1

Intelligent agents are supposed to maximize their performance measure. As we mentioned in Chapter 2, achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it. Let us first look at why and how an agent might do this.

64

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. The agent's performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on. The decision problem is a complex one involving many tradeoffs and careful reading of guidebooks. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the **goal** of getting to Bucharest. Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified. Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider. **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

GOAL FORMULATION

We will consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out how to act, now and in the future, so that it reaches a goal state. Before it can do this, it needs to decide (or we need to decide on its behalf) what sorts of actions and states it should consider. If it were to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution. **Problem formulation** is the process of deciding what actions and states to consider, given a goal. We discuss this process in more detail later. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. Each state therefore corresponds to being in a particular town.

PROBLEM FORMULATION

Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad. Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind. None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.[1] In other words, the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action. If the agent has no additional information—i.e., if the environment is **unknown** in the sense defined in Section 2.3—then it is has no choice but to try one of the actions at random. This sad situation is discussed in Chapter 4.

But suppose the agent has a map of Romania. The point of a map is to provide the agent with information about the states it might get itself into and the actions it can take. The agent can use this information to consider *subsequent* stages of a hypothetical journey via each of the three towns, trying to find a journey that eventually gets to Bucharest. Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions that correspond to the legs of the journey. In general, *an agent with several immediate options of unknown value can decide what to do by first examining* future *actions that eventually lead to states of known value.*

To be more specific about what we mean by "examining future actions," we have to be more specific about properties of the environment, as defined in Section 2.3. For now,

[1] We are assuming that most readers are in the same position and can easily imagine themselves to be as clueless as our agent. We apologize to Romanian readers who are unable to take advantage of this pedagogical device.

we assume that the environment is **observable**, so the agent always knows the current state. For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers. We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities. We will assume the environment is **known**, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.) Finally, we assume that the environment is **deterministic**, so each action has exactly one outcome. Under ideal conditions, this is true for the agent in Romania—it means that if it chooses to drive from Arad to Sibiu, it does end up in Sibiu. Of course, conditions are not always ideal, as we show in Chapter 4.

Under these assumptions, the solution to any problem is a fixed sequence of actions. "Of course!" one might say, "What else could it be?" Well, in general it could be a branching strategy that recommends different actions in the future depending on what percepts arrive. For example, under less than ideal conditions, the agent might plan to drive from Arad to Sibiu and then to Rimnicu Vilcea but may also need to have a contingency plan in case it arrives by accident in Zerind instead of Sibiu. Fortunately, if the agent knows the initial state and the environment is known and deterministic, it knows exactly where it will be after the first action and what it will perceive. Since only one percept is possible after the first action, the solution can specify only one possible second action, and so on.

SEARCH
SOLUTION

The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This

EXECUTION

is called the **execution** phase. Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1. After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

Notice that while the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be. An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going

OPEN-LOOP

on. Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.

3.2.1 Toy problems

The first example we examine is the **vacuum world** first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

2b

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

Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier. Chapter 4 relaxes some of these assumptions.

The **8-puzzle**, an instance of which is shown in Figure 3.4, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

[redacted]

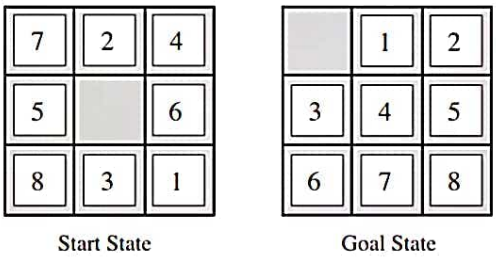


Figure 3.4 A typical instance of the 8-puzzle.

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

What abstractions have we included here? The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding. We have abstracted away actions such as shaking the board when pieces get stuck and ruled out extracting the pieces with a knife and putting them back again. We are left with a description of the rules of the puzzle, avoiding all the details of physical manipulations.

