UNIT II PROBLEM SOLVING

Heuristic search strategies –heuristic functions -Local search and optimization problems – local search in continuous space –search with non-deterministic actions –search in partially observable environments –online search agents and unknown environments

Informed (Heuristic) Search Strategies

Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies

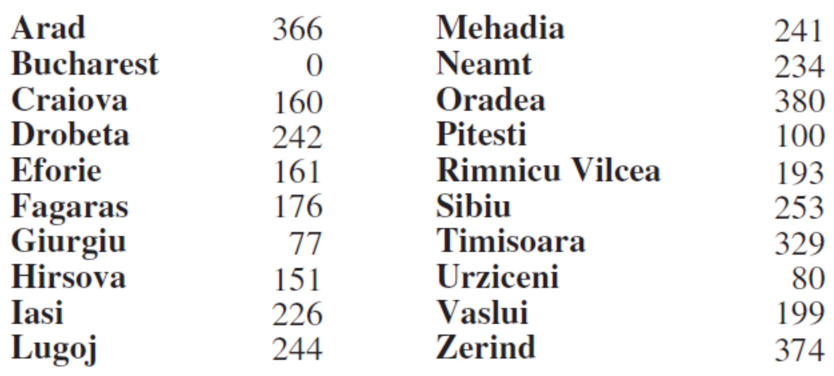
- Greedy best-first search
- A* search
- Search contours
- Satisficing search: Inadmissible heuristics and weighted A*
- Memory-bounded search
- Bidirectional heuristic search

Greedy best-first search

Greedy best-first search is a form of best-first search that expands first the node with the lowest value—the node that appears to be closest to the goal— that is likely to lead to a solution quickly So the evaluation function is f(n) = h(n).
h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

Consider the route-finding problem of Romania where a **straight-line distance** heuristic, $\mathbf{h_{SLD}}$ is used to reach the goal Bucharest from the initial state Arad. The values of $\mathbf{h_{SLD}}$ cannot be computed from the problem description itself (that is, the ACTIONS and RESULT functions), it also takes a certain amount of world knowledge to know that is correlated with actual road distances and therefore, a useful heuristic.

Consider the example figure that shows the progress of a greedy best-first search using to find a path from Arad to Bucharest.



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal.

For this particular problem, greedy best-first search using **h_{SLD}** finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results .

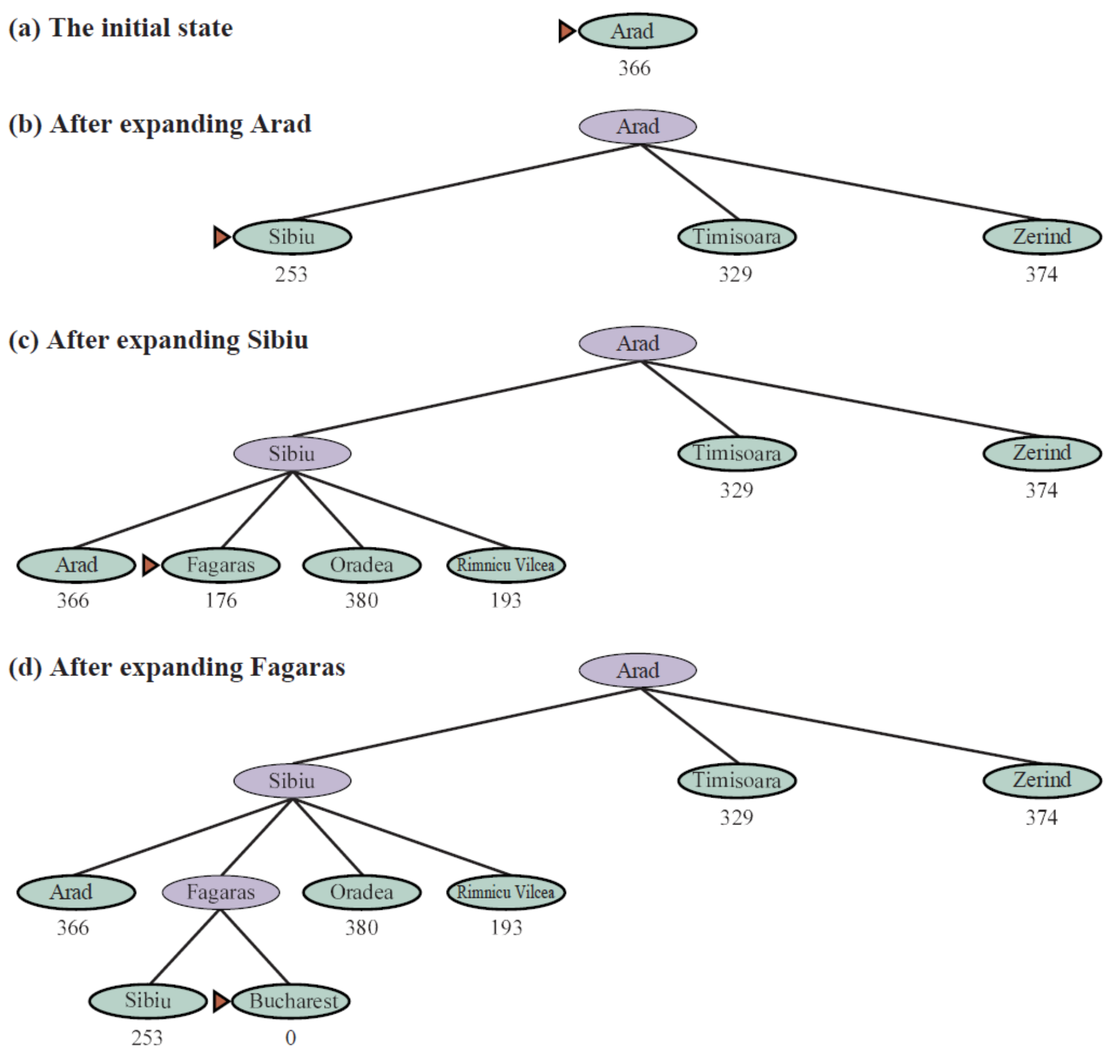


Figure : Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic hSLD. Nodes are labeled with their h-values.

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).

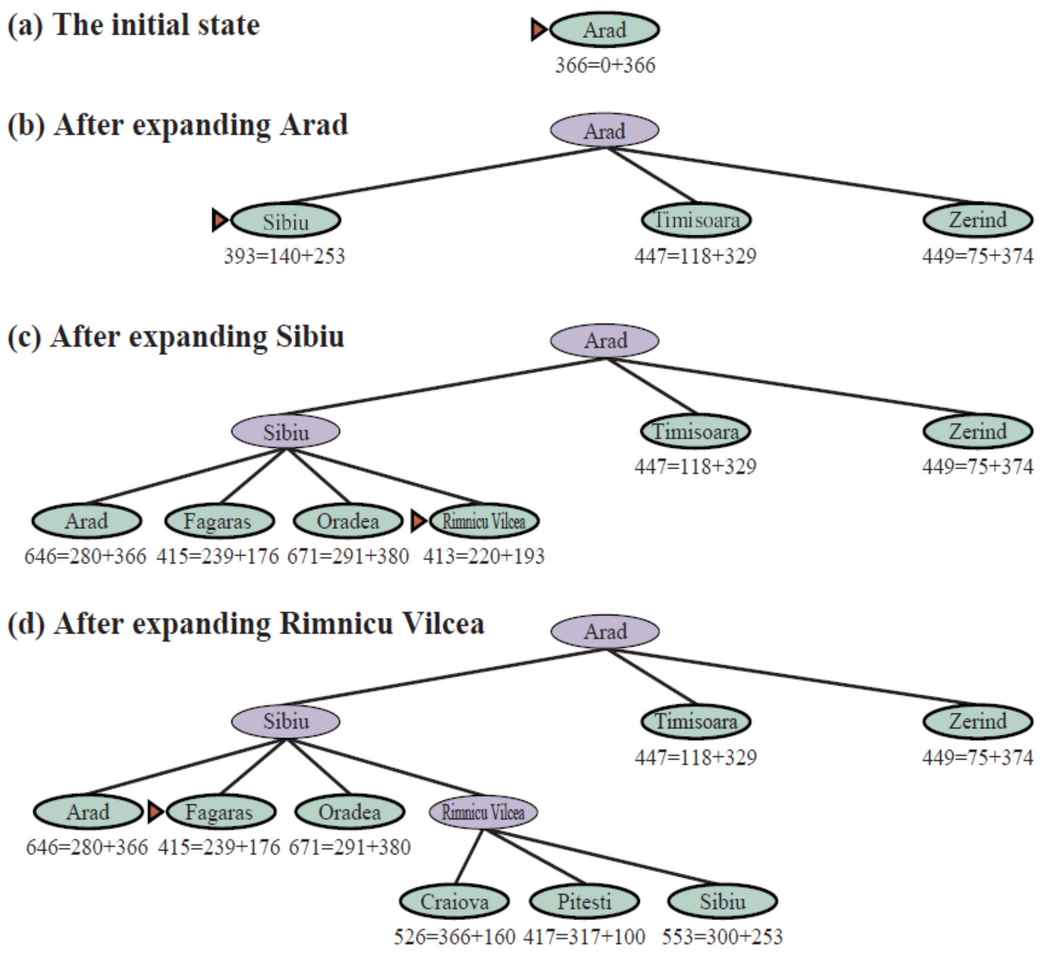
A* Search

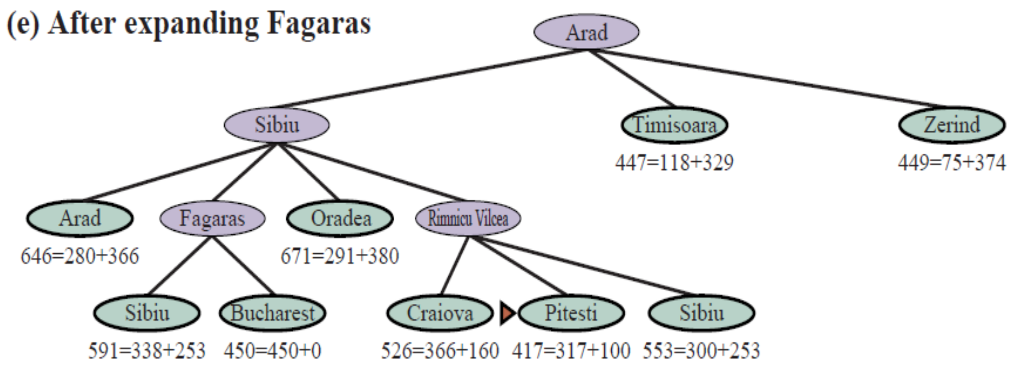
The most common informed search algorithm is **A* search** (pronounced "A-star search"), a best-first search that uses the evaluation function

