OPTIMAL PATH-FINDING ALGORITHMS*

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024

ABSTRACT

Path-finding search occurs in the presence of various sources of knowledge such as heuristic evaluation functions, subgoals, macro-operators, and abstractions. For a given source of knowledge, we explore optimal algorithms in terms of time, space, and cost of solution path, and quantify their performance. In the absence of any knowledge, a depth-first iterative-deepening algorithm is asymptotically optimal in time and space among minimal-cost exponential tree searches. A heuristic version of the algorithm, called iterative-deepening-A*, is shown to be optimal in the same sense over heuristic searches. For solving large problems, more powerful knowledge sources, such as subgoals, macro-operators, and abstraction are required, usually at the cost of suboptimal solutions. Subgoals often drastically reduce problem solving complexity, depending upon the subgoal distance, a new measure of problem difficulty. Macro-operators store solutions to previously solved subproblems in order to speed-up solutions to new problems, and are subject to a multiplicative time-space tradeoff. Finally, an analysis of abstraction concludes that abstraction hierarchies can reduce exponential problems to linear complexity.

*This chapter is based upon two articles [1] and [2] that appeared originally in the journal *Artificial Intelligence*. The material is reprinted here by permission of the publisher, North-Holland, The Netherlands. This research was supported in part by NSF Grant IST 85-15302, an NSF Presidential Young Investigator Award, an IBM Faculty Development Award, and a grant from the Delco Electronics Corporation.

L. Kanal et al. (eds.), *Search in Artificial Intelligence*

1 INTRODUCTION

Search is ubiquitous in artificial intelligence, and the performance of most AI systems is determined by the the complexity of a search algorithm in their inner loops. In order to improve the performance of a search algorithm, generally more knowledge about the problem must be made available to the algorithm. The different possible sources of knowledge include heuristic evaluation functions, subgoals, macro-operators, and abstractions.

We adopt a common model of problem solving search, the **problem space**, and consider the effect of each knowledge source on search complexity. In each case, the knowledge is defined in terms of a problem space, and we explain how and to what extent the knowledge reduces search in that space. We present two main ideas: the technique of iterative-deepening to reduce the memory requirements of brute-force and heuristic searches, and a quantitative theory of planning that views it as problem solving search using subgoals, macro-operators, and abstractions as knowledge sources.

1.1 Problem Space Model

The problem space model was formulated by Newell and Simon[3]. A problem space consists of a set of **states**, and a set of **operators** that are mappings from states to states. An operator is a partial mapping since it may have preconditions which determine what states it can be applied to. A problem, or **problem instance**, is composed of a problem space together with an **initial** state and a set of **goal** states. The task is to find a **solution**, that is, a sequence of operators that map the initial state to one of the goal states. The issues that we will be concerned with are the amount of time and space that are required to find a solution, and the cost of the solution itself.

For example, consider the well-known Eight Puzzle (see Figure 1). It consists of a three by three square frame which holds eight movable square tiles, with one space left over (the blank). The states of the Eight Puzzle are the different possible permutations of the tiles in the frame. The allowable operators are to move a tile which is horizontally or vertically adjacent to the blank into the blank position. The object of the puzzle is to find a sequence of legal moves that maps an initial arrangement of the tiles into the desired goal configuration. The cost

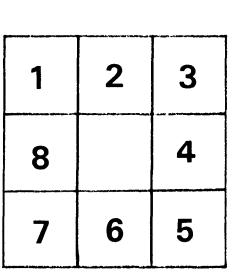

Figure 1: The Eight Puzzle

of a solution to this problem is the number of moves required.

The complexity of a problem will be expressed in terms of two parameters: the branching factor of the problem space, and the depth of solution of the problem. The **node branching factor** (b) of a problem is defined as the number of new states that can be generated by the application of a single operator to a given state, averaged over all states in the problem space. We will assume that the branching factor is constant throughout the problem space. The **depth** (d) of solution of a problem instance is the length of a shortest sequence of operators that map the initial state to a goal state. The time cost of a search algorithm in this model of computation is simply the number of nodes that are expanded. The reason for this choice is that we are interested in asymptotic complexity and the amount of time is proportional to the number of states expanded. Similarly, since the amount of space required is proportional to the number of states that are stored, the asymptotic space cost of an algorithm in this model is the number of states that must be stored.

1.2 Iterative-Deepening

Most of the literature on search has focussed on one particular source of knowledge, namely heuristic evaluation functions. The main goal of that literature is to analyze the performance of various heuristic search algorithms in terms of two primary performance measures: the quality of the solutions they generate, and the amount of time required to find those solutions.

A third performance measure, namely the amount of space or memory required by an algorithm, has been largely ignored. The result is the development of a number of algorithms, such as breadth-first search and A*, that in practice exhaust the available memory on typical computer configurations in a matter of seconds. We present a

general technique, called iterative-deepening, that drastically reduces the memory requirement of many different search algorithms, without sacrificing solution quality nor asymptotic time complexity.

While the idea of iterative-deepening is not new, its generality and time-complexity are not well-known. The iterative-deepening technique can be applied to brute-force uni-directional search, bi-directional search, and heuristic searches such as A*, as well as its more traditional role in two-player game searches. We demonstrate the generality of iterative-deepening, analyze its time, space and solution complexity, and show its optimality for exponential tree searches. Much of this research originally appeared in [1].

1.3 Planning as Search

Heuristic evaluation functions are just one source of knowledge that can be used to reduce problem-solving complexity. Other sources of knowledge include subgoals, macro-operators, and abstractions. Problem-solving using these knowledge sources has traditionally been called planning. Ideally, the term planning applies to problem solving in a real-world environment where the agent may not have complete information about the world or cannot completely predict the effects of its actions. In that case, the agent goes through several iterations of planning a solution, executing the plan, and then replanning based on the perceived result of the solution.

Most of the literature on planning, however, deals with problem solving with perfect information and prediction. What distinguishes planning from heuristic search is that instead of using heuristic evaluation function as a source of knowledge, planning tends to make use of knowledge such as subgoals, macro-operators, and abstraction spaces. We view planning simply as problem-solving search using these other sources of knowledge. The reason for this view is to be able to take advantage of the same models and analytic techniques that have been used so successfully in the study of heuristic seach, and apply them to search using these other sources of knowledge.

As the study of search has matured, the state of the art has been refined from qualitative statements to quantitative results . For example, qualitative wisdom of the form, "the more accurate a heuristic evaluation function is, the more efficient search with that function becomes", has been replaced with theorems such as, "if a heuristic

function exhibits constant absolute error, then best-first search using that function runs in linear time"[4]. The study of planning, however, remains at the stage of qualitative wisdom, such as, "problem solving using a hierarchy of abstraction spaces greatly improves search efficiency". We present the first steps toward a quantitative theory of planning. Our results include the identification of **subgoal distance** as a fundamental measure of problem diffiulty, a multiplicative time-space tradeoff for macro-operators, and an analysis of abstraction that shows that it can reduce exponential problems to linear complexity. Much of this research originally appeared in [2].

2 BRUTE-FORCE SEARCH

Before examining how various knowledge sources reduce search effort, we first consider the case where no knowledge is available, or brute-force search. The most common brute-force search algorithms are breadth-first and depth-first search.

2.1 Breadth-First Search

Breadth-first search expands all the states one step (or operator application) away from the initial state, then expands all states two steps from the initial state, then three steps, etc., until a goal state is reached. Since it always expands all nodes at a given depth before expanding any nodes at a greater depth, the first solution path found by breadth-first search will be one of shortest length. In the worst case, breadth-first search must generate all nodes up to depth d, or $b + b^2 + b^3 + ... + b^d$ which is $O(b^d)$. Note that on the average, half of the nodes at depth d must be examined, and therefore the average case time complexity is also $O(b^d)$.

Since all the nodes at a given depth are stored in order to generate the nodes at the next depth, the minimum number of nodes that must be stored to search to depth d is b^{d-1}, which is $O(b^d)$. As with time, the average case space complexity is roughly one-half of this, which is also $O(b^d)$. This space requirement of breadth-first search is its most critical drawback. As a practical matter, a breadth-first search of most problem spaces will exhaust the available memory long before an appreciable amount of time is used. The reason for this is that the

Figure 2: Graph with linear number of nodes but exponential number of paths

typical ratio of memory to speed in modern computers is a million words of memory for each million instructions per second (MIPS) of processor speed. For example, if we can generate a million states per minute and require a word to store each state, memory will be exhausted in one minute.

2.2 Depth-First Search

Depth-first search avoids this memory limitation. It works by always generating a descendant of the most recently expanded node, until some depth cutoff is reached, and then backtracking to the next most recently expanded node and generating one of its descendants. Therefore, only the path of nodes from the initial node to the current node must be stored in order to execute the algorithm. If the depth cutoff is d, then the space required by depth-first search is only $O(d)$.

Since depth-first search only stores the current path at any given point, it is bound to search all paths down to the cutoff depth. In order to analyze its time complexity, we must define a new parameter, called the **edge branching factor** (e), which is the average number of different operators which are applicable to a given state. For trees, the edge and node branching factors are equal, but for graphs in general the edge branching factor may exceed the node branching factor. For example, the graph in Figure 2 has an edge branching factor of two, while its node branching factor is only one. Note that a breadth-first search of this graph takes only linear time while a depth-first search requires exponential time, since there are 2^d paths to node at depth d. In general, the time complexity of a depth-first search to depth d is $O(e^d)$. Since the space used by depth-first search grows only as the log of the time required, the algorithm is time-bound rather than space-bound in practice.

Another drawback, however, to depth-first search is the requirement for an arbitrary cutoff depth. If branches are not cut off and duplicates are not checked for, the algorithm may not terminate. In general, the depth at which the first goal state appears is not known in advance and must be estimated. If the estimate is too low, the algorithm terminates without finding a solution. If the depth estimate is too high, then a large price in running time is paid relative to that of an optimal search, and the first solution found may not be an optimal one.

2.3 Depth-First Iterative-Deepening

A search algorithm which suffers neither the drawbacks of breadth-first nor depth-first search is depth-first iterative-deepening (DFID). The algorithm works as follows: First, perform a depth-first search to depth one. Then, discarding the nodes generated in the first search, start over and do a depth-first search to level two. Next, start over again and do a depth-first search to depth three, etc., continuing this process until a goal state is reached.

Depth-first iterative-deepening has no doubt been rediscovered many times independently. The first use of the algorithm that is documented in the literature is in Slate and Atkin's Chess 4.5 program[5]. Berliner[6] has observed that breadth-first search is inferior to the iterative-deepening algorithm. Winston[7] shows that for two-person game searches where only terminal node static evaluations are counted in the cost, the extra computation required by iterative-deepening is insignificant.

Since DFID expands all nodes at a given depth before expanding any nodes at a greater depth, it is guaranteed to find a shortest length solution. Also, since at any given time it is performing a depth-first search, and never searches deeper than depth d, the space it uses is $O(d)$.

