Machine Learning: A modern approach

Chapter 3: Solving Problems by searching

The simplest agents discussed in Chapter 2 were the reflex agents, which base their actions on a direct mapping from states to actions. Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn. Goal-based agents, on the other hand, consider future actions and the desirability of their outcomes.

3.1: Problem-solving Agents:

Intelligent agents: maximize their performance measure by adopting a goal and satisfying it.

Goal formulation is therefore the first step in problem solving. Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Problem formulation is the second step. It’s the process of deciding what actions and states to consider, given a goal. But in order to do so, we need to consider our environment, which helps us to attain the given goal. Doing so requires us to make assumptions about the environment. We need to define it so that an agent with several immediate options of unknown value can decide what to do by *first examining future actions that eventually lead to states of known value*.

We say that the environment must be

* Observable: The agent always knows the current state
* Discrete: There are only finitely many actions to choose from at each state
* Known: The agent knows which states are reached by each action
* Deterministic: Each action has exactly one outcome

3.1.1: Well-defined problems & solutions:

A problem can be defined formally by five components:

* Initial state:
  + The state in which the agent starts in (s)
* Actions:
  + A description of possible actions available to the agent (a)
  + Given a particular state(s), action(s) returns the set of actions that can be executed in s
* Transition model:
  + A description of what each model does.
  + Specified by a function: result(s, a)
  + Returns the state that results from doing action a in state s
  + State space:
    - function of state, actions and transition model, defining the set of a states reachable from the initial state by any sequence of actions
    - Forms a graph (directed network) in which the nodes are states and the links between nodes are actions
* Goal test:
  + Determines whether a given state is a goal state
* Path cost function:
  + Assigns numeric cost to each path
  + Problem-solving agent chooses a cost function that reflects its own performance measure
  + Described by the sum of the costs of the individual actions along the path
  + The step costs from taking action a in state s to reach state s’ is denoted by: c(s,a,s’)

3.2: Example Problems:

3.2.1: Toy Problems:

*Vacuum world.* Can be formulated as:

* **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 22 = 8 possible world states. A larger environment with n locations has n · 2n states.
* **Initial state**: Any state can be designated as the initial state.
* **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
* **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
* **Goal test**: This checks whether all the squares are clean.
* **Path cost**: Each step costs 1, so the path cost is the number of steps in the path

*8-puzzle world*. Can be formulated as:

* **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
* **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
* **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
* **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
* **Goal test**: This checks whether the state matches the goal configuration shown in Fig- ure 3.4. (Other goal configurations are possible.)
* **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

The 8-puzzle belongs to the family of **sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved. The 15-puzzle (on a 4 × 4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 × 5 board) has around 1025 states, and random instances take several hours to solve optimally.

*8-queen world.* Place eight queens on a chessboard such that no queen attacks another.

There are two main kinds of formulation:

An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.

A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts.

The first incremental formulation one might try is the following:

• **States**: Any arrangement of 0 to 8 queens on the board is a state.  
• **Initial state**: No queens on the board.  
• **Actions**: Add a queen to any empty square.  
• **Transition model**: Returns the board with a queen added to the specified square.

• **Goal test**: 8 queens are on the board, none attacked.

In this formulation, we have 64 · 63 · · · 57 ≈ 1.8 × 1014 possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

* **States**: All possible arrangements of n queens (0 ≤ n ≤ 8), one per column in the leftmost n columns, with no queen attacking another.
* **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8 × 1014 to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly 10400 states to about 1052 states (Exercise 3.5)—a big improvement, but not enough to make the problem tractable.

*Knuth world.*

Knuth conjectured that, starting with the number 4, a sequence of fac- torial, square root, and floor operations will reach any desired positive integer. The problem definition is very simple:

• **States**: Positive numbers.  
• **Initial state**: 4.  
• **Actions**: Apply factorial, square root, or floor operation (factorial for integers only). • **Transition model**: As given by the mathematical definitions of the operations.  
• **Goal test**: State is the desired positive integer.

3.2.2 Real world problems:

*Route finding problems:*

Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward. Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site:

* **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.
* **Initial state**: This is specified by the user’s query.
* **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
* **Transition model**: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
* **Goal test**: Are we at the final destination specified by the user?
* **Path cost**: This depends on monetary cost, waiting time, flight time, customs and im- migration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

A really good system should include contingency plans—such as backup reservations on alternate flights— to the extent that these are justified by the cost and likelihood of failure of the original plan.

*Touring problems:*

are closely related to route-finding problems, but with an important difference. Consider, for example, the problem “Visit every city at least once, starting and ending in Bucharest.” As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited*.

So the initial state would be In(Bucharest),Visited({Bucharest}), a typical intermediate state would be In(Vaslui),Visited({Bucharest,Urziceni,Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

*The traveling salesperson problem (TSP):*

A touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

*A VLSI layout problem:*

Requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.

The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**.

In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so *that they do not overlap and so that there is room for the connecting wires* to be placed between the cells.

Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

*Robot navigation:*

A generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite.

3.3 Searching for solutions:

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state 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.

