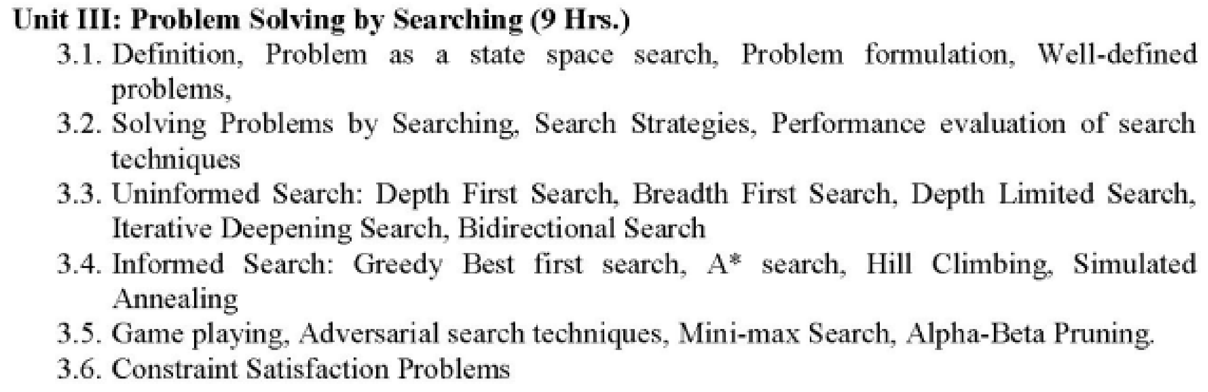
UNIT-3: Problem solving by Searching (9 Hrs)

Syllabus



Unit III: Problem Solving by Searching (9 Hrs.)

3.1. Definition, Problem as a state space search, Problem formulation, Well-defined problems,
3.2. Solving Problems by Searching, Search Strategies, Performance evaluation of search techniques
3.3. Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Bidirectional Search
3.4. Informed Search: Greedy Best first search, A* search, Hill Climbing, Simulated Annealing
3.5. Game playing, Adversarial search techniques, Mini-max Search, Alpha-Beta Pruning.
3.6. Constraint Satisfaction Problems

Definition

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.

The general concept of 'searching' is about looking for something. In computer science, searching techniques are strategies that look for solutions to a problem in a search space. The solutions or 'goal states' could sometimes be an object, a goal, a sub-goal or a path to the searched item.

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

A simple problem-solving agent is design in such a way that it has a simple **"formulate, search, execute"** design. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

In many situations, "to solve a problem can be described as to change the current situation, step by step, from an initial state to a final state. If each state is represented by a node and each possible change is represented by a link, then a problem can be represented as a graph (the **"state space"**), with a **"solution"** corresponding to a path from the initial state to a final state.

In this way, a solution consists of a sequence of operations, each of which changes one state into another one, and the whole sequence changes the initial state into a final state in multiple steps.

Four general steps in problem solving (Steps performed by Problem-solving agent)

1. **Goal formulation**
   A goal is a state that the agent is trying to reach. Goal formulation is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure.

2. **Problem formulations:**
   This is the process of deciding what actions and states to consider for given goal. It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
   1) **Initial State:** It is the starting state or initial step of the agent towards its goal.
   2) **Actions:** It is the description of the possible actions available to the agent.
   3) **Transition Model:** It describes what each action does.
   4) **Goal Test:** It determines if the given state is a goal state.
   5) **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, an optimal solution has the lowest path cost among all the solutions.

3. **Search Method**
   Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
4. **Execute**
   Once the solution is found, the actions it recommends can be carried out.

Problem as a state space search, Problem formulation, Well-defined problems,

Problem as a state space search

A state-space is defined as a set of all possible states of a problem. A state-space representation allows for the formal definition of a problem that makes the move from the initial state to the goal state.

State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

State space search process is used in the field of computer science, including artificial intelligence (AI), in which successive configurations or states of an instance are considered, with the goal of finding a goal state with a desired property.

In, state space search, a state space is formally represented as a tuple S
S: [S, A, Action(s), Result (s, a), Cost (s, a)]

in which:

- S: is the set of all possible states;
- A: is the set of possible actions, not related to a particular state but regarding all the state space;
- Action(s): is the function that establish which action is possible to perform in a certain state;
- Result(s,a): is the function that returns the state reached performing action 'a' in state 's'
- Cost(s,a) is the cost of performing an action 'a' in state 's'.

<u>Example</u>

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. Now, suppose the agent has a non-refundable 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.