The disadvantage of DFID is that it performs wasted computation prior to reaching the goal depth. In fact, at first glance it seems very inefficient. Below, however, we present an analysis of the running time of DFID that shows that this wasted computation does not affect the asymptotic growth of the running time for exponential tree searches. The intuitive reason is that almost all the work is done at the deepest level of the search. Unfortunately, DFID suffers the same drawback

as depth-first search on arbitrary graphs, namely that it must explore all possible paths to a given depth.

More formally, we define a **brute-force** search as a search algorithm that uses no information other than the initial state, the operators of the space, and a test for a solution. Given this definition, it can be shown that **depth-first iterative-deepening is asymptotically optimal in terms of time and space among brute-force tree searches that find optimal solutions.** The argument is as follows:

As mentioned above, since DFID generates all nodes at a given depth before expanding any nodes at a greater depth, it always finds a shortest path to the goal, or any other state. Hence, it always finds an optimal solution.

Next, we examine the running time of DFID on a tree. The nodes at depth d are generated once during the final iteration of the search. The nodes at depth $d-1$ are generated twice, once during the final iteration at depth d, and once during the penultimate iteration at depth $d-1$. Similarly, the nodes at depth $d-2$ are generated three times, during iterations at depths d, $d-1$, and $d-2$, etc. Thus the total number of nodes generated in a depth-first iterative-deepening search to depth d is

$$b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db.$$

Factoring out b^d gives

$$b^d(1 + 2b^{-1} + 3b^{-2} + \cdots + db^{1-d}).$$

Letting $x = 1/b$ yields

$$b^d(1 + 2x + 3x^2 + 4x^3 \cdots + dx^{d-1}).$$

This is less than the infinite series

$$b^d(1 + 2x + 3x^2 + 4x^3 + \cdots),$$

which converges to

$$\frac{b^d}{(1-x)^2} \text{ for } |x| < 1, \text{ or}$$

$$b^d \left(\frac{b}{b-1}\right)^2 \text{ for } b > 1.$$

Since $(b/(b-1))^2$, is a constant that is independent of d, if $b > 1$ then the running time of depth-first iterative-deepening is $O(b^d)$. A similar result was arrived at independently by Stickel and Tyson[9].

To see that this is optimal, we present a simple adversary argument. The number of nodes at depth d is b^d. Assume that there exists an algorithm that examines less than b^d nodes. Then, there must exist at least one node at depth d which is not examined by this algorithm. Since we have no additional information, an adversary could place the only solution at this node and hence the proposed algorithm would fail. Hence, any brute-force algorithm must take at least cb^d time, for some constant c.

Finally, we consider the space used by DFID. Since DFID at any point is engaged in a depth-first search, it need only store a stack of nodes which represents the branch of the tree it is expanding. Since it finds a solution of optimal length, the maximum depth of this stack is d, and hence the maximum amount of space is $O(d)$.

To show that this is optimal, we note that any algorithm which uses $f(n)$ time must use at least $k \log f(n)$ space for some constant k [8]. The reason is that the algorithm must proceed through $f(n)$ distinct states before looping or terminating, and hence must be able to store that many distinct states. Since storing $f(n)$ states requires $\log f(n)$ bits, and $\log b^d$ is $d \log b$, any brute-force algorithm must use kd space, for some constant k.

The value of the constant $(b/(b-1))^2$ gives an upper bound on how much computation is wasted in the lower levels of the search, since it is the limit of the constant coefficient as the search depth goes to infinity. Figure 3 shows a graph of this constant versus the branching factor. As the branching factor increases, the constant approaches one. For branching factors close to one, however, the value of the constant coefficient approaches infinity as the depth goes to infinity.

Note that this constant factor measures the relative amount of computation expended in the levels prior to the search frontier. Even breadth-first search must spend some time in these levels as well, and its complexity is approximately $b^d(b/(b-1))$. Thus, if we compare DFID to breadth-first search, we find that the ratio of their running times is approximately $b/(b-1)$[9], which is even less than $(b/(b-1))^2$.

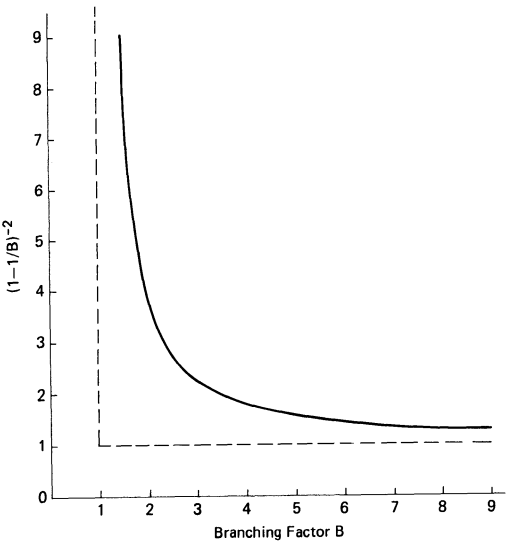

Figure 3: Graph of branching factor vs. constant coefficient of depth-first iterative-deepening as search depth goes to infinity

2.4 Bi-Directional Search

For those problems with a single goal state that is given explicitly and for which the operators have inverses, such as the Eight Puzzle, bi-directional search[10] can be used. Bi-directional search trades space for time by searching forward from the initial state and backward from the goal state simultaneously, storing the states generated, until a common state is found on both search frontiers. Depth-first iterative-deepening can be applied to bi-directional search as follows. A single iteration consists of a depth-first search from one direction to depth k, storing only the states at depth k, and two depth-first searches from the other direction, one to depth k and one to depth $k+1$, not storing states but simply matching against the stored states from the other direction. The search to depth $k+1$ is necessary to find odd length solutions. This is repeated for k from zero (to find solutions of length one) to $d/2$. Assuming that a hashing scheme is used to perform the matching in constant time per node, this algorithm will find an optimal solution of length d in time $O(b^{d/2})$ and space $O(b^{d/2})$.

3 HEURISTIC SEARCH

The source of knowledge that has been most extensively studied in the context of search problems is the heuristic evaluation function. In the case of single-agent problems, a heuristic evaluation function is an efficiently computable function that estimates the cost of reaching a goal from a given state. For example, a common heuristic function for the Eight Puzzle is Manhattan Distance: it is computed by determining for each tile the number of grid units between its current position and its goal position, and summing these values over all tiles. There are a number of search algorithms that utilize heuristic evaluation functions to reduce search, including hill-climbing, best-first search, and A*.[11,1].

3.1 A*

The best-known of these algorithms, A*, works as follows: it maintains an OPEN list of those nodes that have been generated but not yet expanded. Associated with each node n is a cost, $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost of reaching node n from the initial state, and $h(n)$ is the heuristic function or the estimated cost of reaching the goal from node n. At each stage, A* expands the node on OPEN with the least cost, until a goal node is chosen for expansion.

A well-known property of A* is that it always finds a shortest solution path if the heuristic function never overestimates the actual distance to the goal[13]. For example, since Manhattan Distance is a lower bound on the actual number of moves required to solve an instance of the Eight Puzzle, A* using the Manhattan Distance Heuristic is guaranteed to find an optimal solution.

The reason, of course, for using a heuristic search algorithm instead of a brute-force algorithm is that in general the heuristic algorithm expands fewer nodes to solve the same problem. The actual number of node expansions to solve a given problem depends upon the accuracy of the heuristic function. In order to characterize the efficiency of a heuristic search algorithm, we can define a **heuristic branching factor**, analogous to the brute-force branching factor of the problem space. Whereas the brute-force branching factor is defined as the average ratio of the number of nodes at a given depth to the number

of nodes at the next shallower depth, the heuristic branching factor is defined as the average ratio of the number of nodes at a given cost to the number of nodes at the next smaller cost. In the case of A*, cost refers to $g(n) + h(n)$.

To get a feel for the tradeoff between accuracy of a heuristic function and time complexity, it is useful to look at certain values. For example, if the heuristic function is zero everywhere, and all edges have unit cost, A* reduces to breadth-first search and the heuristic branching factor equals the brute-force branching factor. If the heuristic function is an exact estimate everywhere, the heuristic branching factor is one, and A* runs in linear time. Similarly, if the heuristic enjoys constant absolute error, the heuristic branching factor is still one and the running time is linear, but the number of nodes at each level is larger. Finally, if the heuristic suffers from constant relative error, the most realistic assumption, then the heuristic branching factor is greater than one but less than the brute-force branching factor, and the running time of A* is exponential.

A recent result of Dechter and Pearl[14] shows that for a given consistent non-overestimating heuristic function, A* is optimal, in terms of number of nodes expanded, over the class of best-first searches that find optimal solutions. This means that the A* cost function of $g(n) + h(n)$ is the best way to use the heuristic function to find optimal solutions.

Unfortunately, A* suffers from the same memory limitation as breadth-first search. The space required to store the OPEN list causes the algorithm to run out of memory in a matter of seconds on typical computer configurations.

3.2 Iterative-Deepening-A*

As in the case of breadth-first search, depth-first iterative-deepening can be applied to best-first heuristic searches such as A* to remove this memory limitation. The idea is that successive iterations correspond not to increasing depth of search, but rather to increasing values of the total cost of a path. Iterative-deepening-A* (IDA*) works as follows: At each iteration, perform a depth-first search, cutting off a branch when its total cost ($f(n) = g(n) + h(n)$) exceeds a given threshold. This threshold starts at the estimate of the cost of the initial state, and increases for each iteration of the algorithm. At each iteration,

the threshold used for the next iteration is the minimum of all values that exceeded the previous threshold. The algorithm continues until a goal node is expanded. As in the case of A*, if all the operators have equal cost, the algorithm can be terminated as soon as a goal node is generated.

The main advantage of IDA* is its space complexity. Since at any given point it is executing a depth-first search, its space complexity is only linear in the solution depth, as opposed to exponential as in the case of A*.

3.2.1 Optimality of IDA*

Fortunately, IDA* retains the optimality of A* with regard to solution cost. In particular, **given a non-overestimating cost function, iterative-deepening-A* will find a solution of minimal cost if one exists.** The argument is as follows: Since the initial cost cutoff of IDA* is the heuristic estimate of the cost of the initial state, and the heuristic never overestimates cost, the initial cost threshold must be less than or equal to the cost of the optimal solution. Furthermore, since the cost threshold for each succeeding iteration is the minimum value which exceeded the previous cutoff, and no node along the optimal solution path can have a cost greater than the optimal solution path, the cost threshold for some iteration must eventually equal the optimal solution cost. When that happens, the goal node along the optimal path will be expanded, terminating the algorithm, and no other goal node with a greater cost will be expanded in that iteration. Therefore, the first goal noded expanded by IDA* will be one of least cost.

