UNIT - I

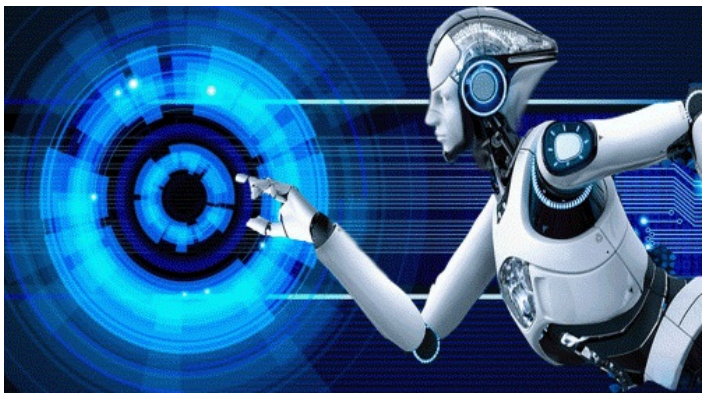
Problem Solving by Search-I: Introduction to AI, Intelligent Agents
Problem Solving by Search–II: Problem-Solving Agents, Searching for Solutions, Uninformed Search Strategies: Breadth-first search, Uniform cost search ,Depth-first search, Iterative deepening Depth-first search, Bidirectional search, Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions, Beyond Classical Search: Hill-climbing search, Simulated annealing search, Local Search in Continuous Spaces, Searching with Non-Deterministic Actions, Searching with Partial Observations, Online Search Agents and Unknown Environment.

What is Artificial Intelligence (AI)?

In today's world, technology is growing very fast, and we are getting in touch with different new technologies day by day.

Here, one of the booming technologies of computer science is Artificial Intelligence which is ready to create a new revolution in the world by making intelligent machines.The Artificial Intelligence is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, Painting, etc.

AI is one of the fascinating and universal fields of Computer science which has a great scope in future. AI holds a tendency to cause a machine to work as a human.



Artificial Intelligence is composed of two words **Artificial** and **Intelligence**, where Artificial defines *"man-made,"* and intelligence defines *"thinking power"*, hence AI means *"a man-made thinking power."*

So, we can define AI as:

"It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and be able to make decisions."

Artificial Intelligence exists when a machine can have human based skills such as learning, reasoning, and solving problems

With Artificial Intelligence you do not need to preprogram a machine to do some work, despite that you can create a machine with programmed algorithms which can work with its own intelligence, and that is the awesomeness of AI.

It is believed that AI is not a new technology, and some people say that as per Greek myth, there were Mechanical men in early days which could work and behave like humans.

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know what is the importance of AI and why we should learn it. Following are some main reasons to learn about AI:

- With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.
- With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.
- With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk.
- AI opens a path for other new technologies, new devices, and new Opportunities.

Goals of Artificial Intelligence

Following are the main goals of Artificial Intelligence:

1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
   - Proving a theorem
   - Playing chess
   - Plan some surgical operation
   - Driving a car in traffic
5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise its user.

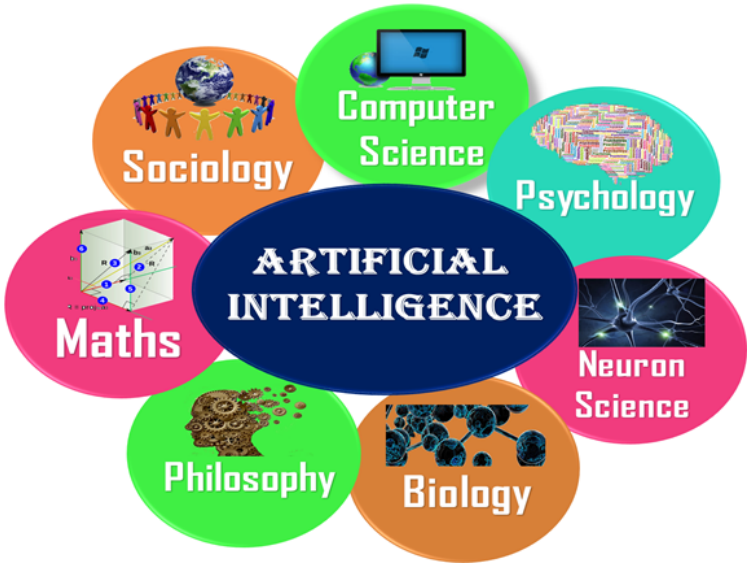
What Comprises Artificial Intelligence?

Artificial Intelligence is not just a part of computer science even though it's so vast and requires lots of other factors which can contribute to it. To create the AI first we should know how intelligence is composed, so the Intelligence is an intangible part of our brain which is a combination of **Reasoning, learning, problem-solving perception, language understanding, etc**.

To achieve the above factors for a machine or software Artificial Intelligence requires the following discipline:

- Mathematics
- Biology
- Psychology
- Sociology
- Computer Science

- Neurons Study
- Statistics



Advantages of Artificial Intelligence:

- **High Accuracy with less errors:** AI machines or systems are prone to less errors and high accuracy as it takes decisions as per pre-experience or information.
- **High-Speed:** AI systems can be of very high-speed and fast-decision making, because of that AI systems can beat a chess champion in the Chess game.
- **High reliability:** AI machines are highly reliable and can perform the same action multiple times with high accuracy.
- **Useful for risky areas:** AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where employing a human can be risky.
- **Digital Assistant:** AI can be very useful to provide digital assistance to the users such as AI technology is currently used by various E-commerce websites to show the products as per customer requirement.
- **Useful as a public utility:** AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purposes, Natural language processing to communicate with the human in human-language, etc.

Disadvantages of Artificial Intelligence

Every technology has some disadvantages, and the same goes for Artificial intelligence. Being such an advantageous technology still, it has some disadvantages which we need to keep in mind while creating an AI system. Following are the disadvantages of AI:

- **High Cost:** The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
- **Can't think out of the box:** Even though we are making smarter machines with AI, they still cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.

- **No feelings and emotions:** AI machines can be an outstanding performer, but still it does not have the feeling so it cannot make any kind of emotional attachment with humans, and may sometimes be harmful for users if the proper care is not taken.
- **Increase dependency on machines:** With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- **No Original Creativity:** As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

Agents in Artificial Intelligence

An AI system can be defined as the study of the rational agent and its environment. The agents sense the environment through sensors and act on their environment through actuators. An AI agent can have mental properties such as knowledge, belief, intention, etc.

What is an Agent?

An agent can be anything that perceives its environment through sensors and acts upon that environment through actuators. An Agent runs in the cycle of **perceiving**, **thinking**, and **acting**. An agent can be:

- **Human-Agent:** A human agent has eyes, ears, and other organs which work for sensors and hand, legs, and vocal tract work for actuators.
- **Robotic Agent:** A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
- **Software Agent:** Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

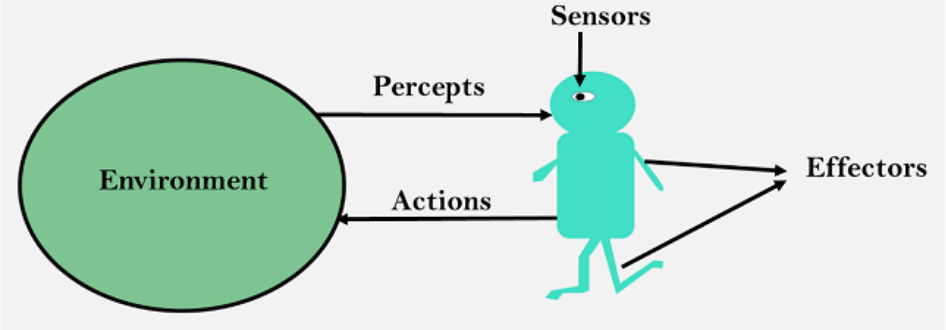
Hence the world around us is full of agents such as thermostat, cellphone, camera, and even we are also agents.

Before moving forward, we should first know about sensors, effectors, and actuators.

Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

Actuators: Actuators are the components of machines that convert energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



Intelligent Agents:

An intelligent agent is an autonomous entity which acts upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.

Following are the main four rules for an AI agent:

- **Rule 1:** An AI agent must have the ability to perceive the environment.
- **Rule 2:** The observation must be used to make decisions.
- **Rule 3:** Decision should result in an action.
- **Rule 4:** The action taken by an AI agent must be a rational action.