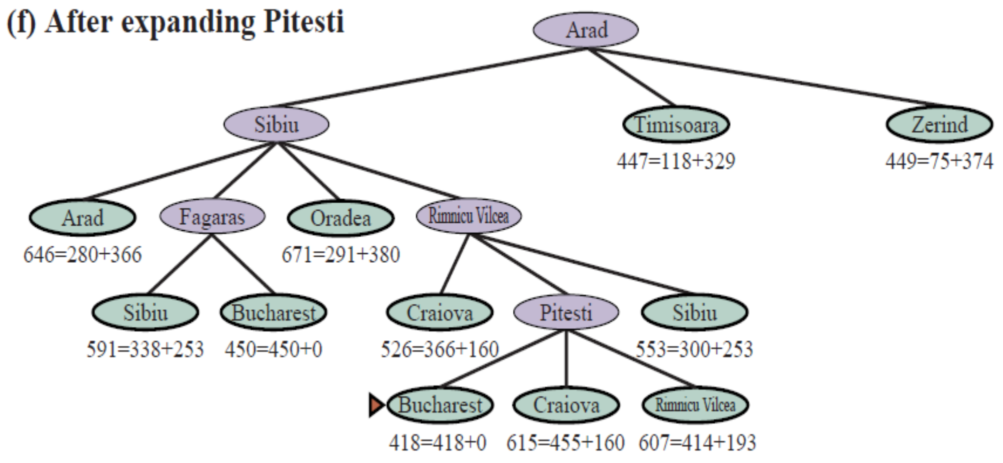
$$f(n) = g(n) + h(n)$$

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

Consider the figure that shows the progress of an A* search with the goal of reaching Bucharest.
Stages in an A∗ search for Bucharest. Nodes are labeled with f = g + h. The h values are the straight-line distances to Bucharest taken from Figure







The values of g are computed from the action costs in the figure and the values of **h**$_{\textbf{SLD}}$ are also given in Figure . Bucharest first appears on the frontier at step (e), but it is not selected for expansion (and thus not detected as a solution) because at f=450 it is not the lowest-cost node on the frontier. Another way is to find the solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. At step (f), a different path to Bucharest is now the lowest-cost node, at f=418 so it is selected and detected as the optimal solution

A* is cost-optimal and depends on certain properties of the heuristic.
1)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*.)
2) 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 a given by:
$h(n) \leq c(n, a, n') + h(n')$
This is a form of the **triangle inequality**, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides

Triangle inequality: If the heuristic h is **consistent**, then the single number h(n) will be less than the sum of the cost c(n, a, a′) of the action from n to n' plus the heuristic estimate h(n′).

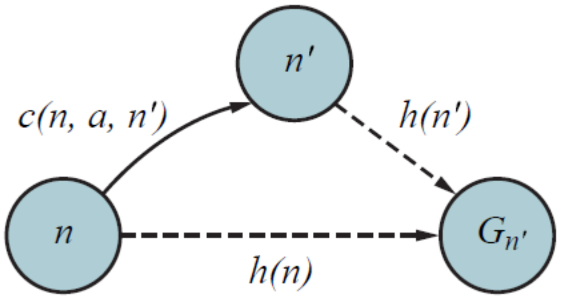


Figure: Triangle Inequality

Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal.

With an inadmissible heuristic, A* may or may not be cost-optimal. The two cases are:
First, if there is even one cost-optimal path on which h(n) is admissible for all nodes on the path , then that path will be found, no matter what the heuristic says for states off the path.
Second, if the optimal solution has cost C* and the second-best has cost C_2 and if h(n)overestimates some costs, but never by more than C_2-C* then A* is guaranteed to return cost-optimal solutions.

Search contours

A useful way to visualize a search is to draw **contours** in the state space, just like the contours in a topographic map. Figure shows an example.

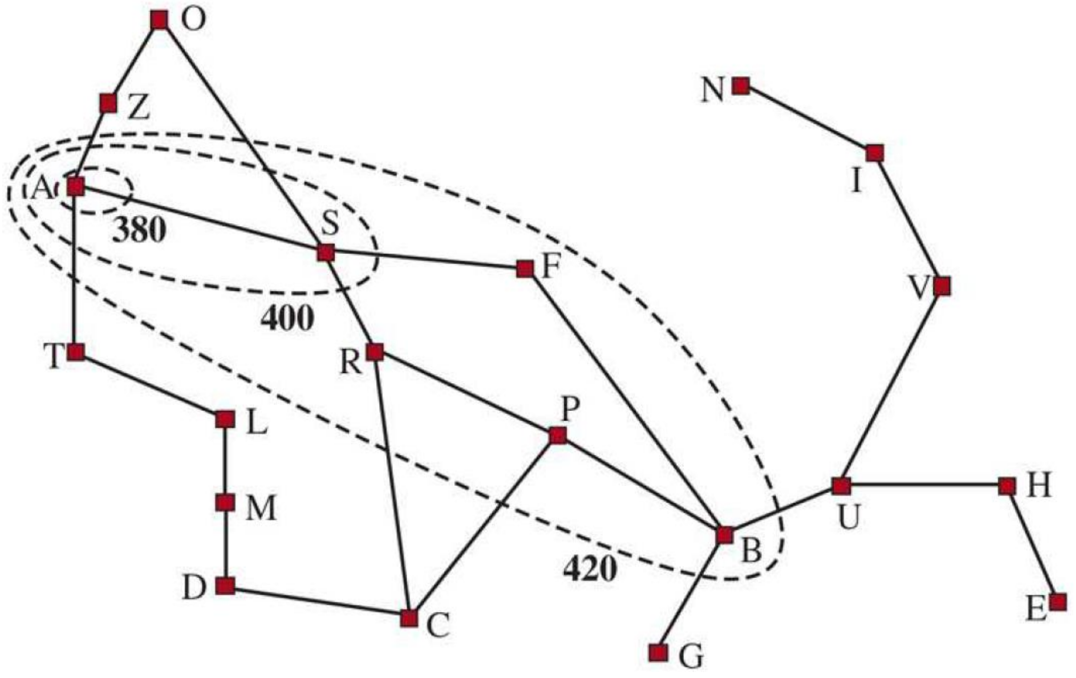


Figure:Map of Romania showing contours at f=380,f=400 and f=420 with Arad as the start state. Nodes inside a given contour have f=g+h costs less than or equal to the contour value.

Inside the contour labelled 400, all nodes have $f(n) = g(n) + h(n) \leq 400$ and so on. As A* expands the frontier node of lowest -cost, an A* search fans out from the start node, adding nodes in concentric bands of increasing -cost.

In uniform-cost search contours have g-cost and not g+h.The contours with uniform-cost search will be "circular" around the start state, spreading out equally in all directions with no preference towards the goal. In A* search using a good heuristic, the bands will stretch toward a goal state and become more narrowly focused around an optimal path

As the path is extended, the costs are monotonic: the path cost always increases along a path, because action costs are always positive.

If C* is the cost of the optimal solution path, then the following holds:
1)A* expands all nodes that can be reached from the initial state on a path where every node on the path has $f(n)< C^*$Then these are surely expanded nodes.
2)A* might then expand some of the nodes right on the "goal contour" (where $f(n)= C^*$)before selecting a goal node.
3)A* expands no nodes with has $f(n)>C^*$

A* with a consistent heuristic is optimally efficient in the sense that any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A*

A* is efficient because it **prunes** away search tree nodes that are not necessary for finding an optimal solution. Thus A* search is complete, cost-optimal, and optimally efficient among all such algorithms

Satisficing search: Inadmissible heuristics and weighted A*

A* search has many good qualities, but it expands a lot of nodes. **Satisficing** solutions explores fewer nodes(taking less time and space) , willing to accept solutions that are suboptimal.

If A* search use an inadmissible heuristic it may miss the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of nodes expanded.

Weighted A* search weights the heuristic value more heavily, leading to the evaluation function $f(n) = g(n) +W \times h(n)$, for some $W > 1$.

There are a variety of suboptimal search algorithms, which can be characterized by the criteria for what counts as "good enough." In bounded suboptimal search, a solution is looked for that is guaranteed to be within a constant factor W of the optimal cost. Weighted A* provides this guarantee.

In bounded-cost search, a solution is looked for whose cost is less than some constant A and in unbounded-cost search, a solution is accepted of any cost, as long as it can be found quickly.

An example of an unbounded-cost search algorithm is **speedy search**, which is a version of greedy best-first search that uses as a heuristic the estimated number of actions required to reach a goal, regardless of the cost of those actions. Thus, for problems where all actions have the same cost it is the same as greedy best-first search, but when actions have different costs, it tends to lead the search to find a solution quickly, even if it might have a high cost.

Memory-bounded search

The main issue with A* is its use of memory. Memory is split between the *frontier* and the *reached* states. In the bestfirst search implementation, a state that is on the frontier is stored in two places: as a node in the frontier (what to expand next can be decided) and as an entry in the table of reached states (to keep track of the visited state).

New algorithms that are designed to conserve memory usage:

Beam search limits the size of the frontier. The easiest approach is to keep only the nodes with the best f-scores, discarding any other expanded nodes. This of course makes the search incomplete and suboptimal, but makes good use of available memory, and the algorithm executes fast because it expands fewer nodes. For many problems it can find good near-optimal solutions.

Uniform-cost or A* search spreads out everywhere in concentric contours, while beam search spreads as exploring only a focused portion of those contours, the portion that contains the best candidates.

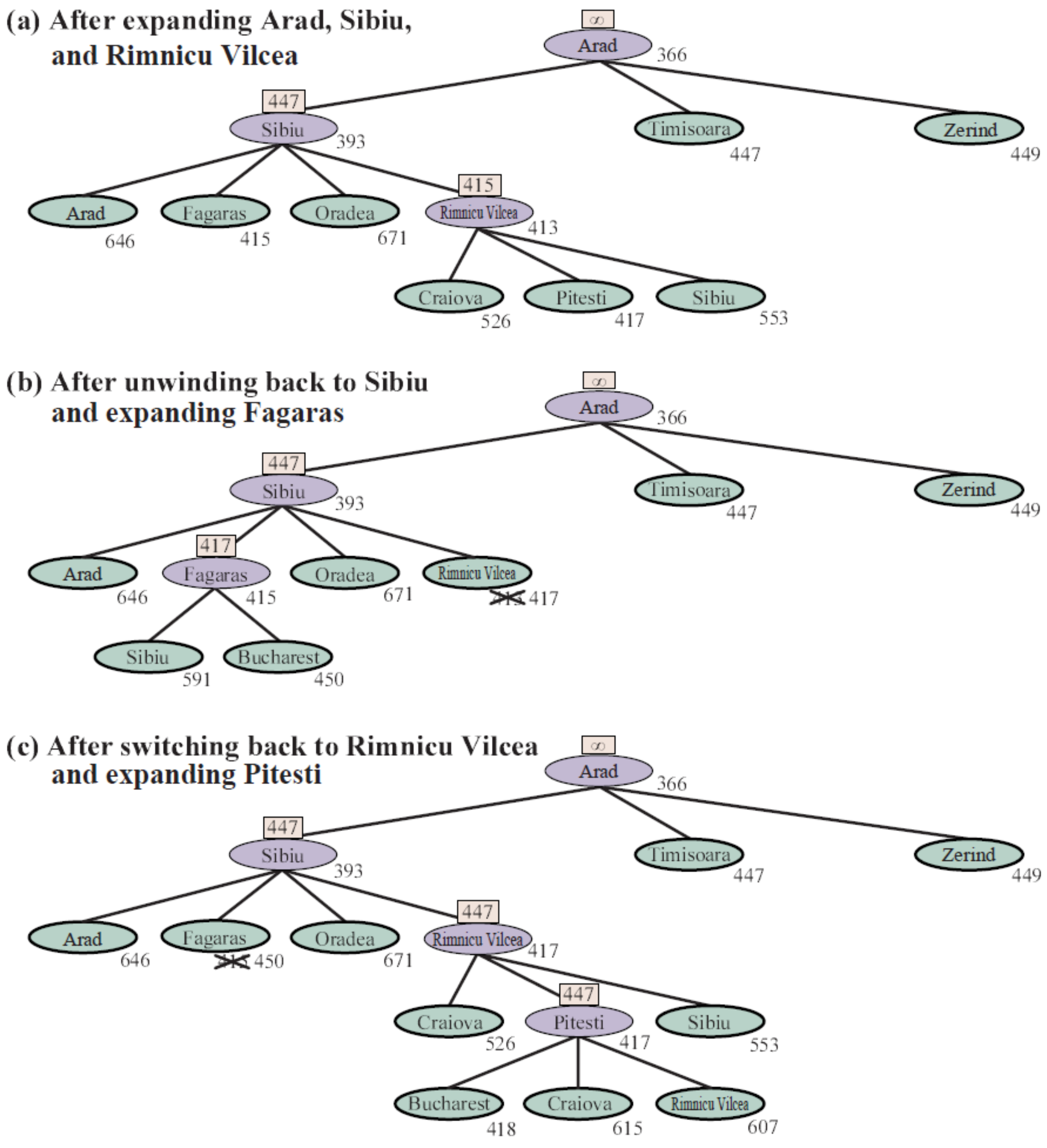
An alternative version of beam search doesn’t keep a strict limit on the size of the frontier but instead keeps every node whose f-score is within ∂ of the best f-score.