Then we need to consider taking various actions. We do this by **expanding** the current state, thereby **generating** a new set of state. From each set of states we must again choose to deploy other actions that lead to other states. We do so until we reach a **leaf node**, a node without any further directions, or children as the tree. The set of all leaf nodes available for expansion at any given point is called a **frontier.**

This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.

Search algorithms can also include **loopy paths,** which is a repeated state in a search tree. Fortunately, there is no need to consider loopy paths. We can rely on more than intuition for this: because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed. Loopy paths are a special form of **redundant paths,** which define a situation in which there are more than one way to get from the first state to the second. We say that the paths with higher path costs are **redundant.**

Redundant paths are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles. Route- finding on a **rectangular grid** is a particularly important example in computer games.

In such a grid, each state has four successors, so a search tree of depth d that includes repeated states has 4d leaves; but there are only about 2d2 distinct states within d steps of any given state. For d = 20, **this means about a trillion nodes but only about 800 distinct states.** Thus, following redundant paths can cause a tractable problem to become intractable. This is true even for algorithms that know how to avoid infinite loops.

The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the **closed list**), which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.

Tree search algorithm with explored set data structure. This makes the algorithm remember every expanded node.


The algorithm has another nice property: the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.

A close up of a logo

Description automatically generated

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.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

A screenshot of a cell phone

Description automatically generated

Up to now, we have not been very careful to distinguish between nodes and states, but in writing detailed algorithms it’s important to make that distinction. A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not. Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.

Now that we have nodes, we need somewhere to put them. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**. The operations on a queue are as follows:

• EMPTY?(queue) returns true only if there are no more elements in the queue. • POP(queue) removes the first element of the queue and returns it.  
• INSERT(element, queue) inserts an element and returns the resulting queue.

Queues are characterized by the *order* in which they store the inserted nodes. Three common variants are the first-in, first-out or **FIFO queue**, which pops the *oldest* element of the queue; the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

The explored set can be implemented with a hash table to allow efficient checking for repeated states. With a good implementation, insertion and lookup can be done in roughly constant time no matter how many states are stored. One must take care to implement the hash table with the right notion of equality between states. Sometimes this can be achieved most easily by insisting that the data structures for states be in some **canonical form**; that is, logically equivalent states should map to the same data structure.

3.3.2 Measuring problem-solving performance:

We can evaluate an algorithm’s performance in four ways:

• **Completeness**: Is the algorithm guaranteed to find a solution when there is one?

• **Optimality**: Does the strategy find the optimal solution, with the lowest path cost?

• **Time complexity**: How long does it take to find a solution?

• **Space complexity**: How much memory is needed to perform the search?

In AI, the graph is often represented *implicitly* by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:

* b, the **branching factor** or maximum number of successors of any node
* d, the **depth** of the shallowest goal node (i.e., the number of steps along the path from the root)
* m, the maximum length of any path in the state space.

Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory. For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how “redundant” the paths in the state space are.

To assess the effectiveness of a search algorithm, we can consider:

* **search cost** which typically depends on the time complexity but can also include a term for memory usage
* **total cost**, which combines the search cost and the path cost of the solution found.

For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometers. Thus, to compute the total cost, we have to add milliseconds and kilometers.

3.4 Uninformed search strategies:

The term means that the strategies have no additional information about 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.

3.4.1 Breadth-first search

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. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Therefore, it uses a FIFO method to perform the search query.

Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. 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. This decision is explained below, where we discuss time complexity.

A screenshot of a cell phone

Description automatically generated

So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of *these* generates b more nodes, yielding b3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

* b+b2 +b3 +···+bd =O(bd)

As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be O(bd−1) nodes in the explored set and O(bd) nodes in the frontier, so the space complexity is O(bd), i.e., it is dominated by the size of the frontier!

A screenshot of a cell phone

Description automatically generated

Two lessons can be learned from Figure 3.13:

* *The memory requirements are a bigger problem for breadth-first search than is the execution time.*
* *time is still a major factor*

In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

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).

Again, the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80 + 97 + 101 = 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

A screenshot of a social media post

Description automatically generated

Uniform-cost searches are optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. Then, because step costs are nonnegative, paths never get shorter as nodes are added.

These two facts together imply 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 does not care about the *number* of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost action. Also, this form of search is not optimal when all step costs are the same, because then it is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal’s depth to see if one has a lower cost; thus uniform-cost search **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. The progress of the search is illustrated in Figure 3.16. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those 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.

Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion.

A close up of a map

Description automatically generated

The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used:

The graph-search version, which avoids repeated states and re- dundant paths, is complete in finite state spaces because it will eventually expand every node.

The tree-search version, on the other hand, is *not* complete. Depth-first tree search 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**.

For example, in Figure 3.16, depth- first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the O(bm) 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.

So, time complexity is worse for depth-first search, but for **tree-search versions of depth-first search, space complexity is far better!**

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. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. For a state space with branching factor b and maximum depth m, depth-first search requires storage **of only O(bm) nodes.**

Using the same assumptions as for Figure 3.13 and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth d = 16, a factor of 7 trillion times less space. **This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI.**