Rational Agent:

A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.

A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theory for various real-world scenarios.

For an AI agent, the rational action is most important because in the AI reinforcement learning algorithm, for each best possible action, the agent gets the positive reward and for each wrong action, the agent gets a negative reward.

Note: Rational agents in AI are very similar to intelligent agents.

Rationality:

The rationality of an agent is measured by its performance measure. Rationality can be judged on the basis of following points:

- Performance measure which defines the success criterion.
- Agent prior knowledge of its environment.
- Best possible actions that an agent can perform.
- The sequence of percepts.

Note: Rationality differs from Omniscience because an Omniscient agent knows the actual outcome of its action and acts accordingly, which is not possible in reality.

Structure of an AI Agent

The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

1. Agent = Architecture + Agent program

Following are the main three terms involved in the structure of an AI agent:

Architecture: Architecture is machinery that an AI agent executes on.

Agent Function: Agent function is used to map a percept to an action.

1. $f:P^* \rightarrow A$

Agent program: Agent program is an implementation of agent function. An agent program executes on the physical architecture to produce function f.

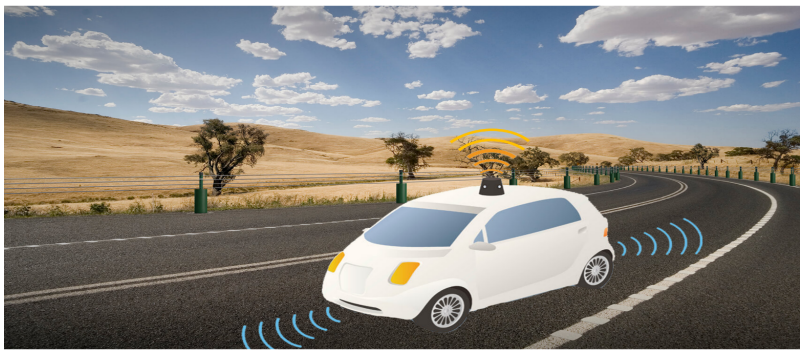
PEAS Representation

PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under the PEAS representation model. It is made up of four words:

- **P:** Performance measure
- **E:** Environment
- **A:** Actuators
- **S:** Sensors

Here performance measure is the objective for the success of an agent's behavior.

PEAS for self-driving cars:



Let's suppose a self-driving car then PEAS representation will be:

Performance: Safety, time, legal drive, comfort

Environment: Roads, other vehicles, road signs, pedestrian

Actuators: Steering, accelerator, brake, signal, horn

Sensors: Camera, GPS, speedometer, odometer, accelerometer, sonar.

Example of Agents with their PEAS representation

Agent	Performance measure	Environment	Actuators	Sensors
1. Medical Diagnose	Healthy patient Minimized cost	Patient Hospital Staff	Tests Treatments	Keyboard (Entry of symptoms)
2. Vacuum Cleaner	Cleanness Efficiency Battery life Security	Room Table Wood floor Carpet Various obstacles	Wheels Brushes Vacuum Extractor	Camera Dirt detection sensor Cliff sensor Bump Sensor Infrared Wall Sensor
3. Part -picking Robot	Percentage of parts in correct bins.	Conveyor belt with parts, Bins	Jointed Arms Hand	Camera Joint angle sensors.

Agent Environment in AI

An environment is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.

The environment is where the agent lives, operates and provides the agent with something to sense and act upon. An environment is mostly said to be non-feminist.

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the search algorithms in AI.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

- **Search:** Searching Is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

a. **Search Space:** Search space represents a set of possible solutions, which a system may have.

b. **Start State:** It is a state from which the agent begins **the search**.

c. **Goal test:** It is a function which observes the current state and returns whether the goal state is achieved or not.

- **Search tree:** A tree representation of a search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action does, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

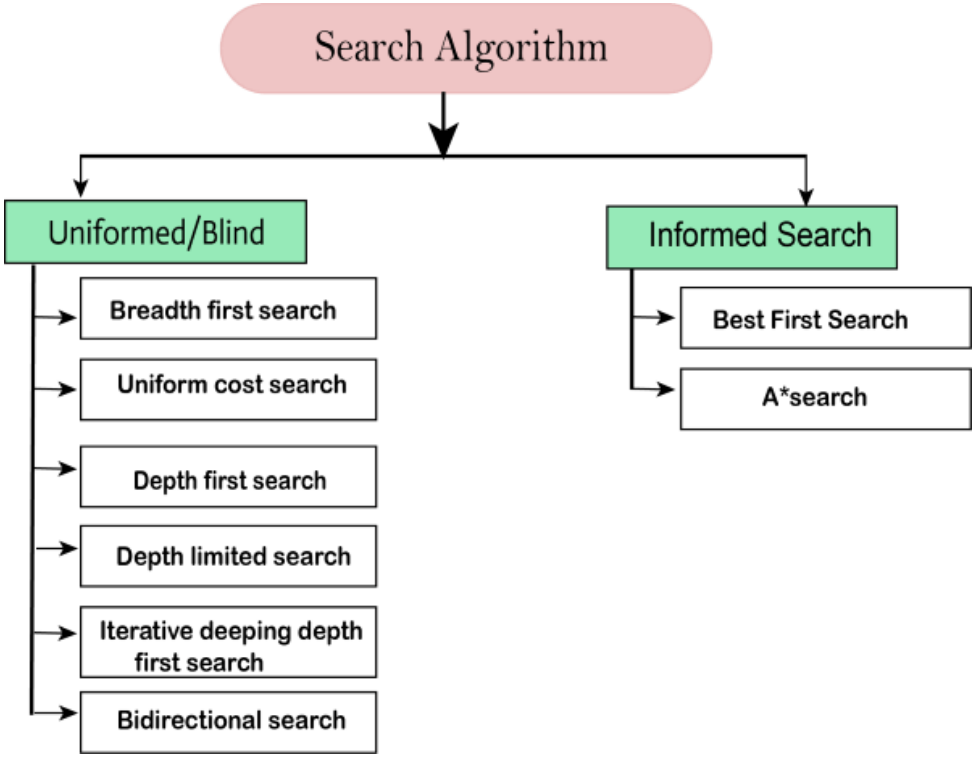
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which a search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search.It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve many complex problems which could not be solved in another way.

An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search

5. Uniform cost search
6. Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- The BFS algorithm starts searching from the root node of the tree and expands all successor nodes at the current level before moving to nodes of the next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

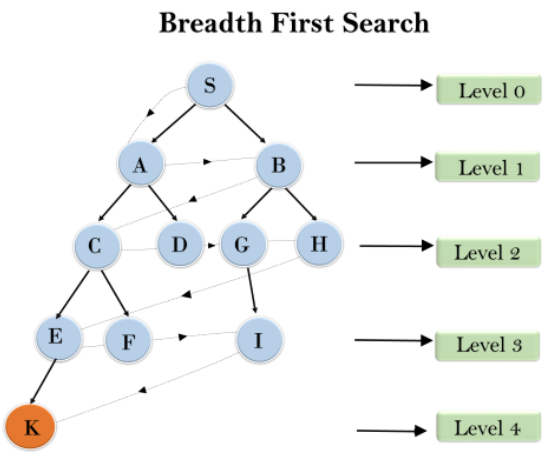
Disadvantages: It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.....+ b^d = O(b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

> Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach the goal node than the BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep reoccurring, and there is no guarantee of finding the solution.
- The DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

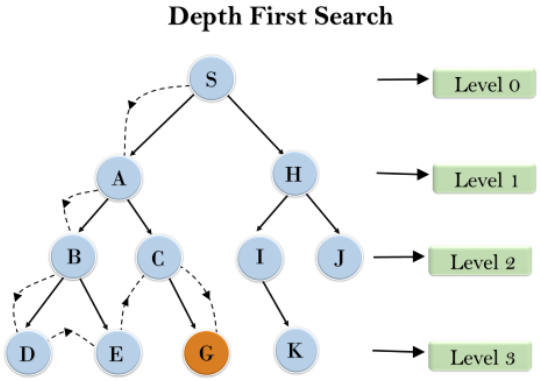
It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still the goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found the goal node.

