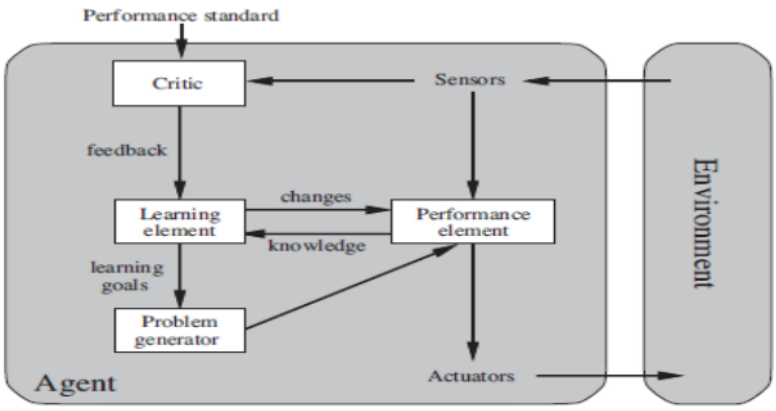
Learning agents have the ability to modify their behavior by drawing on past experiences. They utilize learning algorithms to enhance their performance progressively.



Multi-Agent Systems:

In situations with several intelligent agents, they have the option to work together or compete with each other in order to accomplish their own or shared objectives. Effective coordination and communication among agents is of utmost importance.

Intelligent agents play a crucial role in a wide range of AI applications, including as robotics, autonomous systems, recommendation systems, and others. The development and execution of intelligent agents need the integration of algorithms, models, and technologies that are customized to meet the unique needs of the application.

Module 2

Problem-solving by searching

Problem-solving by searching is a fundamental concept in artificial intelligence (AI) that involves finding a sequence of actions or steps to reach a goal state from an initial state. This process is modeled as a search problem, where the system explores possible states and actions in a systematic manner until a solution is found.

In artificial intelligence, a well-defined problem typically exhibits the following characteristics:

Clearly Stated Goal: The problem should have a clear and unambiguous goal or objective. This goal defines the desired outcome of the problem-solving process.

Initial State: There should be a defined starting point or initial state from which the problem-solving process begins. This is the situation or configuration at the outset.

State Space: The set of all possible states that the system can be in should be well-defined. Each state represents a specific configuration or situation.

Actions: A set of possible actions or operators must be identified. These actions represent the allowable moves or transitions between states.

Transition Model: A transition model describes how the system moves from one state to another based on the application of actions. It specifies the successor states for each action in a given state.

Constraints: Constraints define any limitations or restrictions on the problem. These may include resource limitations, legal moves, or other conditions that must be adhered to during the problem-solving process.

Optimality Criteria: The criteria for determining the optimality of a solution should be well-defined. In some cases, finding any solution may be sufficient, while in others, finding the best or optimal solution is crucial.

Solution Space: The set of possible solutions is well-defined. This encompasses all the potential outcomes that achieve the specified goal.

Objective Function or Evaluation Metric: An objective function or evaluation metric is used to quantify the quality of a solution. This metric guides the search for an optimal or satisfactory solution.

Feasibility: The problem should be feasible, meaning that there exists at least one sequence of actions that leads from the initial state to a state satisfying the goal.

A classic example of a well-defined problem in AI is the 8-puzzle, where the goal is to rearrange a set of numbered tiles in a 3x3 grid to achieve a specified configuration. The initial state, possible actions (move a tile), and the goal state are clearly defined, making it a well-structured problem for AI search algorithms.

2.1 Problem-solving agents

Search-The methodical analysis of states to determine a route from the initial or root state to the desired state is known as search.

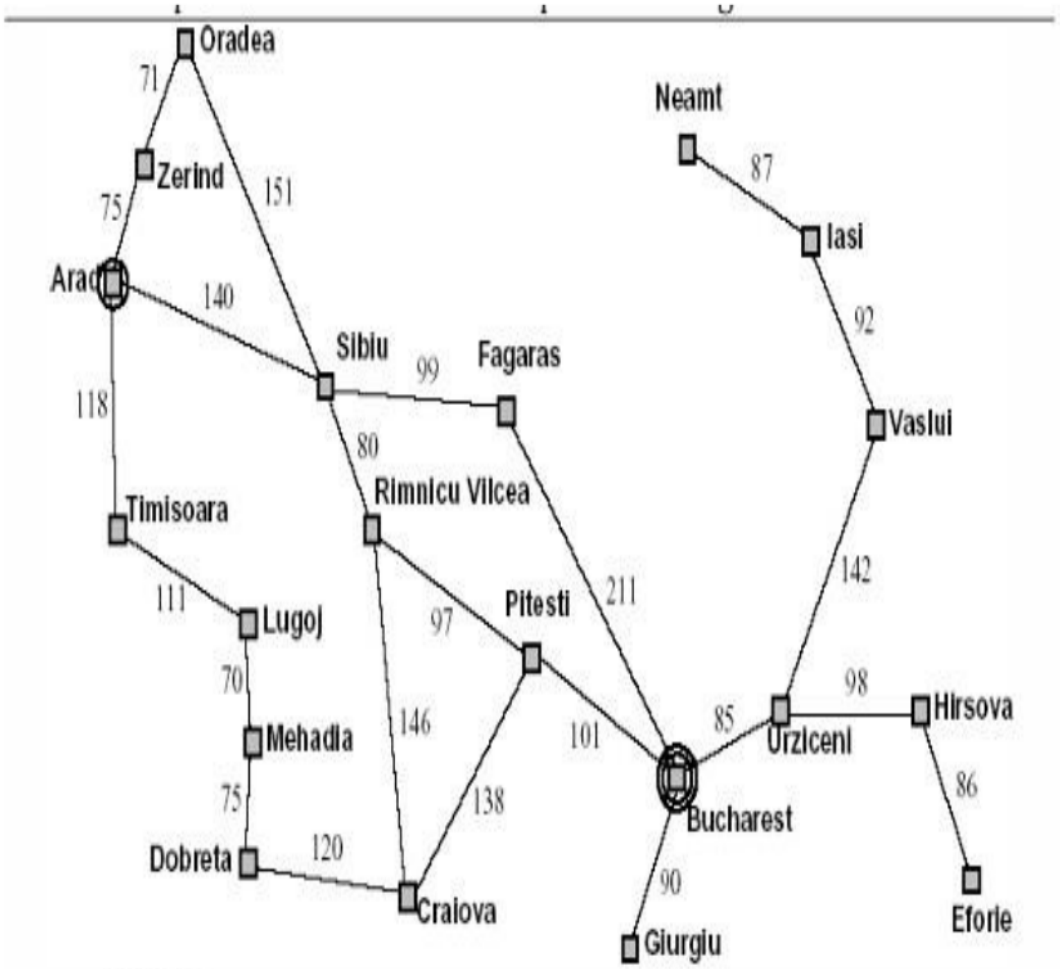
Problem-solving agents in artificial intelligence are computational entities designed to findsolutions to problems. These agents are a fundamental concept in AI and are often implemented using various problem-solving techniques. Problem-solving agents have a well-defined objectiveor goal they aim to achieve. This goal represents the desired outcome or solution to the problem.