To determine the time used by IDA*, consider the final iteration, that is, the one that expands a goal node. In the worst case, i.e. the solution path is the last path explored in that iteration, it must expand all nodes reachable from the initial state along a path in which none of the node values exceed the optimal solution cost. A* must also expand these same nodes in the worst case tie-breaking situation, i.e. the goal node is the last node of its cost expanded. Thus, in the worst case, the final iteration of IDA* expands the same set of nodes as A*. Furthermore, if the graph is a tree, each of these nodes will be expanded exactly once by each algorithm. IDA* must also expand nodes during the previous iterations as well. However, if the heuris-

tic branching factor is strictly greater than one, then each iteration expands an exponentially increasing number of new nodes. In that case, we can use the same argument as in the brute-force case, substituting the heuristic branching factor for the brute-force branching factor, to show that the previous iterations of IDA* do not affect the asymptotic order of the total number of nodes. Thus, IDA* expands the same number of nodes, asymptotically, as A*. Combining this with the time optimality of A* allows us to conclude that IDA* is asymptotically optimal in terms of time for exponential heuristic tree searches that find optimal solutions.

Since at any given point, IDA* is executing a depth-first search, its space complexity is linear in the depth of the solution. Since the fastest algorithm for this problem takes exponential time, we can conclude that the space complexity of IDA* is also asymptotically optimal.

Combining these three facts leads to the following general result: **given a non-overestimating heuristic function with heuristic branching factor greater than one, then iterative-deepening-A* is asymptotically optimal in time and space over the class of best-first searches that find optimal solutions on a tree.**

The assumption that is critical to the performance of IDA* is that the number of nodes generated grow exponentially with the number of iterations, or in other words, that the heuristic branching factor be greater than one. Is this a realistic assumption in practice? Since a heuristic function with constant relative error produces exponential complexity, one approach to this question is to ask if the assumption of constant relative error, i.e. that the error in the estimate grows at the same rate as the magnitude of the actual cost, is valid for most heuristics? Pearl observes that heuristics with better accuracy almost never occur in practice. For example, most physical measurements are subject to constant relative error[11].

Since the error in an actual heuristic function may be difficult to measure, another approach is to ask under what conditions might the number of nodes per iteration grow slower than exponentially. Since the new nodes expanded during any iteration all have costs equal to the threshold of that iteration, the issue is the rate of growth of the number of nodes tied at a given cost value, as a function of that cost. One condition that allows sub-exponential growth of the number of

ties is if the problem space itself does not grow exponentially with depth. In such a case, search is no longer an exponential problem. Assuming, however, that we have an exponential problem space, then we need an exponentially increasing number of different node values to prevent exponential numbers of ties. One way to achieve this is if the cost of each node is a unique positive integer, but this requires that the values of nodes grow exponentially with depth. Another other way to accomplish it would be if the costs of nodes were chosen uniquely from an infinite limiting series such as $1/2, 3/4, 7/8$, etc., but this requires increasing the number of significant digits for node values with increasing depth. Both these assumptions, exponentially increasing node values and increasing precision for node values, are unrealistic in practice.

Thus, we can conclude that heuristic depth-first iterative-deepening is asymptotically optimal for most best-first tree searches which occur in practice.

3.2.2 IDA* in Practice

As an empirical test of the practicality of this algorithm, both IDA* and A* were implemented for the Fifteen Puzzle, a larger 4x4 relative of the Eight Puzzle. The implementations were in Pascal and were run on a DEC 2060. The heuristic function used for both was Manhattan Distance. The two algorithms were tested against 100 randomly generated, solvable initial states. IDA* solved all instances with a median time of 30 CPU minutes, generating over 1.5 million nodes per minute. The average solution length was 53 moves and the maximum was 66 moves. A* solved none of the instances since it ran out of space after about 30,000 nodes were generated. The data from this experiment are summarized in Table 1. These are the first published optimal solution lengths to randomly generated instances of the Fifteen Puzzle. Although the Fifteen Puzzle graph is not strictly a tree, its edge branching factor is only slightly greater than its node branching factor, and hence the iterative-deeepening algorithm is still effective.

An additional benefit of IDA* over A* is that it is simpler to implement since there are no OPEN or CLOSED lists to be managed. A simple recursion performs the depth-first search inside an outer loop to handle the iterations. A related observation is that even though

NUMBER	INITIAL STATE	ESTIMATE	ACTUAL	NODES
1	14 13 15 7 11 12 9 5 6 0 2 1 4 8 10 3	41	57	276,361,933
2	13 5 4 10 9 12 8 14 2 3 7 1 0 15 11 6	43	55	15,300,442
3	14 7 8 2 13 11 10 4 9 12 5 0 3 6 1 15	41	59	565,994,203
4	5 12 10 7 15 11 14 0 8 2 1 13 3 4 9 6	42	56	62,643,179
5	4 7 14 13 10 3 9 12 11 5 6 15 1 2 8 0	42	56	11,020,325
6	14 7 1 9 12 3 6 15 8 11 2 5 10 0 4 13	36	52	32,201,660
7	2 11 15 5 13 4 6 7 12 8 10 1 9 3 14 0	30	52	387,138,094
8	12 11 15 3 8 0 4 2 6 13 9 5 14 1 10 7	32	50	39,118,937
9	3 14 9 11 5 4 8 2 13 12 6 7 10 1 15 0	32	46	1,650,696
10	13 11 8 9 0 15 7 10 4 3 6 14 5 12 2 1	43	59	198,758,703
11	5 9 13 14 6 3 7 12 10 8 4 0 15 2 11 1	43	57	150,346,072
12	14 1 9 6 4 8 12 5 7 2 3 0 10 11 13 15	35	45	546,344
13	3 6 5 2 10 0 15 14 1 4 13 12 9 8 11 7	36	46	11,861,705
14	7 6 8 1 11 5 14 10 3 4 9 13 15 2 0 12	41	59	1,369,596,778
15	13 11 4 12 1 8 9 15 6 5 14 2 7 3 10 0	44	62	543,598,067
16	1 3 2 5 10 9 15 6 8 14 13 11 12 4 7 0	24	42	17,984,051
17	15 14 0 4 11 1 6 13 7 5 8 9 3 2 10 12	46	66	607,399,560
18	6 0 14 12 1 15 9 10 11 4 7 2 8 3 5 13	43	55	23,711,067
19	7 11 8 3 14 0 6 15 1 4 13 9 5 12 2 10	36	46	1,280,495
20	6 12 11 3 13 7 9 15 2 14 8 10 4 1 5 0	36	52	17,954,870
21	12 8 14 6 11 4 7 0 5 1 10 15 3 13 9 2	34	54	257,064,810
22	14 3 9 1 15 8 4 5 11 7 10 13 0 2 12 6	41	59	750,746,755
23	10 9 3 11 0 13 2 14 5 6 4 7 8 15 1 12	33	49	15,971,319
24	7 3 14 13 4 1 10 8 5 12 9 11 2 15 6 0	34	54	42,693,209
25	11 4 2 7 1 0 10 15 6 9 14 8 3 13 5 12	32	52	100,734,844
26	5 7 3 12 15 13 14 8 0 10 9 6 1 4 2 11	40	58	226,668,645
27	14 1 8 15 2 6 0 3 9 12 10 13 4 7 5 11	33	53	306,123,421
28	13 14 6 12 4 5 1 0 9 3 10 2 15 11 8 7	36	52	5,934,442
29	9 8 0 2 15 1 4 14 3 10 7 5 11 13 6 12	38	54	117,076,111
30	12 15 2 6 1 14 4 8 5 3 7 0 10 13 9 11	35	47	2,196,593
31	12 8 15 13 1 0 5 4 6 3 2 11 9 7 14 10	38	50	2,351,811
32	14 10 9 4 13 6 5 8 2 12 7 0 1 3 11 15	43	59	661,041,936
33	14 3 5 15 11 6 13 9 0 10 2 12 4 1 7 8	42	60	480,637,867
34	6 11 7 8 13 2 5 4 1 10 3 9 14 0 12 15	36	52	20,671,552
35	1 6 12 14 3 2 15 8 4 5 13 9 0 7 11 10	39	55	47,506,056
36	12 6 0 4 7 3 15 1 13 9 8 11 2 14 5 10	36	52	59,802,602
37	8 1 7 12 11 0 10 5 9 15 6 13 14 2 3 4	40	58	280,078,791
38	7 15 8 2 13 6 3 12 11 0 4 10 9 5 1 14	41	53	24,492,852
39	9 0 4 10 1 14 15 3 12 6 5 7 11 13 8 2	35	49	19,355,806
40	11 5 1 14 4 12 10 0 2 7 13 3 9 15 6 8	36	54	63,276,188
41	8 13 10 9 11 3 15 6 0 1 2 14 12 5 4 7	36	54	51,501,544
42	4 5 7 2 9 14 12 13 0 3 6 11 8 1 15 10	30	42	877,823
43	11 15 14 13 1 9 10 4 3 6 2 12 7 5 8 0	48	64	41,124,767
44	12 9 0 6 8 3 5 14 2 4 11 7 10 1 15 13	32	50	95,733,125
45	3 14 9 7 12 15 0 4 1 8 5 6 11 10 2 13	39	51	6,158,733
46	8 4 6 1 14 12 2 15 13 10 9 5 3 7 0 11	35	49	22,119,320
47	6 10 1 14 15 8 3 5 13 0 2 7 4 9 11 12	35	47	1,411,294
48	8 11 4 6 7 3 10 9 2 12 15 13 0 1 5 14	39	49	1,905,023
49	10 0 2 4 5 1 6 12 11 13 9 7 15 3 14 8	33	59	1,809,933,698
50	12 5 13 11 2 10 0 9 7 8 4 3 14 6 15 1	39	53	63,036,422

