Solving Problems by Searching

Michael Rose

In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do

This chapter describes one kind of a goal-based agent called a **problem solving agent**. They use atomic representations of the states of the world with no internel representations. Goal based agents that use more advanced factored or structured representations are usually called planning agents.

3.1 | Problem Solving Agents

Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Goal Formulation is the first step in problem solving. **Problem Formulation** is the process of deciding what actions and states to consider, given a goal.

The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as an input and returns a solution in the form of an action sequence. Once a solution is found, the actions in reccommends can be carried out. When an agent is executing its solution sequence, it *ignores its percepts* when choosing an action because it knows in advance what it will be. An agent that carries out its plans like this essentially has its eyes closed. Control theorists call this an **open loop system**, because ignoring the percepts breaks the loop between agent and environment.

3.1.2 | Formulating Problems

We wish to abstract the problem as much as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

3.2 | Example Problems

We must pay attention to the

- states
- initial state
- actions
- transition model
- goal test
- path cost

for each problem that an agent seeks to solve.

An interesting toy problem is given by Donald Knuth, ,in order to illustrate how infinite state spaces can arise.

Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach *any* desired positive integer. For example

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5$$

The problem definition is simple:

- States: positive numbers
- initial state: 4
- actions: apply factorial, square root, or floor operation
- transition model: As given by the mathematical definitions of the operations
- goal test: State is the desired positive integer

As far as we know, there is no bound on how large a number might be constructed in the process of reaching a given target. For example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5 - so the state space for this problem is infinite. Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

3.2.2 | Real-World Problems

Route-finding problems are defined in terms of specified locations and transitions along links between them.

Touring Problems are closely related to route finding problems, but with one distinct difference. The state space for these problems includes not just the current states, but a list of each state visited. The **traveling salesperson** problem is a touring problem in which each city must be visited exactly once, and the aim is to find the shortest tour.

3.3 | Searching for Solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences. These form a **search tree** with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.

The first step is to test whether this is the goal state. Then we need to consider taking various actions. We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby generating a new set of states. Then we must choose which of the new possibilities to consider.

This is the essence of search - following up one option now and putting the others aside for later, in case the choice does not lead to a solution. The set of all leaf nodes available for expansion at any given point is called the **frontier**. The process of expanding nodes on the frontier is continued until either a solution is found or there are no more states to expand.

If a set of nodes recursively refers to a node that was phased out in previous tiers of the tree, we have a **loopy path**. This can vastly expand our state space and can often make tractable problems intractable. Therefore, when we use trees we augment them with the **explored set** data structure to avoid exploring redundant paths. There is a saying, "algorithms that forget their history are doomed to repeat it."

3.3.1 | Infrastructure for Search Algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- n.State: the state in the state space to which the node corresponds
- n.Parent: The node in the search tree that generated this node
- n.Action: The action that was applied to the parent to generate the node
- n.Path-Cost: The cost, traditionally denoted by g(n) of the path from the initial state to the node, as indicated by the parent pointers.

3.3.2 | Measuring problem-solving performance

We evaluate an algorithms performance in 4 ways:

- Completeness
- Optimality
- Time Complexity
- Space Complexity

3.4 | Uninformed Search Strategies

Uninformed Search Strategies have no additional information about the states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state.

All search strategies are distinguished by the order in which nodes are expanded. Strategies that know whether one non-goal state is more promising than another are called **informed search** or **heuristic search** strategies.

3.4.1 | Breadth-First Search

Breadth-First Search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. BFS is an instance of the general graph search algorithm in which the *shallowest* unexpanded node is chosen for expansion. This is achieved by using a FIFO queue for the frontier.

There is one slight tweak on the general graph search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. BFS always has the shallowest path to every node on the frontier. Shallow is not always optimal - but in this case it is optimal if the path cost is a nondecreasing function of the depth of the node.

BFS has an exponential complexity bound of $O(b^d)$. If we assume 1 node is 1000 bytes, the memory requirements are intractable for BFS.

3.4.2 | Uniform-Cost Search

When all step costs are equal, breadth first search is optimal because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g.