Sensors allow the agent to perceive its environment. They provide information about the current state, enabling the agent to make informed decisions. Actuators are responsible for executing actions in the environment. These actions are the agent's means of influencing and changing the current state.

Goal Formulation: Setting goals based on the circumstance at hand and the agent's performance indicator is the first stage in solving an issue. Determining the path that will lead to a desired state is the agent's job**.**

Problem Formulation: The problem-solving agent translates the problem into a formal representation, including the definition of states, actions, and the goal.

The successor function for the Romania problem, for instance, would return { [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] } from the state In(Arad).



1 A simplified Road Map of part of Romania

Example: Route finding problem

Referring to figure 1.19

On holiday in Romania : currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal: be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem formulation

A **problem** is defined by four items:

initial state e.g., "at Arad"

successor function S(x) = set of action-state pairs

e.g., S(Arad) = {[Arad -> Zerind;Zerind],....}

goal test, can be

explicit, e.g., x = at Bucharest"

implicit, e.g., NoDirt(x)

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

c(x; a; y) is the step cost, assumed to be >= 0

A **solution** is a sequence of actions leading from the initial state to a goal state.

An agent with multiple immediate options of uncertain value can select the optimum course of action by first looking at various probable action sequences that lead to states of known value. Search is the process of figuring out the steps to take next from the present condition to get to the desired state.