Iterative-deepening A* search (IDA*) gives 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.

In standard iterative deepening the cutoff is the depth, which is increased by one each iteration. In IDA* the cutoff is the f-cost (g+h); at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

Recursive best-first search (RBFS) attempts to mimic the operation of standard best-first search, but using only linear space. RBFS resembles a recursive depthfirst search, but rather than continuing indefinitely down the current path, it uses the f_limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it’s worth reexpanding the subtree at some later time.

Figure shows how RBFS reaches Bucharest.



Stages in an RBFS search for the shortest route to Bucharest. The *-limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Figure : The algorithm for recursive best-first search.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), $\infty$)
  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, $\infty$
   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.

To determine how much memory is available, and allow an algorithm to use all of it. Two algorithms used are **MA*** (memory-bounded A*) and **SMA*** (simplified MA*). SMA* is—well—simpler, SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node are forgotten, then we will not know which way to go from n but we will still have an idea of how worthwhile it is to go anywhere from n

SMA* expands the best leaf and deletes the worst leaf.To avoid selecting the same node for deletion and expansion, SMA* expands the *newest* best leaf and deletes the *oldest* worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors. SMA* is complete if there is any reachable solution—that is, if d the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, action costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the reached set.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of **thrashing** in disk paging systems.) Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, *memory limitations can make a problem intractable from the point of view of computation time*. Although no current theory explains the tradeoff between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

Bidirectional heuristic search

With unidirectional best-first search ,the evaluation function f(n)=g(n)+h(n) gives us an A* search that is guaranteed to find optimal-cost solutions while being optimally efficient in the number of nodes expanded.

With bidirectional best-first search there is no guarantee to get an optimal-cost solution, nor that it would be optimally efficient, even with an admissible heuristic. With bidirectional search, it turns out that it is not individual nodes but rather *pairs* of nodes (one from each frontier) that can be proved to be surely expanded, so any proof of efficiency will have to consider pairs of nodes.

Consider the new notation, fF (n) = gF (n) + hF (n) for nodes going in the forward direction (with the initial state as root) and fB (n) = gF (n) + hF (n) for nodes in the backward direction (with a goal state as root). Although both forward and backward searches are solving the same problem, they have different evaluation functions because, for example, the heuristics are different depending on whether you are striving for the goal or for the initial state.

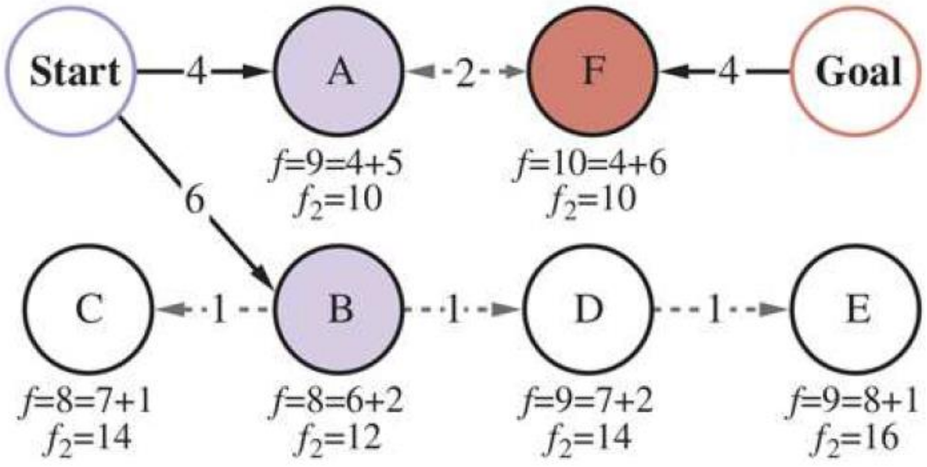
Consider a forward path from the initial state to a node m and a backward path from the goal to a node n .A lower bound is defined on the cost of a solution that follows the path from the initial state m to then somehow gets to n then follows the path to the goal as

lb(m, n) = max(gF (m) + gB(n), fF (m), fB(n))
In other words, the cost of such a path must be at least as large as the sum of the path costs of the two and the cost must also be at least as much as the estimated f cost of either part (because the heuristic estimates are optimistic).

Given that, the theorem is that for any pair of nodes m,n withlb(m,n) less than the optimal cost C* then expand either m or n because the path that goes through both of them is a potential optimal solution. The difficulty is which node is best to expand is not known and therefore no bidirectional search algorithm can be guaranteed to be optimally efficient—any algorithm might expand up to twice the minimum number of nodes if it always chooses the wrong member of a pair to expand first. Some bidirectional heuristic search algorithms explicitly manage a queue of (m,n)pairs, with bidirectional best-first search which has two frontier priority queues, and gives it to an evaluation function that mimics the lb criteria:

f2(n) = max(2g(n), g(n) + h(n))
The node to expand next will be the one that minimizes this f2 value; the node can come from either frontier. This f2 function guarantees that nodes are never expanded (from either frontier) with g(n) > C∗/2 The two halves of the search “meet in the middle” in the sense that when the two frontiers touch, no node inside of either frontier has a path cost greater than the bound C∗/2

Figure works through an example bidirectional search.



Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; to the right, node F is an inverse successor of Goal. Each node is labeled with f=g+h values and f2 = max(2g, g + h) value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-FGoal, has cost C*=4+2+4=10 means that a meet-in-the-middle bidirectional algorithm should not expand any node with g> C*/2=5 and indeed the next node to be expanded would be A or F (each with g=4), leading us an optimal solution. If the nodes are expanded with lowest cost first, then B and C would come next, and D and E would be tied with A, but they all have g> C*/2 and thus are never expanded when f2 is the evaluation function.

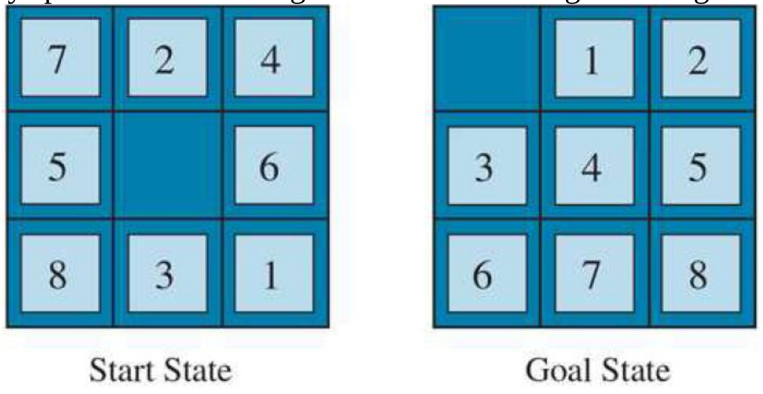
The described approach where the heuristic estimates the distance to the goal (or, when the problem has multiple goal states, the distance to the closest goal) and estimates the distance to the start. This is called a **front-to-end** search. An alternative, called **front-to-front** search, attempts to estimate the distance to the other frontier.

Bidirectional search is sometimes more efficient than unidirectional search, sometimes not. In general, if a very good heuristic is used then A* search produces search contours that are focused on the goal, and adding bidirectional search does not help much. With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred. In the worst case of a poor heuristic, the search is no longer focused on the goal, and bidirectional search has the same asymptotic complexity as A*. Bidirectional search with the evaluation function and an admissible heuristic is complete and optimal.

Heuristic Functions

A heuristic function is used to estimate the cost of cheapest path from node n to a goal node. A heuristic function, or simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.

Consider the example of 8- puzzle, the objective of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure)



A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

Let h1 = the number of misplaced tiles (blank not included) all eight tiles are out of position, so the start state has h1=8 ,h1is an admissible heuristic because any tile that is out of place will require at least one move to get it to the right place.
h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance is the sum of the horizontal and vertical distances— sometimes called the **city-block distance** or **Manhattan distance**.h2 is also admissible because any move can move one tile one step closer to the goal.
Tiles 1 to 8 in the start state of Figure gives a Manhattan distance of

h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.

The effect of heuristic accuracy on performance
One way to characterize the quality of a heuristic is the **effective branching factor** 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 contain N+1 nodes. Thus, $N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^{d.}$

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 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 close to 1, allowing fairly large problems to be solved at reasonable computational cost.

Generating heuristics from relaxed problems

A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

Because the relaxed problem adds edges to the state-space graph, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have *better* solutions if the added edges provide shortcuts. Hence, *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem*.
Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent .

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.

For example if the 8-puzzle actions are described as
A tile can move from square X to square Y if
X is adjacent to Y and Y is blank,

Three relaxed problems can be generated by removing one or both of the conditions:
a. A tile can move from square X to square Y if X is adjacent to Y.
b. A tile can move from square X to square Y if Y is blank.
c. A tile can move from square X to square Y.

Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially *without search*, because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.

A program called **ABSOLVER** can generate heuristics automatically from problem definitions, using the "relaxed problem" method and various other techniques . ABSOLVER generated a new heuristic for the 8-puzzle that was better than any preexisting heuristic and found the first useful heuristic for the famous Rubik's Cube puzzle.

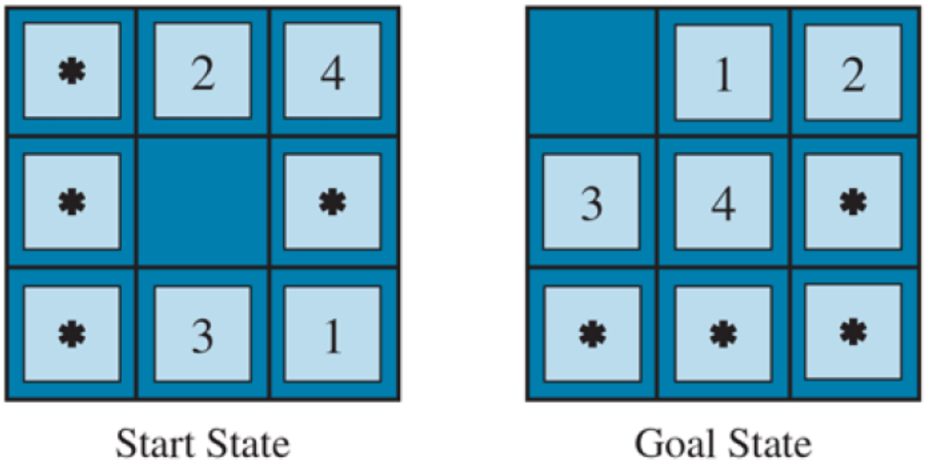
If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem and none of them is clearly better than the others then h(n) can be defined as

$\mathbf{h(n) = max\{h_1(n), \ldots, h_k(n)\}}$.

