	0.0%

Table 4: Nodes Expanded by IDA* on the Eight Puzzle

3.9.3 Fifteen Puzzle

We ran a similar experiment on the Fifteen Puzzle, also using the Manhattan distance heuristic. The average heuristic value is about 37 moves, and the maximum is 62 moves. The average optimal solution length is 52.6 moves. Since the Fifteen Puzzle has over 10^{13} solvable states, we used a random sample of ten billion solvable states to approximate the heuristic distributions. Three different distributions were used, one for the blank in a middle, corner, or side position. The number of nodes of each type at each depth was also computed exactly, for each different initial blank position.

Table 5 is similar to Table 4. Each line is the average of 100,000 random solvable problem instances. Despite over ten orders of magnitude variation in the nodes expanded in individual problem instances, the average values agree with the theoretical prediction to within 1% in most cases. The ratio between the experimental number of node expansions at the last two depths is 2.105, compared to the brute-force branching factor of 2.130.

The results above are for single complete iterations to the given search depths, ignoring any solutions found. How well do these results predict the running time of IDA* to solve a random problem instance? The average optimal solution length for 1000 random Fifteen Puzzle instances is 53.522 moves. Multiplying the last value in Table 5 by b^2 or 4.5386 predicts 372,911,869

Depth	Theoretical	Problems	Experimental	Error
40	42,664	100,000	41,973	1.65%
41	90,894	100,000	91,495	0.66%
42	193,641	100,000	191,219	1.27%
43	412,535	100,000	415,490	0.72%
44	878,864	100,000	870,440	0.96%
45	1,872,330	100,000	1,886,363	0.75%
46	3,988,805	100,000	3,959,729	0.73%
47	8,497,734	100,000	8,562,824	0.77%
48	18,103,536	100,000	18,003,959	0.55%
49	38,567,693	100,000	38,864,269	0.77%
50	82,164,440	100,000	81,826,008	0.41%

Table 5: Nodes Expanded by IDA* on the Fifteen Puzzle

node expansions in a complete iteration to depth 52, or 794,451,446 node generations. Multiplying by $b^2/(b^2-1) = 1.2826$ to account for all the previous iterations predicts about 1.019 billion node generations. Completing the final iteration to find all optimal solutions to the same set of 1000 problem instances generates an average of 1.178 billion nodes. Terminating IDA* when the first solution is found generates an average of 401 million nodes.

3.9.4 Twenty-Four Puzzle

We can also predict the performance of IDA* on problems we can't run experimentally, such as the Twenty-Four Puzzle with the Manhattan distance heuristic. The brute-force branching factor is 2.36761. Sampling ten billion random solvable states yields an approximation of the overall heuristic distribution, which approximates the equilibrium distribution. The average heuristic value is 76 moves. Experiments using more powerful disjoint pattern database heuristics [8] give an average optimal solution length of 100 moves. Our theory predicts that running all iterations up to depth 100 will generate an average of 1.217×10^{19} nodes. On a 440 Mhz Sun Ultra 10 workstation, IDA* with Manhattan distance generates about 7.5 million nodes per second. This predicts an average time to complete all iterations up to depth 100, on a random instance of the Twenty-Four Puzzle, ignoring any solutions found, of about 51,454 years!

3.9.5 Observed Heuristic Branching Factor

If we run IDA* on a single instance of a sliding-tile puzzle, we observe that the ratio between the numbers of nodes generated in successive iterations usually decreases with each iteration, but exceeds the theoretical heuristic branching factor. On the sliding-tile puzzles with Manhattan distance, the cost threshold increases by two in each successive iteration, and hence the theoretical heuristic branching factor is the square of the brute-force branching factor. For example, in the Twenty-Four Puzzle, the observed heuristic branching factor is often greater than 10, whereas b^2 is only 5.6

The reason for this discrepancy is an initial transient in the observed heuristic branching factor. The formula in Theorem 1 is based on the equilibrium heuristic distribution. Starting from a single initial state, it takes many iterations of IDA* for the heuristic distribution at a given level to converge to the equilibrium distribution. This effect is ameliorated in the results presented above because the experimental data is averaged over a large number of initial states, which better approximates the equilibrium distribution. If we run IDA* long enough on a single problem instance, the observed heuristic branching factor eventually converges to the square of the brute-force branching factor.

Why does the observed heuristic branching factor overestimate the theoretical branching factor? The heuristic distribution at the root of the search tree starts with the heuristic value of the initial state, and gradually spreads out to larger and smaller heuristic values with increasing depth. Thus, the frequency of small and large heuristic values is initially zero, underestimating their frequency in the equilibrium heuristic distribution. Underestimating the large values has little effect, since the frequency of these values is multiplied by the relatively small number of nodes at shallow depths. The frequencies of small values, however, are multiplied by the large numbers of nodes at deep depths, and hence underestimating these values significantly decreases the number of node generations, relative to what happens at equilibrium. As the search depth increases in successive iterations, the frequency of nodes with small heuristic values increases, which causes a larger observed heuristic branching factor than occurs at equilibrium.