GOAL STATE

```
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

LEGEND

ESTIMATE	Initial Heuristic Estimate
ACTUAL	Optimal Solution Length
NODES	Total states generated

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Table 1: Optimal solution lengths for 100 randomly generated Fifteen Puzzle instances using iterative-deepening-A* with Manhattan Distance heuristic function

NUMBER	INITIAL STATE	ESTIMATE	ACTUAL	NODES
51	10 2 8 4 15 0 1 14 11 13 3 6 9 7 5 12	44	56	26,622,863
52	10 8 0 12 3 7 6 2 1 14 4 11 15 13 9 5	38	56	377,141,881
53	14 9 12 13 15 4 8 10 0 2 1 7 3 11 5 6	50	64	465,225,698
54	12 11 0 8 10 2 13 15 5 4 7 3 6 9 14 1	40	56	220,374,385
55	13 8 14 3 9 1 0 7 15 5 4 10 12 2 6 11	29	41	927,212
56	3 15 2 5 11 6 4 7 12 9 1 0 13 14 10 8	29	55	1,199,487,996
57	5 11 6 9 4 13 12 0 8 2 15 10 1 7 3 14	36	50	8,841,527
58	5 0 15 8 4 6 1 14 10 11 3 9 7 12 2 13	37	51	12,955,404
59	15 14 6 7 10 1 0 11 12 8 4 9 2 5 13 3	35	57	1,207,520,464
60	11 14 13 1 2 3 12 4 15 7 9 5 10 6 8 0	48	66	3,337,690,331
61	6 13 3 2 11 9 5 10 1 7 12 14 8 4 0 15	31	45	7,096,850
62	4 6 12 0 14 2 9 13 11 8 3 15 7 10 1 5	43	57	23,540,413
63	8 10 9 11 14 1 7 15 13 4 0 12 6 2 5 3	40	56	995,472,712
64	5 2 14 0 7 8 6 3 11 12 13 15 4 10 9 1	31	51	260,054,152
65	7 8 3 2 10 12 4 6 11 13 5 15 0 1 9 14	31	47	18,997,681
66	11 6 14 12 3 5 1 15 8 0 10 13 9 7 4 2	41	61	1,957,191,378
67	7 1 2 4 8 3 6 11 10 15 0 5 14 12 13 9	28	50	252,783,878
68	7 3 1 13 12 10 5 2 8 0 6 11 14 15 4 9	31	51	64,367,799
69	6 0 5 15 1 14 4 9 2 13 8 10 11 12 7 3	37	53	109,562,359
70	15 1 3 12 4 0 6 5 2 8 14 9 13 10 7 11	30	52	151,042,571
71	5 7 0 11 12 1 9 10 15 6 2 3 8 4 13 14	30	44	8,885,972
72	12 15 11 10 4 5 14 0 13 7 1 2 9 8 3 6	38	56	1,031,641,140
73	6 14 10 5 15 8 7 1 3 4 2 0 12 9 11 13	37	49	3,222,276
74	14 13 4 11 15 8 6 9 0 7 3 1 2 10 12 5	46	56	1,897,728
75	14 4 0 10 6 5 1 3 9 2 13 15 12 7 8 11	30	48	42,772,589
76	15 10 8 3 0 6 9 5 1 14 13 11 7 2 12 4	41	57	126,638,417
77	0 13 2 4 12 14 6 9 15 1 10 3 11 5 8 7	34	54	18,918,269
78	3 14 13 6 4 15 8 9 5 12 10 0 2 7 1 11	41	53	10,907,150
79	0 1 9 7 11 13 5 3 14 12 4 2 8 6 10 15	28	42	540,860
80	11 0 15 8 13 12 3 5 10 1 4 6 14 9 7 2	43	57	132,945,856
81	13 0 9 12 11 6 3 5 15 8 1 10 4 14 2 7	39	53	9,982,569
82	14 10 2 1 13 9 8 11 7 3 6 12 15 5 4 0	40	62	5,506,801,123
83	12 3 9 1 4 5 10 2 6 11 15 0 14 7 13 8	31	49	65,533,432
84	15 8 10 7 0 12 14 1 5 9 6 3 13 11 4 2	37	55	106,074,303
85	4 7 13 10 1 2 9 6 12 8 14 5 3 0 11 15	32	44	2,725,456
86	6 0 5 10 11 12 9 2 1 7 4 3 14 8 13 15	35	45	2,304,426
87	9 5 11 10 13 0 2 1 8 6 14 12 4 7 3 15	34	52	64,926,494
88	15 2 12 11 14 13 9 5 1 3 8 7 0 10 6 4	43	65	6,009,130,748
89	11 1 7 4 10 13 3 8 9 14 0 15 6 5 2 12	36	54	166,571,097
90	5 4 7 1 11 12 14 15 10 13 8 6 2 0 9 3	36	50	7,171,137
91	9 7 5 2 14 15 12 10 11 3 6 1 8 13 0 4	41	57	602,886,858
92	3 2 7 9 0 15 12 4 6 11 5 14 8 13 10 1	37	57	1,101,072,541
93	13 9 14 6 12 8 1 2 3 4 0 7 5 10 11 15	34	46	1,599,909
94	5 7 11 8 0 14 9 13 10 12 3 15 6 1 4 2	45	53	1,337,340
95	4 3 6 13 7 15 9 0 10 5 8 11 2 12 1 14	34	50	7,115,967
96	1 7 15 14 2 6 4 9 12 11 13 3 0 8 5 10	35	49	12,808,564
97	9 14 5 7 8 15 1 2 10 4 13 6 12 0 11 3	32	44	1,002,927
98	0 11 3 12 5 2 1 9 8 10 14 15 7 4 13 6	34	54	183,526,883
99	7 15 4 0 10 9 2 5 12 11 13 6 1 3 14 8	39	57	83,477,694
100	11 4 0 8 6 10 5 13 12 7 14 3 1 2 9 15	38	54	67,880,056

GOAL STATE

```
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

LEGEND

ESTIMATE	Initial Heuristic Estimate
ACTUAL	Optimal Solution Length
NODES	Total states generated

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

IDA* generates more nodes than A*, it actually runs faster than A* on the Eight Puzzle, due to the reduced overhead per node.

In another set of experiments conducted by another researcher[1], IDA* was found to perform poorly on the Traveling Salesman Problem. The reason for this was that the number of new nodes added in each successive iteration grew very slowly. One means of rectifying this problem is that instead of increasing the threshold to the minimum value that exceeds the previous threshold, increase it by a sufficient amount to expand an exponentially growing number of new nodes. In that case, the algorithm cannot be terminated as soon as a goal is generated, but must complete that iteration and return the lowest cost goal found during the iteration, in order to guarantee optimal solutions. As long as the threshold increments are not too large, this doesn't seriously effect the performance of the algorithm. In fact, the next threshold can be determined dynamically during the previous iteration by recording how many nodes exceed different candidate next thresholds.

3.2.3 Move Ordering

An obvious technique to try to improve the performance of IDA* is called **move ordering**. The idea is that instead of expanding the successors of a given node in a fixed order, expand the successors of each node in increasing order of their heuristic values. This will not effect the number of nodes generated in any iteration prior to the final one, since these iterations must generate the entire search tree up to the threshold and hence the order is irrelevant. However, since the final iteration is terminated as soon as a goal node is expanded, move ordering may cause a goal to be found sooner, thus potentially speeding up the algorithm by a constant factor.

Unfortunately, the results of experiments on the Fifteen Puzzle are disappointing. The twenty-five easiest problems of the set of one hundred described above were run both with and without move ordering. In only half the instances was the number of nodes generated in the final iteration smaller with move ordering, with no average reduction in nodes expanded. Furthermore, the move ordering slowed down the algorithm by almost 50 per cent, with the net effect that in only one problem instance did the algorithm run faster with move ordering.

[1] Personal communication with Vipin Kumar, 1987.

Since the ordering decisions made high in the tree are the most important, the interpretation of this data is that the moves that look best at the beginning are not significantly more likely to remain the best further down the tree.

A more sophisticated form of move ordering would be to use the results of the previous iteration to order the moves in the next iteration. This requires saving some portion of the ordered tree from one iteration to the next, such as the path that led to the frontier node with the minimum h value. This path is then explored first in the next iteration. Experiments on the efficiency of this technique are currently underway.

3.3 Effectiveness of Heuristic Search

In general, the effect of the heuristic in these algorithms is to reduce the time complexity of the search from $O(b^d)$ to $O(c^d)$, where c is the heuristic branching factor. This results in a significant savings and allows somewhat larger problems to be solved. On the other hand, since the complexity of heuristic search is still exponential, the size of problems that it can solve effectively is severely limited. For example, on modern computers, brute-force techniques are sufficient to find optimal solutions to the Eight Puzzle, and heuristic search can find optimal solutions to the Fifteen Puzzle, but finding optimal solutions to typical instances of the 5x5 Twenty-Four Puzzle is beyond the range of the best known heuristic techniques. Thus, more powerful sources of knowledge are needed, at the expense of sacrificing optimal solutions.

4 SUBGOALS

It is well-known that the use of appropriate subgoals can greatly reduce the amount of search necessary to solve a problem. The reason is that decomposing an exponential problem into two or more simpler problems tends to divide the exponent, and hence drastically reduces the total problem solving effort[15]. In this section we make this notion precise and quantitative. We first consider subgoals in general and then examine three special cases: independent subgoals, serializable subgoals, and non-serializable subgoals.

The simplest case of the use of a subgoal involves an initial state i, a single goal state g, and a single intermediate state s. Without additional knowledge, the best problem solving algorithm is a brute-force search from i to s, followed by a brute-force search from s to g. If we define $d(x, y)$ to be the *distance* from state x to state y, or the length of the shortest sequence of operators that maps x to y, then the time complexity of our subgoal search is $O(b^{d(i,s)} + b^{d(s,g)})$ which is $O(b^{\max(d(i,s),d(s,g))})$. The length of the resulting solution is $d(i, s) + d(s, g)$. Thus, the subgoal will reduce search whenever $\max(d(i, s), d(s, g))$ is less than $d(i, g)$. The penalty for this search reduction, however, is that the solution length $d(i, s) + d(s, g)$ will in general be longer than the optimal solution length $d(i, g)$. Note that the optimal subgoal lies exactly halfway along an optimal solution path from i to g, since this cuts the exponent of b in half yet still results in an optimal solution.

4.1 Subgoal Distance

In general, a subgoal is not a single state but rather a property that is true of a number of states. For example, if we establish a subgoal for the Eight Puzzle of correctly positioning a particular tile, this subgoal is satisfied by any state in which that tile is in its goal position, regardless of the positions of the remaining tiles. Therefore, we formally define a subgoal to be a set of states, with the interpretation that a state is an element of a subgoal set if and only if it has the properties that satisfy the subgoal.

Correspondingly, we define the *subgoal distance* between subgoals S and T as

$$D(S, T) \equiv \max_{s \in S} \min_{t \in T} d(s, t).$$

The rationale for this definition is that the min captures the idea that we consider a subgoal achieved when we reach the first state which satisfies it, and max allows us to do a worst-case analysis over all possible states in the first subgoal. Note that in general, $D(S, T)$ is not equal to $D(T, S)$, due to the asymmetry of the definition.

The usual relationship between successive subgoals in a problem solving strategy is one of set inclusion. This is because latter subgoals in the sequence typically include the previous subgoals as well. For example, when we solve the Eight Puzzle one tile at a time, the

first subgoal is the set of all states where the first tile is correctly positioned, the second subgoal contains all states in which the first two tiles are correct, etc.

Given a sequence of subgoals $S_0, S_1, \ldots, S_n$ where S_0 contains just the initial state and S_n is the goal set, the amount of time to find a solution in the worst case is

$$b^{D(S_0,S_1)} + b^{D(S_1,S_2)} + \cdots + b^{D(S_{n-1},S_n)}$$

which is $O(b^{D_{\max}})$ where

$$D_{\max} \equiv \max_{0 \le i \le n-1} D(S_i, S_{i+1})$$

or the maximum subgoal distance. Thus, the complexity of solving a problem using subgoals depends on the maximum gap between two successive subgoals.

