## **1 Search problems and solutions**

When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a problem-solving agent, and the computational process it undertakes is called search.

Problem-solving agents use atomic representations, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Agents that use factored or structured representations of states are called planning agents.

A search problem can be defined formally as follows:

* A set of possible states that the environment can be in. This is called the state space.
* The initial state that the agent starts in.
* A set of one or more goal states. Sometimes there is one goal state, sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).
* The actions available to the agent. Given a state *s*, ACTIONS(*s*) returns a finite set of actions that can be executed in *s*. We say that each of these actions is applicable in *s*.
* A transition model, which describes what each action does. RESULT(*s*,*a*) returns the state that results from doing action in state *s*.
* An action cost function, denoted by ACTION-COST(*s*, *a*, *s*’) when we are programming or *c*(*s*, *a*, *s*’) when we are doing math, that gives the numeric cost of applying action *a* in state *s* to reach state *s*’. A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length in miles, or it might be the time it takes to complete the action.

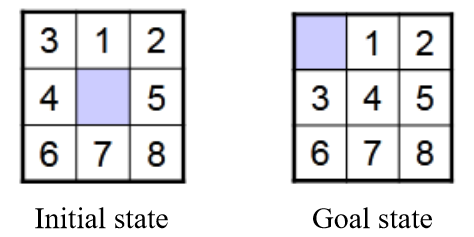
A sequence of actions forms a path, and a solution is a path from the initial state to a goal state. We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs. An optimal solution has the lowest path cost among all solutions.

**2 Example Problems**

The problem-solving approach has been applied to a vast array of task environments. A standardized problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms. A real-world problem, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

**2.1 Standardized problems**

A grid world problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell.

In a sliding-tile puzzle, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. Perhaps the best-known variant is the 8-puzzle, which consists of a grid with eight numbered tiles and one blank space, and the 15-puzzle on a grid. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation of the 8 puzzle is as follows:

STATES: A state description specifies the location of each of the tiles.

INITIAL STATE: Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states.

ACTIONS: While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving Left, Right, Up, or Down. If the blank is at an edge or corner then not all actions will be applicable.

TRANSITION MODEL: Maps a state and action to a resulting state; for example, if we apply Left to the start state, the resulting state has the 4 and the blank switched.

GOAL STATE: Although any state could be the goal, we typically specify a state with the numbers in order.

ACTION COST: Each action costs 1.

**2.2 Real-world problems**

The route-finding problem is defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications, such as Web sites and in-car systems that provide driving directions. The main complications are varying costs due to traffic-dependent delays, and rerouting due to road closures. 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 website:

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: The user’s home airport.

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 new location and the flight’s arrival time as the new time.

GOAL STATE: A destination city. Sometimes the goal can be more complex, such as “arrive at the destination on a nonstop flight.”

ACTION COST: A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flier reward points, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the airlines’ byzantine fare structures. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—what happens if this flight is delayed and the connection is missed?

Touring problems describe a set of locations that must be visited, rather than a single goal destination. The traveling salesperson problem (TSP) is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost < *C* (or in the optimization version, to find a tour with the lowest cost possible). An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles. For example, a search and optimization algorithm for routing school buses in Boston saved $5 million, cut traffic and air pollution, and saved time for drivers and students. In addition to planning trips, search algorithms have been used for tasks such as planning the movements of automatic circuit board drills and of stocking machines on shop floors.