In this case, the goal test is applied to a node when it is *selected for expansion* rather than when it is first generated. A test is also added in case a better path is found to a node currently on the frontier. Because steps are non-negative, paths never get shorter as nodes are added. This implies that uniform cost search expands nodes in order of their optimal path cost. Hence the first goal node selected for expansion must be the optimal solution.

Uniform cost search is guided by path costs rather than depths. Let C^* be the cost of the optimal solution and assume that every action costs at least ϵ. Then the worst case time and space complexity is $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$, which can be much greater than b^d. When all step costs are equal, we get $O(b^{d+1})$.

When all step costs are the same, uniform cost search is similar to BFS, except that the latter stops once it generates a goal and UCS examines all the nodes at the goals depth to see if one has a lower cost. Therefore UCS does strictly more work by expanding nodes at depth d unnecessarily.

3.4.3 | Depth First Search

Depth First Search always expands the *deepest* node in the current frontier of the search tree. As these nodes are expanded, they are dropped from the frontier, so then the search backs up to the next deepest node that still has unexplored successors.

While BFS uses a FIFO queue, DFS uses a LIFO queue (a stack). As an alternative to the graph search style implementation, it is common to implement depth first search with a recursive function that calls itself on each of its children in turn.

The properties of DFS depend strongly on whether the graph search or tree search version is used. The Graph Search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.

The tree search version is *not* complete. DFTS can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces, but does not avoid the proliferation of redundant paths.

In infinite state spaces, both versions fail if an infinite non-goal path is encountered.

The time complexity of DFGS is bounded by the size of the state space (which may be infinite). A DFTS may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; This can be much greater than the size of the state space.

DFS is not particularly better than BFS in terms of time complexity, but it does well in space complexity. For a DFGS there is no advantage, but a depth first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.

For a state space with branching factor b and maximum depth m, depth first search requires storage of only $O(bm)$ nodes. This is important, because when we looked at BFS the memory requirements were enormous. As a comparison, BFS uses 10 exabytes at depth $d = 16$ and DFTS uses only 156 kilobytes. This is a factor of 7 trillion times less space.

This has lead to DFS being the basic workhorse of many areas of AI, including **constraint satisfaction**, **propositional satisfiability**, and **logic programming**.

A variant of DFS called **Backtracking Search** uses even less memory. In backtacking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only $O(m)$ memory is needed rather than $O(bm)$.

3.4.4 | Depth Limited Search

The inabulity of DFS to handle infinite state spaces can be alleviated by supplying DFS with a predetermined path limit l. That is, nodes at depth l are treated as if they have no successors. This is called **depth limited search**.

Unfortunately, if we choose $l < d$, the shallowest goal is beyond the depth limit.

Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$. We can think of DFS as the special cased of Depth Limited Search where $l = \infty$.

Sometimes depth limits can be based on knowledge of the problem. We can sometimes find the **diameter** of the state space, which gives us a good depth limit.

3.4.5 | Iterative Deepening Depth First Search

Iterative Deepening Search is a general strategy, often used in combinated with depth first tree search, that finds the best depth limit. It does this by gradually increasing the limit - first 0, then 1, then 2, and so

on – until a goal is found. This occurs when $l \to d$.

This combined the benefits of DFS and BFS. Like DFS, its memory requirements are $O(bd)$. Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

In general, iterative deepening is the preffered uninformed search method when the search space is large and the depth of the solution is not known.

It is worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithms optimality guarantees while avoiding its memory requirements. The resulting algorithm is called **iterative lengthening search**. Unfortunately, iterative lengthening occurs substantial overhead compared to uniform cost search.

3.4.6 | Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches - one forward from the initial state and the other backward from the goal - hoping that the two searches meet in the middle.

The motivation is that $b^{d/2} + b^{d/2} << b^d$. We could think of this as the area of two small circles being less than the area of one big circle centered on the start and reaching to the goal.

In BDS we replace the goal test with a check to see whether the frontiers of the two searches intersect. If they do, a solution has been found. It is important to note that the first such solution may not be optimal, even if both the searches are Breadth-First Searches.

The time and space complexity of bidirectional search using BFS in both directions is $O(b^{d/2})$. We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search.

3.4.7 | Comparing Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

h

3.5 | Informed (Heuristic) Search Strategies