This is analogous to the plight of a hiker trying to cross a stream on stepping stones. He doesn't care about the length of the stream (the size of the problem space), nor is he is overly concerned with the width of the stream (the optimal solution length), nor the length of the actual path he must travel (the actual solution length). The parameter, however, that determines the difficulty of crossing the stream is the largest jump between two successive stepping stones (the maximum subgoal distance).

The worst case solution length using subgoals is the sum of the lengths of each segment of the solution, or

$$\sum_{i=0}^{n-1} D(S_i, S_{i+1}).$$

The **diameter** of a problem is the maximum distance between any two connected states in the problem space. This is equal to the worst case optimal solution length, for an arbitrary pair of initial and goal states. Since the problem diameter is an obvious upper bound on all subgoal distances, if we have n subgoals then the worst case solution length using subgoals is no more than n times the problem diameter.

Note that computing or even estimating subgoal distances is a hard problem. In principle, subgoal distances can be computed exactly by the formula which defines them, but in practice the amount of computation which is required may be prohibitive. In general, efficiently providing even reliable estimates of subgoal distances is often very

difficult. Note that heuristic evaluation functions, which estimate distance to a set of goal states, are the result of attempts to solve exactly this problem.

4.2 Subgoal Distance as a Predictor of Solution Lengths

The sum of the subgoal distances may be useful as a predictor of the number of moves that humans use to solve problems. In a series of experiments on human problem solving using the Eight Puzzle, Ericsson[16] found that the average number of moves required by ten human subjects to solve eight different problem instances was 38. If we choose the subgoals of positioning one tile at a time in numerical order, then the sum of the resulting subgoal distances (40 moves) is within five per cent of the average human solution lengths in his experiments. While 80 data points from a single problem are insufficient to validate subgoal distance as a predictor of human problem solving difficulty, the data is at least consistent with such a notion.

Given this general discussion of subgoals as a background, we now examine three special cases: independent subgoals, serializable subgoals, and non-serializable subgoals.

4.3 Independent Subgoals

Consider the following somewhat contrived example: You are given two different physical copies of the Eight Puzzle with different initial configurations, and are asked to transform each of them to a different goal state. Ignoring for a moment the obvious decomposition of this problem into subgoals, if the branching factor of each problem is b, the branching factor of the combined problem is $2b$ since an operator from either puzzle can be applied at any point. Similarly, if d_1 and d_2 are the respective distances of each initial state from its corresponding goal state, the depth of the combined problem is d_1+d_2 since this many operators must be applied to achieve the combined goal state. Thus, if we were to solve the combined problem directly without decomposing it into subgoals, its complexity would be $O((2b)^{d_1+d_2})$.

Obviously, a much better strategy for this problem would be to first solve one of the puzzles, ignoring the other, and then solve the remaining puzzle without disturbing the first. While the same number of moves $(d_1 + d_2)$ must be made in this case, the branching factor is

reduced from $2b$ to b since we only consider operators on one puzzle at any given time. This reduces the time complexity from $O((2b)^{d_1+d_2})$ to $O(b^{d_1} + b^{d_2})$ which is $O(b^{max(d_1,d_2)})$. Note that this subgoal decomposition results in an optimal solution since the length of the optimal solution for the compound problem is the sum of the optimal solution lengths for the subproblems.

The above subgoals are an example of *independent subgoals.* A collection of subgoals are independent if each operator only changes the distance to a single subgoal. In other words, we can partition the complete operator set into subsets such that the operators in a given subset are only relevant to a single subgoal.

One of the important properties of independent subgoals, which is clear from the definition, is that an optimal global solution can be achieved by simply concatenating together optimal solutions to the individual subproblems in any order. Even more important is the fact that independent subgoals reduce the branching factor by allowing the problem solver to focus on only a subset of the operators at any given time. In general, independent subgoals tend to divide both the base and the exponent of the complexity function by the number of subgoals.

Before the reader dismisses independent subgoals as trivial and uninteresting, it is worth noting that most goals that we try to satisfy in everyday life are in fact independent or nearly independent. If we consider a typical mix of goals that must be accomplished in a day, from domestic chores through job-related tasks to recreational and social objectives, we find that for the most part different operators apply to different problems, thus justifying the sequential strategy that we usually apply in our daily routine. The only reason that we can cope with these multiple goals at all is their relative independence. The major exception to independence of everyday tasks is that we are often constrained by global resource limits such as time or money. These constraints, however, only introduce dependence among subgoals when we begin to approach the limits they impose.

There are at least two reasons why independent subgoals have not received much previous attention. One is that we do not normally consider independent subtasks as part of the same problem, and another is that we tend to focus our attention on the more interesting case of interacting subgoals, such as serializable subgoals.

4.4 Serializable Subgoals

Even though two subgoals may not be independent, it may be possible to first solve one and then solve the other without ever violating the first. For example, if we solve the top row of the Eight Puzzle as a subgoal, then we can always solve the rest of the problem without disturbing the top row. We define a set of subgoals to be **serializable** if there exists an ordering among the subgoals such that the subgoals can always be solved sequentially without ever violating a previously solved subgoal in the order. In terms of our definition of a subgoal as a set of states, this means that for every state in a subgoal set, there exists a path into the next smaller subgoal set that lies entirely within the original subgoal set.

Note that serialization depends upon the actual order chosen. For example, if we were to first solve the bottom two rows of the Eight Puzzle, we would not in general be able to then solve the top row without disturbing the bottom two rows.

The idea of serializable subgoals comes from the General Problem Solver (GPS) of Newell and Simon[3]. Essentially, the serializable property of subgoals is equivalent to the existence of a triangular difference table for a problem. Solving a subgoal corresponds to removing a difference, and the ordering of the differences corresponds to the ordering of the subgoals that allows each to be solved without violating the proviously solved ones.

Unlike independence, the serializability of a set of subgoals is usually not obvious. For example, the subgoal of first solving a 2x2x2 corner subcube of the 3x3x3 Rubik's Cube allows the remainder of the problem to be solved without disturbing this subcube, but this fact is surprising to most people who can solve the puzzle. In general, proving that a sequence of subgoals is serializable is as difficult as proving that a given problem is solvable from all initial states. For example, if we swap two tiles of the Eight Puzzle by temporarily removing them from the frame, then the original congfiguration of the tiles can no longer be reached by legal moves, but this result was not known until long after the puzzle became popular[17].

Furthermore, the value of serializable subgoals in reducing search is not immediately obvious. The reason is that protecting previously solved subgoals may increase the overall solution length since certain operators cannot be applied at certain points. For example, first

solving the top row of the Eight Puzzle and then leaving it undisturbed while the remainder of the puzzle is solved in general requires more moves than solving the main goal directly. In fact, just solving the bottom two rows of the puzzle without disturbing the top row could require more moves on the average than solving the entire puzzle.

Why then do we preserve serializable subgoals? The reason is that it reduces the branching factor of the space. By protecting previously solved subgoals, the number of legal moves that remain is decreased because operators that violate the subgoals are ruled out. This effect usually more than compensates for the increase in solution length. For example, in Rubik's Cube, protecting a 2x2x2 subcube and twisting only the remaining three faces increases the number of moves required to correctly position the last piece of the puzzle from twelve to fifteen moves in the worst case. This additional search depth, however, is more than compensated for by the fact that protecting the 2x2x2 subcube cuts the branching factor in half.

4.5 Non-Serializable Subgoals

It is often the case that given a collection of subgoals, previously satisfied subgoals must be violated in order to make further progress towards the main goal, regardless of the solution order. Such a collection of subgoals will be called *non-serializable*. In terms of our subset model, non-serializability of a pair of subgoals means that for some state in the first subgoal set, all paths from that state into the second subgoal state temporarily pass outside of the the first subgoal set, even when the second subgoal set is contained in the first.

A classic example of this phenomenon is Rubik's Cube: once part of the puzzle is solved, in general it must be messed up, at least temporarily, in order to make further progress. Non-serializable subgoals also occur in the Eight Puzzle: if the first two tiles in any row are correctly positioned, in general they must be moved in order to correctly position the third tile. Perhaps the simplest example of non-serializable subgoals comes from the blocks world and is commonly known as Sussman's anomaly[18] (see Figure 4). Given a description of a stack of blocks as the conjunction of (ON A B) and (ON B C), regardless of which order we try to solve these subgoals, the first subgoal must be violated in order to solve the second. In this case, however, the non-serializability is not an inherent property of the problem, but

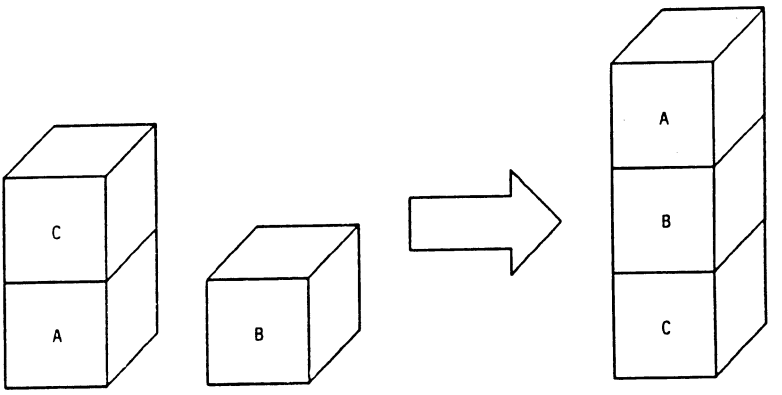

Figure 4: Sussman's blocks world anomaly

an artifact of this particular goal description. If we describe the goal as the conjunction of (ON C Table), (ON B C), and (ON A B), then these three subgoals are serializable. This is in stark contrast with the Rubik's Cube, where careful scrutiny by literally millions of people has failed to yield a sequence of serializable subgoals.

A fair bit of attention has been focussed on the problem of non-serializable subgoals, mostly in the blocks world domain. Waldinger[19] provides a good survey of this research. The essential technique employed was to start with a plan for achieving the subgoals independently, and then modify that plan to deal with the subgoal interactions. This approach assumes that the subgoals are approximately independent. By contrast, we make no such assumption and instead use a full-fledged search to solve a sequence of subgoals where each subgoal includes all the previous subgoals as well.

Non-serializable subgoals do not decrease the branching factor since we must allow any operator to be applied at any point in the solution. Furthermore, passing through these subgoals will have the same effect of increasing solution lengths as do subgoals in general. Given these two facts, one might reasonably ask what the value of non-serializable subgoals is. There are two answers to this question, depending on whether the context is simple problem solving or machine learning. The first will be addressed below and the second will be deferred until section 5.