ɔcĸ

The 8-puzzle belongs to the family of **sliding-block puzzles**, which are often used as test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has $9!/2 = 181,440$ reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5×5 board) has around 10^{25} states, and random instances take several hours to solve optimally.

3.2.2 Real-world problems

TE-FINDING BLEM

We have already seen how the **route-finding problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site:

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

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the byzantine fare structures that airlines impose. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—such as backup reservations on alternate flights—to the extent that these are justified by the cost and likelihood of failure of the original plan.

RING PROBLEM

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the *set of cities the agent has visited.* So the initial state would be $In(Bucharest)$, $Visited(\{Bucharest\})$, a typical intermediate state would be $In(Vaslui)$, $Visited(\{Bucharest, Urziceni, Vaslui\})$, and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

'ELING SPERSON BLEM

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

Y PROBLEM

EAL-WORLD
ROBLEM

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *toy* and *real-world* problems. A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms. A **real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

3

The search problem is a fundamental problem in artificial intelligence (AI) and computer science. It involves finding a solution or a sequence of actions that lead from an initial state to a goal state, given a set of available actions and a description of the problem.

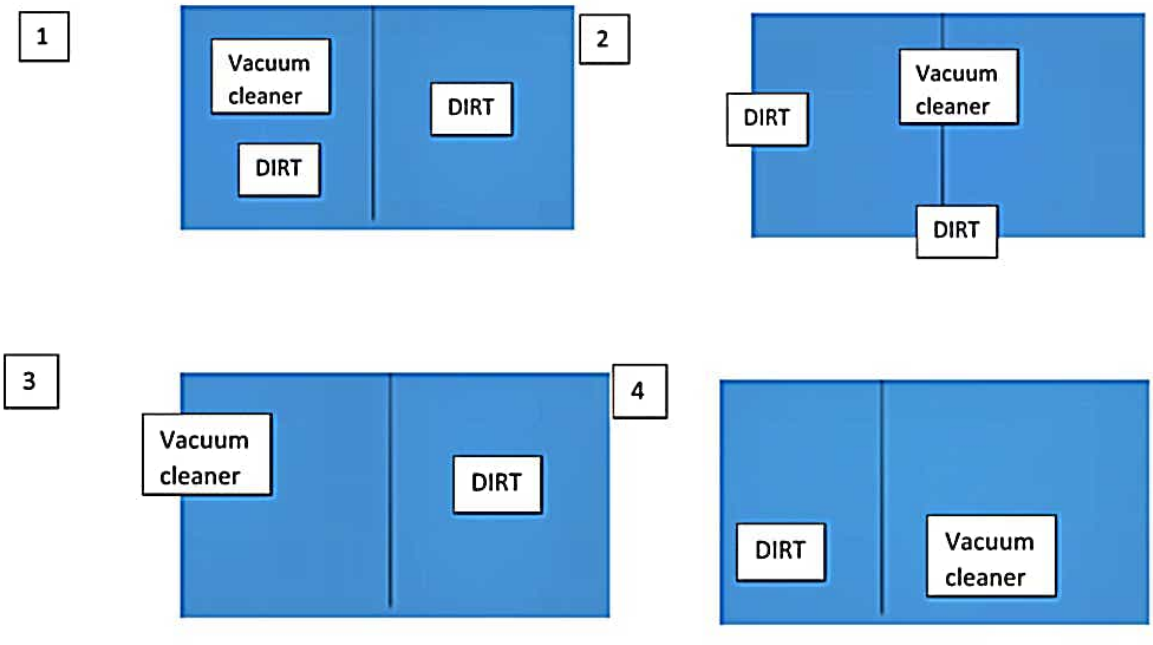
In order to solve the search problem, we need to define several key terms:

