 Rev. October, 1992

Graph Search and STRIPS Planning

AI is founded upon that belief that intelligence is based on computation. It seems likely that a deep understanding of computation will prove useful in the investigation of intelligence. In any computational process efficiency is desirable. This chapter continues the investigation of techniques for improving the efficiency of computations that search.

The theory of computation provides a framework for classifying computations that search, i.e., those computations which can not be done in polynomial time. As noted in earlier chapters, there appears to be a connection between the intuitive classification of search problems and formal complexity classes. Constraint satisfaction corresponds to the complexity class NP. On the other hand, graph search corresponds to the complexity class PSPACE. Graph search problems seem to be quite different from constraint satisfaction problems.

This chapter describes the most pragmatically effective known algorithms for solving graph search problems. This chapter also introduces the closely related problem of STRIPS planning along with simple planning algorithms. Just as SAT can be viewed as a prototypical NP complete problem, STRIPS planning is a prototypical PSPACE complete problem. The most pragmatically effective known algorithms for STRIPS planning are called "nonlinear planners" and are discussed in the next chapter.

Figure 1: A legal move in the eight puzzle.

1 Introduction to Graph Search

Consider the eight puzzle show in figure 1. This puzzle is played on a three by three square board containing eight tiles and an empty square. Any tile which is next to the empty square can be slid into the empty square leaving an opening (empty square) in the place that used to be occupied by the moved tile. Figure 1 shows how one state of the puzzle can be transformed into another configure by sliding a tile into the empty square. Given an initial state of the puzzle the objective is to find a sequence of legal moves that transform the initial state into some desired goal state. Figure 2 also shows a typical goal state.

The problem of finding a sequence of moves that transforms a given initial state into a given goal state can be formulated as a graph search problem. The nodes of the graph are the possible states of the puzzle and the arcs of the graph correspond to legal moves that transform one state into another. The problem is to find a path in this graph from a given initial state to a given goal state.

Figure 2: A graph search problem.

In the eight puzzle there are four types of legal moves. The empty square can move up, move down, move right, or move left. This allows the puzzle to be formulated in terms of four operations U, D, L, and R that move the empty square up, down, right and left respectively. If the empty square is already on the upper edge of the board then we define the operation U to leave the state unchanged and a similar convention is used for the operations D, L, and R. Note that the entire graph search problem can be specified by a triple $<s, g, \{U, D, L, R\}>$ where s is an initial state, g is a goal state, and $\{U, D, L, R\}$ is a set of operations that can be applied to states. The possible states of the puzzle can be represented by simple Lisp data structures and the operations by simple Lisp procedures on those data structures.

Consider a specification of a graph search problem as an initial state, a goal state (both represented by Lisp data structures), and set of legal operations (represented by Lisp procedures). The specification of the search problem is usually much smaller than the implicitly represented graph — for the eight puzzle the problem can be specified with less than a page of Lisp code but the implicitly represented graph contains over 100,000 nodes. If we consider the natural generalization of the eight puzzle to $n \times n$ boards, the size of the specification grows as n^2 but the size of the represented graph grows as $(n^2)!$.

> **Definition:** A graph search problem consists of a set of an initial state, a goal state, and a set of state transition operators, i.e., functions from states to states.[1]

Rubic's cube and the towers of Hanoi are two particularly well known examples of puzzles that can be specified as an initial state, a goal state, and a set of legal operations. As in the eight puzzle, the size of the graph involved is vastly larger than the size of the problem specification. Just as constraint satisfaction problems correspond to the complexity class NP, graph search problems specified by a pair of states and allowed operations correspond to the complexity class PSPACE. Game search also corresponds to PSPACE. However, in practice the techniques that work well for game search are quite different from those that work well for graph search. Game search algorithms are discussed in chapter eight.

The following sections present a series of algorithms for graph search. The most pragmatically useful algorithm is IDA* presented in section 6. The earlier algorithms provide insight into the history and range of possibilities of algorithms for PSPACE complete graph search problems.

2 Depth First and Breadth First Search

In this section, and in sections, 3 through 7 of this chapter we assume a fixed but arbitrary graph search problem $<s, g, \{o_1, \cdots, o_b\}>$ where s is the initial state, g is the goal state, and $\{o_1, \cdots, o_b\}$ is the set of legal operations. I will

[1] This definition does not specify how the states and operators are to be represented. In practice, a state is usually represented by an array or list structure and the operations are represented by computer programs, such as Lisp procedures. In order to formally capture the intuition that graph search corresponds to the complexity class PSPACE the definition of a graph search problem must be made more restrictive — the allowed representation of states and operators must be specified. The definition of a STRIPS planning problem in section /refSTRIPS of this chapter can be viewed as a more restricted definition of a graph search problem that is provably PSPACE complete.

refer to states derived by repeated application of operations to the initial state as "nodes" of the graph search problem. A sequence of operations will be called a *plan*. Note that a plan (a sequence of operations) can be applied to any node to give a new node.

This section presents a naive algorithm for graph search called "algorithm zero". Algorithm zero uses a data structure called `FRINGE` which contains a set of pairs $<nP>$ where n is a node and P is a plan (sequence of operations) that leads from the given initial node to the node n.

Graph search algorithm zero: This algorithm finds a plan that leads from s to g using the operations $o_1, \cdots, o_b$.

1. Initialize `FRINGE` to be the set containing the pair $<s, \emptyset>$ where s is the initial node and $\emptyset$ is the empty plan.

2. Remove a pair $<n, P>$ from `FRINGE`.

3. If n is the goal node g then terminate with success and return the plan P.

4. For each of the operations $o_1, \cdots, o_b$ add the pair $<o_i(n), P; o_i>$ to `FRINGE`.

5. Go to step 2.

The behavior of algorithm zero is quite sensitive the way nodes are selected at step 2. If one always removes the most recently added pair (in which case `FRINGE` behaves as a stack) then the algorithm executes a depth first search. If one always removes the oldest (earliest added) pair (in which case `FRINGE` behaves as a queue) then the algorithm executes breadth first search.

Algorithm zero is not guaranteed to terminate — if there is no plan leading from the initial node to the goal then the procedure will run forever. Even if there is a solution, depth first search may run forever exploring some infinite unsuccessful plan. However, if there is a solution breadth first search will always find a shortest plan that solves the problem.