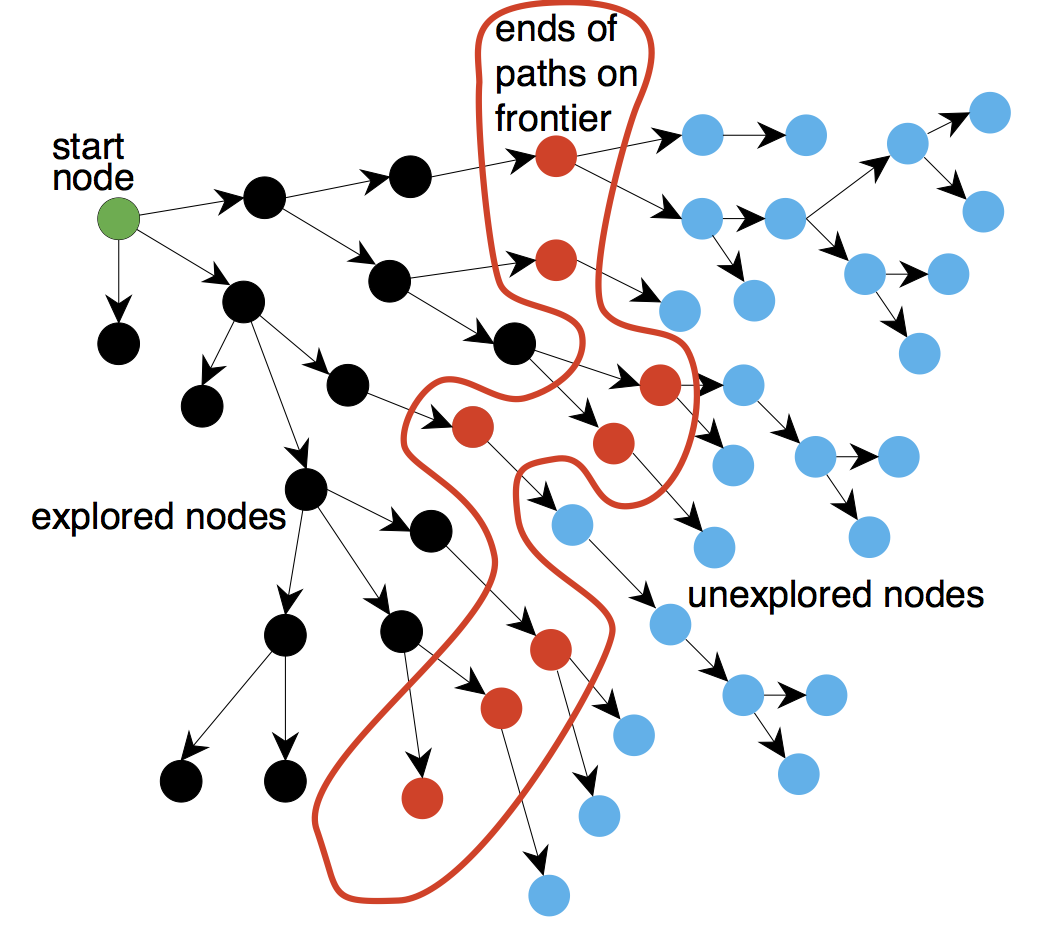
Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking an action in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space.

**3 Search Algorithms**

A search algorithm takes a search problem as input and returns a solution, or an indication of failure. Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

It is important to understand the distinction between the state space and the search tree. The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

When the root node (or start node) of the search tree is at the initial state we can expand the node, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead to, and generating a new node (called a child node or successor node) for each of the resulting states. Each child node now has the root node as its parent node.

Then we must choose which of these 3 child nodes to consider next. This is the essence of search—following up one option now and putting the others aside for later. Suppose we choose to expand the child node in the middle. The result: a set of another 3 unexpanded nodes. Then we expand them too and so on. We call these explored nodes the frontier of the search tree. We say that any state that has had a node generated for it has been reached (whether or not that node has been expanded). The frontier separates two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

**3.1 Best-first search**

A very general approach is called best-first search, in which we choose a node *n*, with minimum value of some evaluation function *f*(*n*). Algorithm:

**function** BEST-FIRST-SEARCH(*problem*, *f*) **returns** a solution node or *failure*

*node* ← NODE(STATE=*problem*.INITIAL)

*frontier* ← a priority queue ordered by *f*, with *node* as an element

*reached* ← a lookup table, with one entry key *problem*.INITIAL and value *node*

**while not** IS-EMPTY(*frontier*) **do**

*node* ← POP(*frontier*)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

**for each** *child* **in** EXPAND(*problem*, *node*) **do**

*s* ← *child*.STATE

**if***s* **is not in** *reached* or *child*.PATH-COST < *reached*[*s*].PATH\_COST **then**

*reached*[*s*] ← *child*

add *child* to *frontier*

**return** *failure*

**function** EXPAND(*problem*, *node*) **yields** nodes

s ← *node*.STATE

**for each** *action* **in** *problem*.ACTIONS(*s*) **do**

*s*’ ← *problem*.RESULT(*s*, *action*)

*cost* ← *node*.PATH-COST + *problem*.ACTION-COST(*s*, *action*, *s*’)

**yield** NODE(STATE=*s*’, PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

On each iteration we choose a node on the frontier with minimum *f*(*n*) value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path. The algorithm returns either an indication of failure, or a node that represents a path to a goal. By employing different *f*(*n*) functions, we get different specific algorithms.