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)



Space Complexity: DFS algorithm needs to store only a single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will be treated as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that the problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

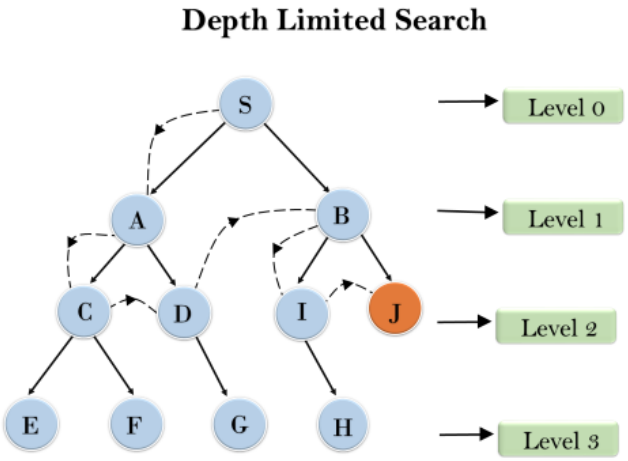
- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example: Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b^{\ell})$.

Space Complexity: Space complexity of DLS algorithm is $O(b \times \ell)$.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.



4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to the BFS algorithm if the path cost of all edges is the same.

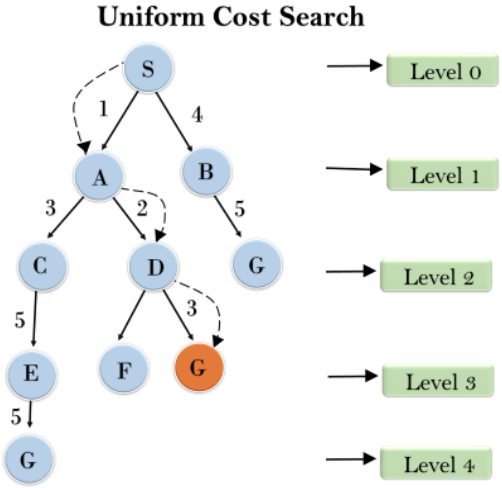
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involved in searching and is only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + \lfloor C^*/\varepsilon \rfloor})}$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful for uninformed search when the search space is large, and depth of the goal node is unknown.

Advantages:

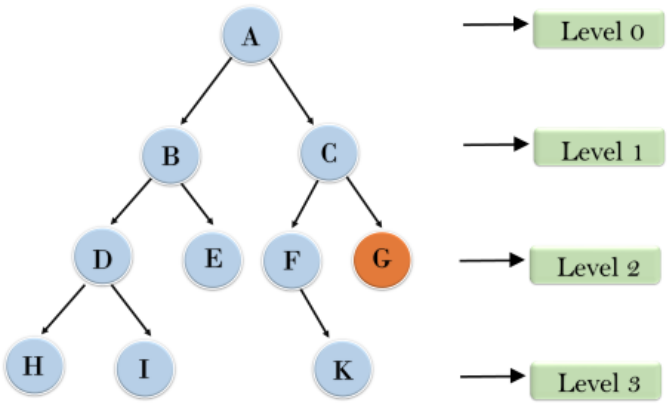
- It Combines the benefits of BFS and DFS search algorithms in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure shows the iterative deepening depth-first search. The IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A

2'nd Iteration----> A, B, C

3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

The IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

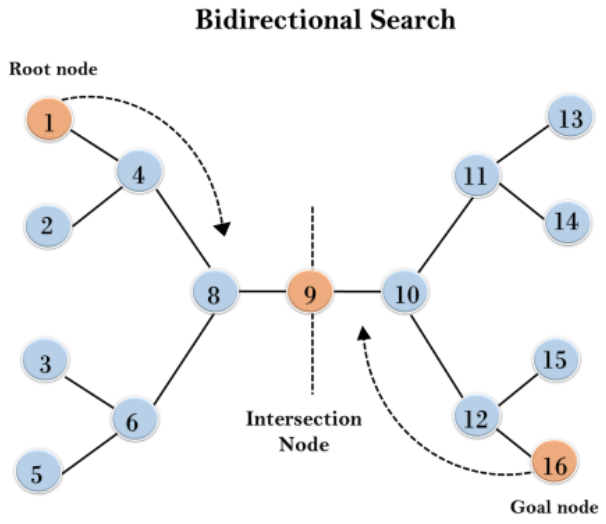
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example:

In the below search tree, a bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

Space Complexity: Space complexity of bidirectional search is **$O(b^d)$**.

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But an informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach the goal node, etc. This knowledge helps agents to explore less of the search space and find the goal node more efficiently.

The informed search algorithm is more useful for large search spaces. Informed search algorithms use the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close the agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as: h(n) <= h*(n)

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues until a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1.) Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

1. f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is a goal node, then return success and terminate the search, else proceed to Step 6.

- **Step 6:** 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.
- **Step 7:** Return to Step 2.

Advantages:

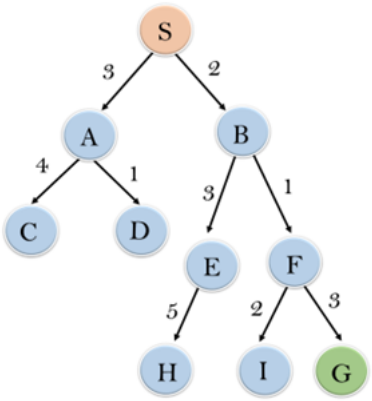
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

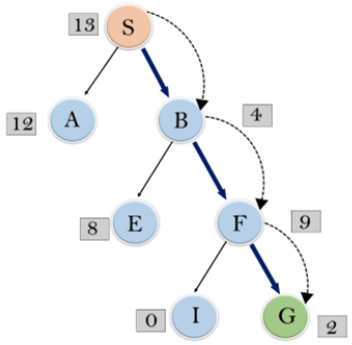
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using the evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iterations for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

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

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

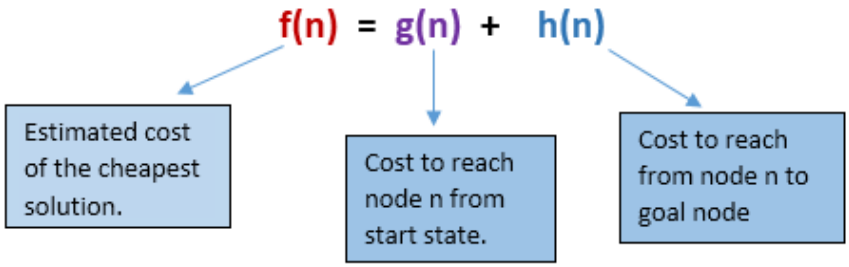
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses the heuristic function h(n), and the cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands the search tree and provides optimal results faster. A* The algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as follows, 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:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stop.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. 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.

Step 5: 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.

Step 6: Return to **Step 2**.

Advantages:

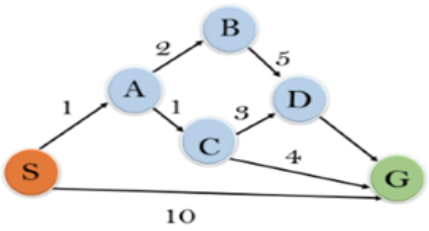
- A* search algorithm is the best algorithm than other search algorithms.
- A* The search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it is mostly based on heuristics and approximation.
- A* The search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from the start state.
Here we will use an OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

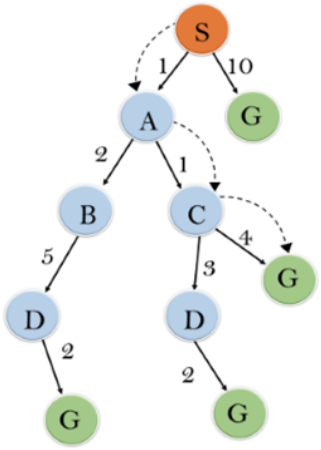
Solution: Initialization: {(S, 5)}

Iteration 1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.



Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* The search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing mathematical problems. One of the widely discussed examples of Hill climbing algorithms is the Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

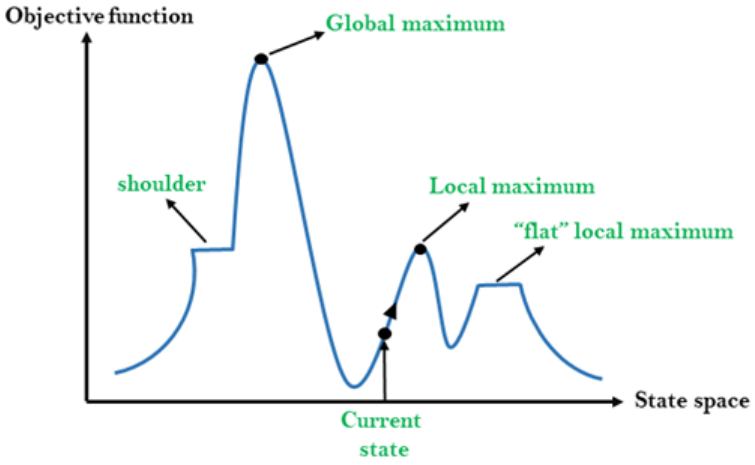
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On the Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of the Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of the state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and sets it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is a goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:

a. If it is a goal state, then return to success and quit.
b. Else if it is better than the current state then assign a new state as a current state.
c. Else if not better than the current state, then return to step2.

- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of a simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is the goal state then return success and stop, else make the current state as the initial state.

- **Step 2:** Loop until a solution is found or the current state does not change.

a. Let SUCC be a state such that any successor of the current state will be better than it.

b. For each operator that applies to the current state:

a. Apply the new operator and generate a new state.

b. Evaluate the new state.

c. If it is a goal state, then return it and quit, else compare it to the SUCC.

d. If it is better than SUCC, then set a new state as SUCC.

e. If the SUCC is better than the current state, then set the current state to SUCC.

- **Step 5:** Exit.

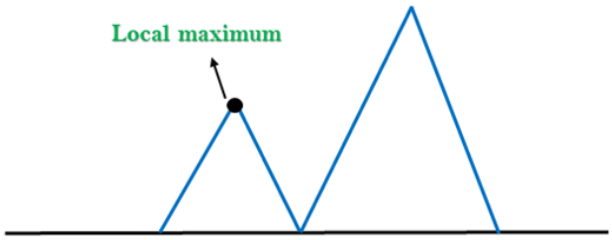
3. Stochastic hill climbing:

Stochastic hill climbing does not examine all its neighbors before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

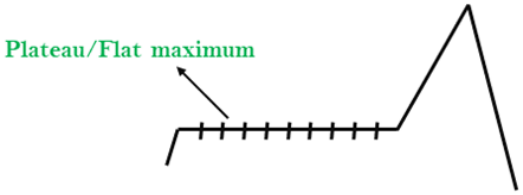
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



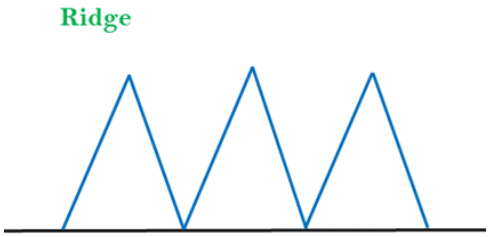
2. Plateau: A plateau is the flat area of the search space in which all the neighboring states of the current state contain the same value, because this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find a non-plateau region.



3. 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.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value is guaranteed to be incomplete because it can get stuck on a local maximum. And if the algorithm applies a random walk, by moving a successor, then it may be complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness.

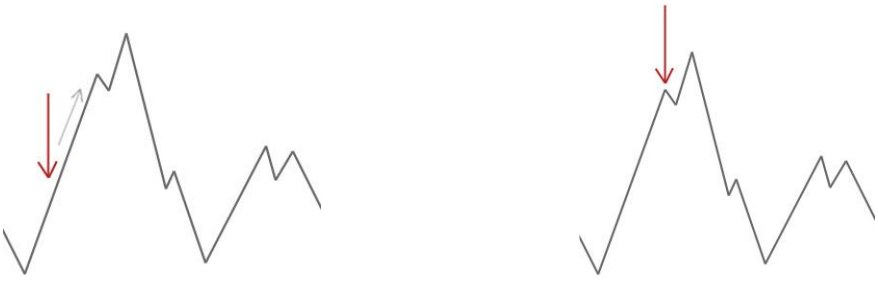
In mechanical terms Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

Advantages of Simulated Annealing

You may be wondering if there is any real advantage to implementing simulated annealing over something like a simple hill climber. Although hill climbers can be surprisingly effective at finding a good solution, they also have a tendency to get stuck in local optimums. As we previously determined, the simulated annealing algorithm is excellent at avoiding this problem and is much better on average at finding an approximate global optimum.

To help better understand, let's quickly take a look at why a basic hill climbing algorithm is so prone to getting caught in local optimums.

A hill climber algorithm will simply accept neighbor solutions that are better than the current solution. When the hill climber can’t find any better neighbors, it stops.



In the example above we start our hill climber off at the red arrow and it works its way up the hill until it reaches a point where it can't climb any higher without first descending. In this example we can clearly see that it's stuck in a local optimum. If this were a real world problem we wouldn't know how the search space looks so unfortunately we wouldn't be able to tell whether this solution is anywhere close to a global optimum.

Simulated annealing works slightly differently than this and will occasionally accept worse solutions. This characteristic of simulated annealing helps it to jump out of any local optimums it might have otherwise got stuck in.

Algorithm Overview

So how does the algorithm look? Well, in its most basic implementation it's pretty simple.

- First we need to set the initial temperature and create a random initial solution.
- Then we begin looping until our stop condition is met. Usually either the system has sufficiently cooled, or a good-enough solution has been found.
- From here we select a neighbor by making a small change to our current solution.
- We then decide whether to move to that neighbor's solution.
- Finally, we decrease the temperature and continue looping.

LOCAL SEARCH IN CONTINUOUS SPACES

Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

Distinction between discrete and continuous environments

- If an environment consists of a finite number of actions that can be deliberated in the environment to obtain the output, it is said to be a discrete environment.
- The game of chess is discrete as it has only a finite number of moves. The number of moves might vary with every game, but still, it's finite.
- The environment in which the actions are performed cannot be numbered i.e. is not discrete, is said to be continuous.
- Self-driving cars are an example of continuous environments as their actions are driving, parking, etc. which cannot be numbered.

None of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors. This section provides a very brief introduction to some local search techniques for finding optimal solutions in continuous spaces.

We begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized. The state space is then defined by the coordinates of the airports: (x1, y1), (x2, y2), and (x3, y3). This is a six-dimensional space; we also say that states are defined by six variables. VARIABLE (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(x1, y1, x2, y2, x3, y3) is relatively easy to compute for any particular state once we compute the closest cities. Let Ci be the set of cities whose closest airport (in the current state) is airport i. Then, in the neighborhood of the current state, where the Cis remain constant, we have

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

This expression is correct locally, but not globally because the sets Ci are (discontinuous) functions of the state. One way to avoid continuous problems is simply to discretize the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount ±δ. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. We could also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length δ 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. For our problem, we have

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

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) 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 we can compute the gradient locally (but not globally); for example,

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

Given a locally correct expression for the gradient, we can perform steepest-ascent hill climb- ing by updating the current state according to the formula

$x \leftarrow x + \alpha \nabla f(x)$,

where α is a small constant often called the step size. In other cases, the objective function might not be available in a differentiable form at all—for example, the value of a particular set of airport locations might be determined by running some large-scale economic simulation package. In those cases, we can calculate a so-called empirical gradient by evaluating the response to small increments and decrements in each coordinate. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space.

Hidden beneath the phrase "α is a small constant" lies 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 f starts to decrease again. The point at which this occurs becomes the new current state. There are several schools of thought about how the new direction should be chosen at this point. For many problems, the most effective algorithm is the venerable 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 x according to Newton's formula

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

To find a maximum or minimum of f, we need to find x such that the gradient is zero (i.e., $\nabla f(x) = 0$). Thus, g(x) in Newton's formula becomes $\nabla f(x)$, 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 Hf (x) is the Hessian matrix of second derivatives, whose elements Hij are given by $\partial^2 f / \partial x_i \partial x_j$. For our airport example, we can see from Equation (4.2) that Hf (x) is particularly simple: the off-diagonal elements are zero and the diagonal elements for airport i are just twice the number of cities in Ci. A moment's calculation shows that one step of the update moves airport i directly to the centroid of Ci, which is the minimum of the local expression for f from Equation (4.1).7 For high-dimensional problems, however, computing the n2 entries of the Hessian and inverting it may be expensive, so many approximate versions of the Newton–Raphson method have been developed. Local search methods suffer from local maxima, ridges, and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get

lost. A final topic with which a passing acquaintance is useful is constrained optimization. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. For example, in our airport-siting problem, we might constrain sites

to be inside Romania and on dry land (rather than in the middle of lakes). 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 8 and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables. Linear programming is probably the most widely studied and broadly useful class of optimization problems. 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.