Goal: Be in Bucharest
Formulate problem
States: various cities
Actions: drive between cities
Solution: Appropriate sequence of cities **e.g.:** Arad, Sibiu, Fagaras, Buchares

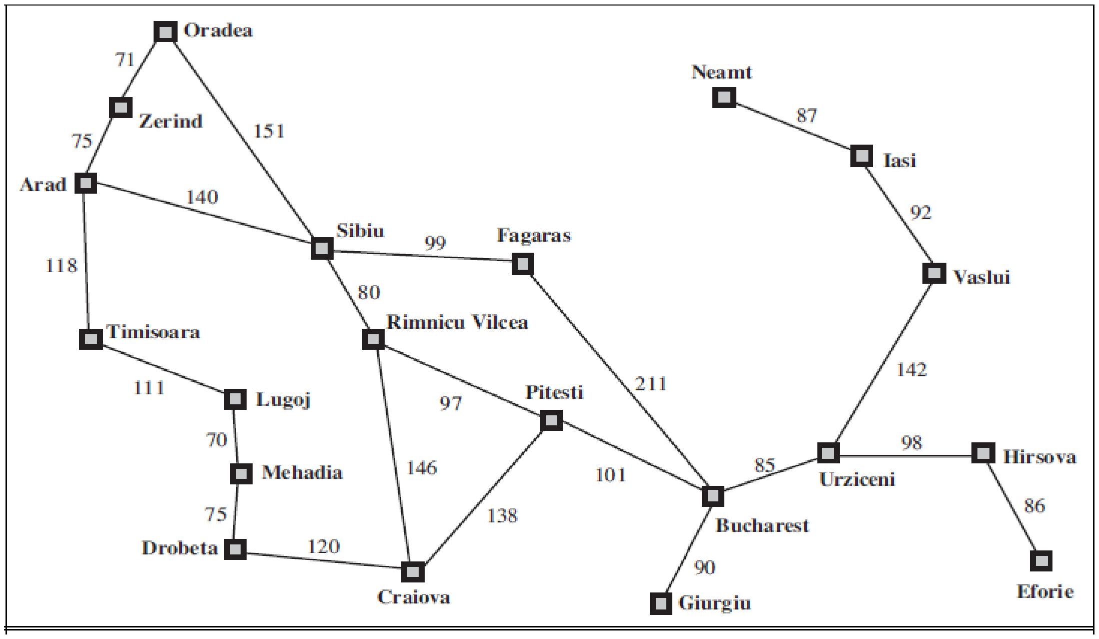


Figure: A simplified road map of part of Romania.

A simple problem-solving agent, it first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Problem Formulation

Problem formulation means choosing a relevant set of states to consider and a feasible set of operators for moving from one state to another. So given a problem we want to formulate the problem in terms of a set of states and a set of operators on actions.

A problem can be defined formally or formulated by five components:

1) **initial state: -**

The **initial state** is the agent starts in. It is a description of starting configuration of the agent. For example, the initial state for our agent in Romania might be described as In(Arad).

2) **Actions: -**

Action signifies the description of the possible **actions** available to the agent. An action takes an agent from one state to another state. By taking an action the agent moves from a current state to its successor state.

For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3) **Transition Model**

Transition Model signifies the description of what each action does; the formal name for this is the **transition model**, specified by a function RESULT (s, a) that returns the state that results from doing action (a) in state (s). For example, we have

RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

4) **Goal test**

The **goal test**, which determines whether a given state is a goal state or not. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set {In(Bucharest)}.

5) **path cost**

A **path** cost function assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to reach Bucharest, time is of the essence, so the cost of a path might be its length in kilometres. The **step cost** of taking action (a) in state (s) to reach state (z) is denoted by **c (s, a, z).**

Problem formulation involves deciding what actions and states to consider, given the goal.

Example: -

The vacuum-cleaner world shown in Figure below.

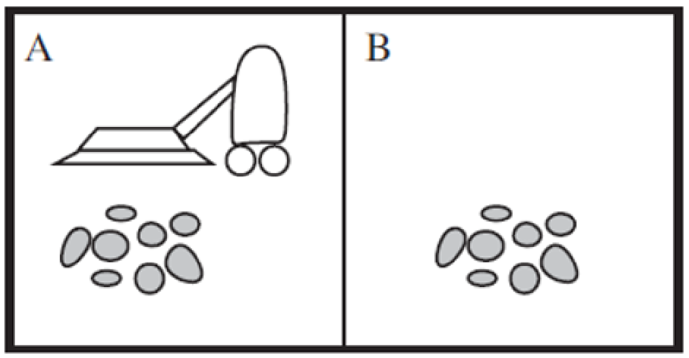


Figure: vacuum-cleaner world with just two locations.