**3.2 Search data structures**

Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with four components:

* *node*.STATE: the state to which the node corresponds;
* *node*.PARENT: the node in the tree that generated this node;
* *node*.ACTION: the action that was applied to the parent’s state to generate this node;
* *node*.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, *g*(*node*) is used as a synonym for PATH-COST.

Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution. We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are:

* IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
* POP(*frontier*) removes the top node from the frontier and returns it.
* TOP(*frontier*) returns (but does not remove) the top node of the frontier.
* ADD(*node*, *frontier*) inserts node into its proper place in the queue.

Three kinds of queues are used in search algorithms:

* A priority queue first pops the node with the minimum cost according to some evaluation function, *f*. It is used in best-first search.
* A FIFO queue or first-in-first-out queue first pops the node that was added to the queue first; it is used in breadth-first search.
* A LIFO queue or last-in-first-out queue (a stack) pops first the most recently added node; it is used in depth-first search.

The *reached* states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

The search tree can sometimes include a path from *A* to *B* and back to *A* again. We say that *A* is a repeated state in the search tree, generated in this case by a cycle (also known as a loopy path). So even though the state space has only 20 states, the complete search tree is infinite because there is no limit to how often one can traverse a loop.

A cycle is a special case of a redundant path. For example, we can get to *B* via the path *A*–*B* or the path *A*–*K*–*S*–*B*. This second path is redundant—it’s just a worse way to get to the same state—and need not be considered in our quest for optimal paths.

**3.3 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, and to correctly report failure when there is not?

To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. In finite state spaces that is straightforward to achieve: as long as we keep track of paths and cut off ones that are cycles, eventually we will reach every reachable state.

To be complete, a search algorithm must be systematic in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state. For example, on the infinite grid, one kind of systematic search is a spiral path that covers all the cells that are steps from the origin before moving out to cells that are steps away. Unfortunately, in an infinite state space with no solution, a sound algorithm needs to keep searching forever; it can’t terminate because it can’t know if the next state will be a goal.

COST OPTIMALITY: Does it find a solution with the lowest path cost of all solutions?

TIME COMPLEXITY: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.

SPACE COMPLEXITY: How much memory is needed to perform the search?

Time and space complexity are considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state-space graph |*V*| + |*E*|, where |*V*| is the number of vertices (state nodes) of the graph and |*E*| is the number of edges (distinct state/action pairs). This is appropriate when the graph is an explicit data structure. But in many AI problems, the graph is represented only implicitly by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of *d*, the depth or number of actions in an optimal solution; *m*, the maximum number of actions in any path; and *b*, the branching factor or number of successors of a node that need to be considered.

**4 Uninformed Search Strategies**

An uninformed search algorithm is given no clue about how close a state is to the goal(s).

**4.1 Breadth-first search**

When all actions have the same cost, an appropriate strategy is breadth-first search, 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. This is a systematic search strategy that is therefore complete even on infinite state spaces. We could implement breadth-first search as a call to Best-First-Search where the evaluation function is the depth of the node—that is, the number of actions it takes to reach the node.

A FIFO queue will be faster than a priority queue, and will give us the correct order of nodes: 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. In addition, *reached* can be a set of states rather than a mapping from states to nodes, because once we’ve reached a state, we can never find a better path to the state. That also means we can do an early goal test, checking whether a node is a solution as soon as it is generated, rather than the late goal test that best-first search uses, waiting until a node is popped off the queue. The algorithm with the early-goal efficiency enhancements:

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*

*node* ← NODE(*problem*.INITIAL)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