The method of search receives an input in the form of an action series, which is the solution to the problem. The proposed course of action is carried out during the execution phase when a solution has been found.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
inputs : percept, a percept
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state  UPDATE-STATE(state, percept)
if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
action ← FIRST(seq);
seq ← REST(seq)
return action
```

State Space: The state space represents all possible configurations or situations the agent might encounter. It is crucial for modeling the problem and guiding the search for solutions.

Action Space: The action space consists of all possible actions that the agent can take to transition from one state to another. Actions are the agent's way of affecting the environment.

Search Algorithms: Problem-solving agents often use search algorithms to explore the state space and find a sequence of actions leading from the initial state to a goal state. Common search algorithms include Breadth-First Search, Depth-First Search, A*, and more.

Knowledge Representation: Agents may possess knowledge about the problem domain. This knowledge is often represented explicitly to guide decision-making.

Learning: Some problem-solving agents can learn from experience. Learning mechanisms allow agents to improve their performance over time based on feedback from the environment.

Heuristics: Heuristics are rules or strategies that aid in decision-making by providing shortcuts or estimates. They are often used in informed search algorithms to guide the exploration of the state space.

Optimization: In some cases, problem-solving agents are concerned with finding not just any solution but the optimal solution. Optimization algorithms aim to find the best possible outcome based on a defined objective function.

Problem-solving agents may need to adapt to changes in the environment or the problem itself. Adaptive agents can modify their strategies based on evolving conditions. Problem-solving agents are central to many AI applications, ranging from classic search problems to morecomplex real-world scenarios, such as robotics, game playing, and natural language understanding. The design and implementation of these agents often involve a combination of algorithms, representation methods, and domain-specific knowledge.

Well-defined problems

Well-defined problems in artificial intelligence (AI) are those for which the following elementsare clearly specified:

Initial State: The starting point of the problem, representing the current situation or configuration.

Goal State: The desired outcome or situation that the problem-solving process aims to achieve.

State Space: The set of all possible states the system can be in, including the initial and goal states.

Actions: A set of permissible actions or operations that can be applied to transition from one state to another.

Transition Model: A description of how the system moves from one state to another based onthe application of actions.

Constraints: Any restrictions or limitations on the problem-solving process, such as resource constraints or legal moves.

Optimality Criteria: Criteria for evaluating and determining the quality of a solution. This could involve finding the most efficient, cost-effective, or optimal solution.

Objective Function or Evaluation Metric: A function or metric used to quantify the quality ofa solution. It guides the search for an optimal or satisfactory solution.

Feasibility: The problem must be feasible, meaning there exists at least one sequence of actionsthat leads from the initial state to a state satisfying the goal.

Repeatability: The problem should be repeatable, meaning that the same initial state and set of actions consistently lead to the same results.

Examples of well-defined problems in AI include:

8-Puzzle: Given a 3x3 grid with numbered tiles, rearrange the tiles to reach a specified goal configuration.

Pathfinding: Find the shortest path from a starting point to a destination in a graph or grid.

Tic-Tac-Toe: Determine the optimal moves to win, lose, or draw in a game of Tic-Tac-Toe.

Traveling Salesman Problem: Determine the shortest possible route that visits a set of cities and returns to the starting city.

Sudoku: Fill in a 9x9 grid with numbers such that each row, column, and 3x3 subgrid contains all the digits from 1 to 9.

The Vacuum World is a classic example used in artificial intelligence to illustrate the concepts of problem-solving and search algorithms. In this scenario, an agent (usually a vacuum cleaner) is placed in a grid world with dirty and clean squares. The objective is to clean all the dirty squares efficiently. Let's define the problem and its solution:

Problem Definition:

State Space: The state space consists of all possible configurations of the environment, represented by the cleanliness status of each square in the grid.

Initial State: The initial state is the starting configuration of the environment, where the vacuum cleaner is placed in a specific square, and some squares may be dirty.