The vacuum-cleaner world consists of just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square or not. It can choose to move left, move right, suck up the dirt, or do nothing. The function of vacuum world agent is, if the current square is dirty, then suck; otherwise, move to the other square. The function of this agent is shown in following figure ‘A’ as follows below.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Figure A. Partial tabulation of a simple agent function for the vacuum-cleaner world

The **vacuum world** can be formulated as a problem as follows:

States:

The state is determined by both the agent location and the dirt locations. The environment with n locations has **($n * 2^n$)** states. Thus, in vacuum world there are $2 \times 2^2 = 8$ possible world states.

1) **Initial state**:
   Any state can be designated as the initial state.
2) **Actions**:
   In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
3) **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 below.
4) **Goal test**:
   This checks whether all the squares are clean.
5) **Path cost**:
   Each step costs 1, so the path cost is the number of steps in the path.

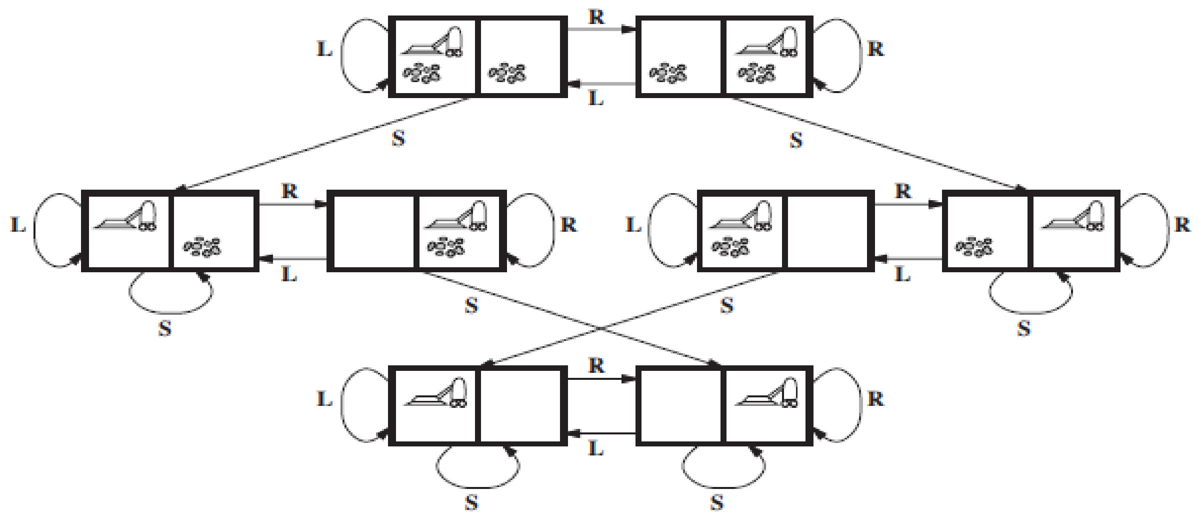


Figure :- The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Well-defined problems

The well-defined problems have specific goals, clearly defined solution paths, and clear expected solutions. Similarly, the ill-defined problems are those that do not have clear goals, solution paths, or expected solution.

In the study of problem solving, any problem in which the initial state or starting position, the allowable operations, and the goal state are clearly specified, and a unique solution can be shown to exist then it is considered as well-defined problems.

A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

Some problems which are simple and well-defined are called well-structured problems and include a set number of possible solutions - solutions are either 100% right or 100% wrong. An example of a well-structured problem is a typical mathematical (2 + 2 = 4) question.

Some of the most popularly used problem solving with the help of artificial intelligence are:

- Travelling Salesman Problem.
- Tower of Hanoi Problem.
- Water-Jug Problem.
- N-Queen Problem.
- Chess.
- Sudoku.
- Crypt-arithmetic Problems.
- Magic Squares

Solving Problems by Searching

- Search is the systematic examination of states to find path from the start/root state to the goal state.
- Problem-solving is commonly known as the method to reach the desired goal or finding a solution to a given situation.
- Problem-solving refers to AI techniques, including various techniques such as forming efficient algorithms, heuristics, and performing root cause analysis etc.
- The searching algorithm helps us to search for solution of particular problem.

The process of solving a problem consists following steps.

- **Defining The Problem:**
  The definition of the problem contains the possible initial as well as final situations which should result in acceptable solution.
- **Analysing The Problem:**
  Analysing the problem and its requirement must be done as few features can have immense impact on the resulting solution.
- **Identification Of Solutions:**
  This phase generates reasonable amount of solutions to the given problem in a particular range.
- **Choosing a Solution:**
  From all the identified solutions, the best solution is chosen basis on the results produced by respective solutions.
- **Implementation:**
  After choosing the best solution, its implementation is done.