SEARCHING WITH NONDETERMINISTIC ACTIONS

In Chapter 3, we assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its precepts provide no new information after each action, although of course they tell the agent the initial state.

When the environment is either partially observable or nondeterministic (or both), percepts become useful. In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals. When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred. In both cases, the future precepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but a contingency plan (also known as a strategy) that specifies what to do depending on what percepts are received.

The erratic vacuum world

As an example, we use the vacuum world, first introduced in Chapter 2 and defined as a search problem in Section 3.2.1. Recall that the state space has eight states, as shown in Figure 4.9. There are three actions—Left, Right, and Suck—and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms in Chapter 3 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.

Now suppose that we introduce nondeterminism 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, we need to generalize the notion of a transition model from Chapter 3. Instead of defining the transition model by a RESULT function that returns a single state, we use a RESULTS function that returns a set of possible outcome states. For example, in the erratic vacuum world, the Suck action in state 1 leads to a state in the set {5, 7}—the dirt in the right-hand square may or may not be vacuumed up.

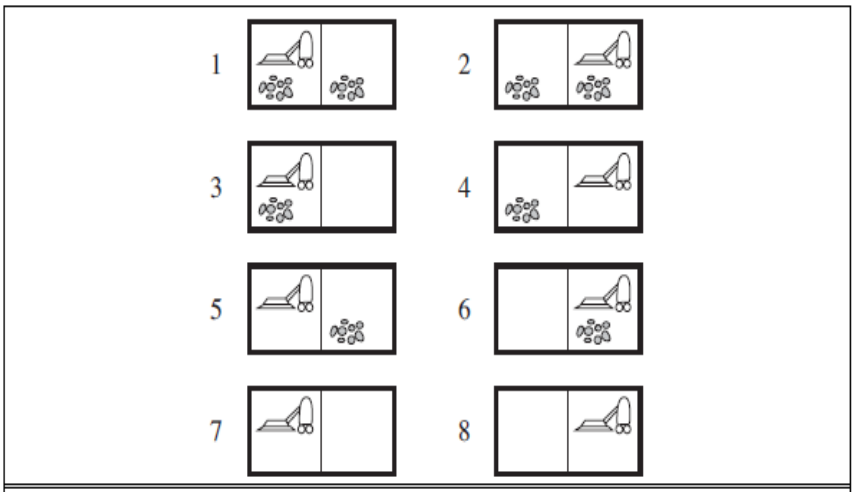


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

We also need to generalize the notion of a solution to the problem. For example, if we start in state 1, there is no single sequence of actions that solves the problem. Instead, we need a contingency plan such as the following:

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

Thus, solutions for nondeterministic problems can contain nested if–then–else statements; this means that they are trees rather than sequences. This allows the selection of actions based on contingencies arising during execution. Many problems in the real, physical world are contingency problems because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

AND–OR search trees

The next question is how to find contingent solutions to non deterministic problems. As in Chapter 3, we begin by constructing search trees, but here the trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call 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. We call these nodes AND nodes. For example, the Suck action in state 1 leads to a state in the set {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 4.10.

A solution for an AND–OR search problem is a subtree 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 given in Equation (4.3). (The plan uses if–then–else notation to handle the AND branches, but when there are more than two branches at a node, it might be better to use a case construct.) Modifying the basic problem-solving agent shown in Figure 3.1 to execute contingent solutions of this kind are straightforward. One may also consider a somewhat different agent design, in which the agent can act before it has found a guaranteed plan and deals with some contingencies only as they arise during execution. This type of interleaving of search and execution is also useful for exploration problems (see Section 4.5) and for game playing (see Chapter 5).

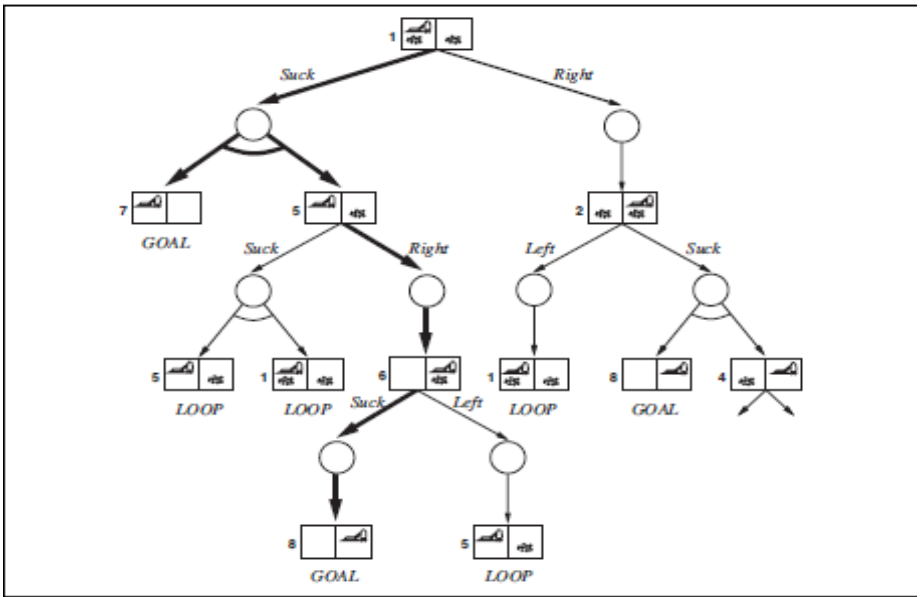


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

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

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, 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]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Figure 4.11 gives a recursive, depth-first algorithm for AND–OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). 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, we ensure that the algorithm 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. Exercise 4.5 investigates this issue.

AND–OR graphs can also be explored by breadth-first or best-first methods. The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence, but the notion of admissibility carries over and there is an analog of the A* algorithm for finding optimal solutions. Pointers are given in the bibliographical notes at the end of the chapter.

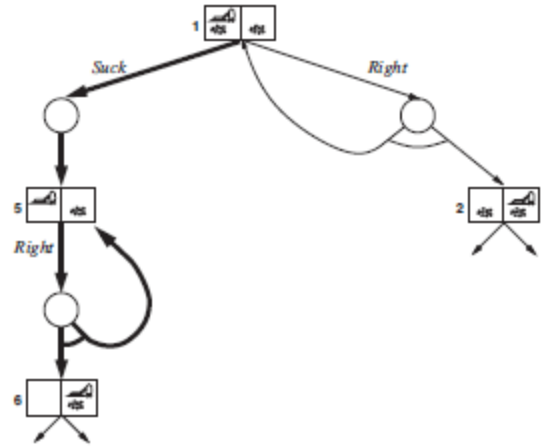


Figure 4.12 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

Try, try again

Consider the 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. For example, moving Right in state 1 leads to the state set {1, 2}. Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state 1, and AND-OR-GRAPH-SEARCH would return with failure. There is, however, a cyclic solution, which is to keep trying Right until it works. We can express this solution by adding a label to denote some portion of the plan and using that label later instead of repeating the plan itself.
Thus, our cyclic solution is

[Suck, L1 : Right , if State =5 then L1 else Suck] .

(A better syntax for the looping part of this plan would be "while State =5 do Right .") In general a cyclic plan may be considered a solution provided that every leaf is a goal state and that a leaf is reachable from every point in the plan. The modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 4.6. The key realization is that a loop in the state space back to a state L translates to a loop in the plan back to the point where the subplan for state L is executed.

Given the definition of a cyclic solution, an agent executing such a solution will eventually reach the goal provided that each outcome of a nondeterministic action eventually occurs. Is this condition reasonable? It depends on the reason for the nondeterminism. If the action rolls a die, then it's reasonable to suppose that eventually a six will be rolled. If the action is to insert a hotel card key into the door lock, but it doesn't work the first time, then perhaps it will eventually work, or perhaps one has the wrong key (or the wrong room!). After seven or eight tries, most people will assume the problem is with the key and will go back to the front desk to get a new one. One way to understand this decision is to say that the initial problem formulation (observable, nondeterministic) is abandoned in favor of a different formulation (partially observable, deterministic) where the failure is attributed to an unobservable property of the key. We have more to say on this issue in Chapter 13.