3 Transposition Tables

Breadth first and depth first search can be modified to guarantee termination whenever the graph being searched is finite. The basic idea is to mark previously visited nodes and to avoid placing previously visited nodes back onto the fringe in step 4. Graph algorithms are often viewed as polynomial time procedures that operate on explicitly given graphs. Given an explicit representation of a graph, where each node is represented by, say, an integer from 1 to the the number of nodes, there is no problem in "marking" nodes as they are visited. However, when graph search is formalized as a PSPACE complete problem we need to carefully consider how one marks previously visited nodes. In PSPACE complete graph search problems nodes are represented by data structures such as arrays or list structures. When we say that a newly generated node is "the same" as some previously visited node, we do not mean that the newly generated data structure occupies the same location in memory as the earlier data structure. If the two data structures in question reside in different parts of memory, then a mark in the earlier data structure will not be visible in the new data structure. I will assume an "intern function" I such that for any two data structures n and m, if n and m represent the same node (e.g., the same state of eight puzzle) then $I(n)$ is identical to (in the same memory location as) $I(m)$. The interned data structure $I(n)$ can be used to "carry side effects" between the distinct data structures n and m. The intern function I can be computed using a hash table or some other dictionary data structure.

The most common source of speedup from marking is due to "transpositions". Consider two operations o_i and o_k and a node n. It is often the case that $o_i(o_k(n))$ equals $o_k(o_i(n))$ — we say that o_i and o_j can be transposed (alternatively, that they commute). Because transposition is a common source of speedup from marking, the hash table used to implement the intern function I is often called a *transposition table.*

Using a transposition table to recognize previously visited nodes we can construct what is commonly called the breadth first search procedure.

Breadth First Search:

1. Initialize `QUEUE` to be the queue containing the pair $<s, \emptyset>$ where s is the initial node and $\emptyset$ is the empty plan.

2. Remove the first pair $<n, P>$ from `QUEUE`.

3. If $I(n)$ is marked then go to step 2.

4. Mark $I(n)$.

5. If n is the goal node g then terminate with success and return the plan P.

6. For each of the operations $o_1, \cdots, o_b$ add the pair $<o_i(n), P; o_i>$ to the end of `QUEUE`.

7. Go to step 2.

In most PSPACE complete graph search problems it is impossible to search all the nodes of the graph. So we do not really care about termination — if there is no solution then the procedure will run much longer than we are willing to wait. However, in those cases where there is a solution path it is important that the search procedure find the path quickly. In analyzing the complexity of graph search procedures we assume that there is a solution path and consider the time complexity of the algorithm as a function of the path length d.

If there are b legal operations in a graph search problem then the breadth first version of algorithm zero will always take order b^d time to find a solution of length d. For certain graph search problems, marking can reduce this from order b^d to order d. Exactly how much savings is generated by marking depends, of course, on the structure of the particular search problem in question. Let $N(d)$ be the number of distinct nodes in the graph that can be reached by a plan of length d. While the naive algorithm must take $O(b^d)$ time, the marking algorithm takes order $O(N(d))$ time.[2]

[2]This analysis assumes that the intern function I can be computed in unit time. This is true on average for hash tables but not in the worst case. Using a dictionary data structure with $\log N$ worst case lookup time, the procedure takes order $N(d) \log N(d)$ time.

4 Least Cost Plans

In this section we examine Dijkstra's shortest path algorithm [Dijkstra, 1959] for finding least cost plans. The graph search procedure can be viewed as a planner — it is searching for a plan, a sequence of operations, that can transform the initial state into the goal state. If the states represents states of actual physical objects, such as a state of robots on an assembly line, then we may wish to consider not only how hard it is to find a plan, but also how difficult it is to physically execute a plan. Each operation can o_i can be associated with a cost $c(o_i)$ which is the cost of physically executing o_i. If P is a plan then we define $c(P)$ to be the sum of the costs of the operations in P. Algorithm one shown below is guaranteed to find a least cost plan whenever a plan exists.

Dijkstra's Shortest Path Algorithm

1. Initialize `FRINGE` to be the set containing the pair $<s, \emptyset>$ where s is the initial node and $\emptyset$ is the empty plan.
2. If `FRINGE` is empty then terminate with failure.
3. Remove a pair $<n, P>$ from `FRINGE` such that $c(P)$ is minimum over all elements of `FRINGE`.
4. If $I(n)$ is marked then go to step 2.
5. Mark $I(n)$.
6. If n is the goal node g then terminate with success and return the plan P.
7. For each of the operations $o_1, \cdots, o_b$ add the pair $<o_i(n), P; o_i>$ to `FRINGE`.
8. Go to step 2.

A complexity analysis of Dijkstra's shortest path algorithm can be given in terms of the following notations.

Definition: $R(c)$ is the set of nodes *reachable* from the start node via a plan of cost c or less.

Definition: $N(c)$ is the number of nodes in $R(c)$.

Definition: $F(c)$ is the set of *fringe* nodes at cost c, i.e., nodes that are not in $R(c)$ but are reachable by a single operation from some element of $R(c)$.

It is easy to see that the plans selected in step 3 of algorithm one have non-decreasing costs — all pairs added to `FRINGE` have a cost larger than the last pair removed. However, the algorithm may process several plans of the same cost. The operation of the algorithm can be divided into "phases". First we process plans of some cost c_1, then we process plans of a larger cost c_2, and so on. Because of marking, the algorithm does not process all possible plans. However, immediately after the algorithm finishes process plans of cost c_i it will have marked all nodes in $R(c_i)$ and `FRINGE` will contain all nodes in $F(c_i)$. Thus, if the procedure finishes processing plans of cost c, and no solution has been found, then there is no solution of cost c. Thus the procedure is guaranteed to find a minimal cost solution.

Now we analyze the time taken by this procedure to find a plan of cost c. If there exists a plan of cost c then the procedure will terminate by the time it finishes processing all plans of cost c. For a fixed branching factor b, the number of nodes placed on `FRINGE` is proportional to $N(c)$. In this procedure, `FRINGE` can be implemented as a priority queue. To process n items in a priority queue takes order $n \log n$ time. Thus the total time spent adding and deleting elements from `FRINGE` is order $N(c) \log N(c)$. Since the total number of items added to `FRINGE` is order $N(c)$, there are at most order N executions of step 2. Even if we allow for $\log N(c)$ time in computing the dictionary lookup function I, the total time other than handling the priority queue is also order $N(c) \log N(c)$. Thus this procedure can finish processing plans of cost c in order $N(c) \log N(c)$ total time. It is important to remember, however, that $N(c)$ usually grows exponentially in c.