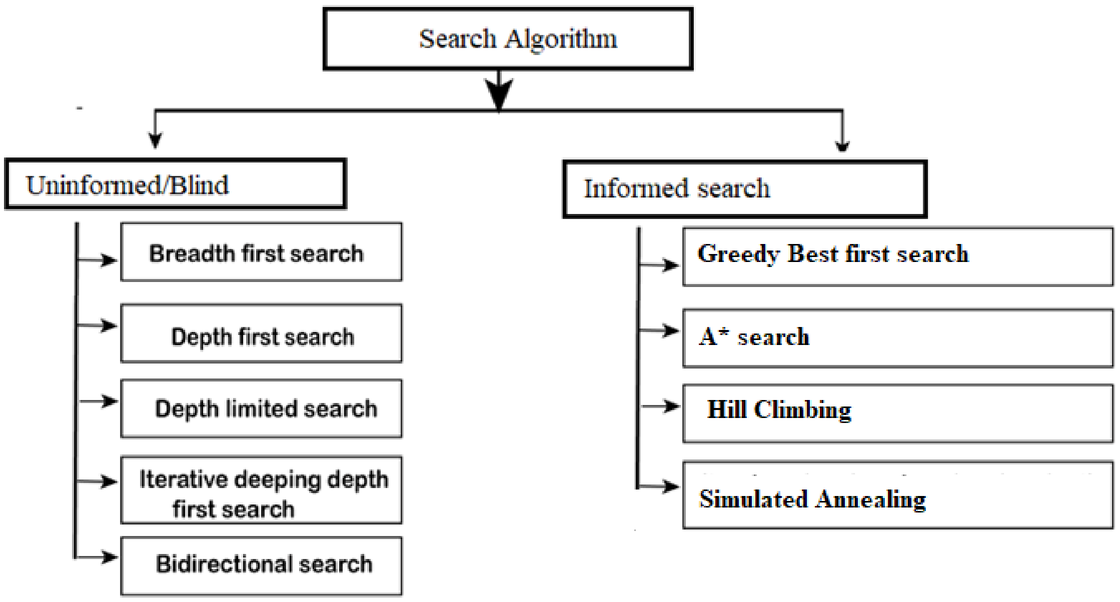
Search Strategies

A problem determines the graph and the goal but not which path to select from the frontier. This is the job of a search strategy. A search strategy specifies which paths are selected from the frontier. Different strategies are obtained by modifying how the selection of paths in the frontier is implemented. We have different types of search strategies like inform, uninform etc… we will see in detail later on…

Search strategies evaluate in terms of four criteria:

- **Completeness:** is the strategy guaranteed to find a solution when there is one?
- **Time complexity:** how long does it take to find a solution?
- **Space Complexity:** how much memory is required to perform the search?
- **Optimality:** does the search strategy find the highest quality solution when there are multiple solutions?

Types of search



Uninformed search (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 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.

Breadth-first search (BFS): -

Breadth-First Search algorithm is a graph traversing technique, where you select a random initial node (source or root node) and start traversing the graph layer-wise in such a way that all the nodes and their respective children nodes are visited and explored.

In BFS search strategy the root node is expanded first, then **all the successors** of the root node are expanded, and then their successors, and so on, until the goal node is found. All the nodes at a given depth in the search tree is expanded before a node in the next depth is expanded. Which is shown in figure below.

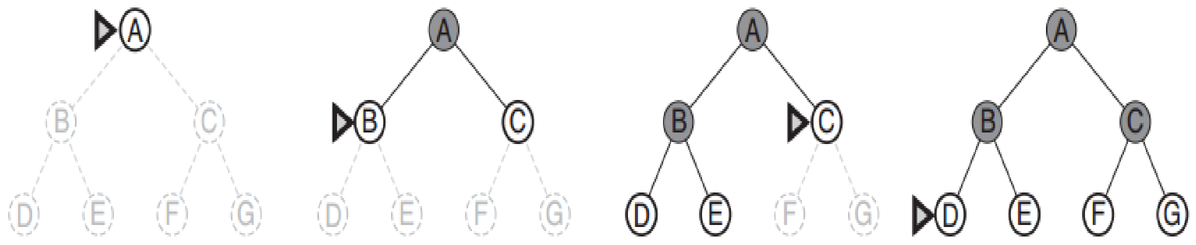
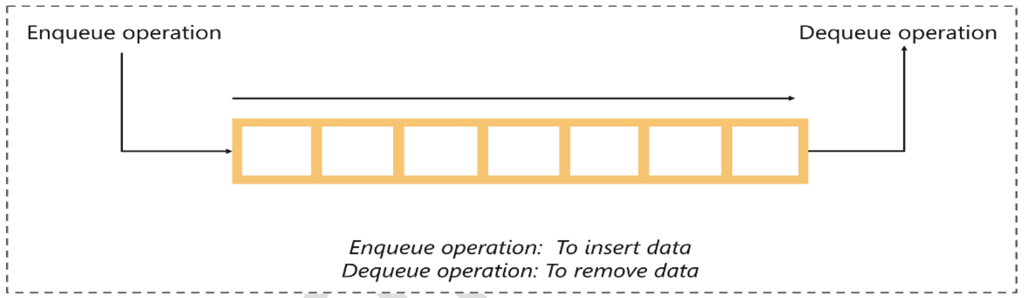
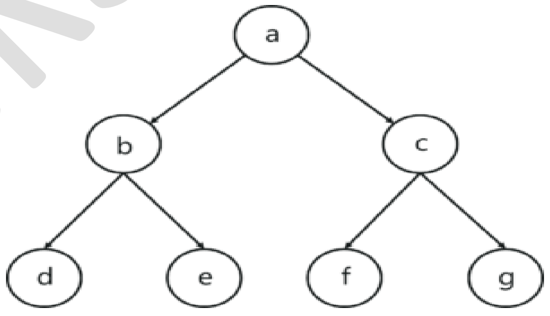