Actions: The agent (vacuum cleaner) can perform actions to move in a direction (up, down, left, right) and to clean the square it is currently in.

Transition Model: The transition model defines the result of applying an action in a given state, updating the cleanliness status and the agent's location.

Goal State: The goal state is a configuration where all squares are clean.

Objective Function or Evaluation Metric: The objective is to minimize the number of actions taken or time required to clean all dirty squares.

Solution: A solution to the Vacuum World problem involves finding a sequence of actions that, when applied starting from the initial state, leads to the goal state (a clean environment). Here's an example solution:

State 1:

Vacuum cleaner at (1, 1), some dirty squares.

Action: Clean.

State 2:

Vacuum cleaner at (1, 1), fewer dirty squares.

Action: Move right.

State 3:

Vacuum cleaner at (2, 1), some dirty squares.

Action: Clean.

State 4:

Vacuum cleaner at (2, 1), fewer dirty squares.

Action: Move right.

State 5:

Vacuum cleaner at (3, 1), clean squares.

Action: Move down.

State 6:

Vacuum cleaner at (3, 2), some dirty squares.

Action: Clean.

State 7:

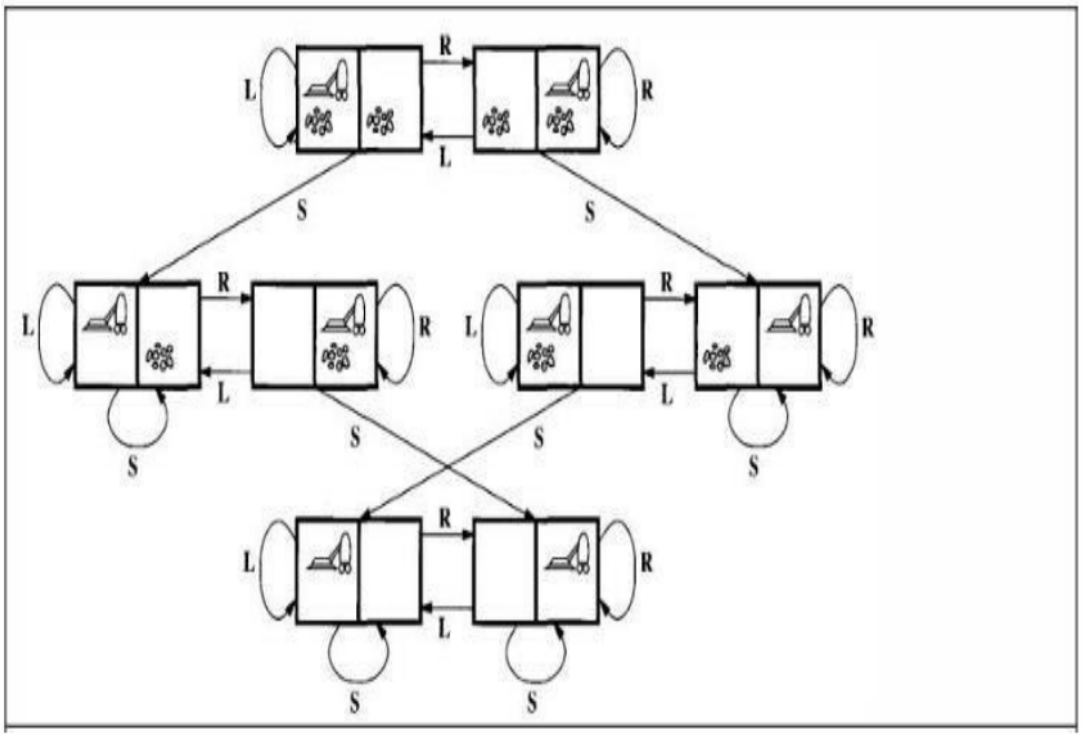
Vacuum cleaner at (3, 2), fewer dirty squares.

Action: Move left.

State 8:

Vacuum cleaner at (2, 2), clean squares.

Action: Move down.



2.2 Search algorithms

Search algorithms are fundamental in artificial intelligence for solving problems by systematically exploring the state space of a given problem. Several problems are there that need to be solved. This is accomplished by searching the state space. The initial state and thesuccessor function together define the state space and produce a search tree. When there are several ways to get to the same state, we typically have a search graph rather than a search tree.

A couple of search tree's expansions for determining a path from Arad to Bucharest are displayed in Figure.

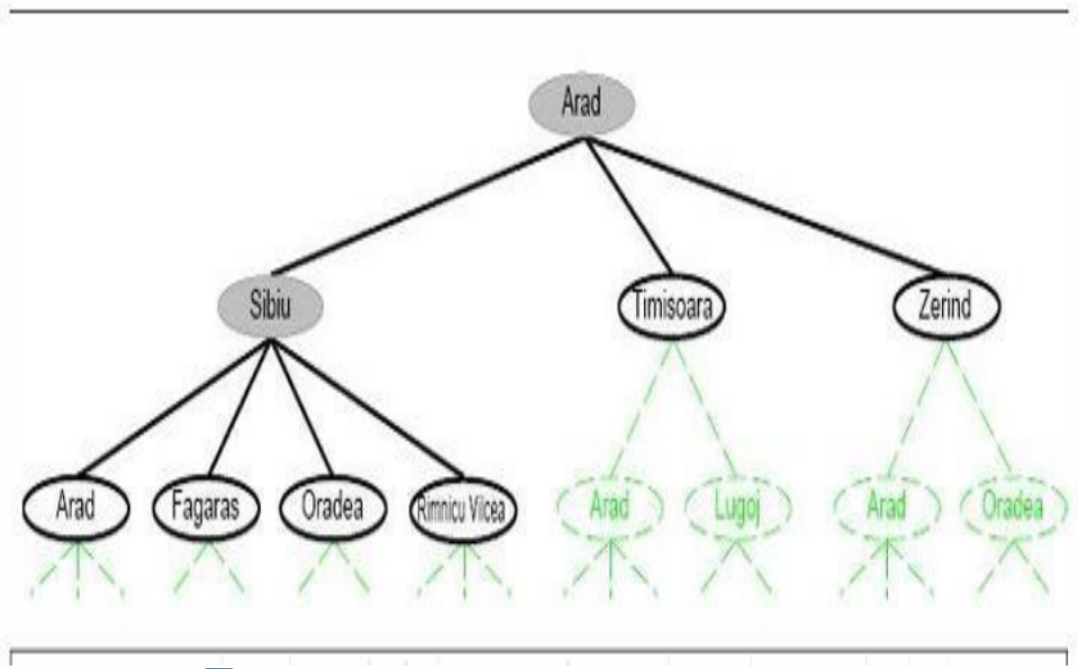


Fig. A partial tree search to locate a path from Arad to Bucharest. Expanded nodes are shaded, created but unexplored nodes are outlined in bold, and not-yet-generated nodes are displayed as a faint dashed line.

A search node that corresponds to the starting state, In(Arad), is the root of the search tree. Checking if this is a goal state is the first step. By applying the successor function to the present state, a new set of states is created by expanding the current state. Three new states are obtained in this instance: In(Sibiu), In(Timisoara), and In(Zerind). We now have to decide which of these three options to investigate further. The essence of searching is to pursue one alternative at a time, setting the others aside for a later time in case the initial option proves fruitless.

MEASURING PROBLEM-SOLVING PERFORMANCE

An algorithm used to solve problems can either produce a solution or fail. Some algorithms, however, have the potential to run indefinitely without producing any results at all.

There are four methods to assess the algorithm's performance:

- Completeness: Is there a guarantee that the method will locate a solution when one exists?

- Optimality: Does the plan produce the best outcome?
- Time complexity: What is the average time to solve a problem?
- Space complexity: What amount of Storage is required for the search to be conducted?

2.3 Uninformed search strategies

Uninformed search strategies, also known as blind search strategies, are algorithms used in artificial intelligence to explore the state space of a problem without using any domain-specific knowledge. These strategies do not take into account the goal or the specific structure of the problem but rather systematically explore the space until a solution is found. Here are some common uninformed search strategies:

Breadth-First Search (BFS):

- Explores the state space level by level, visiting all nodes at the current depth before moving on to the next depth.
- Guarantees finding the shallowest goal state but may require significant memory.

Depth-First Search (DFS):

- Explores the state space by going as deep as possible along one branch before backtracking.
- Uses less memory compared to BFS but may not find the shallowest goal state.

Depth-Limited Search:

- Limits the depth of exploration in DFS to prevent infinite loops.
- Useful when the state space is infinite or very large.

Iterative Deepening Depth-First Search (IDDFS):

- Repeatedly performs DFS with increasing depth limits until the goal is found.
- Combines advantages of both BFS and DFS.

Uniform Cost Search (UCS):

- Expands nodes based on the cost to reach them from the start node.
- Guarantees finding the optimal solution in terms of cost.

Bidirectional Search:

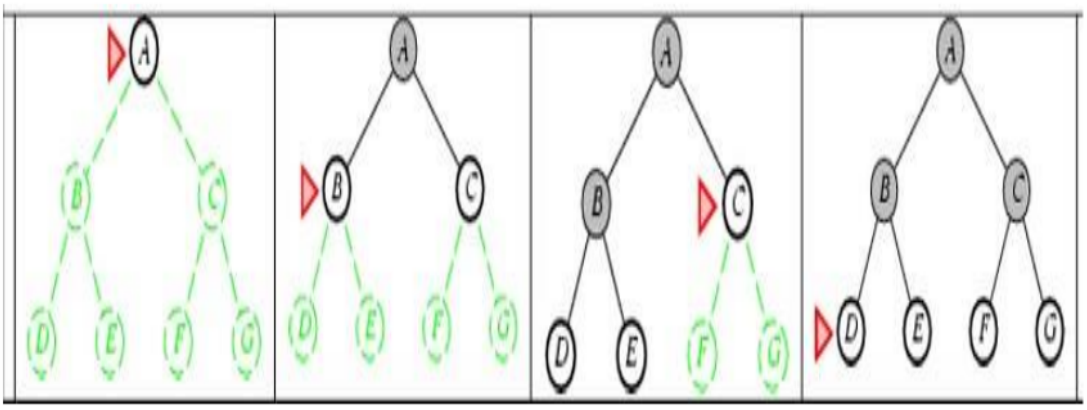
- Simultaneously explores from the start and goal states, with the goal of meeting in the middle.
- Can reduce the effective branching factor and lead to faster solutions.