In a pure problem solving context where the goal is simply to solve a single instance of a problem, the value of non-serializable subgoals is that states which satisfy them are often closer to the goal state on

the average than arbibtrary states, even though the subgoal property cannot be preserved through the rest of the solution. The intuitive reason for this is that if our main goal is a conjunction of subgoals, we expect a state in which most of the subgoals are satisfied to be closer to the goal state than a state in which few subgoals are satisfied. Thus, satisfying even non-serializable subgoals usually constitutes progress toward the goal in terms of the number of remaining moves. For example, it turns out that a state of the 3x3x3 Rubik's Cube in which almost all the components are correctly positioned is closer to the goal state on the average (12 moves) then an arbitrary state ($\geq$ 18 moves), even though the correct pieces must be temporarily moved in order to make further progress. This is also the case with the subgoals of solving the Eight Puzzle one tile at a time since the maximum subgoal distance is only fourteen moves[20] while the maximum distance from the goal state is 30 moves[21]. Note that these are simply empirical results derived from the work described in section 5, and is not always the case, as we will see.

4.6 Block-Serializable Subgoals

Often a sequence of non-serializable subgoals can be made serializable simply by grouping together multiple subgoals into a single subgoal. Such a sequence of subgoals will be called *block-serializable subgoals.* As was mentioned above, the Eight Puzzle subgoals of positioning the tiles one at a time are not serializable since once the first two tiles in a row have been placed, in general they must be moved in order to place the third. We also pointed out, however, that once the entire top row is correctly positioned, the remainder of the puzzle can be solved without disturbing the top row. Thus, even though the individual subgoals are not serializable, they can be grouped together into blocks which are serializable. For example, for the goal state shown in Figure 1, the blocks (1), (2, 3), (4, 5), and (6, 7, 8) form a sequence of serializable subgoals in the sense that once **all** the tiles in a block are correctly placed, they need not be disturbed to complete the remainder of the puzzle. Thus, we find that serializability is often a function of the granularity of the set of subgoals. The tradeoff is that increasing the granularity to achieve serializability also increases the resulting subgoal distances.

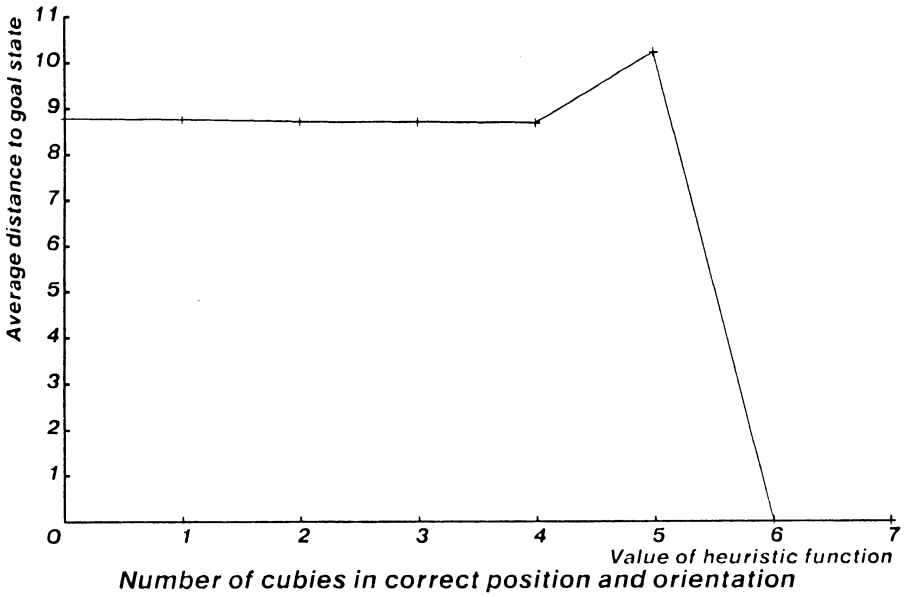

Figure 5: Average distance to goal state vs. number of cubies in correct position and orientation for 2x2x2 Rubik's Cube

4.7 Pathological Subgoals

We stated above that non-serializable subgoals often improve problem solving efficiency by moving the problem solver closer to the goal, and gave Rubik's Cube and the Eight Puzzle as examples. This is not always the case, however. For some problems, solving subgoals does not decrease the distance to the main goal. For example, the obvious subgoals for solving Rubik's Cube are to position the individual components one at a time. Figure 5 shows a graph of the number of components which are correctly positioned versus the average distance to the goal state for the smaller 2x2x2 version of Rubik's Cube. Each data point shows the number of moves to the goal state via the shortest path, averaged over all states with the given number of components correctly positioned. One would expect the distance to the goal to decrease as more of the puzzle is solved.

What we find, however, is that as more components are positioned, the average distance to the goal remains constant. Even more surprisingly, the states in which the puzzle is almost completely solved are further from the goal than the average state! Such a sequence of sub-

goals may be termed **pathological**. When people solve this puzzle, however, they pass through these states on their way to the solution. This raises the following question: by what measure do these subgoals represent progress towards the main goal?

5 MACRO-OPERATORS

The reason that human problem solvers establish and solve these subgoals is that they know what sequence of operators to apply to solve the next subgoal from those states. A sequence of primitive operators is called a **macro-operator**. Progress toward the solution in this context is getting to a state from which the problem solver knows macro-operators that will achieve the next subgoal. Unfortunately, this explanation begs the question since it doesn't address where the macro-operators come from in the first place. In order to address that issue, we must broaden our perspective from pure problem solving to include learning behavior as well.

Strictly speaking, problem solving involves finding a solution to a particular problem instance. Learning, on the other hand, involves a collection of problem instances with some common structure. It may be a single problem instance that is to be solved repeatedly or a collection of related problem instances that are to be solved. In the former case, rote memorization of the solution is sufficient for learning. In the latter case, there must be some common structure to the collection of problem instances such that the fixed cost of the learning plus the marginal cost of solving each problem instance using what was learned is less than the total cost of solving each instance from scratch. In essence, the learning cost is amortized over all the problem instances to be solved.

In the case of a problem such as Rubik's Cube, the task of the human problem solver is not just to solve a particular instance of the problem, but rather to learn a general strategy for solving the problem from any given initial state. In fact, the problem is so difficult that an average problem instance cannot be solved without learning a general strategy. What people learn in order to solve this problem is a collection of macro-operators. Since an individual macro-operator can be used to solve more than one problem instance, the computational cost of learning a macro is amortized over all the problem instances

for which it is useful.

5.1 Macros for Non-Serializable Subgoals

In particular, the macros that are useful for non-serializable subgoals are those that leave previously satisfied subgoals intact, even though they may violate them temporarily. In other words, during the application of the macro, the previous subgoals may be disturbed, but by the end of the macro application, all previously satisfied subgoals will be reestablished.

For example, consider the table of macro-operators for the Eight Puzzle in Table 2, corresponding to the goal state in Figure 1. Each macro-operator is a sequence of primitive moves represented by the first letter of **l**eft, **r**ight, **u**p, or **d**own depending upon which direction a tile is moved. This table of macros can be used to solve the puzzle from any initial state with no search, as follows:

First, locate the blank (or zero tile) in the initial state and note its position. (The label for any given position corresponds to the number of the tile that occupies that position in the goal state.) The position of the blank in the initial state is used as a row index into the 0^{th} column of the macro table to select a particular macro-operator. Applying that macro-operator will correctly move the blank to its goal position in the center of the frame. After that macro is applied, the resulting position of the one tile is used as a row index into the first column of the macro table to select the next macro-operator to be applied. The application of that macro will move the one tile to its correct position, and also return the blank tile to its goal position as well. Similarly, the resulting position of the two tile is used as a row index into the second column of the macro table to select a macro that will correctly position the two tile and also return the one and blank tiles to their goal positions. In general, at the *nth* stage of the algorithm, the position of the *nth* tile is used as a row index into the *nth* column of the macro table to select a macro that will correctly position the *nth* tile, and also restore the previous $n-1$ tiles to their goal positions.

Note that the effect of introducing these macro-operators is to serialize the subgoals of solving one tile at a time. While these subgoals are not serializable with respect to the primitive operator set, they

		TILES						
		0	1	2	3	4	5	6
P	0							
O	1	ul						
S	2	u	rdlu					
I	3	ur	dlurrdlu	dlur				
T	4	r	ldrurdlu	ldru	rdllurdrul			
I	5	dr	uldrurdldrul	lurdldru	ldrulurddlur	lurd		
O	6	d	urdldrul	ulddru	urddluldrrul	uldr	rdlluurdldrrul	
N	7	dl	rulddrul	druuldrdlu	ruldrdluldrrul	urdluldr	uldrurdllurd	urdl
S	8	l	drul	rullddru	rdluldrrul	rulldr	uldrruldlurd	ruld

Table 2: Macro table for the Eight Puzzle

are serializable with respect to this complex set of macro-operators. In general, the serializability of a set of subgoals depends upon the operator set to be applied to the problem.

The power of this approach lies in the fact that all 181,440 solvable initial states of the Eight Puzzle can be solved without any search using only the 35 macros shown. Such a table for the 4×10^{19} initial states of the 3x3x3 Rubik's Cube requires only 238 macros. The reason is that each macro can usefully be applied to a large number of different states. This in turn is due to a structural property of these problems called **operator decomposability**.

If we represent the state of a problem by a vector of state variables, each one of which can take on a number of different values, then **total operator decomposability** requires that the effect of each operator on each state variable be a function only of that state variable. For example, in Rubik's Cube the state variables correspond to the individual movable components of the puzzle, and the values correspond to the different positions and orientations that the components can be in. Rubik's Cube is totally decomposable since the effect of any operator on the position and orientation of any component depends only on the position and orientation of that component and is independent of the other components of the puzzle. **Serial operator decomposability** is a more general form of operator decomposability and requires only that there exist some ordering of the state variables for which the effect of each operator on each state variable depends only on that state variable and previous state variables in the ordering. For example, the Eight Puzzle is serially decomposable as long as the blank tile comes first in the ordering.

If a problem exhibits serial decomposability, which includes total decomposability as a special case, then a macro table similar to the one in Table 2 exists for the problem. If n is the number of state variables and k is the number of different values that each state variable could be assigned, then in general the number of macros in the table is $O(nk)$ whereas the number of states in the problem space is $O(k^n)$. Note that in the case of the Eight Puzzle, the states are actually permutations of the tiles so the number of states is actually $O(k!)$. Since n and k are the problem size parameters, if we hold n fixed and allow k to grow, the number of macros grows linearly, the number of states grows polynomially, and hence the number of macros grows as the n^{th} root of the number of states. Alternatively, if we hold k fixed and allow n to grow, the number of macros still grows linearly, but the number of states grows exponentially, and the number of macros grows only as the log of the number of states.

Thus, operator decomposability allows an exponential number of problem instances to be solved without any search, using only a linear amount of knowledge expressed as macro-operators. The amount of time required to learn the macros is $O(b^{D_{max}})$, where the subgoals are to map each state variable to its goal value. This is the same amount of time that would be required to solve a single instance of the problem using the same subgoals, with the difference being that in this case, the learning only has to be done once for all possible problem instances. The solution lengths resulting from this technique are simply the sum of the subgoal distances. A complete treatment of serially decomposable problems, including the formal definition of the property and the macro learning algorithm, can be found in [20].