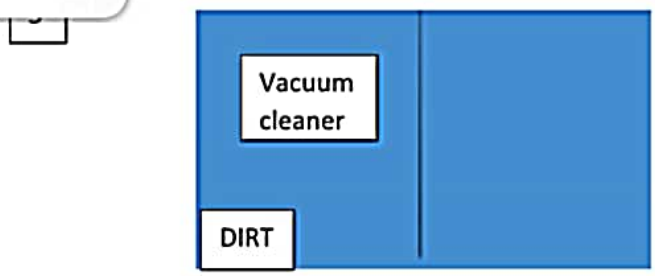
1. State space: The set of all possible states that the system can be in. It is the space of all possible configurations that the system can take.
2. Initial state: The starting state of the system. It is the state from which the search process begins.
3. Goal state: The desired or target state of the system. It is the state that the search process is trying to reach.
4. Actions: The set of possible actions that can be taken to change the state of the system.
5. Transition model: The function that describes the effects of taking an action in a given state. It maps a state and an action to a new state.
6. Action cost function: A function that assigns a cost to each action. It is used to determine the best sequence of actions that lead to the goal state with the lowest cost.

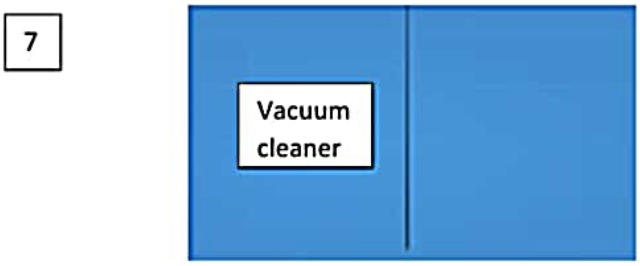
In summary, the search problem involves navigating through a state space by applying actions, using a transition model to generate new states, and using an action cost function to determine the optimal sequence of actions that lead from an initial state to a goal state.

4 **Vacuum cleaner problem** is a well-known search problem for an agent which works on Artificial Intelligence. In this problem, our vacuum cleaner is our agent. It is a goal based agent, and the goal of this agent, which is the vacuum cleaner, is to clean up the whole area. So, in the classical vacuum cleaner problem, we have two rooms and one vacuum cleaner. There is dirt in both the rooms and it is to be cleaned. The vacuum cleaner is present in any one of these rooms. So, we have to reach a state in which both the rooms are clean and are dust free.

So, there are eight possible states possible in our **vacuum cleaner problem**. These can be well illustrated with the help of the following diagrams:







Here, states 1 and 2 are our initial states and state 7 and state 8 are our final states (goal states). This means that, initially, both the rooms are full of dirt and the **vacuum cleaner** can reside in any room. And to reach the final goal state, both the rooms should be clean and the **vacuum cleaner** again can reside in any of the two rooms.

The **vacuum cleaner** can perform the following functions: move left, move right, move forward, move backward and to suck dust. But as there are only two rooms in our problem, the vacuum cleaner performs only the following functions here: move left, move right and suck.

5a

Q. Differentiate btw uniformed search algorithm and informed search algorithm. Explain about DFS and BFS in detail.

Ans Difference btw uniform Search & informed Search Algorithm.

Parameters	informed search	Uniformed Search
Known as	It is also known as Heuristic search	It is also known as Blind search
Using knowledge	It uses knowledge for the searching process	It doesn't use knowledge for searching process.
Performance	It finds a solution more quickly.	It find a solution slow compared to informed search
Completion	It may or may not be complete.	It is always complete.
Cost-factor	Cost is low.	Cost is high.
Time	Consumes less time because of quick searching.	Consumes moderate time because of slow searching.
Efficiency	It is more efficient.	It is comparitively less efficient.
Implementation	It is less lengthy while implemented.	It is more lengthy while implemented.
Size of Search problem	Having a wide scope in terms of handling large search problems.	Solving a massive search task is challenging.
Examples of Algorithms	A* Search, AO* search, Hill climbing Algorithm.	Deapth first search (DFS), Breadth first search (BFS) Branch and Bound.

5b

Ans: Heuristic function: It 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 agent is from the goal. This method, however, might not always give the best solution, but it guarantee to find a good solution in reasonable time. This 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

btw the pair of states. The value of the heuristic function is always positive.