Uninformed search strategies are suitable for problems where little or no information is available about the structure of the state space or the optimal solution. While these algorithms are simple and easy to implement, they may not be the most efficient for certain types of problems, especially those with large state spaces or complex structures. Informed search strategies, which use domain-specific knowledge, may be more effective in such cases.

Breadth-First Search (BFS)

Breadth-First Search (BFS) is an algorithm used for tree and graph traversal. It explores all the vertices at the current depth before moving on to the vertices at the next depth. BFS is often employed to find the shortest path between two nodes in an unweighted graph.

A straightforward approach known as "breadth-first search" expands the root node first, followed by all of its descendants, then those descendants, and so on. Generally speaking, nodes in the search tree extend to a certain depth before any nodes at a higher level do. The implementationof breath-first-search involves invoking TREE-SEARCH with an empty fringe, so creating a first-in-first-out (FIFO) queue that guarantees the expansion of the nodes that are visited first. Onthe other hand, breadth-first search is produced while executing TREE-SEARCH(problem,FIFO- QUEUE()). Shallow nodes increase before deeper nodes because the FIFO queue places all newly generated successors at the end of the queue.



Let's analyze the time complexity for an adjacency list representation of an undirected graph.

In BFS, each vertex and each edge of the graph are explored exactly once. The time complexityis often expressed in terms of the number of vertices (|V|) and the number of edges (|E|) in the graph.

For an adjacency list representation, the time complexity is O(|V| + |E|), where:

|V| is the number of vertices. |E| is the number of edges.

Visiting each vertex once: O(|V|) time as each vertex is processed once.

Exploring each edge once: O(|E|) time, as each edge is traversed once.

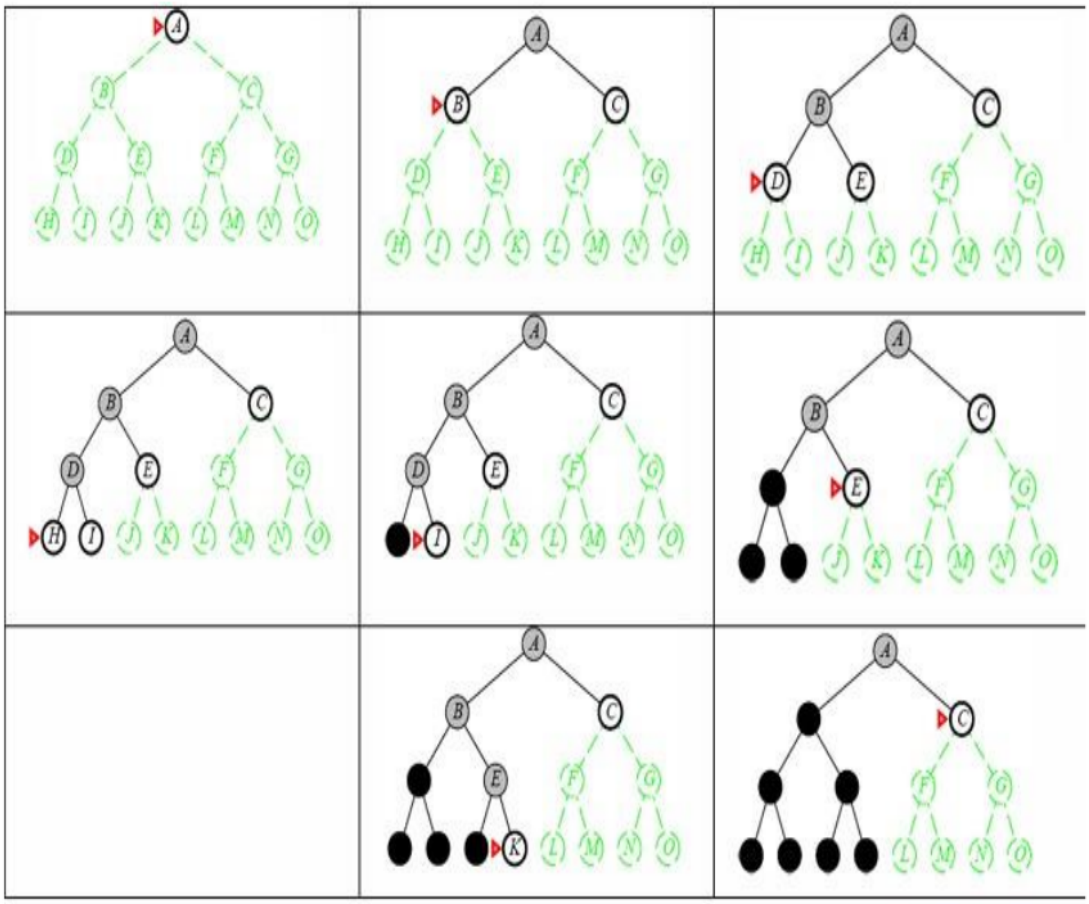
In the worst case, where every vertex and every edge must be explored, the overall timecomplexity is O(|V| + |E|).

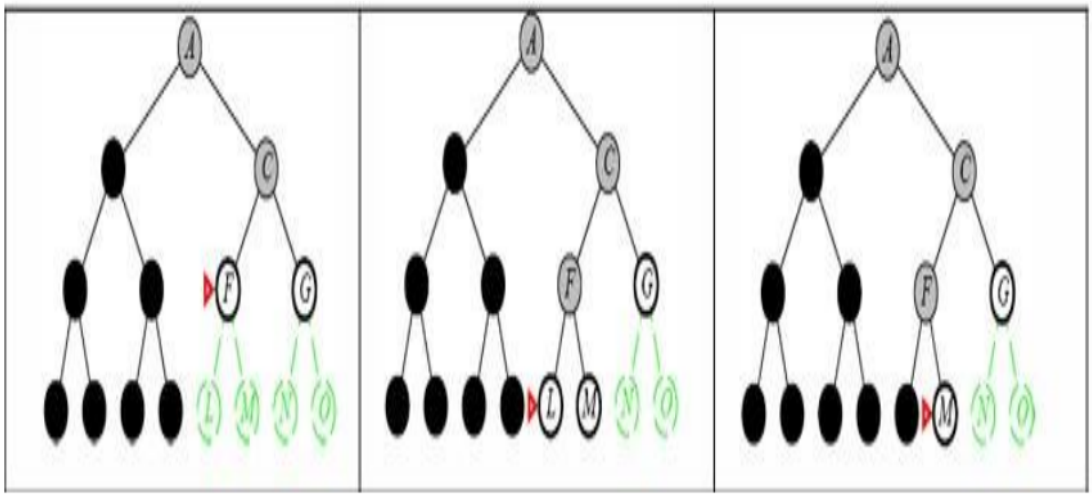
It's important to note that the space complexity of BFS, in addition to the time complexity, is influenced by the use of a queue to store vertices. The space complexity is also O(|V|) for the visited set and the queue.

Depth-first search

When using depth-first search, the deepest node on the search tree's current fringe is always expanded. Figure shows how the search is progressing. The search moves straight down the search tree to the lowest level, where there are no successors for the nodes. The next shallowest node with undiscovered successors is when the search "backs up" as those nodes grow and are removed from

the fringe.





This tactic can be applied by using TREE-SEARCH with a stack, or a last-in-first-out (LIFO) queue.

The memory requirements for depth-first-search are extremely low.For every node on the route,it only needs to keep the one path from the root to the leaf node as well as the remaining, unexpanded sibling nodes. When the node has grown, and its offspring have been thoroughly investigated, it can be deleted from memory.

In a state space where the maximum depth is m and the branching factor is b, depth-first search necessitates the storage of just bm + 1 nodes.

One limitation of depth-first search :

Depth-first search has the disadvantage of having the potential to make a bad decision and become stuck traveling down an extremely long (or even infinite) path when an alternative decision would result in a solution close to the search tree's root. Deep-first-search, for instance, will examine every node in the left subtree, even if node C is a goal node.

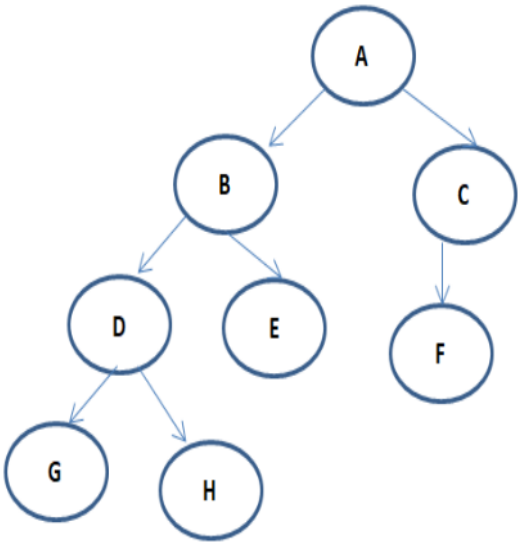
Depth-Limited Search

Depth-Limited Search (DLS) is a variation of depth-first search where the search is limited to a certain depth or level in the search tree. This limitation helps prevent infinite loops in cases where the tree is infinitely deep.

The basic idea of Depth-Limited Search is as follows

- Start at the root node.
- Explore each branch of the tree up to a certain depth limit.
- If the goal is not found within the specified depth, backtrack to the previous level.
- Continue the search, increasing the depth limit if needed.

Let's illustrate Depth-Limited Search with an example. Consider the following tree:



Let's say we want to find node F starting from node A with a depth limit of 2.

Depth 0: Start at A.

Depth 1: Explore A's children: B and C.

Go to B.

Explore B's children: D and E.

Depth 2: Go to D.

Explore D's children: G and H.

The depth limit is reached for D, so backtrack to B.

Go to E.

The depth limit is reached for E, so backtrack to A.

Backtrack to Depth 1:

Explore C's children: F.

Node F is found.

In this example, the Depth-Limited Search successfully found node F within the specified depth limit. If the depth limit was set to 1, the search would not have reached node F. The key is to limit the depth of the search to avoid infinite exploration and unnecessary backtracking.

Algorithm