5 The A* Algorithm

In this section we consider an algorithm, called A*, that exploits additional information about distances in the graph. The A* algorithm is based on the following definitions.

> **Definition:** $h^*(n)$ is the cost of a minimal cost plan that leads from n to the goal node.

> **Definition:** An *admissible heuristic function* is a function h from nodes to costs such that for any node n we have that $h(n) \leq h^*(n)$.

An admissible heuristic function tells us, without searching, that the least cost plan form the given node to the solution costs at least so much, i.e., we are at least so far from the goal. For example, in the eight puzzle we can take $h(n)$ to be the number of tiles out of position at node n. Any solution must move every tile, and we will need at least $h(n)$ moves. Since we need at least $h(n)$ moves to solve the puzzle starting at node n, we have that $h(n) \leq h^*(n)$ and h is an admissible heuristic.

In the remainder of this section, and in section 6, we assume a fixed but arbitrary admissible heuristic function. In practice the algorithms in this section can be used with heuristic functions that are not admissible, i.e., a function h that heuristically approximates h^* but which is not strictly uniform as defined above. The use of a heuristic function in selecting the next node for expansion was introduced by Doran and Michie [Doran and Michie, 1966]. The concept of an admissible heuristic function, and an early variant of the A* procedure presented below, was introduced by Hart Nilsson and Raphael [Hart *et al.*, 1968].

The heuristic function h can be used to assign a "cost" to a plan.

> **Definition:**

The *directly projected cost* of a plan P is $c(P) + h(n)$ where n is the node reached by following P from the initial node.

Whenever h is admissible the directly projected cost of a plan P is a lower bound on the cost of any solution that begins with P. Longer plans have larger costs, and one might expect that longer plans also have larger directly projected cost. However, there are "pathological" heuristic functions, which are still admissible, under which an extension of a plan P can have smaller directly projected cost than P. The existence of these pathological heuristic functions motivates the following definition.

Definition:

The *projected cost* of a plan $o_1; o_2; \cdots; o_k$ is the maximum over all $j \leq k$ of the directly projected cost of the prefix plan $o_1; o_2; \cdots; o_j$.

Like directly projected costs, whenever h is admissible the projected cost of P is a lower bound on the cost of any solution that starts with P. However, if h is pathological then the projected cost of h can be larger (and hence more informative) than the directly projected cost of P. Note that, for any heuristic function, any extension of a plan P must have a projected cost that is at least as large as the projected cost of P.

The following definition can be used to characterize the "well behaved" admissible heuristic functions.

Definition: An admissible heuristic h is called *monotone* if for any node n and operation o_i we have that $c(o_i) + h(o_i(n)) \geq h(n)$.

If h is monotone then projected cost equals directly projected cost. It seems that most admissible heuristic functions are monotone. However, the algorithm given here does not assume that the heuristic function is monotone and uses projected cost rather than directly projected cost. We can now define the algorithm A* as follows.

A* Graph search algorithm.

1. Initialize FRINGE to be the set containing the pair $<s, \emptyset>$ where s is the initial node and $\emptyset$ is the empty plan.

2. If FRINGE is empty then terminate with failure.

3. Remove a pair $<n, P>$ from FRINGE which minimizes the projected cost of P.

4. If $I(n)$ is marked then go to step 2.

5. If n is the goal node g then terminate with success and return the plan P.

6. Mark $I(n)$.

7. For each of the operations $o_1, \cdots, o_b$ add the pair $<o_i(n), P; o_i>$ to FRINGE.

8. Go to step 2.

The only difference between A* and algorithm one is the method of selecting the next pair at step 3. Since the projected cost of a plan can not decrease as operations are added, the new pairs generated in step 7 involve plans whose projected cost is at least as large as the projected cost of the last plan removed in step 3. This implies that the projected cost of the plans removed in step 3 is non-decreasing. To analyze both the correctness and the running time of the above A* algorithm we use to following definitions.

> **Definition:** $R^*(c)$ is the set of nodes n such that there exists a plan from the initial node to n with projected cost no larger than c.

> **Definition:** $F^*(c)$ is the set of nodes that are not in $R^*(c)$ but are reachable by a single operation from some element of $R^*(c)$.

> **Definition:** $N^*(c)$ is the number of nodes in $R^*(c)$.

The projected cost of a plan P is always at least as large as the classical cost $c(P)$. So $R^*(c)$ is a subset of $R(c)$ and $N^*(c)$ is less than or equal to $N(c)$. If the values of h are very large, then projected costs are much larger than classical costs, $R^*(c)$ is a small subset of $R(c)$, and $N^*(c)$ is much smaller than $N(c)$.

As previously mentioned, the monotonicity condition ensures that the plans selected in step 3 have non-decreasing projected costs. Immediately after the algorithm finishes processing plans of projected cost c or less it will have marked all nodes in $R^*(c)$ and `FRINGE` will contain all nodes in $F^*(c)$. This implies that if the procedure finishes processing plans of cost c, and no solution has been found, then there is no solution of cost c. Thus the procedure is guaranteed to find a minimal cost solution. By an argument similar to that given for algorithm one, the procedure terminates in time proportional to $N^*(c) \log N^*(c)$ where c is the cost of a minimal cost solution plan.

6 Iterative Deepening and IDA*

Iterative deepening uses a bounded version of depth first search — a depth first search which fails and backtracks whenever it generates a path longer than the given bound. To ensure that any solution will eventually be found, the depth first search is repeated with ever increasing bounds. Because the number of nodes searched generally grows exponentially with the size of the search, the time spent in early searches is negligible compared to the time taken of the last search (or perhaps the last several searches). Iterative deepening with admissible heuristic cutoff was first proposed and analyzed in the context of graph search by Korf [Korf, 1985]. Prior to Korf's work iterative deepening was recognized as an essential technique in the construction of high performance computer chess programs.