Best-First Search Algorithm (Greedy search):

→ Greedy best-first search algorithm always select 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.

→ It 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 this algorithm we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e

$f(n) = g(n)$, where $h(n)$ = estimated cost from node n to goal.

→ This 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 goal node, then return success and terminate the search, else proceed to step 6.

Step 6: For each successor node, algorithm checks for evaluation function $f(n)$, and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

Step 7: Return to step 2.

A* Search Algorithm:

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

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

Algorithm of A* Search:

Step 1: 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 stops.

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 evaluation function for n and place into open list.

Step 5: Else if node n is already in OPEN and CLOSED, the it should be attached to the back pointer which reflects the lowest g(n) value.

Step 6: Return to Step 2.

3.5.2 A* search: Minimizing the total estimated solution cost

A* SEARCH

6

The most widely known form of best-first search is called **A* search** (pronounced "A-star search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

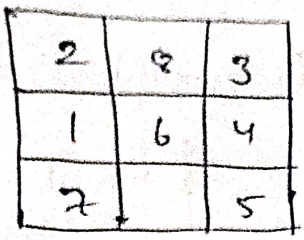
$$f(n) = g(n) + h(n) .$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

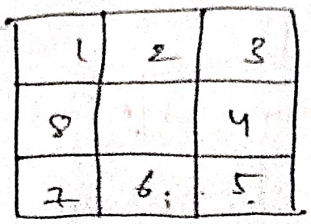
$$f(n) = \text{ estimated cost of the cheapest solution through } n .$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g.