This composite heuristic picks whichever function is most accurate on the node, because the h_i components are admissible, h is admissible. Furthermore, h dominates all of its component heuristics. The only drawback is that h(n) takes longer to compute. If that is an issue, an alternative is to randomly select one of the heuristics at each evaluation, or use a machine learning algorithm to predict which heuristic will be best. Doing this can result in a heuristic that is inconsistent , but in practice it usually leads to faster problem solving.

Generating heuristics from subproblems: Pattern databases

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. A subproblem of the 8-puzzle instance given in **Figure .**



The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.

The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in this example, every possible configuration of the four tiles and the blank. Then to compute an admissible heuristic for each state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized hence used in solving many problems. By working backward from the goal, the exact solution cost of every instance encountered is immediately available. This is an example of **dynamic programming**,

Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem— because only one tile is moved at a time.

Generating heuristics with landmarks

A perfect heuristic can be generated by precomputing and storing the cost of the optimal path between every pair of vertices.

A better approach is to choose a few **landmark points(sometimes called "pivots" or "anchors")** from the vertices. Then for each landmark and for each other vertex in the graph, compute C*(v,L), and store the exact cost of the optimal path from v to L. Given the stored tables C* , an efficient heuristic can be created: the minimum, over all landmarks, of the cost of getting from the current node to the landmark, and then to the goal:

$$h_L(n) = \min_{L \in Landmarks} C^*(n,L) + C^*(L, goal)$$

If the optimal path happens to go through a landmark, this heuristic will be exact; if not it is inadmissible—it overestimates the cost to the goal. Some route-finding algorithms save even more time by adding **shortcuts**—artificial edges in the graph that define an optimal multi-action path.

A heuristic that is both efficient and admissible: is called a **differential heuristic** (because of the subtraction).

There are several ways to pick landmark points.

- Selecting points at random is fast, as better results are got if the spread the landmarks are not too close to each other.
- A greedy approach is to pick a first landmark at random, then find the point that is furthest from that, and add it to the set of landmarks, and continue, at each iteration adding the point that maximizes the distance to the nearest landmark.
- For the differential heuristic it is good if the landmarks are spread around the perimeter of the graph. Thus, a good technique is to find the centroid of the graph, arrange pie-shaped wedges around the centroid, and in each wedge select the vertex that is farthest from the center.

Landmarks work especially well in route-finding problems because of the way roads are laid out in the world: a lot of traffic actually wants to travel between landmarks, so civil engineers build the widest and fastest roads along these routes; landmark search makes it easier to recover these routes.

Learning to search better

An agent *learns* how to search better using an important concept called the **metalevel state space**. Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an ordinary state space where each state on the path is an object-level search tree.

For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree.

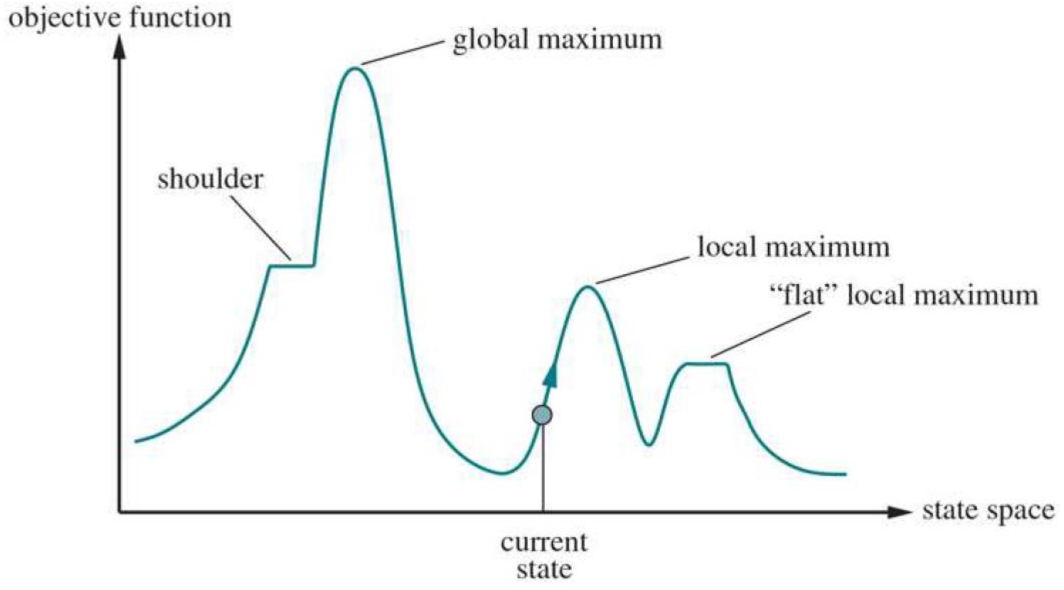
For harder problems, there will be many such missteps, and a **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees. The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost.

Learning heuristics from experience
One way to invent a heuristic is to devise a relaxed problem for which an optimal solution can be found easily. An alternative is to learn from experience.
“Experience” here means solving lots of 8-puzzles, for instance. Each optimal solution to an 8-puzzle problem provides an example (goal, path) pair. From these examples, a learning algorithm can be used to construct a function that can (with luck) approximate the true path cost for other states that arise during search. Most of these approaches learn an imperfect approximation to the heuristic function, and thus risk inadmissibility. This leads to an inevitable tradeoff between learning time, search run time, and solution cost. Techniques such as machine learning, reinforcement learning methods are also applicable to search.
Some machine learning techniques work better when supplied with **features** of a state that are relevant to predicting the state’s heuristic value, rather than with just the raw state description.

Local Search and Optimization Problems

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory; and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable. Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function**.

To understand local search, consider the states of a problem laid out in a **state-space landscape**, as shown in Figure.



Each point (state) in the landscape has an **"elevation,"** defined by the value of the objective function. If elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**—this is known as **hill climbing**. If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**—this is known as **gradient descent**.

Hill-climbing search

The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor. The **hill-climbing** search algorithm keeps track of one current state and on each iteration moves to the neighboring state with highest value—that is, it heads in the direction that provides the **steepest ascent**. It terminates when it reaches a "peak" where no neighbor has a higher value. Hill climbing does not look ahead beyond the immediate neighbors of the current state

Algorithm: Hill Climbing Search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
     neighbor ← a highest-valued successor state of current
     if VALUE(neighbor) ≤ VALUE(current) then return current
     current ← neighbor
```

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

Unfortunately, hill climbing can get stuck for any of the following reasons:
LOCAL MAXIMA: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.

RIDGES: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

PLATEAUS: A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible.

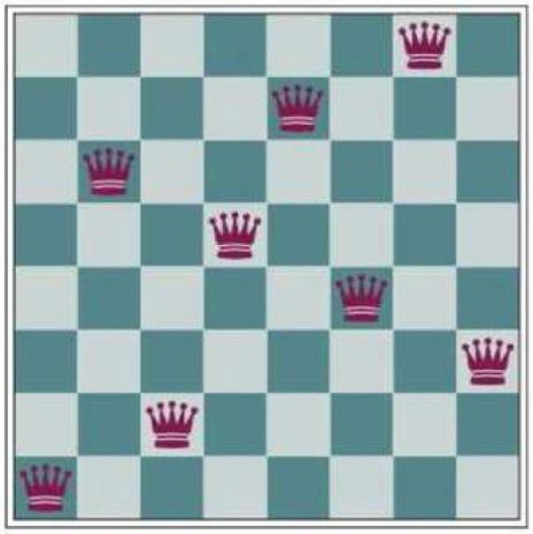
Many variants of hill climbing have been invented. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

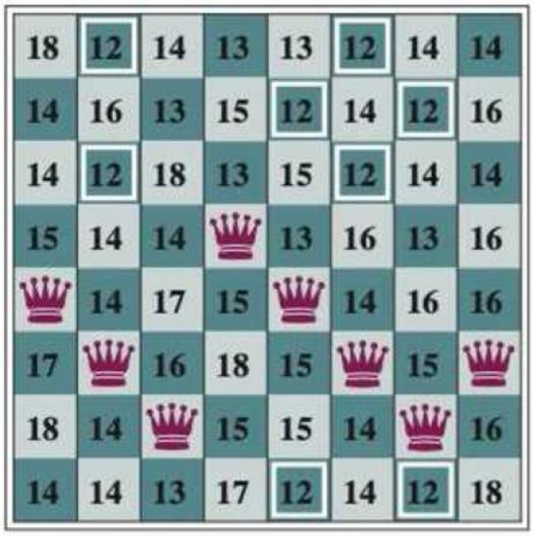
Another variant is **random-restart hill climbing**, which adopts the quote "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is complete with probability 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability of success, then the expected number of restarts required is 1/p . The expected number of steps is the cost of one successful iteration plus (1-p)/p times the cost of failure. For 8-queens, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in seconds

The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaus, random-restart hill climbing will find a good solution very quickly.

Example:
To illustrate hill climbing, Consider the **8-queens problem** . The **complete-state formulation is used here** , which means that every state has all the components of a solution, but they might not all be in the right place. In this case every state has 8 queens on the board, one per column. The initial state is chosen at random, and the successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8x7=56 successors). The heuristic cost function is the number of pairs of queens that are attacking each other; this will be zero only for solutions. (It counts as an attack if two pieces are in the same line, even if there is an intervening piece between them.)

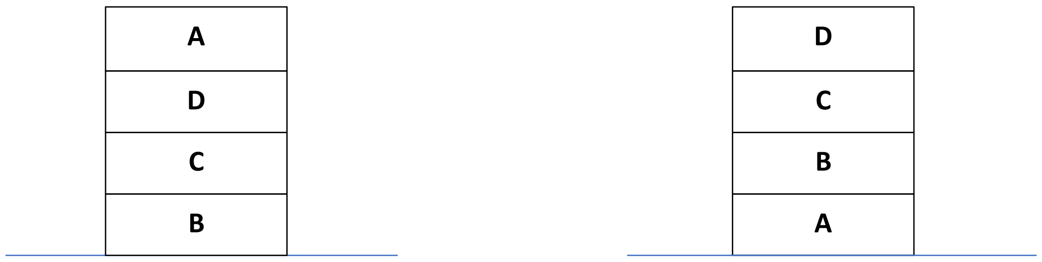
Figure a : The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.)The figure b shows the h values of all its successors.





Blocks world problem

Consider the blocks world problem with the four blocks A,B,C,D with the start and goal states given below



Initial State **Goal State**

Assume the following two operations :

(i)Pick a block and put it on the table.

(ii)Pick a block and place it on another block

Solve the above problem using Hill Climbing algorithm and a suitable heuristic function. Show the intermediate decisions and states

Solution:

Define the heuristic function

h(x)=+1 for all the blocks in the structure if the block is correctly positioned or

h(x)=-1 for all uncorrectly placed blocks in the structure.

Simulated annealing

A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient. Therefore, **Simulated annealing** is a hill climbing algorithm with a random walk that yields both efficiency and completeness.

In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.
The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The schedule input determines the value of the “temperature” T as a function of time.