```
function depthLimitedSearch(node, goal, depthLimit):if
    node is a goal:
        return true  // Goal found
    if depthLimit is 0:
        return false  // Reached depth limit, backtrack
    for each child of node:

        if depthLimitedSearch(child, goal, depthLimit - 1):
            return true // Goal found in child
    return false  // Goal not found at this level
```

Iterative Deepening Depth-First Search (IDDFS)

It is a combination of depth-first search and breadth-first search. It repeatedly performs depth-limited searches with increasing depth limits until the goal is found. This approach ensures that the algorithm explores progressively deeper levels of the search space, similar to breadth-first search, while maintaining the low memory requirements of depth-first search.

```
function iterativeDeepeningDFS(root, goal):
    depthLimit = 0
    while true:

        result = depthLimitedSearch(root, goal, depthLimit)
        if result == "found":
```

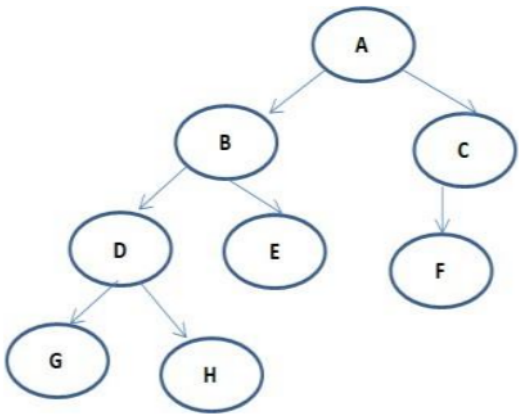
```
            return "Goal found"

        else if result == "not_found":

            depthLimit += 1 // Increase depth limit for the next iterationfunction
depthLimitedSearch(node, goal, depthLimit):
    if node is a goal:

        return "found"  // Goal found
    if depthLimit is 0:
        return "not_found" // Reached depth limit, backtrackfor
    each child of node:
        result = depthLimitedSearch(child, goal, depthLimit - 1)
        if result == "found":
            return "found"  // Goal found in child
    return "not_found"  // Goal not found at this level
```

let's consider an example using the same tree structure as before:



Let's say find node F using IDDFS:

Iteration 1 (depthLimit = 0):

Perform depth-limited search with depth limit 0. No nodes will be explored, and the result is "not_found."

Iteration 2 (depthLimit = 1):

Perform depth-limited search with depth limit 1.
Explore A, B, C. No goal is found at this depth.
Increase depth limit for the next iteration.

Iteration 3 (depthLimit = 2):

Perform depth-limited search with depth limit 2.
Explore A, B, D, E, C, F. Node F is found.
Return "Goal found."

IDDFS successfully found node F by gradually increasing the depth limit in each iteration until the goal was reached. The advantage of IDDFS is that it combines the efficiency of depth-first search with the completeness of breadth-first search.

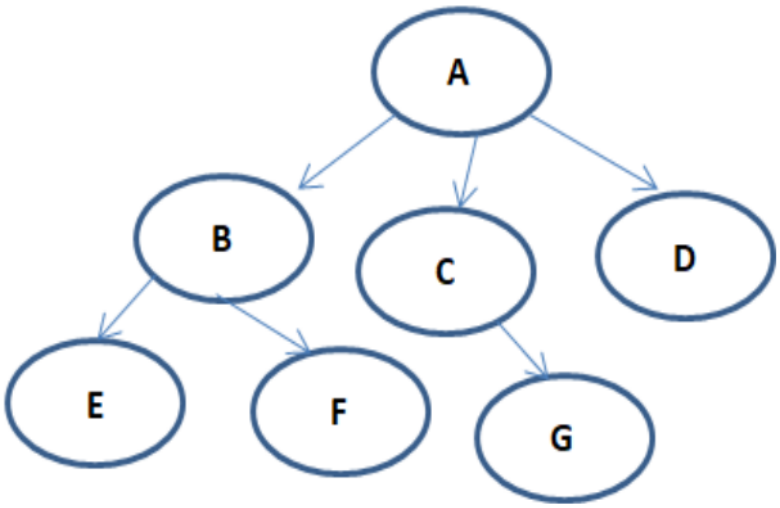
Uniform Cost Search (UCS)

It is a graph search algorithm that explores the graph by expanding the node with the lowest cost.It is typically used for finding the lowest-cost path in a weighted graph. UCS is an informed search algorithm and guarantees finding the optimal solution if all edge costs are non-negative.

The general idea of Uniform Cost Search:

- Start with the initial node and an empty priority queue.
- Add the initial node to the priority queue with a cost of 0.
- While the priority queue is not empty:
    - Pop the node with the lowest cost from the priority queue.
    - If the node is the goal, return the solution.
    - Otherwise, expand the node by generating its successors.
    - For each successor, calculate the cost to reach that successor and add it to the priority queue.
- If the priority queue becomes empty and the goal is not reached, then there is no solution.

Consider the following graph:



Let's assign some costs to the edges:

A to B: 3

A to C: 2

A to D: 5

B to E: 4B

to F: 7C to

G: 1

We want to find the lowest-cost path from node A to node G using UCS.

Initialize the priority queue with the initial node A and cost 0.

Iteration 1:

Pop A from the priority queue.

Expand A and generate B, C, and D.

Add B with cost 3, C with cost 2, and D with cost 5 to the priority queue.

Iteration 2:

Pop C from the priority queue (lowest cost).

Expand C and generate G.

Add G with cost 2+1=3 to the priority queue.

Iteration 3:

Pop B from the priority queue (lowest cost).

Expand B and generate E and F.

Add E with cost 3+4=7 and F with cost 3+7=10 to the priority queue.

Iteration 4:

Pop G from the priority queue (lowest cost).

Goal reached. The lowest-cost path from A to G is A -> C -> G with a cost of 3.

UCS successfully finds the lowest-cost path from node A to node G by considering the cumulative cost of each path.

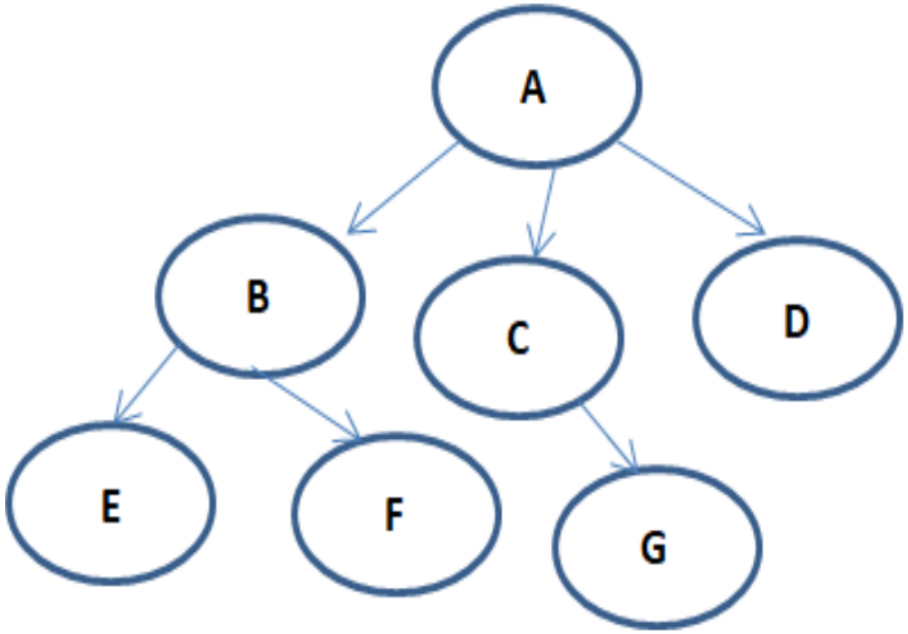
Bidirectional Search

It is an algorithm that simultaneously explores the search space from both the start and goal nodes, hoping to meet in the middle. It involves two simultaneous searches: one from the start node towards the goal, and the other from the goal node towards the start.

The general idea of Bidirectional Search:

- Initialize two queues, one for the forward search from the start node and one for the backward search from the goal node.
- Expand nodes alternately from the two queues.
- Check if a node exists in both searches. If so, a path is found, and the searches meet in the middle.
- Continue until the searches meet or both queues are empty.

Consider the following graph:



We want to find the path from node A to node G using Bidirectional Search.Initialize two queues, one starting from A and the other starting from G.

Iteration 1:

Forward search expands A and generates B, C, and D.

Backward search expands G.

Iteration 2:

Forward search expands B, C, D.

Backward search expands G, C.

Iteration 3:

Forward search expands E, F (from B), C (from D).

Backward search expands G, C.

Iteration 4:

Forward search expands F, G (from C).

Backward search expands G.

Meet in the middle:

Both searches have now reached node G, and a path is found: A -> C -> G.

Bidirectional Search efficiently explores the graph from both the start and goal nodes, reducing the overall search space. This can be particularly useful in scenarios where the search space is vast, and the goal is relatively close to the start. The efficiency gain comes from exploring two smaller subproblems simultaneously rather than focusing on a single, potentially larger search space.

2.4 Informed search strategies

Informed search strategies, also known as heuristic search strategies, utilize domain-specific knowledge or heuristics to guide the search process efficiently towards the goal state. These strategies incorporate additional information about the problem to make more informed decisions about which paths to explore. Here are some common informed search strategies:

Best-First Search:

Expands nodes based on a heuristic evaluation function that estimates the cost from the current state to the goal. Not necessarily optimal but can be more efficient than uninformed strategies.

A* Search:

Combines the benefits of both uniform cost search and best-first search by using both the actual cost to reach a node and a heuristic function. Guarantees finding the optimal solution if the heuristic is admissible (never overestimates the true cost).

Greedy Best-First Search:

Expands nodes based solely on the heuristic function without considering the cost to reach them. Quick but may not find the optimal solution.

Recursive Best-First Search (RBFS):

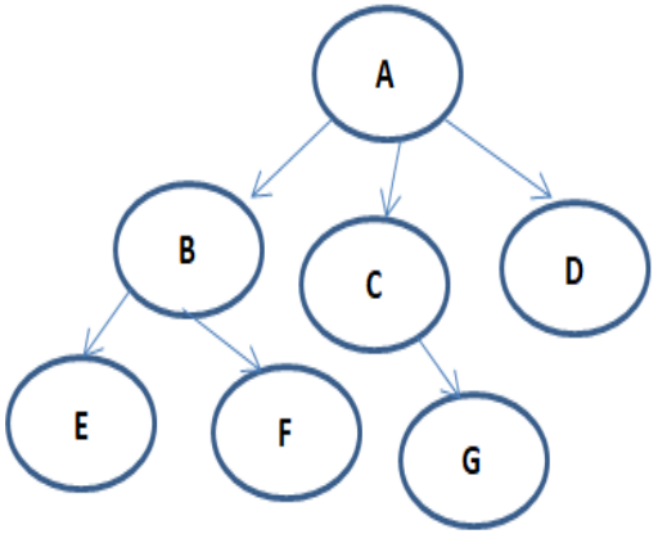
An enhancement of best-first search that uses iterative deepening to handle memory limitations.

Best-First Search is an informed search algorithm that uses an evaluation function to decide which node to expand next. The choice is based on an estimated cost-to-goal from the current node. Unlike Uniform Cost Search, Best-First Search doesn't necessarily consider the total cost of the path; instead, it focuses on the estimated cost from the current node to the goal.