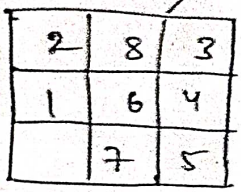
A* Algorithm: → $f(n) = g(n) + h(n)$



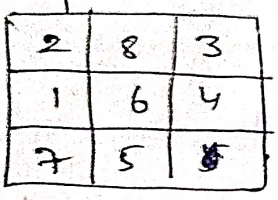
Initial state



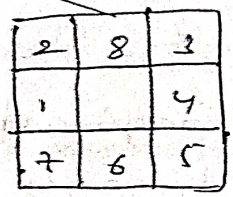
Goal state



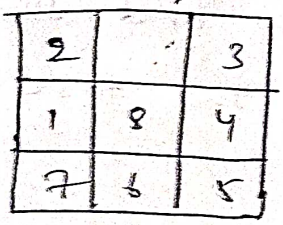
$g(n) = 1$
$h(n) = 5$
$f(n) = 6$



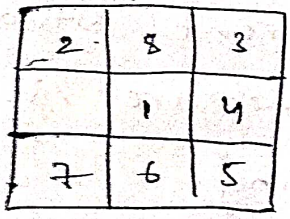
$g(n) = 1$
$h(n) = 5$
$f(n) = 6$



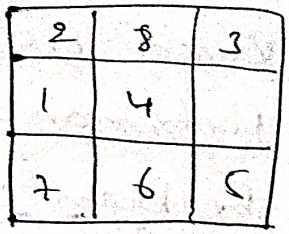
$g(n) = 1$
$h(n) = 3$
$f(n) = 4$



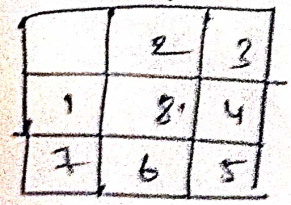
$g(n) = 2$
$h(n) = 3$
$f(n) = 5$



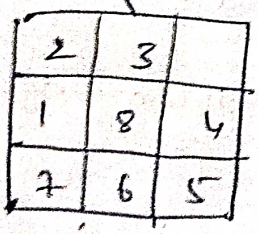
$g(n) = 2$
$h(n) = 3$
$f(n) = 5$



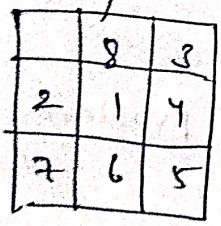
$g(n) = 2$
$h(n) = 4$
$f(n) = 6$



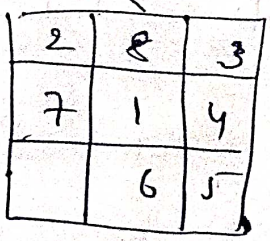
$g(n) = 3$
$h(n) = 2$
$f(n) = 5$



$g(n) = 3$
$h(n) = 4$
$f(n) = 7$

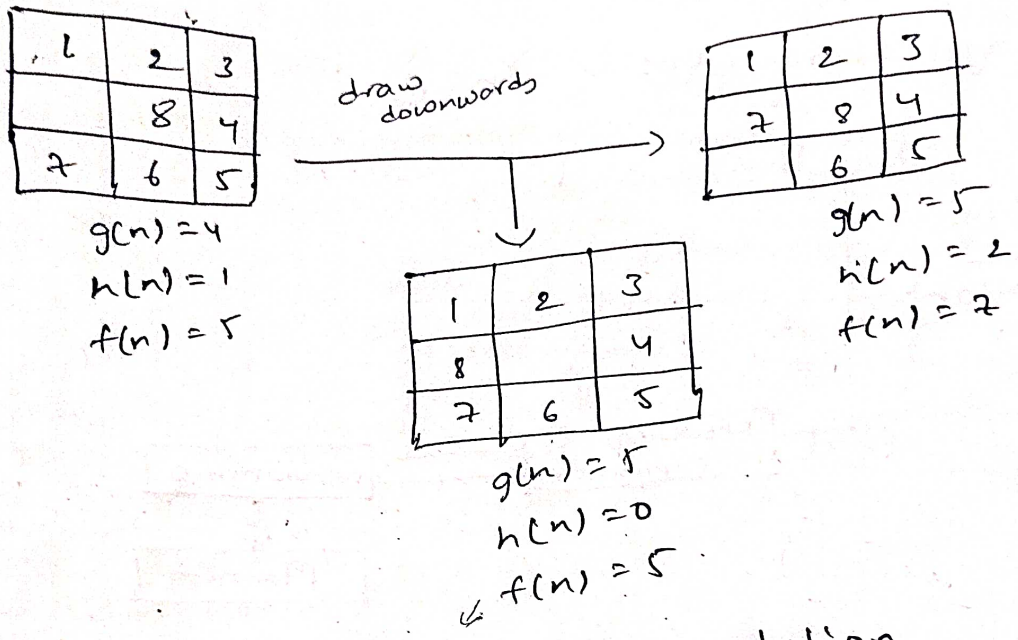


$g(n) = 3$
$h(n) = 3$
$f(n) = 6$



$g(n) = 3$
$h(n) = 4$
$f(n) = 7$

↓



∴ h(n) = 0 , this is the solution

→ A* algorithm maintains a tree of paths starting from initial state.

→ It extends those paths 1 edge at a time.

→ It continues until final state is reached.

7a

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 to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

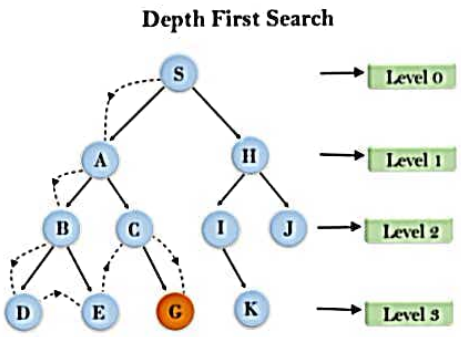
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

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 goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found 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 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 to the goal node.