*frontier* ← a FIFO queue, with *node* as an element

*reached* ← {*problem*.INITIAL}

**while not** IS-EMPTY(*frontier*) **do**

*node* ← POP(*frontier*)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

**for each** *child* **in** EXPAND(*problem*, *node*) **do**

*s* ← *child*.STATE

**if** *problem*.IS-GOAL(*s*) **then return** *child*

**if***s* **is not in** *reached* **then**

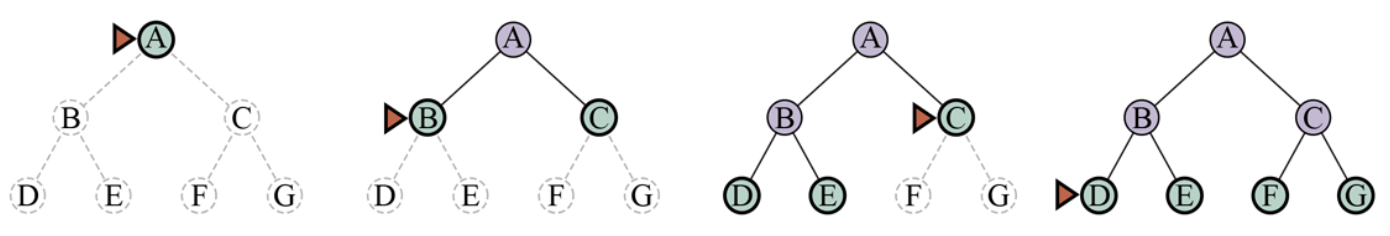
add *s* to *reached*

add *child* to *frontier*

**return** *failure*

**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*

**return** BEST-FIRST-SEARCH(*problem*, PATH-COST)



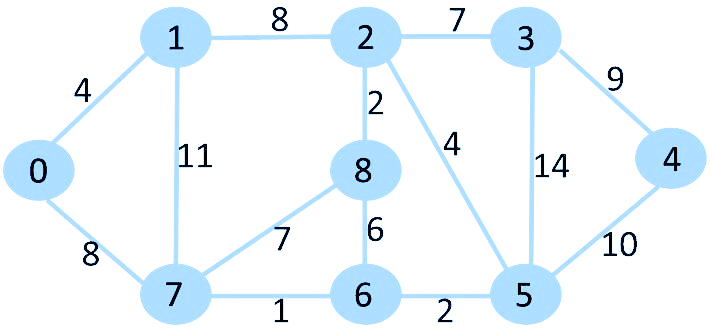
Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth it has already generated all the nodes at depth so if one of them were a solution, it would have been found. That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don’t have that property. It is complete in either case.

All the nodes remain in memory, so both time and space complexity are *O*(*bd*). Exponential bounds like that are scary. As a typical real-world example, consider a problem with branching factor *b* = 10 processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth would take less than 3 hours, but would require 10 terabytes of memory. The memory requirements are a bigger problem for breadth-first search than the execution time. But time is still an important factor. At depth *d* = 14 even with infinite memory, the search would take 3.5 years. In general, exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.

**4.2 Dijkstra’s algorithm or uniform-cost search**

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra’s algorithm by the theoretical computer science community, and uniform-cost search by the AI community. The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST-FIRST-SEARCH with PATH-COST as the evaluation function, as shown in the algorithm above.

The complexity of uniform-cost search is characterized in terms of *C*\*, the cost of the optimal solution, and , a lower bound on the cost of each action, with . Then the algorithm’s worst-case time and space complexity is which can be much greater than *bd*. This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, is just *bd*+1 and uniform-cost search is similar to breadth-first search.

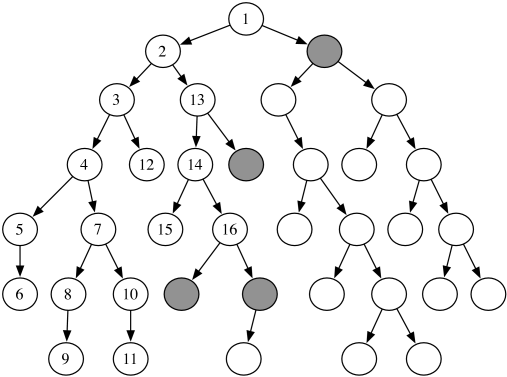


Optimal path for 0→4: 0→7→6→5→4. Cost=21

Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are .

**4.3 Depth-first search**

Depth-first search always expands the deepest node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function *f* is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. Search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then “backs up” to the next deepest node that still has unexpanded successors. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.



For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.

In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles. Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete.

Then why would anyone consider using depth-first search rather than breadth-first or best-first? The answer is that for problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don’t keep a *reached* table at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere.

For a finite tree-shaped state-space, a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only *O*(*bm*) where *b* is the branching factor and *m* is the maximum depth of the tree. Some problems that would require exabytes of memory with breadth-first search can be handled with only kilobytes using depth-first search. Because of its parsimonious use of memory, depth-first tree-like search has been adopted as the basic workhorse of many areas of AI, including constraint satisfaction, propositional satisfiability, and logic programming.

A variant of depth-first search called backtracking search uses even less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by modifying the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of *O*(*m*) actions; a significant savings over *O*(*bm*) states for depth-first search. With backtracking we also have the option of maintaining an efficient set data structure for the states on the current path, allowing us to check for a cyclic path in *O*(1) time rather than *O*(*m*). For backtracking to work, we must be able to *undo* each action when we backtrack. Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly.

**4.4 Depth-limited and iterative deepening search**

To keep depth-first search from wandering down an infinite path, we can use depth-limited search, a version of depth-first search in which we supply a depth limit *l*, and treat all nodes at depth *l* as if they had no successors. The time complexity is *O*(*bl*) and the space complexity is *O*(*bl*). Unfortunately, if we make a poor choice for *l* the algorithm will fail to reach the solution, making it incomplete again. Algorithm:

**function** ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*

**for** *depth* = 0 **to**  **do**

*result* ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)

**if** *result* *cutoff* **then return** *result*

**function** DEPTH-LIMITED-SEARCH(*problem*, *l*) **returns** a node or *failure* or *cutoff*

*frontier* ← a LIFO queue(stack) with NODE(*problem*.INITIAL) as an element

*result* ← *failure*

**while not** IS-EMPTY(*frontier*) do

node ← POP(*frontier*)

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

**if** DEPTH(*node*) > *l* **then**

*result* ← *cutoff*

**else if not** IS-CYCLE(*node*) **do**

**for each** *child***in** EXPAND(*problem*, *node*) **do**

add *child* to *frontier*

**return** *result*

Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than *l*. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check all cycles, then the algorithm may get caught in a loop.

Iterative deepening search solves the problem of picking a good value for *l* by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the failure value rather than the cutoff value. Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: when there is a solution, or on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.

The time complexity is *O*(*bd*) when there is a solution, or *O*(*bm*) when there is none. Each iteration of iterative deepening search generates a new level, in the same way that breadth-first search does, but breadth-first does this by storing all nodes in memory, while iterative-deepening does it by repeating the previous levels, thereby saving memory at the cost of more time.

**4.5 Bidirectional search**

The algorithms described above start at an initial state and can reach any one of multiple possible goal states. An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that *bd*/2 + *bd*/2 is much less than *bd*.

For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state *s*’ is a successor of *s* in the forward direction, then we need to know that *s* is a successor of *s*’ in the backward direction. We have a solution when the two frontiers collide.

There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. We describe bidirectional best-first search. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier.

The general best-first bidirectional search algorithm is shown below. We pass in two versions of the problem and the evaluation function, one in the forward direction (subscript *F*) and one in the backward direction (subscript *B*). When the evaluation function is the path cost, we know that the first solution found will be an optimal solution, but with different evaluation functions that is not necessarily true. Therefore, we keep track of the best solution found so far, and might have to update that several times before the TERMINATED test proves that there is no possible better solution remaining.

**function** BiBF(*problemF, fF, problemB, fB*) **returns** a solution node or *failure*

*nodeF* ← NODE(*problemF*.INITIAL)

*nodeB* ← NODE(*problemB*.INITIAL)

*frontierF* ← a priority queue ordered by *fF*, with *nodeF* as an element

*frontierB* ← a priority queue ordered by *fB*, with *nodeB* as an element

*reachedF* ← a lookup table, with one key *nodeF*.STATE and value *nodeF*

*reachedB* ← a lookup table, with one key *nodeB*.STATE and value *nodeB*

*solution* ← *failure*

**while not** TERMINATED(*solution*, *frontierF,* *frontierB*) **do**

**if** *fF*(TOP(*frontierF*)) < *fB*(TOP(*frontierB*))**then**

*solution* ← PROCEED(*F, problemF, frontierF, reachedF, reachedB, solution*)

**else** *solution* ← PROCEED(*B, problemB, frontierB, reachedB, reachedF, solution*)

**return** *solution*

//expand node on frontier, check against the other frontier in reached2

//*dir* means direction: *F* for forward, *B* for backward

**function** PROCEED(*dir*, *problem*, *frontier, reached, reached2, solution*) **returns** a solution

*node* ← POP(*frontier*)