In practice iterative deepening has dramatic advantages over all of the procedures presented in the previous sections. First, iterative deepening is analytically superior — it runs in time $N^*(c)$ rather than $N^*(c) \log N^*(c)$.[3] Second,

[3]This analysis requires the assumption that the search space grows exponentially and

iterative deepening, and depth first search in general, has the even more important pragmatic advantage that a transition from one node to the next in the search tree can be done by side effect rather than by copying. For example, if the board of the eight puzzle is represented by an array then one can move to the next position by setting values in that array. In depth first search only one board position is needed at any given time and so copying is unnecessary. In breadth first search or A* one must maintain a separate copy of each node on the `FRINGE` data structure. A third advantage of iterative deepening is reduced space requirements. If one does not use a transposition table then the space required by iterative deepening is linear, rather than exponential, in the length of the solution path. Finally, iterative deepening is easier to implement. The procedures described can be implemented directly in nondeterministic lisp. Nondeterministic Lisp provides highly efficient automatic backtracking and automatic backtrackable side effects.

In order to run in time proportional to $N^*(c)$ we must detect repeated visits to the same node. As in the case of Algorithm one and A*, this can be done with an intern function I. However, depth first search makes simple marking insufficient. Rather than remember whether $I(n)$ is marked or unmarked, we store a number $M(I(n))$ in the data structure $I(n)$. The number $M(I(n))$ is the value of the largest cost c such that there has been a call to the procedure `FIND-PLAN` on node n and cost c. Recording this number prevents redundant computations of the function `FIND-PLAN`.

To compute (`FIND-PLAN` n c) do the following:

1. If $c < 0$ then return failure.
2. If n is the solution node then return the empty plan.
3. If $h(n)$ is greater than c then return failure.
4. If $M(I(n))$ has been set to a value greater than or equal to c then return failure.
5. Set $M(I(n))$ to c.

---

that transposition table lookups take constant time.

6. If there is some operation o_i such that (FIND-PLAN $o_i(n)$ $c - c(o_i)$) returns a plan P, then return the plan $o_i; P$.

7. Otherwise, return failure.

To analyze the time required to compute the value of FIND-PLAN we make the following assumptions.

1. The functions h and I can be computed in unit time.

2. The cost of each operation o_i is a positive integer.

3. $N^*(c)$ grows exponentially in c, i.e., there exists a *minimum branching factor* $B > 1$ such that for all positive integer costs $c > 1$ we have $N^*(c) \geq BN^*(c-1)$.

We now show that the total time required to compute (FIND-PLAN s c) is order $N^*(c)$. Now consider computing (FIND-PLAN s c) where s is the initial node and c is an integer. First, note that the total time taken by (FIND-PLAN s c) is proportional to the total number of executions of step 6 — if there are b operators then the total number of recursive calls to the procedure is proportional to b times the number of executions of step 6. Because of the filter at step 4, step 6 will never be executed twice for the same value of n and c. The total number of calls to step 6 can be no larger than the number of pairs $<n, c'>$ such that a recursive call (FIND-PLAN n c') is generated and such that the test at step 3 is passed, i.e., $h(n) \leq c'$. All recursive calls generated by the procedure have the form (FIND-PLAN n $c - c(P)$) where P is a plan from the initial node to n. If $h(n) \leq c - c(P)$ then $C(p) + h(n) \leq c$, i.e., the projected cost of P must be no larger than c in order for step 6 to be executed. For a fixed node n, the number of such values of c' such that a call of the form (FIND-PLAN n c') is generated, and leads to an execution of step 6, is equal to the number of distinct values of $c(P)$ where P is a plan from s to n with projected cost no larger than c. These projected costs must be in the interval $[g(n) + h(n), c]$ where $g(n)$ is the least cost plan from the initial node to n. This interval is the set of values c'' such that n is a member of $R^*(c'')$. The total number of executions of step 6 can be no larger than the sum over all nodes n in $R^*(c)$ of the number of distinct values $c'' \leq c$ that n is a member of $R^*(c'')$. This implies that the total number of executions of step 6 is less than or equal to

$$N^*(c) + N^*(c-1) + N^*(c-2) + \cdots + N^*(0).$$

Given that that $N^*(c) \geq BN^*(c)$, we now have that the total number of executions of step 6 is less than or equal to

$$N^*(c)(1 + \frac{1}{B} + \frac{1}{B^2} + \frac{1}{B^3} + \cdots) = N^*(c)\frac{B}{B-1}.$$

Note that the procedure `FIND-PLAN` may explore a given node more than once, but the overhead due to repeated explorations of the same node results in at most a factor of $\frac{B}{B-1}$ slow-down in the procedure. If the minimal branching factor is large (say 2 or more) then this additional factor is quite manageable and much smaller than $\log N^*(c)$ for practical values of $N^*(c)$.

The procedure `FIND-PLAN` requires an a-priori upper bound on the cost of allowed plans and returns the first plan it encounters that has an acceptable cost. The following procedure can be used to find a least cost plan.

The algorithm IDA*.

1. Initialize c to 0.
2. If `(FIND-PLAN` s c`)` returns a plan P then return plan P.
3. Set c to $c + 1$.
4. Go to 2.

Under the assumption that all costs are integers, this procedure returns a least cost plan. Under the above assumptions that h and I take unit time, and that $N^*(c)$ grows exponentially in c, we can show that the above procedure takes order $N^*(c)$ time where c is the cost of a minimal cost plan from s to the goal node. The time taken by this procedure is dominated by the time taken by the calls to `FIND-PLAN`. Since `(FIND-PLAN` s c'`)` takes order $N^*(c')$ time the overall procedure takes time on the order of

$$N^*(c) + N^*(c-1) + N^*(c-2) + \cdots + N^*(0).$$

Under the assumption that $N^*(c)$ grows exponentially in c, this sum is less than or equal to

$$N^*(c)(1 + \frac{1}{B} + \frac{1}{B^2} + \frac{1}{B^3} + \cdots) = N^*(c)\frac{B}{B-1}.$$

So the total time is order $N^*(c)$.

The overhead due to iterative deepening appears as the factor $\frac{B}{B-1}$ in the analysis. For most values of B and $N^*(c)$ encountered in practice, the factor of $\frac{B}{B-1}$ is much smaller than $\log N^*(c)$. Depth first search also allows a host of constant reducing optimizations not possible in breadth first search. In practice, iterative deepening is much faster than queue-based approaches.