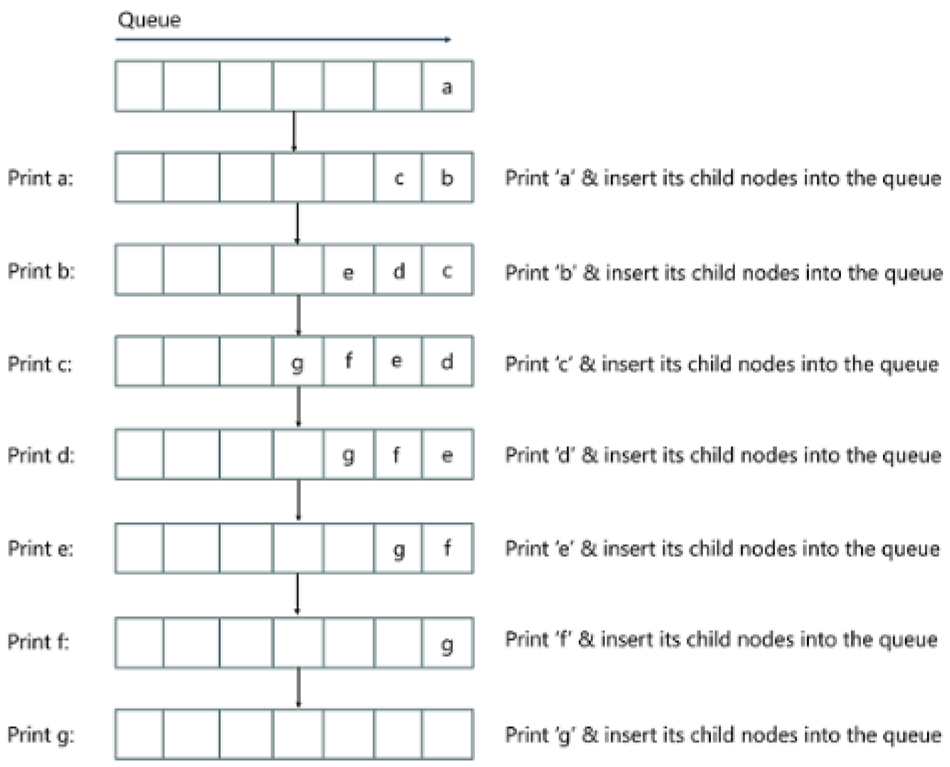
Figure: - The breadth-first search on a search tree.

Breadth-first search always expands the *shallowest unexpanded node*. To achieve this, we will take the help of a First-in First-out (FIFO) queue for the frontier. The newly generated nodes always go to the back of the queue, while the older nodes get expanded first. Which is shown in figure below.


Enqueue operation
Dequeue operation
Enqueue operation: To insert data
Dequeue operation: To remove data

<u>**Example: -**</u>




Queue
Print a:
Print b:
Print c:
Print d:
Print e:
Print f:
Print g:
a
c b
e d c
g f e d
g f e
g f
g
Print 'a' & insert its child nodes into the queue
Print 'b' & insert its child nodes into the queue
Print 'c' & insert its child nodes into the queue
Print 'd' & insert its child nodes into the queue
Print 'e' & insert its child nodes into the queue
Print 'f' & insert its child nodes into the queue
Print 'g' & insert its child nodes into the queue

⇨ Performance measures criteria for BFS algorithm

1) **BFS is Complete: -**
BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

2) **Optimality: -**
BFS is generally not optimal because it simply does not take costs into consideration when determining which node to replace on the fringe. The special case where BFS is guaranteed to be optimal, if all edge costs are equivalent, because this reduces BFS to a special case of uniform cost search.