SEARCHING WITH PARTIAL OBSERVATIONS

We now turn to the problem of partial observability, where the agent's precepts do not suffice to pin down the exact state. As noted at the beginning of the previous section, if the agent is in one of several possible states, then an action may lead to one of several possible outcomes—even if the environment is deterministic. The key concept required for solving partially observable problems is the belief state, representing the agent's current

belief about the possible physical states it might be in, given the sequence of actions and perceptions up to that point. We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as non deterministic actions.

Searching with no observation

When the agent's precepts provide no information at all, we have what is called a sensor less problem or sometimes a conformant problem. At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable. Moreover, sensorless agents can be surprisingly useful, primarily because they don't rely on sensors working properly. In manufacturing systems, for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all. The high cost of sensing is another reason to avoid it: for example, doctors often prescribe a broad spectrum antibiotic rather than using the contingent plan of doing an expensive blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.

We can make a sensorless version of the vacuum world. Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of dirt. In that case, its initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}. Now, consider what happens if it tries the action Right. This will cause it to be in one of the states {2, 4, 6, 8}—the agent now has more information! Furthermore, the action sequence [Right,Suck] will always end up in one of the states {4, 8}. Finally, the sequence [Right,Suck,Left,Suck] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can coerce the world into state 7.

To solve sensorless problems, we search in the space of belief states rather than physical states.10 Notice that in belief-state space, the problem is fully observable because the agent always knows its own belief state. Furthermore, the solution (if any) is always a sequence of actions. This is because, as in the ordinary problems of Chapter 3, the precepts 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 non deterministic.

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by ACTIONSP, RESULTP, GOAL-TESTP, and STEP-COSTP .

Then we can define the corresponding sensorless problem as follows:
• Belief states: The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2N states, although many may be unreachable from the initial state.
• Initial state: Typically the set of all states in P, although in some cases the agent will have more knowledge than this.
• Actions: This is slightly tricky. Suppose the agent is in belief state b={s1, s2}, but ACTIONSP (s1) = ACTIONSP (s2); then the agent is unsure of which actions are legal. If we 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:

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

On the other hand, if an illegal action might be the end of the world, it is safer to allow only the intersection, that is, the set of actions legal in all the states. For the vacuum world, every state has the same legal actions, so both methods give the same result.
• Transition model: The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\} . \quad (4.4)$$

With deterministic actions, b' is never larger than b. With nondeterminism, we have

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

which may be larger than b, as shown in Figure 4.13. The process of generating the new belief state after the action is called the prediction step; the notation b = PREDICTP (b, a) will come in handy.

• Goal test: The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if all the physical states in it satisfy GOAL-TESTP . The agent may accidentally achieve the goal earlier, but it won't know that it has done so.

• Path 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. (This gives rise to a new class of problems, which we explore in Exercise 4.9.) For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

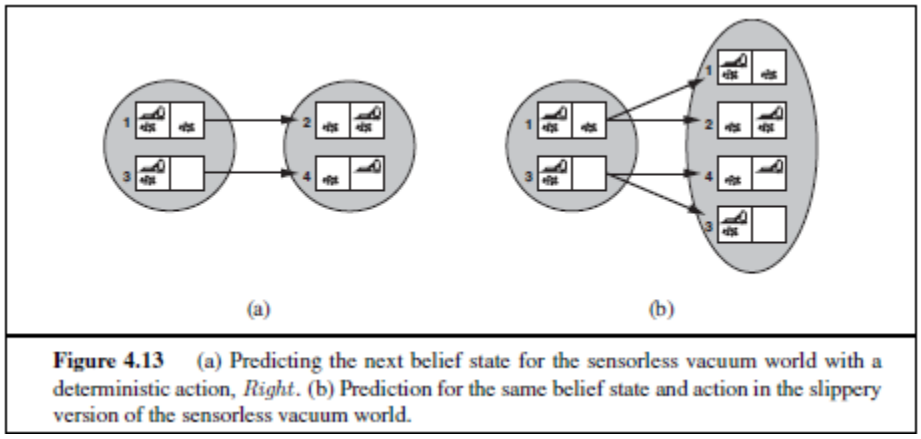


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Figure 4.14 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of 28 =256 possible belief states.

The preceding definitions enable the automatic construction of the belief-state problem formulation from the definition of the underlying physical problem. Once this is done, we can apply any of the search algorithms of Chapter 3. In fact, we can do a little bit more than that. In "ordinary" graph search, newly generated states are tested to see if they are identical to existing states. This works for belief states, too; for example, in Figure 4.14, the action sequence [Suck,Left,Suck] starting at the initial state reaches the same belief state as [Right,Left,Suck], namely, {5, 7}. Now, consider the belief state reached by [Left], namely, {1, 3, 5, 7}. Obviously, this is not identical to {5, 7}, but it is a superset. It is easy to prove (Exercise 4.8) that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Hence, we can discard a path reaching {1, 3, 5, 7} if {5, 7} has already been generated. Conversely, if {1, 3, 5, 7} has already been generated and found to be solvable, then any subset, such as {5, 7}, is 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 as we have described it is seldom feasible in practice. The difficulty is not so much the vastness of the belief-state space—even though it is exponentially larger than the underlying physical state space; in most cases the branching factor and solution length in the belief-state space and physical state space are not so different. The real difficulty lies with the size of each belief state. For example, the initial belief state for the 10×10 vacuum world contains 100×2100 or around 1032 physical states—far too many if we use the atomic representation, which is an explicit list of states.

One solution is to represent the belief state by some more compact description. In English, we could say the agent knows "Nothing" in the initial state; after moving Left, we could say, "Not in the rightmost column," and so on. Chapter 7 explains how to do this in a formal representation scheme. Another approach is to avoid the

standard search algorithms, which treat belief states as black boxes just like any other problem state. Instead, we can look

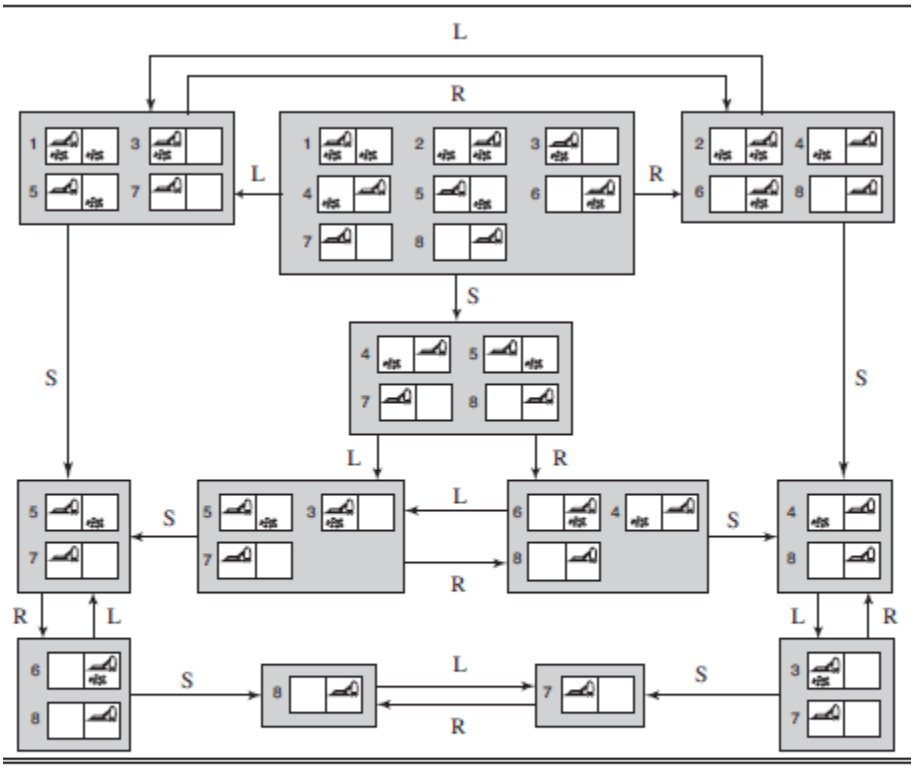


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled links. Self-loops are omitted for clarity.

inside the belief states and develop incremental belief-state search algorithms that build up the solution one physical state at a time. For example, in the sensorless vacuum world, the initial belief state is {1, 2, 3, 4, 5, 6, 7, 8}, and we have to find an action sequence that works in all 8 states. We can do this by first finding a solution that works for state 1; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on. 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, it is usually the case that a small subset of the belief state, consisting of the first few states examined, is also unsolvable. In some cases, this leads to a speedup proportional to the size of the belief states, which may themselves be as large as the physical state space itself.