There are several variations of the DFS algorithm, including:

1. Recursive DFS: This version of DFS uses recursion to traverse the graph. It starts at the root node and recursively explores each neighbor until it reaches a leaf node or a node that has already been visited. When it reaches a dead end, it backtracks to the previous node and explores other unvisited neighbors.
2. Iterative DFS: This version of DFS uses a stack data structure to keep track of the visited nodes and the order in which they are visited. It starts at the root node, pushes it onto the stack, and iteratively explores each neighbor until it reaches a dead end. When it reaches a dead end, it pops the last node from the stack and explores other unvisited neighbors.
3. Bidirectional DFS: This version of DFS explores the graph from both the start node and the goal node simultaneously until they meet at a common node. It is often used in conjunction with other search algorithms to speed up the search process.
4. Multi-start DFS: This version of DFS explores the graph from multiple starting nodes simultaneously. It can be used to speed up the search process and find multiple solutions to a problem.

7b

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.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of 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.

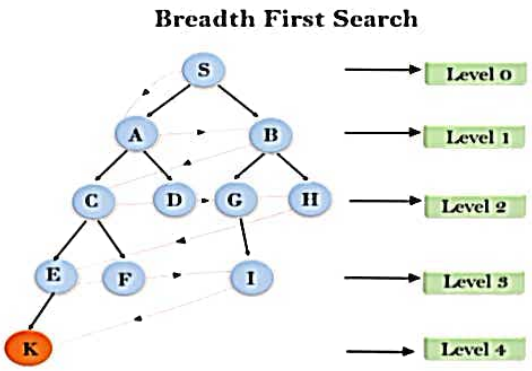
Disadvantages:

- 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.

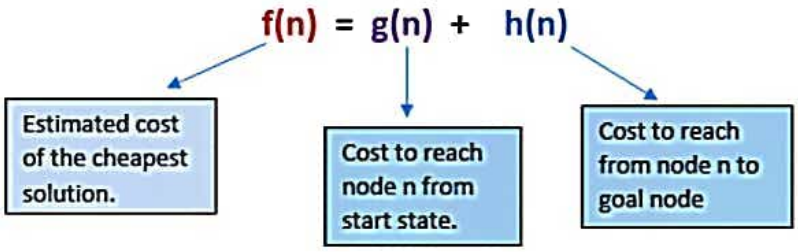
Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

In summary, BFS visits all the nodes at a given distance from the starting node before moving on to nodes at a greater distance, while DFS explores the graph by visiting all the nodes along a single path before backtracking and exploring other paths. BFS ensures that it finds the shortest path from the starting node to the goal node, while DFS may not find the shortest path and can get stuck in infinite loops if the graph is cyclic.

8 2.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and 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 solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* 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 following, and this sum is called as a **fitness number**.



At each point in the search space, only those node is 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 stops.

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 evaluation function for n' and place into 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.

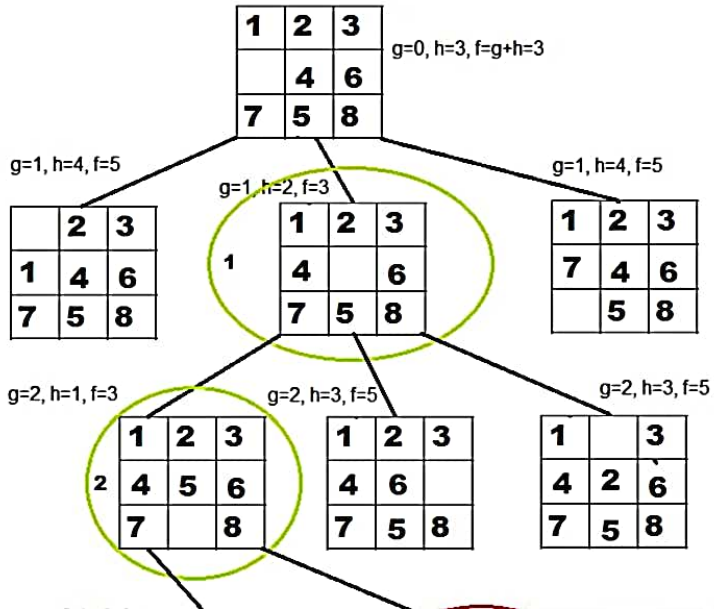
Step 6: Return to **Step 2**.