3) **Time Complexity**
Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Let us assume all the non-goal nodes have *b* successors (*b* is the branching factor of the tree), and generating each successor from the parent node takes constant time. Then, the root of the search tree generates *b* nodes, and so it consumes *b* time. Then each of the *b* nodes in depth 1 further generates *b* nodes, consuming b^2 time, and so on, until the goal is reached in depth *d*. Hence, the **time complexity** is:

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

where *b is the branching factor and d is the depth*

4) <u>Space complexity: -</u>

Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$. for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$, figure below shows the Time and memory requirements for breadth-first search.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure: - The time and memory required for BFS.

From above Figure we conclude that

1) The memory requirements are a bigger problem for breadth-first search than the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take.

2) The time is still a major factor. If your problem has a solution at depth 16, it will take about 350 years for breadth-first search to find it.
In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

<u>Advantages:</u>

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

<u>Disadvantages:</u>

- 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.

Depth first Search (DFS)

Depth-first search is a 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 and it uses a stack data structure for its implementation.

DFS always expands the *deepest* node in the frontier of the search tree. The search proceeds to the deepest level of the search tree, which does not have any children yet (otherwise it is not the deepest node). The node is expanded, and then the successor is expanded, and this process is continued until the goal node is reached, or the node has no more successors. If the latter has occurred, the search *backs-up* to the previous node and explore its other successor, if any of them is still unexplored which is shown in figure(a) below.

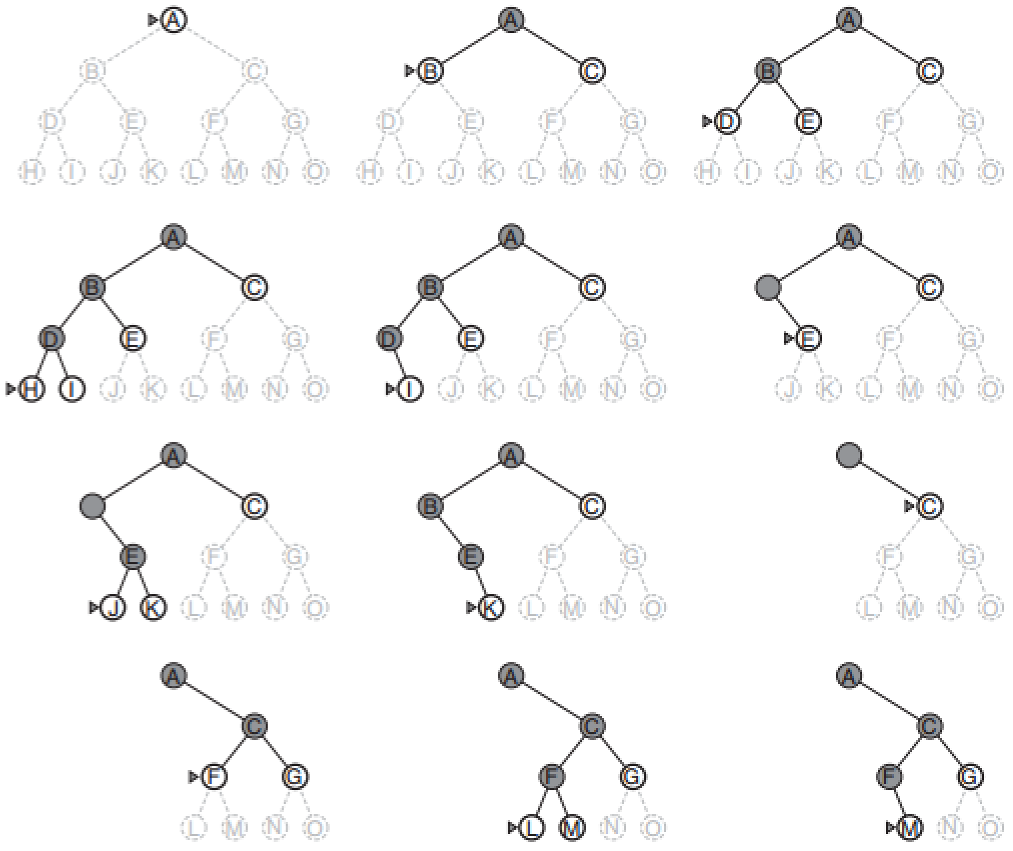


Figure a: Depth-first search on a search tree

In depth-first search, the order in which the nodes are expanded does not depend on the location of the goals which is shown in figure below.