Even the most efficient solution algorithm is not of much use when no solutions exist. Many things just cannot be done without sensing. For example, the sensorless 8-puzzle is impossible. On the other hand, a little bit of sensing can go a long way. For example, every 8-puzzle instance is solvable if just one square is visible—the solution involves moving each tile in turn into the visible square and then keeping track of its location.

Searching with observations

For a general partially observable problem, we have to specify how the environment generates precepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem

specification includes a PERCEPT(s) function that returns the percept received in a given state. (If sensing is nondeterministic, then we use a PERCEPTS function that returns a set of possible percepts.) For example, in the local-sensing vacuum world, the PERCEPT in state 1 is [A, Dirty]. Fully observable problems are a special case in which PERCEPT(s)=s for every state s, while sensorless problems are a special case in which PERCEPT(s)=null .

When observations are partial, it will usually be the case that several states could have produced any given percept. For example, the percept [A, Dirty] is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be {1, 3}. The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated. We can think of transitions from one belief state to the next for a particular action as occurring in three stages, as shown in Figure 4.15:

• The prediction stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is ˆb =PREDICT(b, a).11
• The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state:

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\} .$$

• The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state bo is just the set of states in ˆb that could have produced the percept:

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

Notice that each updated belief state bo can be no larger than the predicted belief state ˆ b; observations can only help reduce uncertainty compared to the sensorless case. Moreover, for deterministic sensing, the belief states for the different possible precepts will be disjoint, forming a partition of the original predicted belief state.

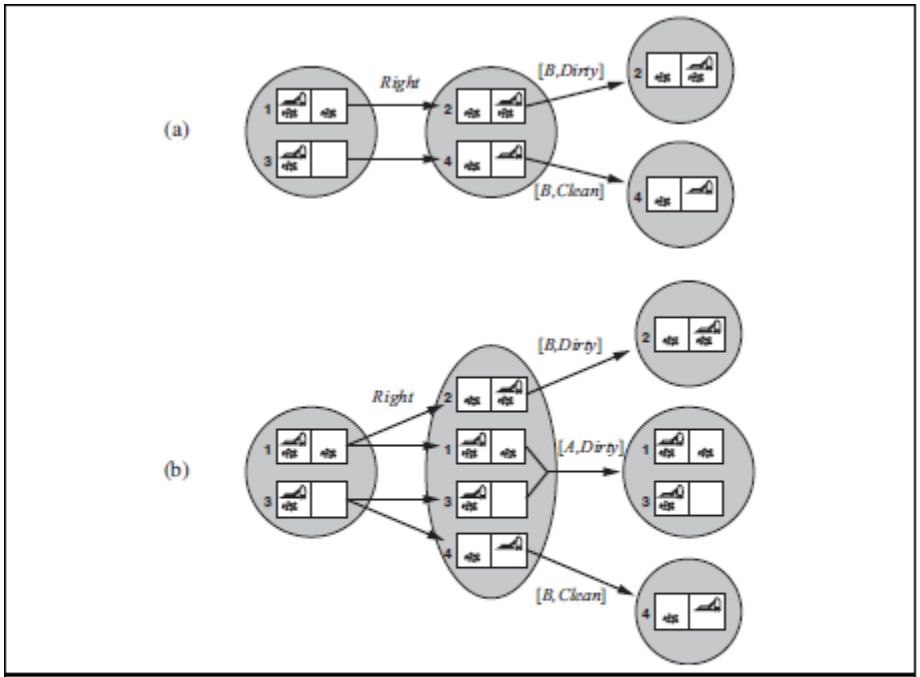


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are $[B, Dirty]$ and $[B, Clean]$, leading to two belief states, each of which is a singleton. (b) In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are $[A, Dirty]$, $[B, Dirty]$, and $[B, Clean]$, leading to three belief states as shown.

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\} .$$

Again, the nondeterminism in the partially observable problem comes from the inability to predict exactly which percept will be received after acting; underlying nondeterminism in the physical environment may contribute to this inability by enlarging the belief state at the prediction stage, leading to more precepts at the observation stage.

Solving partially observable problems

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem and the PERCEPT function. Given

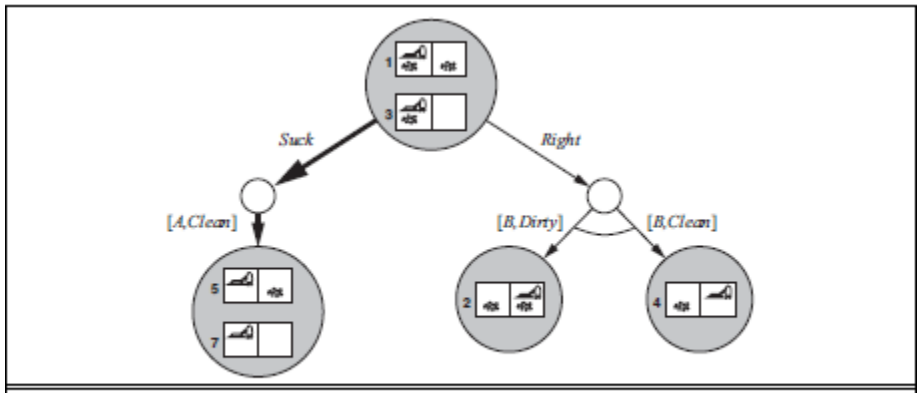


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

such a formulation, the AND–OR search algorithm of Figure 4.11 can be applied directly to derive a solution. Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [A, Dirty]. The solution is the conditional plan

[Suck, Right, if Bstate ={6} then Suck else [ ]] .

Notice that, because we supplied a belief-state problem to the AND–OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state. This is as it should be: in a partially observable environment the agent won't be able to execute a solution that requires testing the actual state.

As in the case of standard search algorithms applied to sensorless problems, the AND–OR search algorithm treats belief states as black boxes, just like any other states. One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state, just as for sensorless problems. One can also derive incremental search algorithms, analogous to those described for sensorless problems, that provide substantial speedups over the black-box approach.

An agent for partially observable environments

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent in Figure 3.1: the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution. There are two main differences. First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if–then–else

expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly. Second, the agent will need to maintain its belief state as it performs actions and receives percepts. This process resembles the prediction–observation–update process in Equation (4.5) but is actually simpler because the percept is given by the environment rather than calculated by the

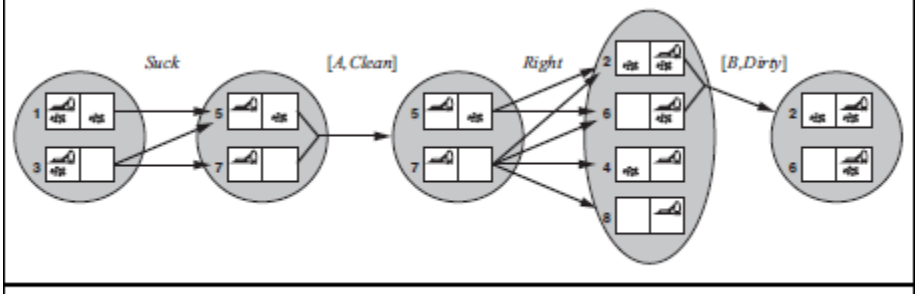


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

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

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

Figure 4.17 shows the belief state being maintained in the kindergarten vacuum world with local sensing, wherein any square may become dirty at any time unless the agent is actively cleaning it at that moment.12

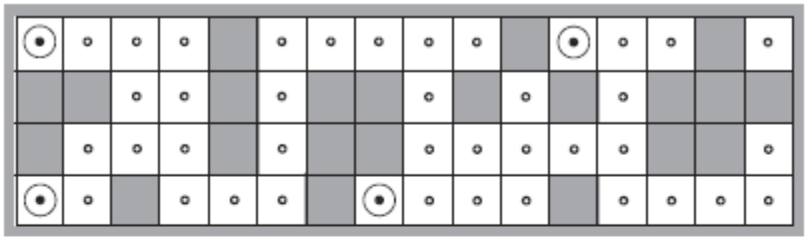
In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. This function goes under various names, including monitoring, filtering and state estimation. Equation (4.6) is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence. If the agent is not to "fall behind," the computation has to happen as fast as percepts are coming in. As the environment becomes more complex, the exact update computation becomes infeasible and the agent will have 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, as explained in Chapter 15. Here we will show an example in a discrete environment with deterministic sensors and nondeterministic actions.