7 Quotienting

Consider the eight puzzle. Suppose that instead of being asked to place each tile in a specified position, we want to solve the more modest goal of placing tile number one in the upper left hand corner, independent of the positions of the other tiles. If all we care about is tile number one, then all we care about is the position of tile one and the position of the space. Two states of the board should be considered to be *equivalent* they both have tile one and the space in the same place. This notion of equivalent states can be captured by an operation that erases the numbering of the irrelevant tiles. Figure 3 shows the operation of erasing irrelevant numbers — each irrelevant number is replaced by "?". This erasing operation is a many-to-one mapping — different states of the original eight puzzle can map to the same erased state. A map that preserves all information relevant to solving the stated problem will be called a search problem *homomorphism*. A many-to-one search problem homomorphism can greatly reduce the size of the search problem. There are 9! potential states of the eight puzzle, but only

Figure 3: A homomorphism mapping states to erased states.

72 erased states. Each erased state represents precisely 7! different unerased states. The erasure function divides, or "quotients", the number of states by 7!. The graph search problem on the erased states is called the *quotiented* graph search problem because the elements of the quotient graph represent uniform size equivalence classes of the original problem. We will refer to the result of erasing operations as quotient graphs even in cases where the equivalence classes are not of uniform size.

We now formalize the general notion of a search problem homomorphism. In this section we take a graph search problem to be a triple $<s, G<o_1, \cdots o_b>>$ if an initial node s, a goal predicate G and a tuple $<o_1, \cdots, o_b>$ of legal operations. To problem is to find sequence of operations (a plan) that leads from the initial node to some node n such that $G(n)$ holds. Note that this is a slightly different formalization from the one used in previous sections — the goal node g has been replaced by a goal predicate G.

> **Definition:** A *graph search homomorphism* between two graph search problems $<s, G<o_1, \cdots o_b>>$ and $<s', G'<o'_1, \cdots o'_b>>$ is a mapping H from the nodes of the first problem to the nodes of

the second problem such that H satisfies the following conditions.

- $s' = H(s)$
- $G(n)$ holds if and only if $G'(H(n))$ holds.
- $o'_i(H(n)) = H(o_i(n))$

The first condition in the above definition specifies the initial node of the quotient graph in terms of the initial node of the original graph and the "erasing" homomorphism H. The second condition states that none of the erased information is relevant to determining whether a node is a goal node, i.e., one can determine whether or not the original node was a goal node by looking at the erasure of that node. The third condition determines the operation o'_i in terms of the operation o_i and the erasing function H. This condition states that the result of erasing and then applying an operator is the same as the result of applying the corresponding operator and then erasing (erasure commutes with operation application). This condition is easily checked for the erasure operation proposed for the eight puzzle.

Quotient Lemma: If H is a graph search homomorphism between $<s, G<o_1, \cdots o_b>>$ and $<s', G'<o'_1, \cdots o'_b>>$, and $o'_{i_1}; o'_{i_2}; \cdots ; o'_{i_k}$ is a plan that solves the latter (quotiented) problem, then the "lifted" plan $o_{i_1}; o_{i_2}; \cdots ; o_{i_k}$ is a solution to the former (original) problem.

Proof: We are given that $G'(o'_{i_k}(o'_{i_{k-1}}(\cdots(o'_{i_1}(s')))))$ holds. The first condition in the definition of graph search homomorphism implies that $G'(o'_{i_k}(o'_{i_{k-1}}(\cdots(o'_{i_1}(H(s))))))$ holds. Repeated application of the third condition gives that $G'(H(o_{i_k}(o_{i_{k-1}}(\cdots(o_{i_1}(s))))))$ holds. Finally, the second condition gives that $G((o_{i_k}(o_{i_{k-1}}(\cdots(o_{i_1}(s)))))$ holds so $o_{i_1}; o_{i_2}; \cdots ; o_{i_k}$ is a solution to the original problem.

The quotient lemma implies that solving the original graph search problem can be reduced to solving the quotient graph search problem. Since there are far fewer nodes in the quotient graph, searching the quotient graph can be more efficient. Improving search efficiency by searching a a quotient graph requires the use of a transposition table (the intern function I used in the graph search algorithms). Consider a graph search problem with b operators and where each operator has unit cost and where the least cost solution pat has cost c. Without a transposition table it will take order b^c time to find a solution in *either* the original graph search problem or the quotient graph search problem — without a transposition table there is no advantage to quotienting. With a transposition table, however, one can construct an algorithm that runs in $N^*(c) \log N^*(c)$ time. Let $N^*(c)$ be the number of nodes reachable by plans of projected cost c or less in the original problem and let $N'^*(c)$ be the analogous number in the quotiented problem. In many cases $N'^*(c)$ is much smaller than $N^*(c)$.

8 STRIPS Planning

Within artificial intelligence graph search has sometimes been used as a model of common sense planning. Suppose that you want to have a barbecue. The goal is to have certain people come to the house and to serve them (and yourself) some good food. One might attempt to think of this a graph search problem where you are currently in some state of the world and you want to select a sequence of actions that results in some desired state of the world. Unfortunately, you do not know all aspects of the world — the world is far too complex to represent in some simple data structure that you can give to your favorite automatic graph search procedure. However, the world can be *partially described* by stating properties of the world that are currently true, such as you are currently at the office, you currently have less than ten dollars in your wallet, and so on. STRIPS planning is a method of solving graph search problems using incomplete knowledge of the initial state.

We assume that each node of a graph search problem, i.e., each state of the world or state of a puzzle, can be partially described in terms of propositions. We assume a set of proposition symbols where each proposition symbol is associated with a truth function on nodes — each proposition is either true or false at each node. The behavior of the operators can be partially described in terms of these propositions. We also assume that we a set Σ of proposition symbols that are known to be true at the initial node. We further assume that we are given a set Ω of "goal propositions". The planning problem is to find a plan (sequence of operations) such that at any node where all propositions in Σ are true, applying plan α results in a node where all propositions in Ω are true. This desired property of the plan α can be expressed in dynamic logic notation as follows.

> **Definition:** We say that the specification $\Sigma \rightarrow [\alpha]\Omega$ *holds* in a graph search problem if, at each node where every proposition in Σ is true, applying the plan α results in a node where each proposition in Ω is true.

STRIPS planning not only deals with partial information about nodes in the search graph (states of the world), it also deals with partial information about the behavior of operators. Partial information about each operator can be phrased in terms of the proposition symbols used to describe nodes.

> **Definition:** A *STRIPS operator specification* consists of a set of *operator symbols* where each operator symbol is associated with a *prerequisite list*, an *add list* and a *delete list* each of which is a set of proposition symbols.
>
> **Definition:** A STRIPS operator specification is said to *hold* (or be *valid*) in a graph search problem if for each operator o_i, and each node n such that every prerequisite of o_i is true at n, we have the following conditions.
>
> - All propositions on the add list of o_i are true at the node $o_i(n)$.
> - If P is a proposition that is *not* on the delete list of o_i, and P is true at n, then P is true at $o_i(n)$.

If the prerequisites of an operator are met, then an application of that operator is guaranteed to achieve (make true) the propositions on its add list. Furthermore, if the prerequisites of an operator are met then any proposition not on the delete list of that operator is guaranteed to be "preserved" in the sense that if it was true before then operation is applied then it is true after the operation is applied. For example, suppose that each node of the graph search problem is a pair of numbers $<x, y>$. Suppose that some of the propositions are of the form $P(x)$, i.e., they only depend on the value of x, and other propositions are analogously of the form $P(y)$. Furthermore suppose that some operators are of the form $x := f(x, y)$, i.e., the operator may change the value of x but does not alter the value of y. In this case any proposition of the form $P(y)$ is preserved by an operator of the form $x := f(x, y)$. So no proposition of the form $P(y)$ should be on the delete list of $x := f(x, y)$ — all propositions on the delete list of $x := f(x, y)$ should be propositions that depend on the value of x. As a more concrete example, consider the operator $x := x + y$ and the proposition $E(x)$ that is true just in case x is an even number. Because there are nodes at which $x := x + y$ does not preserve $E(x)$, the proposition $E(x)$ must be included on the delete list of the operator $x := x + y$. $E(x)$ must be on the delete list of $x := x + y$ even though $x := x + y$ can sometimes cause $E(x)$ to go from false to true. In the formulation of a STRIPS operator description we assume that all propositions can be treated as simple atomic symbols. A more sophisticated treatment of operators and propositions is discussed below.

> **Definition:** A *STRIPS planning problem* consists of a STRIPS operator specification, a set Σ of initial propositions and a set Ω of goal propositions.
>
> **Definition:** A *solution* to a STRIPS planning problem is a plan (sequence of operators) α such that, in every graph where the STRIPS operator specification holds, we have $\Sigma \rightarrow [\alpha]\Omega$.

In a STRIPS planning problem we are given a STRIPS operator specification rather than a particular graph search problem. Since we are not given any particular graph, the graph search methods described earlier can not be directly applied. Fortunately, any strips planning problem can be reduced to a classical graph search problem by constructing a particular search graph from the partial information given in the operator specification. The *free graph* of a STRIPS planning problem is a graph in which the nodes are subsets of proposition symbols and in which the operators map sets of propositions to sets of propositions in accordance with prerequisite lists, add lists and delete lists of the operators. More precisely, we have the following definition.

> **Definition:** The *free graph search problem* of a STRIPS planning problem is the graph search problem $<\Sigma, \Omega, <o_1, \cdots o_b>>$ where Σ is the given set of initial propositions, Ω is the given set of goal propositions, and each operator o_i maps a set of propositions Γ to a set of propositions $o_i(\Gamma)$ in accordance with the following rules.
>
> - If every prerequisite of o_i is true in Γ then $o_i(\Gamma)$ equals Γ minus the elements of the delete list of o_i plus the elements of the add list of o_i.
> - If some prerequisite of o_i is not a member of Γ then $o_i(\Gamma)$ is the empty set.

In the free graph search problem associated with a STRIPS planning problem the nodes are sets of proposition symbols. A proposition symbol P is taken to be true at a node Γ just in case Γ contains P. The free graph gives a concrete graph search problem that can be searched using the standard graph search techniques. The free graph search problem differs from the graph search problems discussed earlier in that the goal is to find a plan that takes the start node Σ to some node that *contains* the goal set Ω — a solution is not required to take Σ precisely to the single node Ω. The precise relationship between the free graph and the original STRIPS planning problem is captured in the following lemmas.

> **First Free Graph Lemma:** If a plan α takes the node Σ in the free graph search problem to some node that contains Ω then α is a solution to the given STRIPS planning problem, i.e., the statement $\Sigma \rightarrow [\alpha]\Omega$ holds in *all* graph search problems for which the given operator specification holds.
>
> **Proof:** Consider an arbitrary plan β. We define the set of propositions $\beta(\Sigma)$ the be the node in the free graph search problem that results from applying the plan β to the initial node Σ. (If β is the sequence $o^1; \cdots; o^k$ then $\beta(\Sigma)$ is the set $o^k(\cdots o^1(\Sigma)))$.) It can be

shown by induction on the number of operations in β that in any graph where the given operator description holds, and any node n in that graph where every proposition in Σ holds, every proposition in $\beta(\Sigma)$ holds at the node $\beta(n)$. This equivalent to the statement that $\Sigma \rightarrow [\beta]\beta(\Sigma)$ holds in any graph where the given operator specification holds. Now if α is such that $\alpha(\Sigma)$ contains Ω, then $\Sigma \rightarrow [\alpha]\Omega$ must hold in any graph search problem where the given operator description holds.

Second Free Graph Lemma: If α is a solution to the STRIPS planning problem then α takes the initial node Σ of the graph search problem to a node that contains Ω.

Proof: The operator specification of a STRIPS planning problem holds in the associated free graph search problem. Therefore any plan α that is a solution to the STRIPS planning problem must "work" in the free graph.

One can find a solution to a STRIPS planning problem by using standard graph search techniques on the associated free free graph. The first free graph lemma states that this method of finding solutions is sound — any plan that solves the free graph problem is indeed a solution to the original STRIPS planning problem. The second lemma states that this method is complete — if there is a solution to the STRIPS planning problem then it can be found by searching the free graph.

The notion of a "free model" is a general method of converting problems with incomplete information into problems with complete information. For STRIPS planning we have converted the incomplete information in the operator specifications into complete information about operations on the free graph. Free models are most often discussed in the context of equational theorem proving. Unfortunately, many problems involving incomplete information can not be associated with a free model. For example, if we allow disjunctions of propositions in the add lists of operators then no free graph can be constructed.

We have reduced the STRIPS planning problem to the problem of finding a solution to a graph search problem of the type discussed earlier. This implies that STRIPS planning is solvable using polynomial space. In fact, it can shown that the STRIPS planning problem is PSPACE complete.

9 A Backward Chaining Planner

For the STRIPS planning problem one can construct a backward chaining search procedure that searches back from the goal set Ω. In many applications the number of propositions known to hold at the initial node is very large. In such cases the number of operators whose preconditions are met in the start node can also be very large, resulting in a large initial branching factor in the search. On the other hand, the number of propositions in the goal set Ω can be quite small. If number of operators that add an element of Ω is much smaller than the number of operators whose prerequisite list is a subset of Σ, then a backward chaining search will have a smaller branching factor than a forward chaining search. The backward chaining procedure is based on the following definition.

> **Definition:** Let Ω be a set of goal propositions, let o_i be an operator, and let $\mathbf{F}$ be a new proposition symbol that is not mentioned in the operator specification (the proposition $\mathbf{F}$ is false at every node of the free graph). The *weakest precondition* for goal Ω with respect to o_i is defined by the following conditions.
>
> - If Ω contains the special symbol $\mathbf{F}$, or if some member of Ω is deleted by o_i, then the weakest precondition of Ω with respect to o_i is the set $\{\mathbf{F}\}$.
> - Otherwise, the weakest precondition of Ω with respect to o_i is the set Ω minus all the propositions added by o_i plus all prerequisites of o_i.

> **Lemma:** If Γ is the weakest precondition for Ω with respect to o_i then either Γ contains $\mathbf{F}$ and there is no set Γ' such that $o_i(\Gamma')$ contains Ω, or Γ does not contain $\mathbf{F}$ in which case Γ is the least set Γ' such that $o_i(\Gamma')$ contains Ω.

Definition: The weakest precondition of a goal set Ω with respect to a plan α is defined by induction on the number of steps in α as follows.

- The weakest precondition of the empty plan is the goal set Ω.
- The weakest precondition of the plan $o_i; \beta$ equals the weakest precondition of Γ with respect to o_i where Γ is the weakest precondition of Ω with respect to β.

Lemma: If Γ is the weakest precondition of goal set Ω with respect to plan α then, either Γ contains **F** and there is no set Γ' such that $\alpha(\Gamma')$ contains Ω, or Γ is the least set Γ' such that $\alpha(\Gamma'')$ contains Γ.

Lemma: A plan α is a solution to a STRIPS planning problem with initial propositions Σ and goal set Ω if and only if Σ contains the weakest precondition of Ω with respect to α.

We can now define a backward chaining procedure for solving STRIPS planning problems. We define a procedure of four arguments `FIND-PLAN(`Σ`, `Ω`, `c`, `h`)` where Σ the initial set of propositions, Ω is a goal set, c is a cost bound, and h is a heuristic function. The function h takes an initial proposition set and a goal proposition set and returns a lower bound on the cost of a solution plan. The procedure nondeterministically returns a plan that solves the given problem and that has cost c or less. If there is no such plan then the procedure fails.

Procedure for computing `FIND-PLAN(`Σ`, `Ω`, `c`, `h`)`**:**

1. If Ω is a subset of Σ then return the empty plan.
2. If $h(\Sigma, \Omega) > c$, or $c = 0$, then fail.
3. Let o_i be an operator satisfying the following two conditions.
   (a) No element of the delete list of o_i is a member of Ω.

(b) Some element of the add list of o_i is a member of Ω.

4. Let Ω' be the weakest precondition of Ω with respect to o_i.

5. Let β be `FIND-PLAN(`Σ`,` Ω'`,` $c - c(o_i)$`,` h`)`

6. Return the plan $\beta; o_i$.

The nondeterminism in this procedure arises form the nondeterministic choice of operator in step 3. To prove that this procedure always finds a plan of cost c when such a plan exists we must show that if a plan of cost c exists then there exists a solution plan of cost c whose last step is an operations that passes the tests of step 3a and 3b. The last step of any solution plan must satisfy 3a. However, there can exist solution plans whose last step does not satisfy 3b. However, if $\beta; o_i$ is a solution plan where the step o_i does not satisfy 3b, then the plan β must also be a solution plan. By stripping off all steps at the end of β that do not satisfy 3b we can construct a (smaller cost) plan whose last step does satisfy 3b. Thus, if there exists a solution plan of cost c or less then there exists a solution plan of cost c or less whose last step satisfies both 3a and 3b.

10 Problems

1. Consider a graph search problem for which the number of nodes reachable by paths of length n is linear in n (i.e., is less than or equal to cn for some constant c). In this problem we compare iteratively deepened depth first search with simple breadth first search. In both cases we assume dynamic programming so that the same node is not visited repeatedly. A node at depth three will be visited exactly once by each phase of the iterative deepening procedure that search to depth three or greater. In the following questions we assume that the hashing operations used to determine if a node has been visited before take constant time. What is the order of running time of the iteratively deepened depth first procedure to find a solution of length n? What is the order of growth of the time taken by the simple breadth first procedure?

2. This problem considers the degree to which a heuristic function improves the performance of a search problem. Consdider a graph search problem with 2 operators (the search has branching factor 2). In this problem we will ignore the posibility that any node can be reached by more than one path. There are 2^d different nodes that can be reached by paths of length d. We also assume that each operator has a cost of 2. We allow for more than one goal node and for any node n we define $h^*(n)$ to be the cost of the shortest path from n to a goal node (which will be twice the number of operations in the path). The definition of h^* implies that at each node n one of the operators must move one step closer to the goal, i.e., there is an operator o such that $h^*(o(n))$ is one less than $h^*(n)$. Such an operator will be called a *good move* from the node n. In this problem we assume that there is only one good move from n. The operator other than the good move will be called the bad moves. We assume that the bad move does not change the distance to the goal, i.e., if o' is a bad move from n then $h^*(o'(n))$ is one greater than $h^*(n)$. We assume a heuristic function h such that for any node n we have that $h(n) = \frac{1}{2}h^*(n)$. (This is unrealistic because in this case we could get a perfect heuristic function by dividing h by c. However, this unrealistic assumption may model more realistic cases where the expected value of h is $\frac{1}{2}h^*$.) Note that since $h^*(n)$ is always an even integer, $h(n)$ is always an integer. This heuristic function is monotone so the projected cost of a path is just the cost of the operators (in this case just the number operators) in the path plus $h(n)$ where n is the final node in the path. We define the *added cost* of a path to be the projected cost of the path minus the projected cost of the empty path.