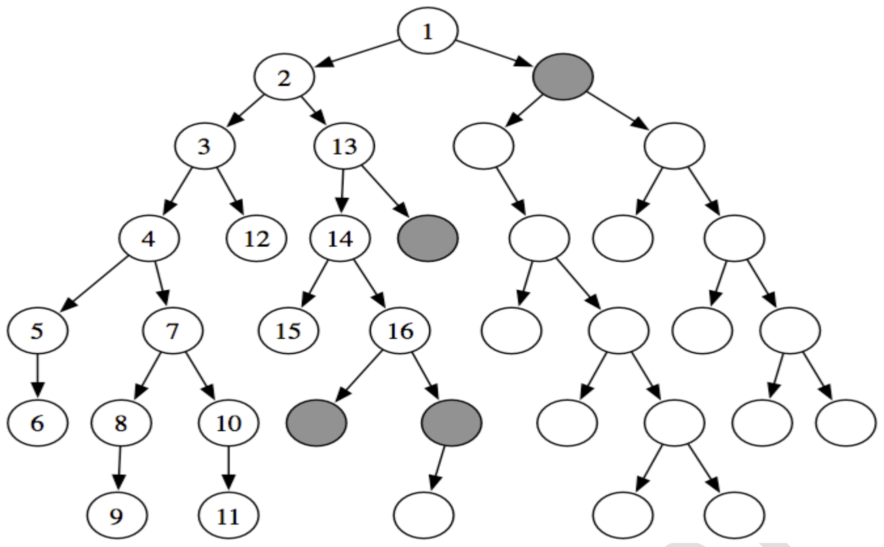


Figure b: The order nodes are expanded in depth-first search

As shown in figure b above, the start node is the root of the tree (the node at the top) and the nodes are ordered from left to right so that the leftmost neighbour is added to the stack last. The first sixteen nodes expanded are numbered in order of expansion and the shaded nodes are the nodes at the ends of the paths on the frontier after the first sixteen steps.
DFS uses a stack data structure for its implementation.

The properties of DFS:

1) Completeness:

DFS is not complete, consider that our search start expanding the left sub tree of the root for so long path (may be infinite) when different choice near the root could lead to a solution, now suppose that the left sub tree of the root has no solution, and it is unbounded, then the search will continue going deep infinitely, in this case we say that **DFS is not complete**.

2) Optimality:

Consider the scenario that there is more than one goal node, and our search decided to first expand the left sub tree of the root where there is a solution at a very deep level of this left sub tree, in the same time the right sub tree of the root has a solution near the root, here comes the non-optimality of DFS that it is not guaranteed that the first goal to find is the optimal one, so we conclude that **DFS is not optimal**.

3) Time complexity

Time taken by DFS depends on the depth of the entire search tree (which could be infinite, if loopy paths are not eliminated). Even in the case of a search tree with finite depth, the **time complexity** will be O(b^d), where b is the branching factor and d is the depth.

4) **Space complexity**

The main reason for the use of DFS over BFS in many areas of AI is because of its very less space consumption. In DFS, we need to store only the nodes which are present in the path from the root to the current node and their unexplored successors. For state space with branching factor *b* and maximum depth *d*, DFS has **space complexity** O(*bd*)

Advantage:

1) 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.
2) It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