The example concerns a robot with the task of localization: working out where it is, given a map of the world and a sequence of percepts and actions. Our robot is placed in the maze-like environment of Figure 4.18. The robot is equipped with four sonar sensors that tell whether there is an obstacle—the outer wall or a black square in the figure—in each of the four compass directions. We assume that the sensors give perfectly correct data, and that the robot has a correct map of the environment. But unfortunately the robot's navigational system is broken, so when it executes a Move action, it moves randomly to one of the adjacent squares. The robot's task is to determine its current location.

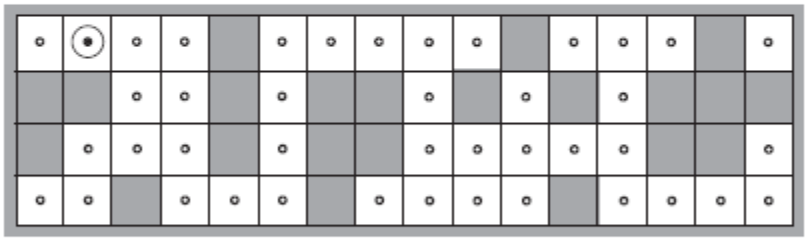
Suppose the robot has just been switched on, so it does not know where it is. Thus its initial belief state b consists of the set of all locations. The robot receives the percept NSW, meaning there are obstacles to the north, west, and south, and does an update using the equation bo =UPDATE(b), yielding the 4 locations shown in Figure 4.18(a). You can inspect the maze to see that those are the only four locations that yield the percept NWS.

Next the robot executes a Move action, but the result is nondeterministic. The new belief state, ba =PREDICT(bo,Move), contains all the locations that are one step away from the locations in bo. When the second percept, NS, arrives, the robot does UPDATE(ba,NS) and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b). That's the only location that could be the result of

UPDATE(PREDICT(UPDATE(b,NSW),Move),NS) .



(a) Possible locations of robot after $E_1 = NSW$



(b) Possible locations of robot After $E_1 = NSW, E_2 = NS$

Figure 4.18 Possible positions of the robot, ⊙, (a) after one observation $E_1 = NSW$ and (b) after a second observation $E_2 = NS$. When sensors are noiseless and the transition model is accurate, there are no other possible locations for the robot consistent with this sequence of two observations.

With non deterministic 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 there were one or more long east-west corridors, then a robot could receive a long sequence of NS percepts, but never know where in the corridor(s) it was.

ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

So far we have concentrated on agents that use offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an online search13 agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't. Of course, there is a tradeoff: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an exploration problem and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however. Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment (Chapter 17 relaxes these assumptions), but we stipulate that the agent knows only the following:

• ACTIONS(s), which returns a list of actions allowed in state s;
• The step-cost function c(s, a, s
)—note that this cannot be used until the agent knows
that s
is the outcome; and
• GOAL-TEST(s).

Note in particular that the agent cannot determine RESULT(s, a) except by actually being in s and doing a. For example, in the maze problem shown in Figure 4.19, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

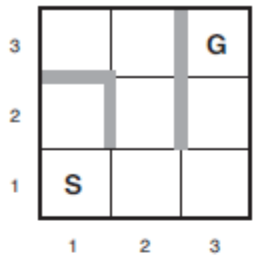


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

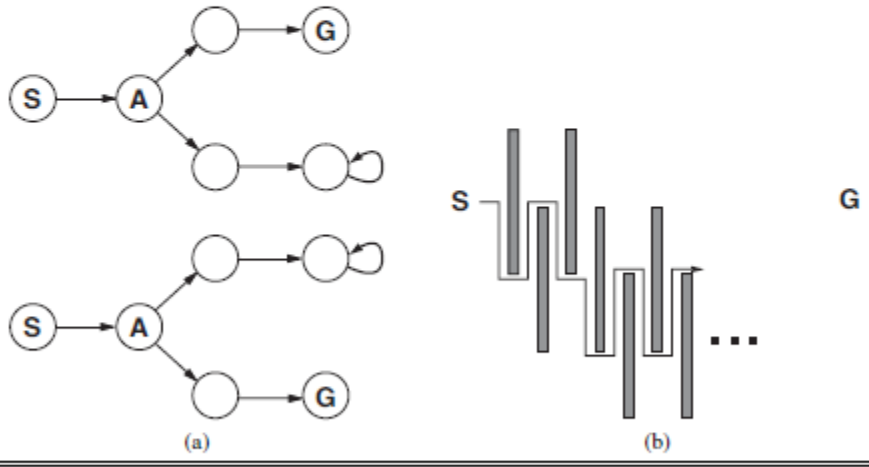


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Finally, the agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state. For example, in Figure 4.19, the agent might know the location of the goal and be able to use theManhattan-distance heuristic.

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 of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow if it knew the search space in advance—that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms, this is called the competitive ratio; we would like it to be as small as possible.

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example, if some actions are irreversible i.e., they lead to a state from which no action leads back to the previous state—the online search might accidentally reach a dead-end state from which no goal state is reachable. Perhaps the term "accidentally" is unconvincing—after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that no algorithm can avoid dead ends in all state spaces. Consider the two dead-end state spaces in Figure 4.20(a). To an online search algorithm that has visited states S and A, the two state spaces look identical, so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an adversary argument—we can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses.

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we simply assume that the state space is safely explorable—that is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.20(b) shows. For this reason, it is common to describe 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 receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide 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.

For example, offline algorithms such as A∗ can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies. To avoid traveling all the way across the tree 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 backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.21. This agent stores its map in a table, RESULT[s, a], that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. 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 physically. 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 a 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.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← 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 ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

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, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one RANDOM WALK might consider using a random walk to explore the environment. 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.14 On the other hand, the process can be very slow. Figure 4.22 shows an environment in which a random walk will take exponentially many steps to find the goal because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

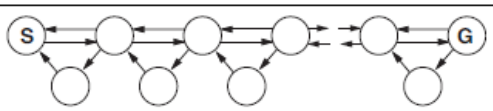


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

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. Figure 4.23 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor's is the cost to get to s plus the estimated cost to get to a goal from there—that is, c(s, a, s) + H(s). In the example, there are two actions, with estimated costs 1+9 and 1+2, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic.

Since the best move costs 1 and leads to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure 4.23(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

An agent implementing this scheme, which is called learning real-time A* (LRTA*), is shown in Figure 4.24. 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 s 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.

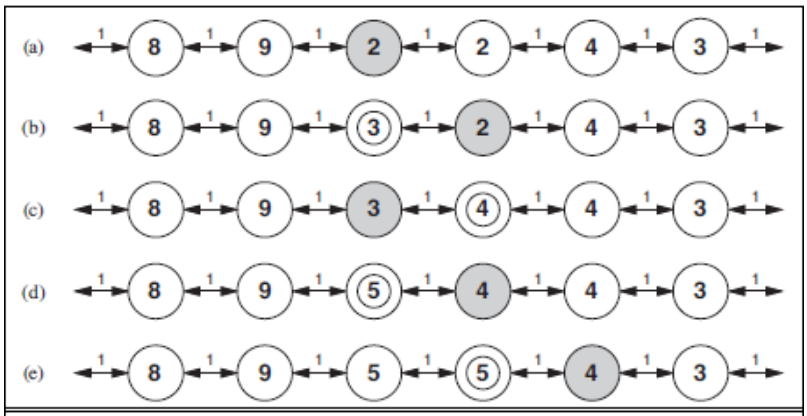


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

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

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

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 n states in O(n2) 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. We discuss this family, developed originally for stochastic environments, in Chapter 21.

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. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*. In Chapter 21, we show that 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.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.19, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,1) to (1,2), the agent still has no idea that the Down action goes back to (1,1) or that the Up action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, that Down reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the RESULT function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent.