Search
Drew McDermott
(2015-09-16)

Search is a term that is used a lot in AI. Its meaning is a bit vague; it seems to come up whenever an algorithm shuffles through a lot of data structures, usually rather simple data structures. The only restriction on the term is that the data are drawn from a discrete space of some kind. Optimization algorithms on continuous spaces are not thought of as doing search in the sense we will discuss.

The focus of this note is what you might call classical search. Suppose you are looking for a solution to a problem. There might be more than one, so we can think of it as a solution space. It’s defined by a solution test, a function that tests whether a candidate solution is in fact in the solution space.

There might or might not be a cost function we are trying to minimize;\(^1\) if no such function is part of the problem statement, then presumably any solution will do. Otherwise, we might require that our algorithm find a minimum-cost solution; or at least favor algorithms that usually find lower-cost solutions before higher-cost solutions. In this note, we neglect the cost function and think about algorithms for finding any solution.

The kind of approach we’ll focus on is to find a superset of the solution space that includes objects that can be considered to be incomplete or defective solutions. The superset is called the state space. A search problem is defined by the following list of items:

1. An initial state, init
2. A successors function, succs, of type State \(\Rightarrow\) List[State].
3. A goal test function, goalTest of type State \(\Rightarrow\) Boolean, which returns true when its argument is in the solution space.\(^2\) The goalTest function should run in polynomial time.

A solution is found by applying the successors function to the initial state, then repeatedly to previously generated states until a solution state is found. This is essentially equivalent to a definition of a nondeterministic-polynomial (NP) problem. Because succs usually returns more than one state, a program solving a problem posed in these terms can be considered

\(^1\)Equivalently, a value function we are trying to maximize.
\(^2\)Or an element of the solution space can be quickly extracted from it.
to be making a nondeterministic choice of successor. The goal-test function embodies the fact that it is easy to verify that a state is a solution to the problem once we have it; it’s finding it that’s difficult.

For example, suppose we’re trying to solve the classical problem of coloring a map using \(N\) colors (or fewer), under the constraint that adjacent regions can’t have the same color. We’ll just represent a map as a graph with one node per region, whose edges represent the adjacency relation. (We won’t require that the graph be planar.)

There is an obvious way to define this as a search problem. The solution space is the set of assignments of colors to nodes (regions) that don’t violate the adjacency constraint. The state space is the set of partial assignments that don’t violate it; a partial assignment might leave some nodes unassigned. The initial state is the empty assignment. The goal test is the test whether all nodes are assigned.

However, a key observation is that the state space, initial state, and successors function are not determined by the solution space. Sometimes the “obviously correct” search space can be improved upon. For example, a different way to approach map coloring is to let the state space be the set of all complete assignments of colors to nodes, regardless of whether the assignments violate some constraints. The initial state can be a random complete assignment, and the successors function can calculate all the assignments reachable by changing one node’s color. (I have no idea if this is an improvement or not.)

In the literature, there is considerable confusion regarding the notion of path to a state. On the one hand, there is a clear definition in terms of the successors function: The path to a state \(s\) is a sequence of states \(s_0, \ldots, s_n\), where \(s_0\) is the initial state, \(s_n = s\), and \(s_{i+1} \in \text{succs}(s_i)\). (There can easily be multiple paths to a state.) I’ll use the term successor path for this sort of sequence. For some problems, there is a domain-dependent notion of path. If the problem is to find a route from one geographical point, \(A\), to another, \(B\), then one state space is the set of routes from \(A\) to some point \(B'\). (A solution is an element of the state space with \(B' = B\).) Unfortunately, this sort of example leads people to conflate the two notions of “path to a state,” the clean mathematical definition, and the domain-dependent notion. The latter idea is better incorporated into the states themselves. In the geographical case, the states happen to be paths, but the notion of successor path can

---

Assuming the colors are interchangeable, it would be better to assign an arbitrary color to some country, to break the symmetry. Otherwise we’ll make the space \(N\) times bigger than it has to be, because for every state there is another equally good state that is a permutation of it. We’ll overlook this in what follows.
be defined independently. Further analysis of these issues must be deferred until cost functions are introduced.

The file `SearchAlg.scala` gives one way of approaching classical search using Scala. The class `SearchProb[P]` defines a search problem as consisting of 3 things: an initial state, a successors function, and a goal test. P is the type of state-space objects. (We parameterize over this type rather than assume a fixed class `State`.)

In the companion object `SearchProb` we put the key function, `trySolve`. It takes 2 arguments: a `SearchProb` and another object of type `SearchManager`. This second object manages a collection `frontier` of states (called, unexcitingly, the `frontier`). We say that a state has been explored when `succs` has been applied to it. All such states are passed to the search manager, which has several jobs to do:

- Keep track of the frontier in an appropriate data structure.
- Decide which state to work on next.
- File new states in the data structure.
- Decide when to delete states that are unlikely to lie on the path to a goal state.
- Detect when a state is equivalent to a previously produced state, and can therefore be discarded immediately.

The last task is important because it is easy to generate more states than can possibly be examined.

Two simple (and standard) search managers are given in `SearchAlg.scala`: `DepthFirstSearchManager` and `BreadthFirstSearchManager`. The first maintains a simple `List` of states. New states are stored at the front of the list. The second maintains a `Queue` (another datatype built in to Scala). It adds the states to the `end` of the queue. The result is that `DepthFirstSearchManager` causes all descendents of an explored state to be explored before any of its siblings, and `BreadthFirstSearchManager` does the opposite.

For example, see `MapColor.scala`, an application of this paradigm to map coloring.

The decision whether to use one of these strategies — or several others — is independent of how the state space is defined. However, frontier management is not the only parameter we can vary. Another is setting a depth limit to a search. The depth of a state is the length of the successor path.
to it. A *depth-limited search* is one in which a parameter \( L \) is supplied such that no state of depth \( \geq L \) is explored.\(^4\)

Another useful strategy is *iterative deepening*, in which a wave of depth-first searches are attempted, with successively deeper depth limits. At first glance, this strategy seems wasteful, because the states produced in previous waves are forgotten when the next wave is attempted. However, it is not much more wasteful than search itself, because the number of states explored on each wave tends to grow exponentially, so the number explored on wave \( k \) is greater than the number explored in all previous waves.

\(^4\)There can be more than one successor path to a state, but we are concerned with particular encounters with a state after applying the successors function \( d \) times. Even realizing that the current state has been seen before is nontrivial. By the way, we could make the depth accessible to a `SearchManager` by including it in the state, but this tactic is unnatural; the depth has nothing to do with the state space, but with the order in which the states are explored.