How A* solves the 8-Puzzle problem.

We first move the empty space in all the possible directions in the start state and calculate the **f-score** for each state. This is called expanding the current state.

After expanding the current state, it is pushed into the **closed** list and the newly generated states are pushed into the **open** list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.

This solves the issue of generating redundant child states, as the algorithm will expand the node with the least **f-score**.



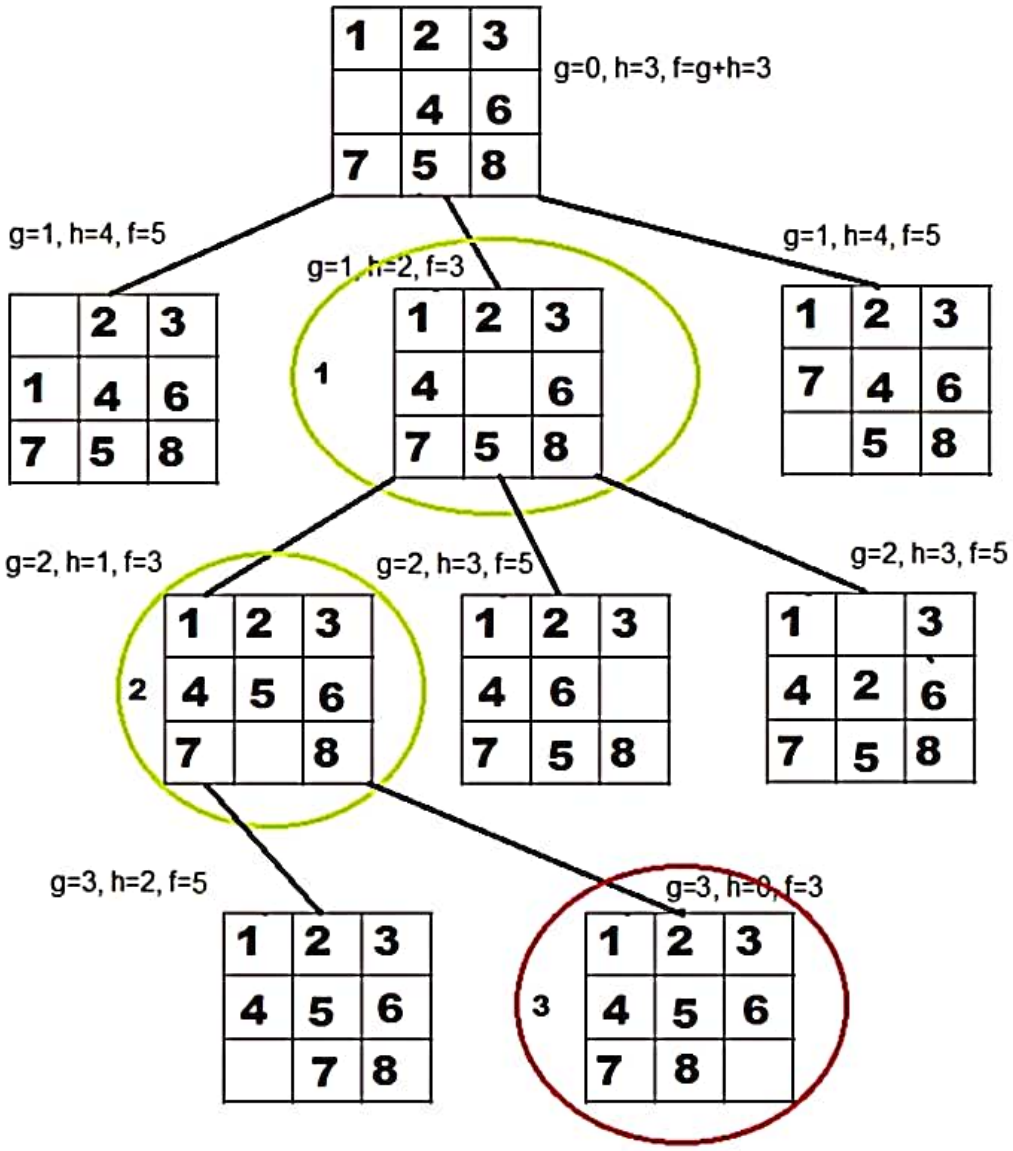


Fig 2. A* algorithm solves 8-puzzle

Disadvantage:

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

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 treat as it has no successor nodes further.