5.2 Macros in Arbitrary Single-Goal Problem Spaces

While macro-operators are particularly powerful for serially decomposable problems, they can be used to effectively reduce search in arbitrary problem spaces as well. We will first address the special case of problems with a single common goal state and then consider finding paths between arbitrary pairs of initial and goal states.

For example, consider the problem of navigating to your home along a network of roads. The states of this problem are different locations. If we adopt the definition of a primitive operator as one that does not admit intermediate states from which other operators can be

applied, then the primitive operators in this space are sections of road between adjacent intersections. Clearly, we don't solve the problem of finding our way home from scratch every time we do it. Rather, with practice we learn enough so that the problem is normally solved with no search. Most of the knowledge in this task can be represented as macro-operators which indicate which sequence of roads to take to get home from various locations. Since different macros will share common paths, particularly as they get close to the goal, the entire set of macros can be viewed as a tree structure, rooted at the goal, where each branch has associated with it the direction to the goal.

Problem solving using this macro network consists of two steps: first search from the initial state to a state on the macro network, and then follow the macro network to the goal state. This reduces the search time at the expense of the space to store the macro network. In fact, we can quantify this space-time tradeoff.

Assume that the states in the macro network are uniformly distributed over the problem space so that the average distance from an arbitrary state to the closest state in the macro network is constant over the whole space. Actually, all that is required is that the distribution of the macro network match the distribution with which initial states are selected. Furthermore, we assume that a state on the macro network can be recognized in constant average time. This can be accomplished by hashing the states in the macro network. Finally, we assume that finding a path to the goal from any state in the macro network requires no search. This would be the case if every state in the network had associated with it the operator to apply to move toward the goal state.

Given these assumptions, let n be the total number of states in the space and let $\frac{n}{k}$ be the number of states in the macro network, with k being the ratio of the size of the space to the size of the macro network. The probability that any given state is in the macro network is therefore $\frac{1}{k}$. Thus, starting from an arbitrary initial state, the expected number of states we would have to examine before finding one in the macro network is k. Since search time is proportional to the number of states explored and no search is required within the network itself, the total search time is $O(k)$ in the average case. The amount of space required to store the macro network is linear in the number of states in the network since only the operator which moves

toward the goal need be stored with each state. Thus, the space required is $O(\frac{n}{k})$. Note that the product of the space to store the network ($\frac{n}{k}$) and the time to search for a path from the initial state to the macro network (k) equals the number of states in the original space (n), independent of the size of the macro network. In other words, there is a multiplicative tradeoff between the size of the macro network and the amount of search required to solve problem instances in the resulting space.

5.3 Macros in Arbitrary Path-Finding Problems

The above analysis dealt only with the case of a single common goal state. We now relax that assumption and consider the use of macro networks to find paths between arbitrary pairs of initial and goal states. One solution to this problem is the following: First find a path from the initial state to the goal state of the macro network as above. Next, find a path from the actual goal state to the goal state of the network. Then, the first path followed by the inverse of the second path is a path from the initial state to the actual goal state, assuming that the operators are invertible.

A practical example of this approach is the system used by most overnight couriers. All packages are flown from their sources to a single common airport near the population center of the country, resorted, and then flown from this central hub to their destinations. This allows a quadratic number of source-destination pairs to be served by a linear number of flights.

The disadvantage of this approach is that the length of the path from initial state to goal state is proportional to the size of the problem space and not to the distance between the states. An overnight letter doesn't mind if it goes through Memphis to get from Los Angeles to San Francisco, but a passenger certainly would.

An alternative approach requires that the macro network be structured as a more highly connected graph rather than a tree. Problem solving then involves three steps: 1) finding a path from the initial state to a state on the macro network, 2) finding a path from the goal state to a state on the macro network, and finally 3) finding a path within the macro network between these two states on the macro network. In this case, the solution length depends on the distance between initial and goal states and on the density of the macro network,

but is independent of the size of the problem space. The computational advantage of this approach over simply solving the problem directly in the original space is that much of the search will occur in the macro network, and since this network is sparser than the original space, this search will be more efficient. This idea is known as **abstraction.**

6 ABSTRACTION

The value of abstraction is well-known in artificial intelligence. The basic idea is that in order to efficiently solve a complex problem, a problem solver should at first ignore low level details and concentrate on the essential features of the problem, filling in the details later. The idea readily generalizes to multiple hierarchical levels of abstraction, each focused on a different level of detail. Empirically, the technique has proven to be very effective in reducing the complexity of large problems.

Like many ideas in AI, the value of abstraction in human problem solving was pointed out by George Polya in *How to Solve It*[22]. The first explicit use of abstraction in an AI program was in the planning version of GPS[3]. The most thorough exploration of abstraction to date is Sacerdoti's work on the ABSTRIPS system[23].

This section presents a quantitative analysis of abstraction in problem solving. The essential reason that abstraction reduces complexity is that the total complexity is the sum of the complexities of the multiple searches, and not their product[15]. Our goal is to formalize and quantify this intuitive explanation. The questions we address are: how much search efficiency is gained by the use of abstraction, and what is the optimum level of detail for each level of abstraction. To do so, we first formalize a model of abstraction. Next, we consider the special case of a single level of abstraction. Finally, we turn our attention to the general case of multiple abstraction levels. The analysis is done in the average case. The main result is that an abstraction hierarchy can reduce the amount of search to solve a problem from linear in the size of the problem space to logarithmic. A practical result of the analysis is that many levels of abstraction, with only small differences between them, reduce search the most.

6.1 A Model of Abstraction in Problem Solving

We model the states of an abstract space as a subset of the states in the original problem space (the base space). For example, in the road navigation problem, the states are street intersections and the operators are sections of road between intersections. In this domain, a suitable set of abstract states would be the set of major intersections, say at the centers of towns. An alternative model is for each state of the abstract space to correspond to a subset of the states in the base space. In the navigation example, a state of the abstract space would then correspond to a region in the base space. In many cases, however, operators only apply to particular states as opposed to sets of states. For example, while an interstate highway may serve an entire area, it can only be accessed from particular points in that area. In any case, the main difference between the subset and region models is that in the region model, no effort is required to get into the abstract space, since any state in a region is already a member of the region as a whole. This reduces the search complexity by a constant factor, and for this reason we adopt the more restricted subset model.

The operators of the abstract space map states in the abstract space to other states in the abstract space. In the case of GPS and Sacerdoti's work, the abstract operators were a subset of the primitive operator set that mapped abstract states to abstract states. In the transportation example, the operators of the abstract space would be direct means of transportation between major cities, such as interstate highways. Alternatively, the abstract operators may be macro-operators that go between abstract states. A necessary property of the macro-operators is that they be stored or otherwise known to the system and do not require search to find. In our example of road navigation, the abstract operators might be driving routes between major cities that are sequences of different roads marked by signs. Regardless of which type of abstract operators we choose, the assumption of known operators between abstract states is an important aspect of our model. An abstract space requires both a set of abstract states and a set of abstract operators between those states. If paths between the abstract states must be found by search among the operators of the base space, then abstraction by itself makes no improvement in search efficiency. For example, if we have to search for routes between nearby major cities, then the abstraction of major cities is of no use

in routefinding. Note that we do not require an abstract operator between every pair of abstract states, but simply that the set of abstract states be connected by abstract operators.

For purposes of this analysis, we assume that the relative distribution of abstract states over the base space, while not necessarily uniform, is the same for all levels of abstraction. In our example, this corresponds to the observation that while there are more towns in the Eastern U.S. than in the West, the relative distribution of large cities is roughly the same as that for small cities. An average case analysis can be done independent of the actual distribution, as long as the distribution remains constant over different levels of abstraction.

6.2 Single Level of Abstraction

We begin our analysis with the special case of a single level of abstraction. As in the case of the macro network, let the number of states in the base space be n and the number of states in the abstract space be $\frac{n}{k}$, with k being the ratio of the size of the base space to the size of the abstract space. The issue we address is the expected amount of search required to find a solution to an average problem, using the abstraction. By average problem, we mean one where the initial and goal states are randomly selected from all states with equal probability.

We assume that the amount of time to search for a solution is proportional to the number of states visited in the search. The expected number of states visited is the sum of the expected number of states visited in the three phases of the search: getting into the abstract space, searching in the abstract space, and getting to the actual goal in the base space. Since we assume that the distribution of states in the abstract space is the same as in the base space, the probability that any randomly selected state is a member of the abstract space is $\frac{1}{k}$. Thus, the expected number of states that must be generated in a search before a state in the abstract space is encountered is k. This is both the expected number of states to be searched in going from the initial state to the abstract space and also in going from the abstract space to the goal state, since this latter search can be performed by searching from the goal state to the abstract space.

Since the total number of abstract states is $\frac{n}{k}$, to find a path between an arbitrary pair of abstract states, we would expect to have to

examine $\frac{n}{2k}$ states on the average. Thus, the total expected number of states expanded is $t = 2k + \frac{n}{2k}$. Note that this is actually an upper bound on the expected number of states because it ignores the slight possibility that the goal state will be found before the abstract space is encountered when searching from the initial state.

What value of k minimizes this total search time? The minimum occurs at $k = \frac{\sqrt{n}}{2}$, or $\frac{n}{k} = 2\sqrt{n}$, giving a value of $t = 2\sqrt{n}$. Thus, for a single level of abstraction, if the base space is of size n, the optimum size for the abstract space is on the order of $\sqrt{n}$. This abstraction reduces total search time from $O(n)$ without any abstraction, to $O(\sqrt{n})$.

6.3 Multiple Levels of Abstraction

We now consider the general case of multiple hierarchical levels of abstraction. In order to solve a problem in a hierarchy of abstraction spaces, we first map the initial state to the nearest state in the first level of abstraction, then map this state to the nearest state in the second level of abstraction, etc., until the highest level of abstraction is reached. At the same time, this process is repeated starting from the goal state and working up through successive levels of abstraction until the paths from the initial and goal states meet at the highest abstraction level. The questions we want to answer are how much do multiple levels of abstraction reduce search, how many levels of abstraction should there be, and what should the ratios between their sizes be in order to minimize the search time to solve an average problem.

Again, let the number of states in the base space be n. Let k_1 be the ratio between the size of the base space and the size of the first level of abstraction, k_2 be the ratio of the sizes of the first and second levels of abstraction, etc. In order to make the formular for the search complexity at the top level the same as in the lower levels, let the highest level of abstraction consist of a single state. The task at the top level then is to find paths from both the initial and goal states to this single common state. With this slight simplification, the expected amount of time to find a solution to an average problem in this hierarchy of abstraction spaces is $2k_1 + 2k_2 + \cdots + 2k_m$, where m is the number of levels of abstraction, since two searches must be made at each level to find a state in the next higher level, one coming from the initial state and the other from the goal state. As in the case of

a single abstraction, this is actually an upper bound on the expected time since it ignores the possibility that the actual goal state will be found before the highest level of abstraction is reached.