```
function SIMULATED-ANNEALING($problem$, $schedule$) returns a solution state
    $current \leftarrow problem$.INITIAL
    for $t$ = 1 to $\infty$ do
        $T \leftarrow schedule(t)$
        if $T = 0$ then return $current$
        $next \leftarrow$ a randomly selected successor of $current$
        $\Delta E \leftarrow$ VALUE($current$) – VALUE($next$)
        if $\Delta E > 0$ then $current \leftarrow next$
        else $current \leftarrow next$ only with probability $e^{-\Delta E/T}$
```

Instead of picking the *best* move, however, it picks a *random* move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the *schedule* T lowers to 0 slowly enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$, is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.
Simulated annealing was used to solve VLSI layout problems beginning in the 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

Local beam search

The **local beam search** algorithm keeps track of states rather than just one. It begins with randomly generated states. At each step, all the successors of all states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the best successors from the complete list and repeats.

In a random-restart search, each search process runs independently of the others. *In a local beam search, useful information is passed among the parallel search threads.* Local beam search can suffer from a lack of diversity among the k states—they can become clustered in a small region of the state space, making the search little more than a k -times slower version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the top k successors, stochastic beam search chooses successors with probability proportional to the successor’s value, thus increasing diversity.

Evolutionary algorithms or Genetic algorithms

Evolutionary algorithms can be seen as variants of stochastic beam search that are explicitly motivated by the metaphor of natural selection in biology: there is a population of individuals (states), in which the fittest (highest value) individuals produce offspring (successor states) that populate the next generation, a process called **recombination**.

The different terminologies used in evolutionary algorithms are

- The size of the population.
- The representation of each individual.

- In **genetic algorithms**, each individual is a string over a finite alphabet (often a Boolean string), just as DNA is a string over the alphabet **ACGT**. In **evolution strategies**, an individual is a sequence of real numbers, and in **genetic programming** an individual is a computer program.
- The mixing number, ρ, which is the number of parents that come together to form offspring. The most common case is ρ = 2: two parents combine their "genes" (parts of their representation) to form offspring. It is possible to have ρ >2 , which occurs only rarely in nature but is easy enough to simulate on computers.
- The **selection** process for selecting the individuals who will become the parents of the next generation: one possibility is to select from all individuals with probability proportional to their fitness score. Another possibility is to randomly select n individuals(n> ρ), and then select the ρ most fit ones as parents.
- The recombination procedure. One common approach (assuming ρ = 2), is to randomly select a **crossover point** to split each of the parent strings, and recombine the parts to form two children, one with the first part of parent 1 and the second part of parent 2; the other with the second part of parent 1 and the first part of parent 2.
- The **mutation rate**, which determines how often offspring have random mutations to their representation. Once an offspring has been generated, every bit in its composition is flipped with probability equal to the mutation rate.
- The makeup of the next generation. This can be just the newly formed offspring, or it can include a few top-scoring parents from the previous generation (a practice called **elitism**, which guarantees that overall fitness will never decrease over time). The practice of **culling**, in which all individuals below a given threshold are discarded, can lead to a speedup .

A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

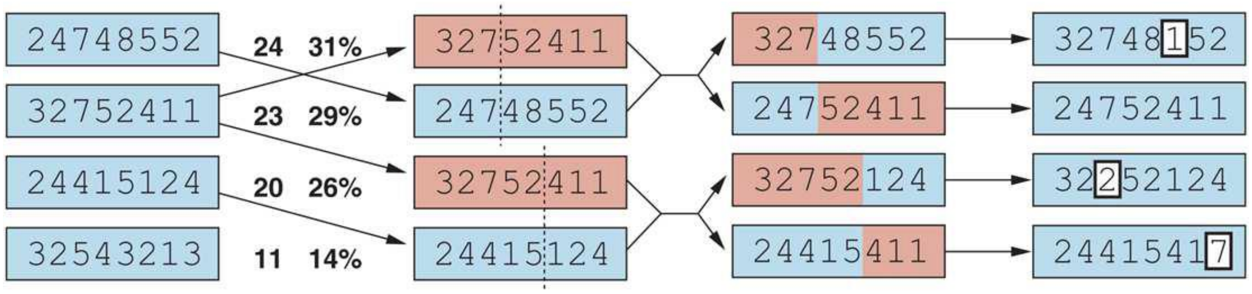
```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
     weights ← WEIGHTED-BY(population, fitness)
     population2 ← empty list
     for i = 1 to SIZE(population) do
        parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child ← REPRODUCE(parent1, parent2)
        if (small random probability) then child ← MUTATE(child)
        add child to population2
     population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Genetic algorithms are similar to stochastic beam search, but with the addition of the crossover operation. In practice, genetic algorithms have their place within the broad landscape of optimization methods , particularly for complex structured problems such as circuit layout or job-shop scheduling, and more recently for evolving the architecture of deep neural networks .

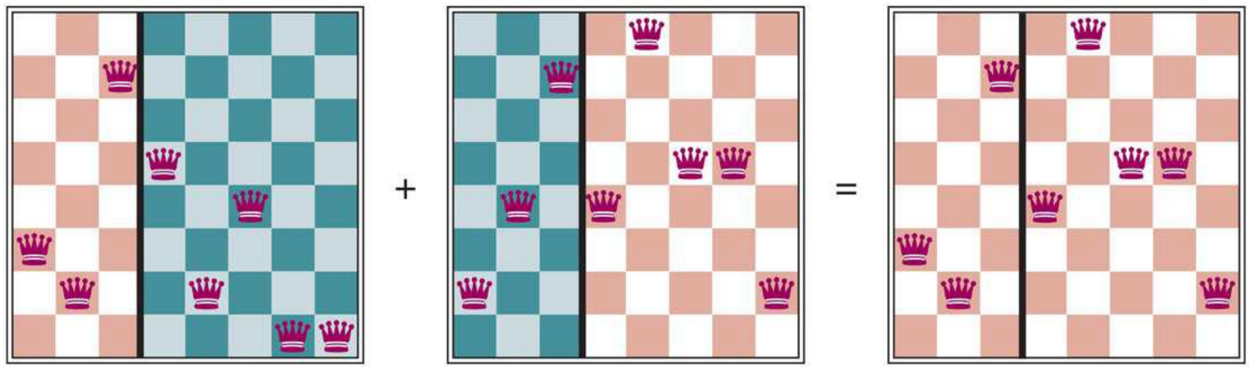
Example

Figure (a) shows a population of four 8-digit strings, each representing a state of the 8- queens puzzle: the c-th digit represents the row number of the queen in column c . In (b), each state is rated by the fitness function. Higher fitness values are better, so for the 8- queens problem the number of *nonattacking* pairs of queens has been used for a solution(8x7/2=28). The values of the four states in (b) are 24, 23, 20, and 11. The fitness scores are then normalized to probabilities, and the resulting values are shown next to the fitness values in (b).



(a) Initial Population (b) Fitness Function (c) Selection (d) Crossover (e) Mutation

In (c), two pairs of parents are selected, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each selected pair, a crossover point (dotted line) is chosen randomly. In (d), the parent strings are crossed over at the crossover points, yielding new offspring. For example, the first child of the first pair gets the first three digits (327) from the first parent and the remaining digits (48552) from the second parent. The 8-queens states involved in this recombination step are shown in Figure



The 8-queens states corresponding to the first two parents in Figure (c) and the first offspring in Figure d) .The green columns are lost in the crossover step and the red columns are retained. Finally, in (e), each location in each string is subject to random mutation with a small independent probability.

Local Search in Continuous Spaces

Consider the example to place three new airports anywhere in Romania, such that the sum of squared straight-line distances from each city on the map to its nearest airport is minimized. The state space is then defined by the coordinates of the three airports;(x1,y1) ,(x2,y2) and (x3,y3) This is a *six-dimensional* space; the states are defined by six **variables**. In general, states are defined by an n-dimensional vector of variables, x . Moving around in this space corresponds to moving one or more of the airports on the map. The objective function f(x)=f(x1,y1,x2,y2,x3,y3) is relatively easy to compute for any particular state once the closest cities are computed. Let C_i be the set of cities whose closest airport (in the state x) is airport i . Then,

$$f(\mathbf{x}) = f(x_1,y_1,x_2,y_2,x_3,y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2 \, .$$

This equation is correct not only for the state x but also for states in the local neighborhood of x . However, it is not correct globally; since the set of closest cities for that airport changes, and recomputation is needed for C_i .

One way to deal with a continuous state space is to **discretize** it. For example, instead of (x_i,y_i) allowing the locations to be any point in continuous two-dimensional space, this could be limited to fixed points on a rectangular grid with spacing of size δ (delta). Then instead of having an infinite number of successors, each state in the space would have only 12 successors, corresponding to incrementing one of the 6 variables by $\pm$ δ. Then apply any of the local search algorithms to this discrete space. Alternatively, the branching factor can be made finite by sampling successor states randomly, moving in a random direction by a small amount δ . Methods that measure progress by the change in the value of the objective function between two nearby points are called **empirical gradient** methods.

Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space. Reducing the value of δ over time can give a more accurate solution, but does not necessarily converge to a global optimum in the limit.

An objective function can be expressed in a mathematical form such that it uses calculus to solve the problem analytically rather than empirically. Many methods attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. Thus

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right) .$$

In some cases, a maximum can be found by solving the equation $\nabla f = 0$. In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means the gradient can be computed *locally* (but not *globally*); for example,

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_1 - x_c) \, .$$

Given a locally correct expression for the gradient, the steepest-ascent hill climbing can be performed by updating the current state according to the formula

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}) \,,$$

where (alpha) is a small constant often called the **step size**. There exist a huge variety of methods for adjusting α . The basic problem is that if α is too small, too many steps are needed; if α is too large, the search could overshoot the maximum. The technique of **line search** tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling α—until starts to decrease again. The point at which this occurs becomes the new current state.

For many problems, the most effective algorithm is the **Newton–Raphson** method. This is a general technique for finding roots of functions—that is, solving equations of the form g(x)=0. It works by computing a new estimate for the root according to Newton's formula

$$x \leftarrow x - g(x)/g'(x) \,.$$

To find a maximum or minimum of f , x has to be found such that the *gradient* is a zero vector (i.e. ∇f(x) = 0,). Thus, g(x) in Newton's formula becomes ∇f(x) , and the update equation can be written in matrix–vector form as

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}) \,,$$

Where $H_f(x)$ is the **Hessian** matrix of second derivatives, whose elements $H_{i,j}$ are given by $\partial^2 f / \partial x_i \partial x_j$.

Local search methods suffer from local maxima, ridges, and plateaus in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing are often helpful. High-dimensional continuous spaces are, however, big places in which it is very easy to get lost.