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

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

Advantages:

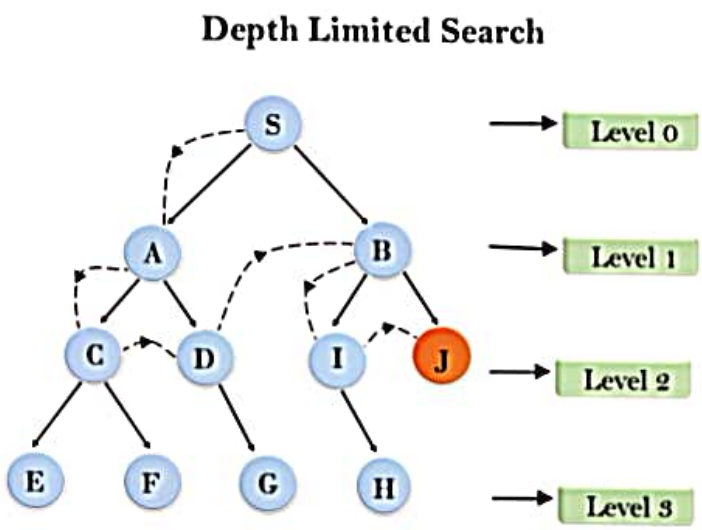
Depth-limited search is Memory efficient.

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 $\ell > d$.

10a) Uniform Search algorithms:

1) Breadth first search
2) Uniform cost search
3) deapth first search
4) Deapth limited Search
5) Iterative deeping depth first search
6) Bidirectional search.

→ Time complexity of BFS can be obtained by the no of nodes traversed in BFS until the shallowest Node. where d= depth of shallowest sol^n and b is a node at every state.

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

→ Let c^* is the cost of the optimal solution, and ε is each step to get closer to the goal node. Then the no of steps is $= c^*/\varepsilon + 1$. Here we have taken +1, as we start from state 0 and end to c^*/ε.

Hence worst case time complexity of Uniform cost search is $O(b^{1+[c^*/\varepsilon]})$.

→ Time complexity of DFS will be equivalent to the node traversed by the alg.

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

→ Time complexity of DLS is $O(b^l)$.

→ let b is branching factor and depth is d then the worst-case time complexity of IDDFS is

$O(b^d)$

→ Time complexity of bidirectional search is $O(b^d)$.

10b

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form 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

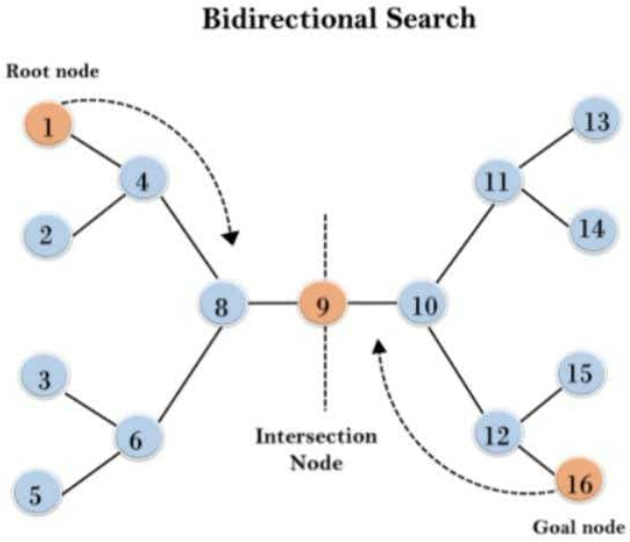
Disadvantages:

- 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, 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.