**for each** *child* **in** EXPAND(*problem*, *node*) **do**

*s* ← *child*.STATE

**if***s* **is not in** *reached* **or** PATH-COST(*child*) < PATH\_COST(*reached*[*s*]) **then**

*reached*[*s*] ← *child*

add *child* to *frontier*

**if** *s* **is in** *reached2* **then**

*solution2* ← JOIN-NODES(*dir, child, reached2*[*s*])

**if** PATH-COST(*solution2*) < PATH\_COST(*solution*) **then**

*solution* ← *solution2*

**return** *solution*

Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

**5 Informed (Heuristic) Search Strategies**

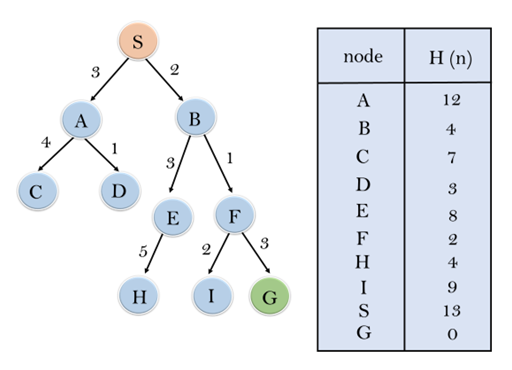
An informed search strategy uses domain-specific hints about the location of goals and can find solutions more efficiently than an uninformed strategy. The hints come in the form of a heuristic function, denoted *h*(*n*), which equals estimated cost of the cheapest path from the state at node *n* to a goal state.

It may seem odd that the heuristic function operates on a node, when all it really needs is the node’s state. It is traditional to use *h*(*n*) rather than *h*(*s*) to be consistent with the evaluation function *f*(*n*) and the path cost *g*(*n*).

**5.1 Greedy best-first search**

Greedy best-first search is a form of best-first search that expands first the node with the lowest *h*(*n*) value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function *f*(*n*) = *h*(*n*). Algorithm is as follows:

1. Expand the nodes of the initial node, put them into the OPEN list, and put the initial node in the CLOSED list.
2. While the OPEN list is not empty, remove the node *n* from the OPEN list which has the lowest value of H(*n*), and place it in the CLOSED list. Its successors are put into the OPEN list.
3. Expand the node *n*, and generate the successors of node *n*.
4. Check each successor of node *n*. If any successor node is a goal node, then return success and terminate the search, else proceed to step 6.
5. For each successor node, the algorithm checks for evaluation function *f*(*n*), and then checks if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
6. Return to step 2.

Example for the graph on the left:

1. OPEN: [A, B]; CLOSED: [S].

2. H(B)=4, H(A)=12 ⇒ B is expanded and put into the CLOSED list.

OPEN: [E, F, A]; CLOSED: [S, B].

3. H(F)=2, H(E)=8 ⇒ F is expanded.

OPEN [I, G, E, A], CLOSED [S, B, F].

4. H(G)=0, H(I)=9 ⇒ G.

Open [I, E, A], Closed [S, B, F, G].

The solution path: S → B → F → G

Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is *O*(|*V*|). With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching *O*(*bm*).

**5.2 A\* search**

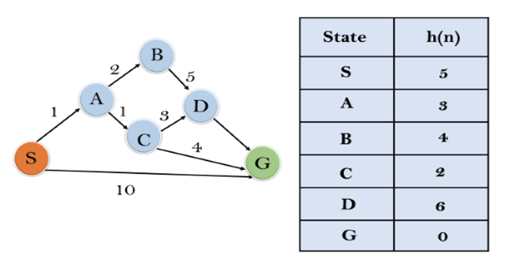
The most common informed search algorithm is A\* search (pronounced “A-star search”), a best-first search that uses the evaluation function

*f*(*n*) = *g*(*n*) + *h*(*n*).

where *g*(*n*) is the path cost from the initial state to node *n* and *h*(*n*) is the estimated cost of the shortest path from *n* to a goal state, so we have *f*(*n*) = estimated cost of the best path that continues from *n* to a goal.

In A\* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs, and this sum is called a fitness number. At each point in the search space, only those nodes are expanded which have the lowest value of *f*(*n*), and the algorithm terminates when the goal node is found. Algorithm of A\* search:

1. Place the starting node in the OPEN list.
2. If the OPEN list is empty then return failure and stop.
3. Select the node from the OPEN list which has the smallest value of evaluation function (*g*+*h*).
4. If node *n* is the goal node then return success and stop.
5. Else expand node *n* and generate all of its successors, and put *n* into the CLOSED list.
6. For each successor *n'*, check whether *n'* is already in the OPEN or CLOSED list, if not then compute the evaluation function for *n'* and place it into the OPEN list.
7. Else if node *n'* is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest *g*(*n'*) value.
8. Return to Step 2.