Part a. Let G be the number of good moves in a path and let B be the number of bad moves in a path. Give an expression for the added cost of a path as a function of G and B.

Part b. Let $M(k)$ be the number of nodes that can be reached by paths with *added cost* of *exactly* k. The empty path has an added cost 0 so $M(0) = 1$. Only the good move from the initial node results in a path of added cost 1 so $M(1) = 1$. Give a recurrence relation for $M(k)$. In particular express $M(k+2)$ as a function of $M(k+1)$ and $M(k)$. (Hint: Draw a picture of the search graph.)

Part c. Give the order of growth of $M(k)$. (Hint: Solve the recurrance relation from part b.)

Part d. Give the order of running time needed to find a path of length d with breadth first search and with A^* using the heuristic function h. Your answer should indicate that A^* with h can find paths roughly twice as long as breadth first search. As the branching factor becomes larger (with only one good move from a position) the performance of A^* relative to breadth first search becomes even better.

3. Consider a graph search problem in which each node is a non-negative integer. Consider the following operations on numbers, viewed as operators in a graph search problem.

- D (for "Double") $D(n)$ is $2n$.
- H (for "Half") $H(n)$ is $\lfloor \frac{n}{2} \rfloor$.
- S (for "Successor") $S(n)$ is $n + 1$.
- E (for "make-Even") If n is odd then $E(n)$ is $n - 1$, else E(n), is n.

The plan (sequence of operations) $H; E; D; E$ maps the number 11 to the number 8 ($H(11) = 5$; $E(5) = 4$; $D(4) = 8$; and $E(8) = 8$).

Consider the following propositions about numbers.

- The proposition **Even** is true of a node n just in case n is even.
- The proposition **Odd** is true of a node n just in case n is Odd.
- The proposition **One-Two** is true of a number n just in case the prime factorization of n contains a single factor of 2. For example **One-Two**(6) is true but **One-Two**(12) is false.

Each operator can be applied at any node. In the following table each operator has been assigned an empty prerequisite list. Make a copy of this table with the add lists and delete lists filled in. Each add list and delete list should be a subset of the propositions **Even**, **Odd**, and **One-Two**. The delete lists should be as small as possible and the add lists as large as possible so that the operator specification holds of the actual operators.

Operator	Prerequisite List	Delete List	Add List
D	$\emptyset$		
H	$\emptyset$		
S	$\emptyset$		
E	$\emptyset$		

In the following table each operator has been assigned prerequisites. Make a copy of this table with minimal delete lists and maximal add lists inserted.

Operator	Prerequisite List	Delete List	Add List
D	**Odd**		
H	$\emptyset$		
S	**Even**		
E	$\emptyset$		

4. GPS was developed in the late 50's by Simon and Newell as a model of human problem solving. The GPS procedure for STRIPS planning is given below.[4]

[4] This specification of the GPS procedure is derived from the description on page 153 of [Winston, 1984].

To find a plan α such that $\Sigma \rightarrow [\alpha]\Omega$ do the following:

1. If Σ is a subset of Ω return the empty plan.
2. Select some operator o_i such that some proposition on the add list of o_i is a member of Σ but not a member of Ω
3. Let Ω' be the prerequisites of o_i.
4. Recursively find a plan β such that $\Sigma \rightarrow [\beta]\Omega'$.
5. Let Σ' be the result of running the program $\beta; o_i$ from start state Σ, i.e., Σ' is the largest set of propositions such that $\Sigma \rightarrow [\beta; o_i]\Sigma'$.
6. Recursively find a plan γ such that $\Sigma' \rightarrow [\gamma]\Omega$.
7. Return the procedure $\beta; o_i; \gamma$.

Step 2 of the above procedure is non-deterministic. This allows for many different possible executions. An implementation of this procedure would have to search the space of all possible executions.

We now consider the "blocks world". In a propositional STRIPS formulation of a blocks world problem with n blocks there are order n propositions of the form `CLEAR(A)` and order n^2 propositions of the form `ON(A,B)`. There are order n^3 operators of the form `MOVE(A,B,C)` which moves `A` from `B` to `C`. `MOVE(A,B,C)` has prerequisites `CLEAR(A)`, `CLEAR(C)`, and `ON(A,B)`. The operation `MOVE(A,B,C)` deletes the propositions `ON(A,B)` and `CLEAR(C)` and adds the propositions `ON(A,C)` and `CLEAR(B)`.

The Sussman anomaly is a blocks world planning where the initial state is given by the propositions

```
CLEAR(C),   ON(C,A),  ON(A,PLACE1)

CLEAR(B),  ON(B,PLACE2),  CLEAR(PLACE3)
```

and the goal consists of the two propositions `ON(A,B)`, `ON(B,C)`. Show that there exists a three step plan that solves the Sussman anomaly but that no execution of the GPS planning procedure can find this plan. Also give a solution to the Sussman anomaly that can be found by GPS.

5. Give an example of a planning problem for which there exists a plan that solves the problem but where GPS can not find a solution.

6. The `FIND-PLAN` procedure given in the notes can always find a minimal length plan, so it can find the three step solution to the Sussman anomaly described in problem 3. Give an example of an unachievable subgoal that might be included in Ω' at step 4 of the top level application of the procedure to the Sussman anomaly, i.e., a subgoal such that no sequence of operations applied to the initial state can make that subgoal true. Describe a filter that could be used in blocks world planning to eliminate unachievable subgoals. Such a filter can be implemented as part of the heuristic function h.

References

[Dijkstra, 1959] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[Doran and Michie, 1966] J. Doran and D. Michie. Experiments with the graph traverser program. *Proc. of the Royal Society of London*, 294(2):235–259, 1966.

[Hart *et al.*, 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimal cost paths. *IEEE Trans. Syst. Science and Cybernetics*, 4(2):100–107, 1968.

[Korf, 1985] Richard E. Korf. Iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:100–107, 1985.

[Winston, 1984] Patrick Winston. *Artificial Intelligence, Second Addition.* Addison Wesley, 1984.