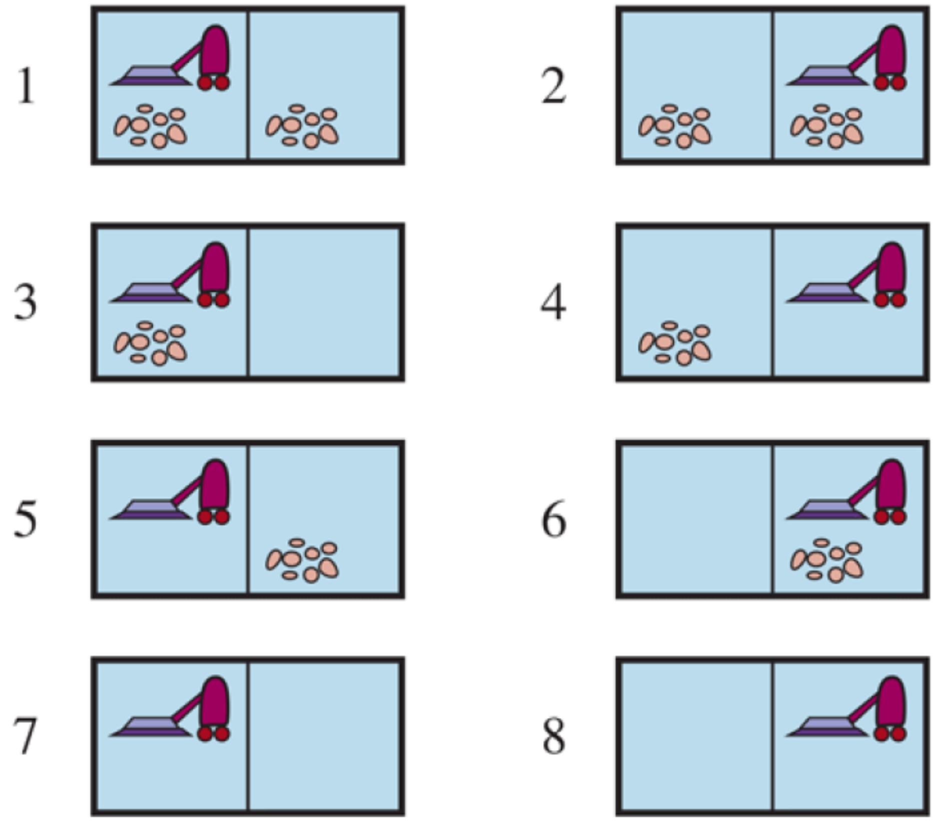
An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a **convex set** and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables.
A set of points is convex if the line joining any two points in S is also contained in S . A **convex function** is one for which the space "above" it forms a convex set; by definition, convex functions have no local (as opposed to global) minima.

Linear programming is probably the most widely studied and broadly useful method for optimization. It is a special case of the more general problem of **convex optimization**, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region. Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables. Several important problems in machine learning and control theory can be formulated as convex optimization problems.

Search with Nondeterministic Actions

When the environment is partially observable, however, the agent doesn't know for sure what state it is in; and when the environment is nondeterministic, the agent doesn't know what state it has transitions after taking an action. A set of physical states that the agent believes that are possible is known as **belief state**. In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence, but rather a conditional plan (sometimes called a contingency plan or a strategy) that specifies what to do depending on what percepts agent receives while executing the plan.

The erratic vacuum world
The vacuum world has eight states, as shown in Figure .



There are three actions—*Right*, *Left*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is fully observable, deterministic, and completely known, then the problem is easy to solve with any of the algorithms, and the solution is an action sequence. For example, if the initial state is 1, then the action sequence [*Suck*, *Right*, *Suck*] will reach a goal state, 8.

Suppose that nondeterminism is introduced in the form of a powerful but erratic vacuum cleaner. In the **erratic vacuum world**, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.

To provide a precise formulation of this problem, the notion of **transition model has to be generalized**. Instead of defining the transition model by a RESULT function that returns a single outcome state, RESULTS function can be used that returns a set of possible outcome states. For example, in the erratic vacuum world, the *Suck* action in state 1 cleans up either just the current location, or both locations:

RESULTS(1, Suck) = {5,7}

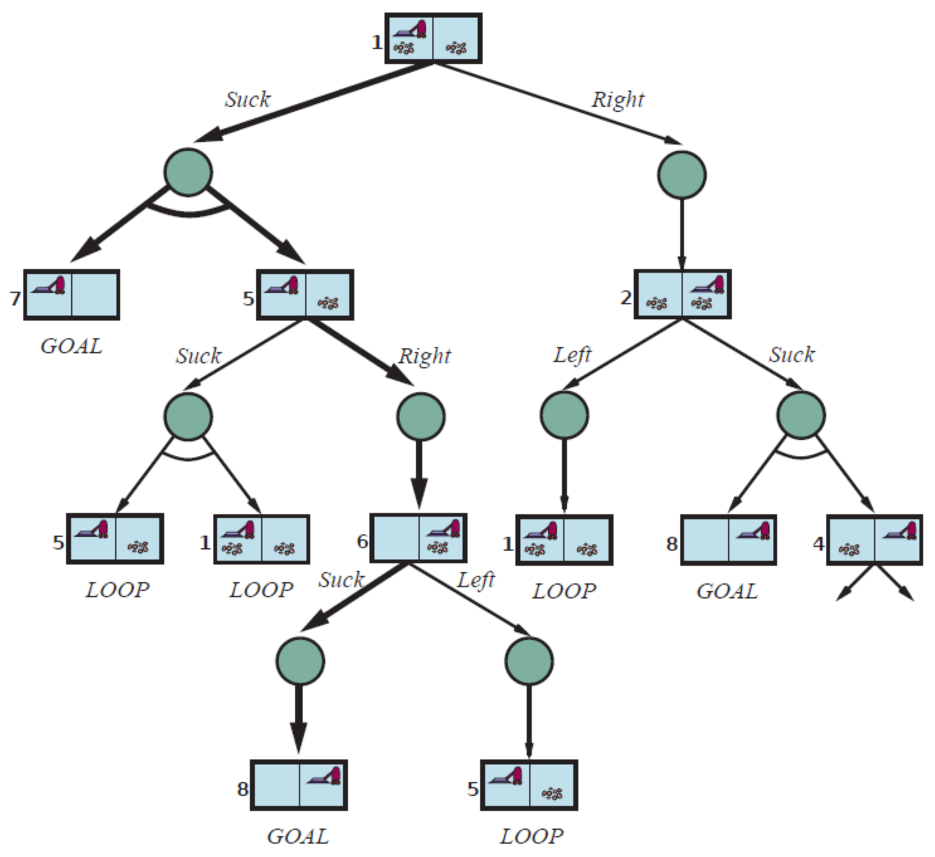
If the start state is 1, no single *sequence* of actions solves the problem, but the following **conditional plan** does:

[Suck,if State = 5 then [Right, Suck] else [ ]] .

A conditional plan can contain **if–then–else** steps; this means that solutions are *trees* rather than sequences. Here the conditional in the **if** statement tests to see what the current state is; this is something the agent will be able to observe at runtime, but doesn't know at planning time. Many problems in the real, physical world are contingency problems, because exact prediction of the future is impossible.

AND–OR search trees

In a deterministic environment, the only branching is introduced by the agent's own choices in each state: these nodes **OR nodes**. In the vacuum world, for example, at an OR node the agent chooses *Left or Right or Suck*. In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action,these nodes **AND nodes**. For example, the *Suck* action in state 1 results in the belief state {5.7}, so the agent would need to find a plan for state 5 *and* for state 7. These two kinds of nodes alternate, leading to an **AND–OR tree** as illustrated in Figure



The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

A solution for an AND–OR search problem is a subtree of the complete search tree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes. The solution is shown in bold lines in the figure; it corresponds to the plan

Figure gives a recursive, depth-first algorithm for AND–OR graph search.

```
function AND-OR-SEARCH($problem$) returns a conditional plan, or $failure$
   return OR-SEARCH($problem$, $problem$.INITIAL, [ ])

function OR-SEARCH($problem$, $state$, $path$) returns $a$ $conditional$ $plan$, $or$ $failure$
   if $problem$.IS-GOAL($state$) then return the empty plan
   if IS-CYCLE($path$) then return $failure$
   for each $action$ in $problem$.ACTIONS($state$) do
      $plan \leftarrow$ AND-SEARCH($problem$, RESULTS($state$, $action$), $[state] + path$])
      if $plan \neq failure$ then return $[action] + plan]$
   return $failure$

function AND-SEARCH($problem$, $states$, $path$) returns $a$ $conditional$ $plan$, $or$ $failure$
   for each $s_i$ in $states$ do
      $plan_i \leftarrow$ OR-SEARCH($problem$, $s_i$, $path$)
      if $plan_i = failure$ then return $failure$
   return [if $s_1$ then $plan_1$ else if $s_2$ then $plan_2$ else ... if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

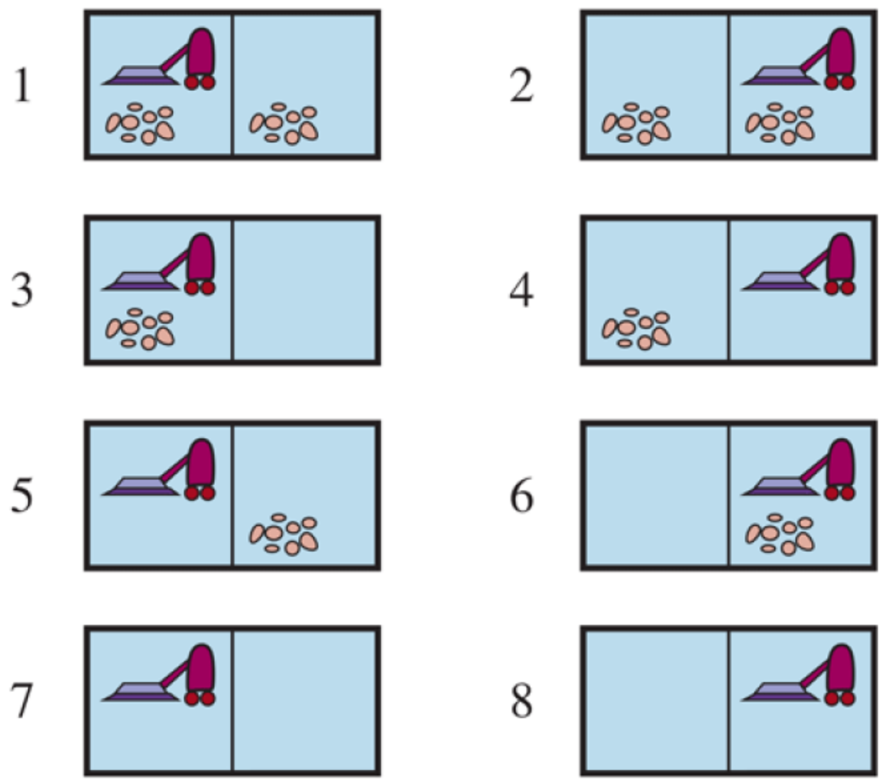
AND–OR graphs can be explored either breadth-first or best-first. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems. If the current state is identical to a state on the path from the root, then it returns with failure. This doesn't mean that there is *no* solution from the current state; it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, the algorithm ensures to terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some *other* path from the root, which is important for efficiency.

A *slippery* vacuum world, which is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location.

Search in Partially Observable Environments
a)Searching with no observation
When the agent's percepts provide *no information at all*, then it is a **sensorless** problem (or a **conformant** problem). The sensorless agent has no hope of solving a problem if it has no idea what state it starts in, but sensorless solutions are surprisingly common and useful, primarily because they *don't* rely on sensors working properly.

Sometimes a sensorless plan is better even when a conditional plan with sensing is available. For example, doctors often prescribe a broad spectrum antibiotic rather than using the conditional plan of doing a blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic. The sensorless plan saves time and money, and avoids the risk of the infection worsening before the test results are available.

Consider a sensorless version of the (deterministic) vacuum world.



Assume that the agent knows the geography of its world, but not its own location or the distribution of dirt. In that case, its initial belief state is {1,2,3.4,5,6,7,8}. Now, if the agent moves *Right* it will be in one of the states {2,4,6,8} —the agent has gained information without perceiving anything! After [*Right*,*Suck*] the agent will always end up in one of the states{4,8}.Finally, after [*Right*,*Suck*,*Left*,*Suck*] the agent is guaranteed to reach the goal state 7, no matter what the start state.

The solution to a sensorless problem is a sequence of actions, not a conditional plan (because there is no perceiving). The space of belief states are searched rather than physical states. In belief-state space, the problem is *fully observable* because the agent always knows its own belief state. Furthermore, the solution (if any) for a sensorless problem is always a sequence of actions. This is because, as the percepts received after each action are completely predictable—they're always empty! So there are no contingencies to plan for. This is true *even if the environment is nondeterministic*.

To introduce new algorithms for sensorless search problems ,the existing algorithms of the underlying physical problem can be transformed into a belief-state problem, in which belief states are searched rather than physical states. The original problem P has components ACTIONSp, RESULTSp etc., and the belief-state problem has the following components:

STATES: The belief-state space contains every possible subset of the physical states. If P has N states, then the belief-state problem has 2^N belief states, although many of those may be unreachable from the initial state.

INITIAL STATE: Typically the belief state consisting of all states in P.
ACTIONS: Suppose the agent is in belief state b={ s1,s2 }, but ACTIONS$_P$(s1) ≠ ACTIONS$_P$(s2); then the agent is unsure of which actions are legal. Assume that illegal actions have no effect on the environment, then it is safe to take the *union* of all the actions in any of the physical states in the current belief state b :

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s)\,.$$

If an illegal action might lead to catastrophe, it is safer to allow only the *intersection*, that is, the set of actions legal in *all* the states.

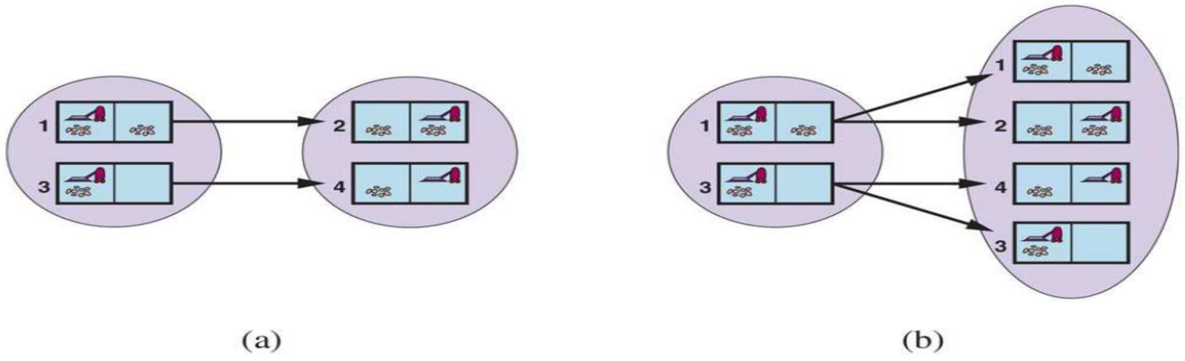
TRANSITION MODEL: For deterministic actions, the new belief state has one result state for each of the current possible states (although some result states may be the same):

$$b' = \text{RESULT}(b,a) = \{s' : s' = \text{RESULT}_P(s,a) \text{ and } s \in b\}\,.$$

With nondeterminism, the new belief state consists of all the possible results of applying the action to any of the states in the current belief state:

$$\begin{aligned} b' = \text{RESULT}(b,a) &= \{s' : s' \in \text{RESULTS}_P(s,a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}_P(s,a)\,, \end{aligned}$$

The size of b’ will be the same or smaller than b for deterministic actions, but may be larger than b with nondeterministic actions
GOAL TEST: The agent *possibly* achieves the goal if *any* state in the belief state satisfies the goal test of the underlying problem, IS-GOAL$_P$(s) . The agent *necessarily* achieves the goal if *every* state satisfies IS-GOAL$_P$(s) .
ACTION COST: This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values.



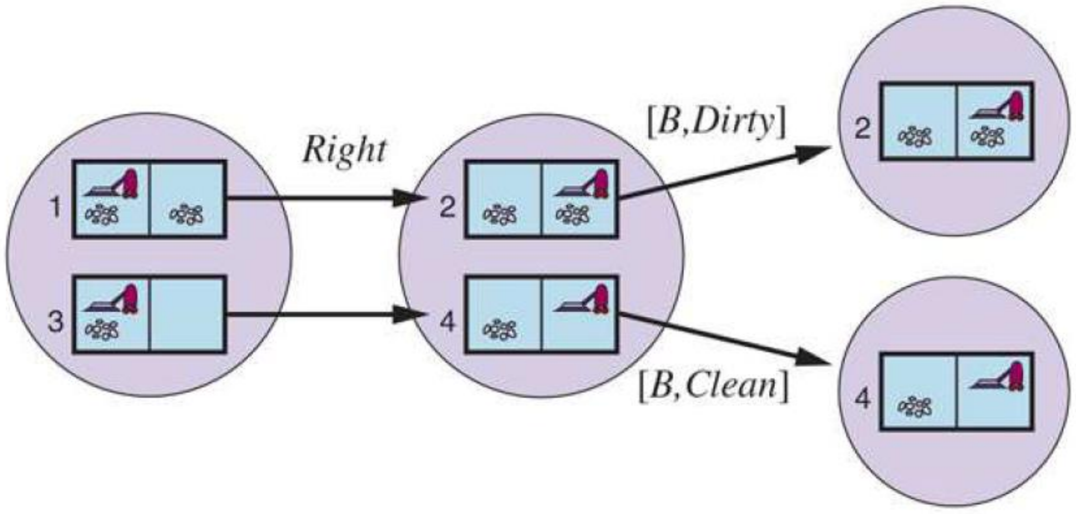
Predicting the next belief state for the sensorless vacuum world with the deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Advantages and drawbacks
In ordinary graph search, newly reached states are tested to see if they were previously reached. This works for belief states, too; Conversely, if any state already been generated and found to be solvable, then any *Subset* of that state is also guaranteed to be solvable. This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.
Even with this improvement, however, sensorless problem-solving is not used in practice. **One issue** is the vastness of the belief-state space. One solution is to represent the belief state by some more compact description. Another approach is to avoid the standard search algorithms, which treat belief states as black boxes just like any other problem state Instead use the belief states and develop **incremental belief-state search** algorithms that build up the solution

Just as an AND–OR search has to find a solution for every branch at an AND node, this algorithm has to find a solution for every state in the belief state; the difference is that AND– OR search can find a different solution for each branch, whereas an incremental belief-state search has to find *one* solution that works for *all* the states. The main advantage of the incremental approach is that it is typically able to detect failure quickly—when a belief state is unsolvable.

b)Searching in partially observable environments

Many problems cannot be solved without sensing. For a partially observable problem, the problem specification will specify a function PERCEPT(s) that returns the percept received by the agent in a given state. If sensing is nondeterministic, then the PERCEPTS function can be used to return a set of possible percepts. For fully observable problems, PERCEPT(s) = s for every state s, and for sensorless problems PERCEPT(s) = null
Consider the example in figure, In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are *[R,Dirty]* and *[R,Clean]*, leading to two belief states, each of which is a singleton



The transition model for partially observable problems consists of three stages

- The **prediction** stage computes the belief state resulting from the action, RESULT(b, a). The notation, $\hat{b}\ (estimated)$= RESULT(b, a) is the synonym for PREDICT(b, a) .
- The **possible percepts** stage computes the set of percepts that could be observed(o) in the predicted belief state

$$\text{Possible-Percepts}(\hat{b}) = \{o : o = \text{Percept}(s) \text{ and } s \in \hat{b}\}\,.$$

- The **update** stage computes, for each possible percept, the belief state that would result from the percept. The updated belief state b_0 is the set of states in $\hat{b}$ that could have produced the percept:

$$b_o = \text{Update}(\hat{b},o) = \{s : o = \text{Percept}(s) \text{ and } s \in \hat{b}\}\,.$$

The three stages can be put together to obtain the possible belief states resulting from a given action and the subsequent possible percepts is given by:

$$\begin{aligned}\text{Results}(b,a) = \{b_o : b_o = \text{Update}(\text{Predict}(b,a),o) \text{ and}\\ o \in \text{Possible-Percepts}(\text{Predict}(b,a))\}\,.\end{aligned}$$

c)Solving Partially Observable Problems

The RESULTS function for a nondeterministic belief-state problem from an underlying physical problem, given the PERCEPT function along with the AND–OR search algorithm can be applied directly to derive a solution. The solution is the conditional plan

d) An agent for partially observable environments

An agent for partially observable environments formulates a problem, calls a search algorithm (such as AND-OR-SEARCH) to solve it, and executes the solution. There are two main differences between this agent and the one for fully observable deterministic environments. First, the solution will be a conditional plan rather than a sequence; to execute an if–then–else expression, the agent will need to test the condition and execute the appropriate branch of the conditional. Second, the agent will need to maintain its belief state as it performs actions and receives percepts.

Given an initial belief state b, an action a, and a percept o , the new belief state is:

$$b' = \text{UPDATE}(\text{PREDICT}(b, a),o)\,.$$

This equation is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence. In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including **monitoring**, **filtering**, and **state estimation.** As the environment becomes more complex, the agent will only have time to compute an approximate belief state, perhaps focusing on the implications of the percept for the aspects of the environment that are of current interest. Most work on this problem has been done for stochastic, continuous-state environments with the tools of probability theory

Consider an example in a discrete environment with deterministic sensors and nondeterministic actions. The example concerns a robot with a particular state estimation task called **localization**: working out where it is, given a map of the world and a sequence of percepts and actions.

The robot is placed in the maze-like environment. The robot is equipped with four sonar sensors that tells whether there is an obstacle.The percept is in the form of a bit vector, one bit for each of the directions northeast, south, and west in that order, so 1011 means there are obstacles to the north, south, and west, but not east.

With nondeterministic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information. Sometimes the percepts don't help much for localization

If the sensors are faulty reasoning can be done with Boolean logic, then every sensor bit has to be considered as being either correct or incorrect, which is the same as having no perceptual information at all. In this case probabilistic reasoning allows to extract useful information from a faulty sensor as long as it is wrong less than half the time.

Online Search Agents and Unknown Environments

Offline search algorithms compute a complete solution before taking their first action. In contrast, an **online search** agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.

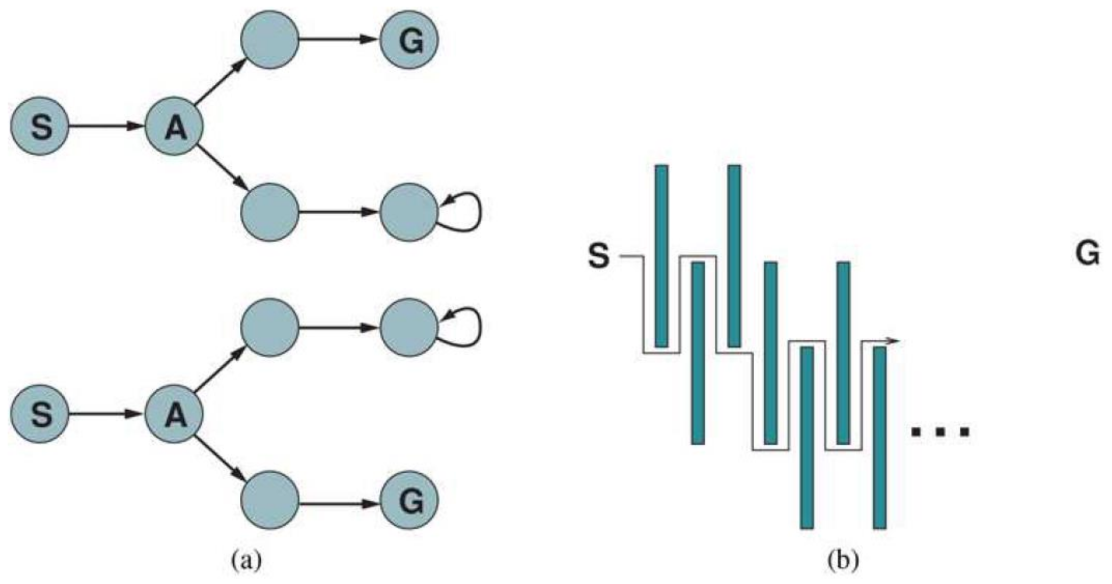
A canonical example of online search is the **mapping problem**

Online search problems

An online search problem is solved by interleaving computation, sensing, and acting
Consider the following assumptions used:
ACTIONS(s), the legal actions in state ;
(s.a,s'), the cost of applying action a in state s to arrive at state s' .
Is-GOAL(*s*), the goal test.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost that the agent incurs as it travels. It is common to compare this cost with the path cost the agent would incur *if it knew the search space in advance*—that is, the optimal path in the known environment. In the language of online algorithms, this comparison is called the **competitive ratio(small value)**

Online explorers are vulnerable to **dead ends**: states from which no goal state is reachable. In general, *no algorithm can avoid dead ends in all state spaces.* Consider the two dead-end state spaces in Figure (a) . An online search algorithm that has visited states S and A cannot tell if it is in the top state or the bottom one. Therefore, there is no way it could know how to choose the correct action in both state spaces. This is an example of an **adversary argument**.



Figure(a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and even natural terrain all present states from which some actions are **irreversible** —there is no way to return to the previous state. The exploration algorithm is only guaranteed to work in state spaces that are **safely explorable**—that is, some goal state is reachable from every reachable state. State spaces with only reversible actions, such as mazes and 8-puzzles, are clearly safely explorable

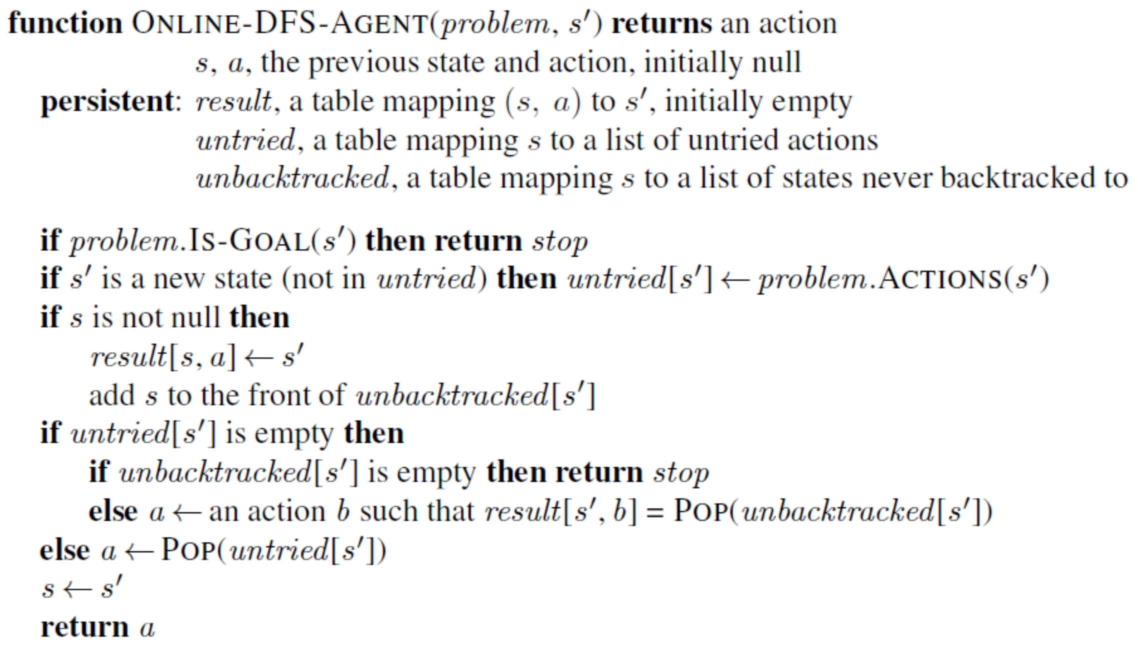
Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. For this reason, it is common to characterize the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent in an observable environment receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The updated map is then used to plan where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously: offline algorithms explore their *model* of the state space, while online algorithms explore the real world.

An online algorithm, on the other hand, can discover successors only for a state that it physically occupies. To avoid traveling all the way to a distant state to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property because (except when the algorithm is backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first exploration agent (for deterministic but unknown actions) is shown in Figure .



```
function ONLINE-DFS-AGENT(problem, $s'$) returns an action
                 $s$, $a$, the previous state and action, initially null
  persistent: result, a table mapping $(s, a)$ to $s'$, initially empty
              untried, a table mapping $s$ to a list of untried actions
              unbacktracked, a table mapping $s$ to a list of states never backtracked to

  if problem.IS-GOAL($s'$) then return stop
  if $s'$ is a new state (not in untried) then untried[$s'$] $\leftarrow$ problem.ACTIONS($s'$)
  if $s$ is not null then
      result[$s, a$] $\leftarrow s'$
      add $s$ to the front of unbacktracked[$s'$]
  if untried[$s'$] is empty then
      if unbacktracked[$s'$] is empty then return stop
      else $a \leftarrow$ an action $b$ such that result[$s', b$] = POP(unbacktracked[$s'$])
  else $a \leftarrow$ POP(untried[$s'$])
  $s \leftarrow s'$
  return $a$
```

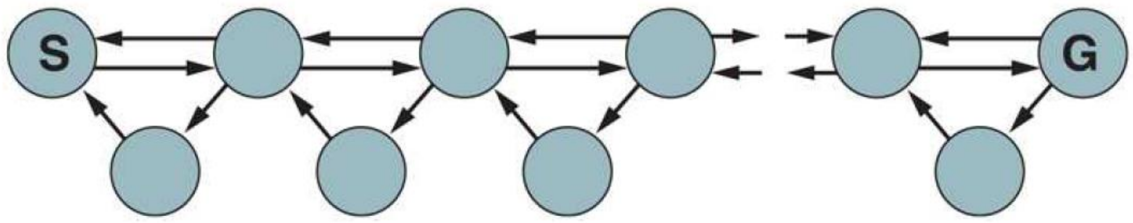
This agent stores its map in a table, *result*[*s*,*a*], that records the state resulting from executing action in state . Whenever the current state has unexplored actions, the agent tries one of those actions. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack in the physical world. In depth-first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps another table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

Online local search
Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, the basic algorithm is not very good for exploration because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot teleport itself to a new start state.

A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite and safely explorable. On the other hand, the process can be very slow.

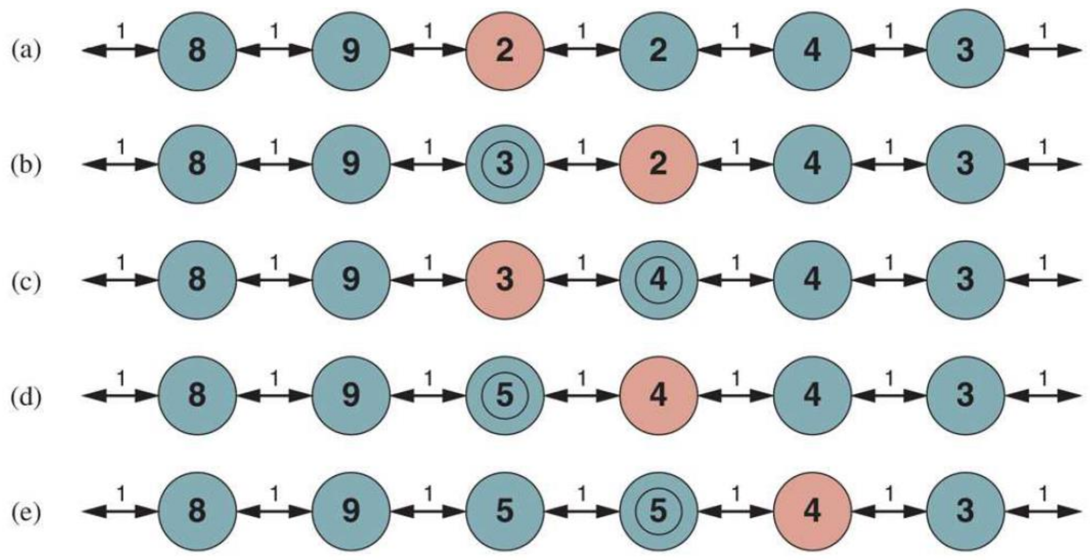
Figure shows an environment in which a random walk will take exponentially many steps to find the goal, because, for each state in the top row except S, backward progress is twice as likely as forward progress.



An environment in which a random walk will take exponentially many steps to find the goal.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited.H(s) starts out being just the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

LRTA*(Learning Real Time A*) Algorithm:
LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.
Figure shows the Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with , the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.



Like ONLINE-DFS-AGENT, it builds a map of the environment in the *result* table. It updates the cost estimate for the state it has just left and then chooses the “apparently best” move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state are always assumed to lead immediately to the goal with the least possible cost, namely h(s) . This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

```
function LRTA*-AGENT($problem$, $s'$, $h$) returns an action
                $s$, $a$, the previous state and action, initially null
   persistent: $result$, a table mapping $(s, a)$ to $s'$, initially empty
                $H$, a table mapping $s$ to a cost estimate, initially empty

   if IS-GOAL($s'$) then return $stop$
   if $s'$ is a new state (not in $H$) then $H[s'] \leftarrow h(s')$
   if $s$ is not null then
       $result[s, a] \leftarrow s'$
       $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($s, b, result[s, b], H$)
   $a \leftarrow \operatorname{argmin}_{b \in \text{ACTIONS}(s)}$ LRTA*-COST($problem, s', b, result[s', b], H$)
   $s \leftarrow s'$
   return $a$

function LRTA*-COST($problem, s, a, s', H$) returns a cost estimate
   if $s'$ is undefined then return $h(s)$
   else return $problem$.ACTION-COST($s, a, s'$) + $H[s']$
```

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of states in steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that one can define by specifying the action selection rule and the update rule in different ways.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a “map” of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*,these updates eventually converge to *exact* values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the lowest-cost successor—that is, pure hill climbing is then an optimal strategy.