4 Conclusions

We first show how to compute the exact number of nodes at different depths, and the asymptotic branching factor, of brute-force search trees where different nodes have different numbers of children. We begin by writing a set of recurrence relations for the generation of the different node types. By expanding these recurrence relations, we can determine the exact number of nodes at a given depth, in time linear in the depth. We can also use the ratio of the numbers of nodes at successive depths to approximate the asymptotic branching factor with very high precision. Alternatively, we can rewrite the recurrence relations as a set of simultaneous equations involving the relative frequencies of the different types of nodes, and solve them analytically for small numbers of node types. We give the asymptotic branching factors for Rubik's Cube, the Five Puzzle, and the first nine square sliding-tile puzzles.

We then use these results to predict the time complexity of IDA*. We characterize a heuristic by the distribution of heuristic values, which can be obtained by random sampling, for example. We compare our predictions with experimental data on Rubik's Cube, the Eight Puzzle, and the Fifteen Puzzle, getting agreement within 1% for Rubik's Cube and the Fifteen Puzzle, and exact agreement for the Eight Puzzle. In contrast to previous results, our analysis and experiments indicate that on an exponential tree, the asymptotic heuristic branching factor is the same as the brute-force branching factor, and the effect of a heuristic is to reduce the effective depth of search, rather than the effective branching factor, relative to a brute-force search.

5 Generality and Further Work

To what extent can these results can be applied to other problems? Our main result is Theorem 1. It says that the number of nodes n for which $f(n) = g(n) + h(n) \leq c$ is a convolution of two distributions. The first is the number of nodes of a given cost in the brute-force search space, and the second is the number of nodes with a given heuristic value. In order to apply this to a particular problem, however, we have to determine the size of the brute-force search space, and the heuristic distribution. Thus, we have decomposed the problem of predicting the performance of a heuristic search algorithm into two simpler problems.

How could we use this analysis to predict the performance of A*? The main difference between A* and IDA* is that A* detects duplicate nodes, and doesn't reexpand them. Theorem 1 applies to A* as well, but N_i is the number of nodes in the problem-space graph, rather than its tree expansion. Unfortunately, the only technique known for computing the number of nodes at a given depth in a search graph is exhaustive search to that depth. As a result, these values are unknown for even regular problem spaces such as the Fifteen Puzzle or Rubik's Cube. The relevant heuristic distribution $P(h)$ for analyzing A* is the overall heuristic distribution $D(h)$, because each state occurs only once in the problem space.

As another example, could we predict the performance of IDA* on the traveling salesman problem? In a problem space that constructs a tour one by adding one city at a time, each node represents a partial tour, and the number of nodes at depth d is the number of permutations of $n-1$ elements taken d at a time. Computing the distribution for a heuristic such as the cost of a minimum spanning tree of the remaining cities is difficult, however. It depends on the depth of search, and the particular problem instance. If the edge costs and heuristic values are real numbers rather than integers, the discrete convolution of Theorem 1 becomes a continuous convolution, and the summation becomes an integral. While we can't solve this problem currently, Theorem 1 tells us what distributions we need, and how to combine them.

The running time of IDA* depends on the branching factor, the heuristic distribution, and the optimal solution cost. Predicting the optimal solution cost for a given problem instance, or even the average optimal solution cost, is an open problem, however. Since the number of nodes in a problem-space tree grows by a factor of b with each succeeding depth, a lower bound on the maximum optimal solution depth is the log base b of the number of reachable states, rounded up to the next larger integer. This can be used as an estimate of the average solution depth. For example, this method predicts a depth of 22 moves for the Eight Puzzle, which equals the average optimal solution length. For Rubik's Cube, this method predicts a value of 18 moves, which is the median optimal solution length. For the Fifteen Puzzle, however, we get an estimate of only 40 moves, while the average solution depth is 53.5 moves. The reason this method doesn't accurately predict the maximum solution depth is that it assumes that all states in the search tree are unique. For all these problems, however, there are multiple paths to the same state, giving rise to duplicate nodes in the tree representing the same state.

6 Acknowledgments

We would like to thank Eli Gafni, Elias Koutsoupias, and Mitchell Tsai for several helpful discussions. R. Korf was supported by NSF grant IRI-9619447.

References

[1] Culberson, J., and J. Schaeffer. Pattern Databases, *Computational Intelligence*, Vol. 14, No. 4, pp. 318-334, 1998.

[2] Edelkamp, S. and R.E. Korf, The branching factor of regular search spaces, *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, July, 1998, pp. 299-304.

[3] Gaschnig, J. *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1979.

[4] Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-4, No. 2, July 1968, pp. 100-107.

[5] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[6] Korf, R.E., Finding optimal solutions to Rubik's Cube using pattern databases, *Proceedings of the National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, July, 1997, pp. 700-705.

[7] Korf, R.E., and M. Reid, Complexity analysis of admissible heuristic search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, July, 1998, pp. 305-310.

[8] Korf, R.E, A. Felner, Disjoint pattern database heuristics, to appear in *Artificial Intelligence*, 2001.

[9] Pearl, J. *Heuristics*, Addison-Wesley, Reading, MA, 1984.

[10] Pohl, I., Practical and theoretical considerations in heuristic search algorithms, *Machine Intelligence 8*, 1977, pp. 55-72