Example for the graph on the left:

The heuristic value of all states is given in the table, so the *f*(*n*)= *g*(*n*) + *h*(*n*), where *g*(*n*) is the cost to reach any node from the start state.

Initialization: {(S, 5)}

1. {(S → A, 4), (S → G, 10)}

2. {(S → A → C, 4), (S → A → B, 7), (S → G, 10)}

3. {(S → A → C → G, 6), (S → A → C → D, 11), (S → A → B, 7), (S → G, 10)}

4. The solution: S → A → C → G; it provides the optimal path with cost 6.

Whether A\* is cost-optimal depends on certain properties of the heuristic. A key property is admissibility: an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.)

A slightly stronger property is called consistency. A heuristic *h*(*n*) is consistent if, for every node *n* and every successor *n'* of *n* generated by an action we have: *h*(*n*) ≤ *c*(*n*, *a*, *n'*) + *h*(*n'*).

Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A\* is cost-optimal. In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path, so we never have to re-add a state to the frontier, and never have to change an entry in *reached*. But with an inconsistent heuristic, we may end up with multiple paths reaching the same state, and if each new path has a lower path cost than the previous one, then we will end up with multiple nodes for that state in the frontier, costing us both time and space. Because of that, some implementations of A\* take care to only enter a state into the frontier once, and if a better path to the state is found, all the successors of the state are updated (which requires that nodes have child pointers as well as parent pointers).

**5.3 Recursive best first search**

A\* search has many good qualities, but it expands a lot of nodes. We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are “good enough”—what we call satisficing solutions. If we allow A\* search to use an inadmissible heuristic—one that may overestimate—then we risk missing the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of nodes expanded.

Iterative-deepening A\* search (IDA\*) is to A\* what iterative-deepening search is to depthfirst: IDA\* gives us the benefits of A\* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory.

**function** RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution or *failure*

*solution*, *fvalue*  ← *RBFS(problem, NODE(problem.INITIAL),*∞*)*

**return** *solution*

**function** RBFS(*problem*, *node*, *f\_limit*) **returns** a solution or *failure,* and a new *f*-cost limit

**if** *problem*.IS-GOAL(*node*.STATE) **then return** *node*

*successors* ← LIST(EXPAND(*node*))

**if** *successors* is empty **then return** *failure,* ∞

**for each** *s* **in** *successors* **do //**update *f* with value from previous search

*s.f* ← max(*s.*PATH-COST *+ h*(*s*)*, node.f*)

**while** *true* **do**

*best* ← the node in *successors* with lowest *f*-value

**if** *best.f* > *f\_limit* **then return** *failure*, *best.f*

*alternative* ← the second-lowest *f*-value among *successors*

*result*, *best.f* ← RBFS(*problem*, *best*, min(*f\_limit*, *alternative*))

if result ≠ *failure* **then return** *result, best.f*

RBFS is somewhat more efficient than IDA\*, but still suffers from excessive node regeneration. RBFS is optimal if the heuristic function *h*(*n*) is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. It expands nodes in order of increasing *f*-score, even if *f* is nonmonotonic.

IDA\* and RBFS suffer from using too little memory. Between iterations, IDA\* retains only a single number: the current *f*-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexploring the same states many times over.

One way to characterize the quality of a heuristic is the effective branching factor *b*\*. If the total number of nodes generated by A\* for a particular problem is *N* and the solution depth is *d* then *b*\* is the branching factor that a uniform tree of depth *d* would have to have in order to contain *N* + 1 nodes. Thus,

*N* + 1 = 1 + *b*\* + (*b*\*)2 + ⋯ + (*b*\*)*d*.

For example, if A\* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually for a specific domain (such as 8-puzzles) it is fairly constant across all nontrivial problem instances. Therefore, experimental measurements of *b*\* on a small set of problems can provide a good guide to the heuristic’s overall usefulness. A well-designed heuristic would have a value of *b*\* close to 1, allowing fairly large problems to be solved at reasonable computational cost.