In order to minimize this expression, we must minimize the sum of the k_is. The constraint on the k_is is that their product must equal n, since they represent the ratios between the number of states at each level and n is the total number of states. Thus, the problem becomes one of factoring a number such that the sum of the factors is minimized, or in other words, finding a *minimum sum factorization.*

If the factors are constrained to be integers, then the prime factorization of n is a minimum sum factorization. However, in our case the k_is are ratios which need not be integral but rather can be arbitrary rational numbers. In that case, the minimum sum factorization of a number n consists of $\ln n$ factors, each of which is equal to e. To see this, first note that all the factors must be equal, because two unequal factors could each be replaced by the square root of their product, which would reduce the sum without changing the product. This implies that there must be $log_k n$ factors of k, and minimizing their sum, $k \log_k n$, yields $k = e$.

This implies that the optimum abstraction hierarchy for a problem space with n states consists of $\ln n$ levels of abstraction and that the ratio of the sizes of successive levels of abstraction is e. The average case time to find a solution given such an optimum abstraction hierarchy is proportional to $2k_1 + 2k_2 + \cdots + 2k_m$ or $2e \ln n$. Thus, the use of such an abstraction hierarchy can reduce the expected search time from $O(n)$ to $O(\log n)$. This improvement makes combinatorial problems tractable. For example, if n is an exponential function of problem size, then $logn$ is linear.

This same result was arrived at by Kleinrock and Kamoun[24] in the context of a closely related but somewhat different problem, that of hierarchical routing in large computer networks. In their problem, no search is performed, but each node stores a routing table which gives a destination node for each cluster at each level of abstraction. Their problem of minimizing the storage space for these routing tables turns out to be equivalent to minimizing search time in our model.

In practice, there are a number of different factors that constrain the design of abstraction spaces, and hence an optimal abstraction hierarchy in the above sense will not be achievable in general. The

point of the exercise is to illustrate the analysis of such a hierarchy and to see how much abstraction can reduce search complexity in the best case. The result does suggest, however, that in general a deep abstraction hierarchy, in terms of number of levels, will reduce search more than a shallow one.

The only quantitative empirical data in the literature on the use of abstraction in problem solving is Sacerdoti's comparison of STRIPS and ABSTRIPS on five different problems. For these problems, ABSTRIPS provided an approximately logarithmic speedup over STRIPS in time to find a solution. While five data points are too small a sample to validate the model, Sacerdoti's data is at least consistent with our theory.

6.4 Space Requirements of Abstraction

What about the space required to store abstract problem spaces? According to the model we used for the macro network, the amount of space for an abstract space would be proportional to the number of abstract states. Thus, the optimal single level of abstraction would require $O(\sqrt{n})$ space and the optimal abstraction hierarchy would require $O(n)$ space, just for the first level.

In reality, however, the actual space requirements for abstraction are usually insignificant. In order to see this, it is necessary to distinguish two different kinds of problems: extensional and intensional. An extensional problem is one in which all the states of the problem space have an **extension** or exist in the world. For example, road navigation is an extensional problem since all the states are actual physical locations. An intensional problem, on the other hand, is characterized by a small set of rules which generate the entire problem space. The Eight Puzzle is an example of an intensional problem since all the states don't actually exist at once but rather are generated by the set of legal moves.

In an extensional problem the space requirement of an abstract problem space is negligible. This is because the space required is no more than linear in the number of states, and all the states already exist in the world. The small additional space is easily absorbed in the space required to represent the problem space. For example, in road navigation abstract states and abstract operators can be created simply by putting up road signs. Furthermore, the same abstractions

can be used by any problem solver for any problem instance and hence the space cost is amortized over all problem solvers and all problem instances. Thus, while road signs indicating the direction to my house are not practical, more general road signs are cost effective.

In the case of intensional problems, the additional space requirements are often avoided by creating abstract problems which are also intensional. For example, an abstract problem space for the Eight Puzzle can be created by ignoring the positions of some of the tiles. The states of the resulting abstract problem need not be represented explicitly, any more than those of the original problem, but rather implicitly by the legal moves. Similarly, in the case of ABSTRIPS, the abstract problems were created by ignoring certain details of the states. In general, when confronted with an intensional problem we design intensional abstractions.

Thus, in both extensional and intensional problems we find that in practice we incur no significant space overhead, albeit for different reasons in the two cases.

7 CONCLUSIONS

The standard algorithms for brute-force search have serious drawbacks. Breadth-first search uses exponential space, and exhausts the available memory on typical computer configurations in a matter of seconds in practice. Depth-first search may use far more time than breadth-first search, and is not guaranteed to find a shortest path to a solution. Depth-first iterative-deepening, however, finds optimal solutions, uses only linear space, and takes no more time, asymptotically, than breadth-first search on a tree. In fact, DFID is asymptotically optimal in terms of time and space, over all brute-force tree searches that are guaranteed to find optimal solutions.

DFID can also be applied to bi-directional search, heuristic best-first search, and two-person game searches. Since A* is a best-first search, it suffers from the same memory restriction as breadth-first search. IDA* is asymptotically optimal in time and space among exponential heuristic tree searches that find optimal solutions. Since almost all heuristic searches have exponential complexity, iterative-deepening-A* is an optimal tree search in practice. An additional benefit is that IDA* is easier to implement, and actually funs faster

than A* due to reduced overhead per node. Empirically, IDA* is the only known algorithm that can find optimal paths for randomly generated instances of the Fifteen Puzzle within practical time and space constraints.

In addition to heuristic evaluation functions, other sources of knowledge that can be used to reduce search complexity include subgoals, macro-operators, and abstraction spaces. We have taken the first steps in the quantitative analysis of problem solving performance using these sources of knowledge.

In the case of subgoals, we identified a measure called the subgoal distance, which is the parameter that determines, along with the branching factor, the time complexity of a search using subgoals. We then explored independent, serializable, and non-serializable subgoals in turn. We found that independent subgoals divide the branching factor of a problem, serializable subgoals reduce the branching factor by ruling out certain operators, and that solving non-serializable subgoals may reduce the distance to the main goal.

Even in those cases where the distance to the goal is not reduced, it may be worthwhile to solve non-serializable subgoals if the problem solver has available macro-operators which will achieve the goal from those subgoals. In the special case of serially decomposable problems, we found that exponentially increasing numbers of initial states can be solved by storing only logarithmically more macro-operators. In the more general case of single goal problems, a macro network provides a multiplicative tradeoff between search time and storage space for macros. This can be extended to handle arbitrary initial and goal states by a central hub approach, but the penalty is that solution lengths are then proportional to the problem diameter rather than the distance between initial and goal states.

A more highly connected macro network for dealing with the general case of arbitrary pairs of initial and goal states amounts to an abstract problem space. We found that for a single level of abstraction, the optimal size for an abstract space is the square root of the size of the base space and that this reduces search from linear in the size of the space to the square root of the size of the space. In the case of multiple hierarchical levels of abstraction, we found that the optimal hierarchy has a logarithmic number of levels with a constant ratio between their sizes and that this hierarchy reduces search from linear

in the number of states in the problem space to logarithmic. Since the number of states is often exponential in the problem size, this explains why abstraction hierarchies can reduce exponential complexity to linear complexity.

8 ACKNOWLEDGMENTS

Judea Pearl originally suggested the application of iterative-deepening to A*. I would like to thank him and Herbert Simon for many helpful discussions concerning this research. Andy Mayer implemented the A* algorithm that was compared with IDA*. An anonymous referee suggested the shortcomings of depth-first search on a graph with cycles. Vipin Kumar pointed out the limitations of synthesized monotone heuristic functions, and that IDA* may perform poorly on problems such as TSP.

References

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

[2] Korf, R.E., Planning as search: A quantitative approach, *Artificial Intelligence*, 1987.

[3] Newell, A., and H.A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J., 1972.

[4] Pohl, I., First results on the effect of error in heuristic search, in *Machine Intelligence 5,* B. Meltzer and D. Michie (eds.), American Elsevier, New York, 1970, pp. 219-236.

[5] Slate, D.J., and L.R. Atkin, CHESS 4.5 - The Northwestern University chess program, in *Chess Skill in Man and Machine*, Frey, P.W. (Ed.), Springer-Verlag, New York, 1977.

[6] Berliner, H., Search, in Artificial Intelligence Syllabus, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 1983.

[7] Winston, P.H., *Artificial Intelligence*, Addison-Wesley, Reading, Mass., 1984.

[8] Hopcroft, J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.

[9] Stickel, M.E., and W.M. Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, Ca., August, 1985.

[10] Pohl, I., Bi-directional search, in *Machine Intelligence 6*, Meltzer, B. and D. Michie (Eds.), American Elsevier, New York, 1971, pp. 127-140.

[11] Pearl, J., *Heuristics*, Addison-Wesley, Reading, Ma., 1984.

[12] Pearl, J., personal communication, 1984.

[13] 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*, SSC-4, No. 2, 1968, pp. 100-107.

[14] Dechter, R., and J. Pearl, Generalized best-first search strategies and the optimality of A*, *Journal of the Association for Computing Machinery*, Vol. 32, No. 3, July 1985, pp. 505-536.

[15] Minsky, M., Steps toward artificial intelligence, in *Computers and Thought*, Feigenbaum and Feldman (eds.), McGraw-Hill, New York, 1963.

[16] Ericsson, K.A., Approaches to descriptions and analysis of problem solving processes: The 8-puzzle, Ph.D. Thesis, University of Stockholm, 1976.

[17] Johnson, W.A. and W.E. Storey, Notes on the 15 puzzle, *American Journal of Mathematics*, Vol. 2, 1879, pp. 397-404.

[18] Sussman, G.J., *A Computer Model of Skill Acquisition,* American Elsevier, New York, 1975.

[19] Waldinger, R., Achieving several goals simultaneously, in *Readings in Artificial Intelligence*, N.J. Nilsson and B. Webber (Eds.), Tioga, Palo Alto, Ca., 1981, pp. 250-271.

[20] Korf, R.E., Macro-operators: A weak method for learning, *Artificial Intelligence*, Vol. 26, No. 1, 1985, pp. 35-77.

[21] Schofield, P.D.A., Complete solution of the eight puzzle, in *Machine Intelligence 3,* B. Meltzer and D. Michie (eds.), American Elsevier, New York, 1967, pp. 125-133.

[22] Polya, G., *How to Solve It*, Princeton University Press, Princeton, N.J., 1945.

[23] Sacerdoti, E.D., Planning in a hierarchy of abstraction spaces, *Artificial Intelligence*, Vol. 5, 1974, pp. 115-135.

[24] Kleinrock, L., and Kamoun, F., Hierarchical routing for large networks, *Computer Networks* **1** (1977) 155-174.

[25] Mero, L., A heuristic search algorithm with modifiable estimate, *Artificial Intelligence*, Vol. 23, 1984, pp. 13-27.