Here's the general idea of Best-First Search:

1. Initialize a priority queue with the initial node.
2. While the priority queue is not empty:
3. Dequeue the node with the highest priority (based on the evaluation function).
4. If the node is the goal, return the solution.
5. Otherwise, expand the node by generating its successors.
6. Calculate the evaluation function value for each successor and enqueue them.
7. If the priority queue becomes empty and the goal is not reached, then there is no solution.

Let's illustrate Best-First Search with an example. Consider the following graph:



Assume that the goal is to reach node G from node A, and we have the following heuristic values (estimated costs from each node to the goal):

A: 5

B: 3

C: 2

D: 7

E: 4

F: 6

G: 0

Initialize the priority queue with the initial node A.

Iteration 1:

Dequeue A.

Expand A and generate B, C, D.

Enqueue C (priority 2), B (priority 3), D (priority 7).

Iteration 2:

Dequeue C.

C is the goal. Return the solution: A -> C.

Best-First Search prioritizes nodes based on the heuristic values, choosing the node that is estimated to be closer to the goal. It successfully finds the path from A to G by prioritizing nodeswith lower heuristic values during the search. The choice of heuristic is crucial in influencing thesearch direction and efficiency of the algorithm.

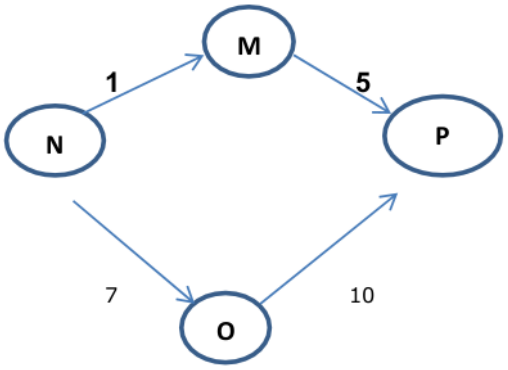
A* Search

A* Search is a popular path finding algorithm used in computer science and artificial intelligence for finding the shortest path between two points on a graph. It's an informed search algorithm that uses a combination of the cost to reach a node (g), the estimated cost from the current nodeto the goal (h), and a heuristic function (f = g + h) to guide the search.

A* Search is comprehensive and ideal. If h(n) is a valid heuristic, then A* is the best option. The straight-line distance hSLD is a clear illustration of an acceptable heuristic. It can't be exaggerated.

If h(n) is an acceptable heuristic, that is, if h(n) never overestimates the cost to achieve the objective, then A* Search is ideal.

How Does the A* Algorithm Work?



Weighted Graph

Take a look at the above weighted graph, which shows nodes and their distances from one another. Suppose you have to go from point A to point D. Given that source A is where we started, this will have a starting heuristic value. Thus, the outcomes are

f(M) = g(M) + h(M)

f(M) = 0 + 6 = 6

Next, take the path to other neighboring vertices :

f(M-N) = 1 + 4

f(M-P) = 5 + 2

Now, compute the weights by following the path from these nodes to the destination:

f(M-N-O) = (1+ 7) + 0

f(M-P-O) = (5 + 10) + 0

You can see that node N provides the optimal route, thus you should follow it to go to your destination.

Advantages:

- A* is a complete algorithm, meaning it will always find a solution if one exists. As long as the search space is finite, A* is guaranteed to find the optimal path.
- When a good heuristic is used, A* can be very efficient in finding the shortest path. The heuristic guides the search towards the goal, reducing the number of nodes that need to be expanded and making the algorithm well-suited for large and complex search spaces.
- A* can be adapted to different scenarios by using different heuristic functions.

Heuristic function

A heuristic function is a function that estimates the cost from a given node to the goal. This estimation guides the search process by helping the algorithm prioritize nodes that are likely to lead to a shorter path. A good heuristic is admissible and consistent to ensure the optimality and efficiency of the A* algorithm and choosing a good heuristic function is important. The effectiveness of a heuristic function determines its quality. Greater problem information corresponds to longer processing times.

Here are some commonly used heuristic functions:

Manhattan Distance:

For grid-based pathfinding, the Manhattan distance heuristic is often used. It calculates the distance between two points only along grid lines, which is suitable for movement in four directions (up, down, left, right).

h(n) = |n.x - goal.x| + |n.y - goal.y|

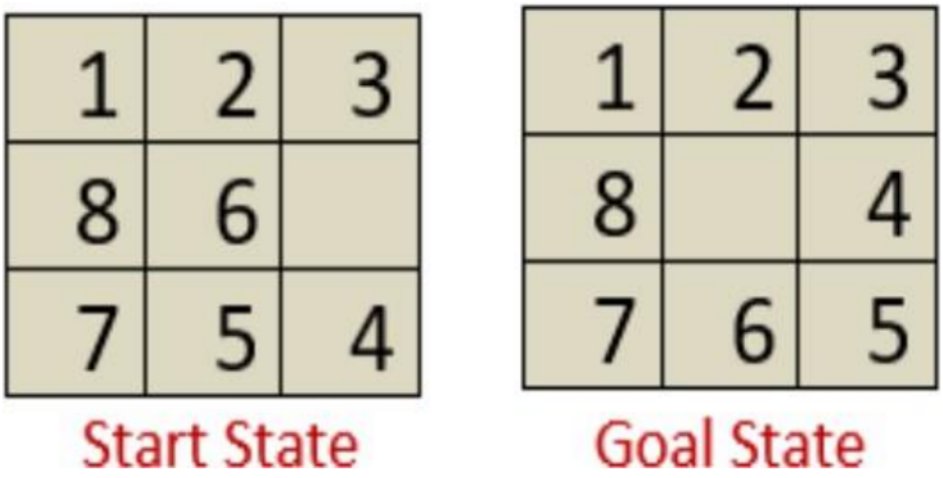
Euclidean Distance:

The Euclidean distance is a straight-line distance between two points in Euclidean space. It is suitable for continuous spaces and diagonal movement.

h(n) = sqrt((n.x - goal.x)^2 + (n.y - goal.y)^2)

Heuristic functions can be used to solve some toy problems more effectively, like tic tac toe, 8-puzzle, and 8-queen.

Consider the eight puzzles example, where we have a goal state and a start state. It is our responsibility to slide the tiles from the current/start state and arrange them in the goal state's prescribed arrangement. There are four possible movements: down, up, and left. The current/start state can be changed in a number of ways to reach the goal state, however a heuristic function h(n) can help us find a faster solution.



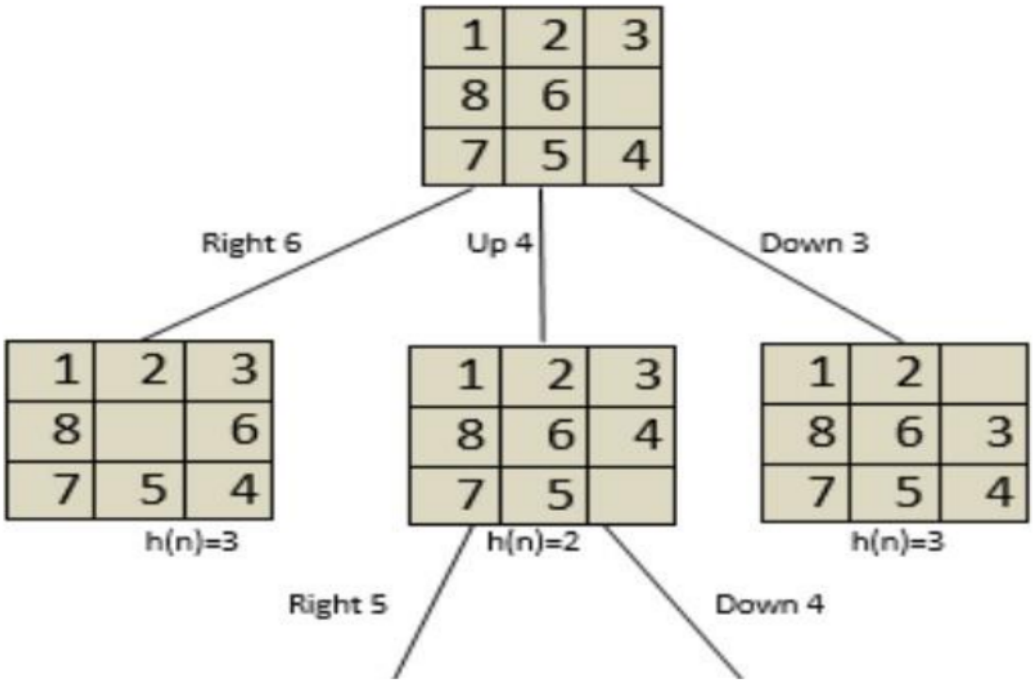
5

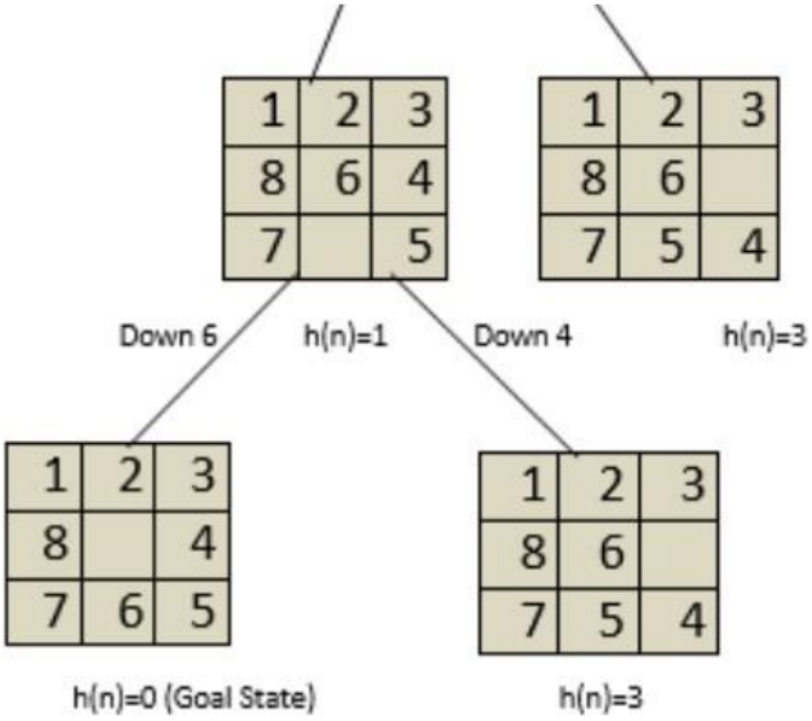
The following defines a heuristic function for the 8-puzzle problem:

h(n)=The quantity of misplaced tiles.

Thus, 6, 5, and 4 are the three tiles that are out of position in total. The empty tile that exists inthe goal state should not be counted. that is, h(n)=3. We must now reduce the value of h(n) = 0.

To reduce the h(n) value to zero, we can build a state-space tree as seen below:





The above state space tree shows that from h(n)=3 to h(n)=0, the objective state is minimized.

Characteristics of a Heuristic function

The following characteristics of a heuristic search algorithm result from using a heuristic function in the algorithm:

Admissible Condition: If an algorithm produces an optimal solution, it is considered admissible.

Completeness: If an algorithm ends with a solution (assuming there is one), it is considered complete.

Dominance Property: If h1 is superior to h2 for all values of node n, then A1 is said to dominate A2 if there exists two acceptable heuristic algorithms, A1 and A2, with h1 and h2 heuristic functions.

Optimality Property: An algorithm is the most effective and will undoubtedly produce an optimal

solution if it is complete, admissible, and dominates alternative algorithms.