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

For reference

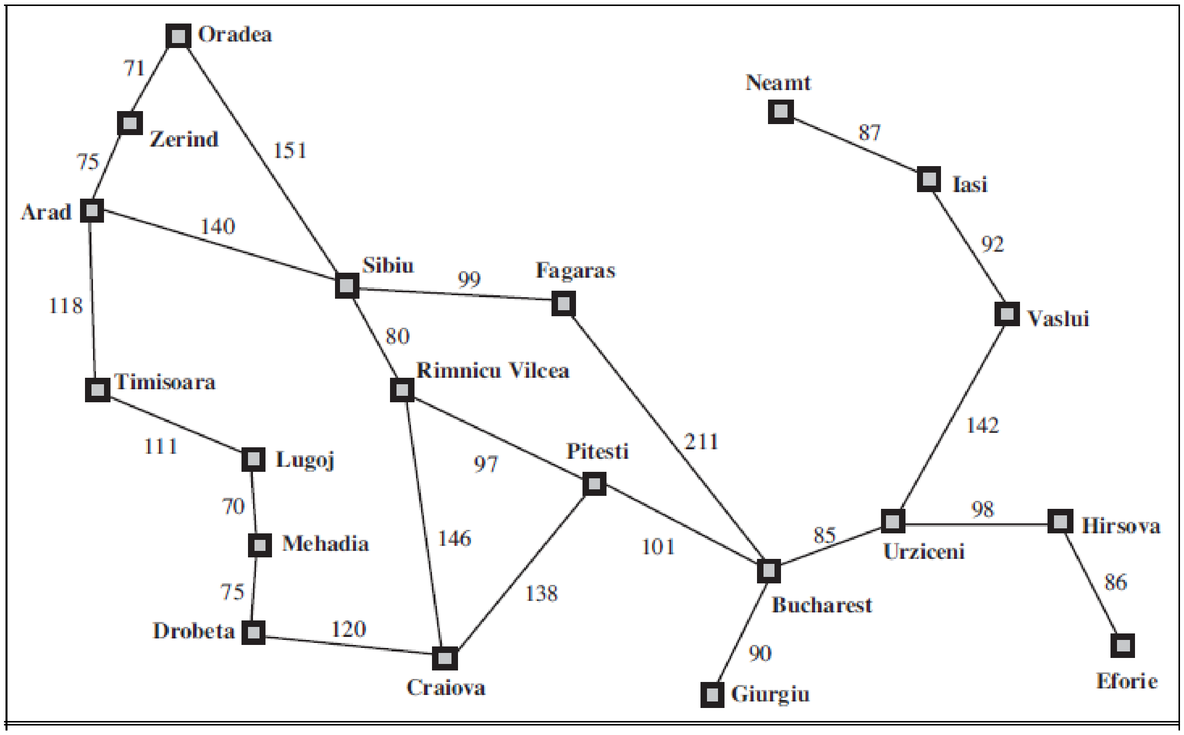


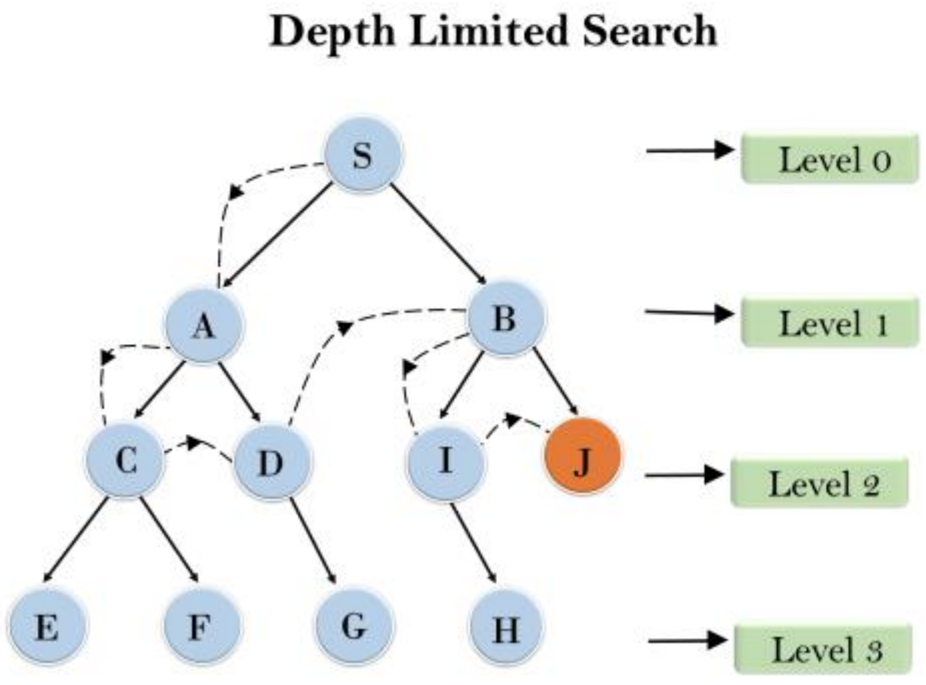
Figure: A simplified road map of part of Romania.

Depth Limit Search (DLS)

In depth first search, for infinite space tree we cannot reach the goal node, since it will continue to traverse down the branch infinitely. To overcome this problem depth-limited search has been introduced. In depth-limited search a limit has been added to the search so that it can get out of the traversing loop and reach the goal node.

Hence, 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.

Example:



1) **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.

2) **Time Complexity:** Time complexity of DLS algorithm is **O(b^{ℓ})**.

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

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

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

- Standard failure value: It indicates that problem does not have any solution.
- Cut-off 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.

Iterative deepening depth-first search (IDS)

Iterative deepening depth-first search is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. The Iterative deepening combines the benefits of depth-first and breadth-first search.

As **BFS** goes level by level, but requires more space. The space required by BFS is O(n) where n is number of nodes in tree and DFS first traverses' nodes going through one adjacent of root, then next adjacent. The problem with this approach is, if there is a node close to root, but not in first few subtrees explored by DFS, then DFS reaches that node very late and also, DFS may not find shortest path to a node which is shown in figure below.

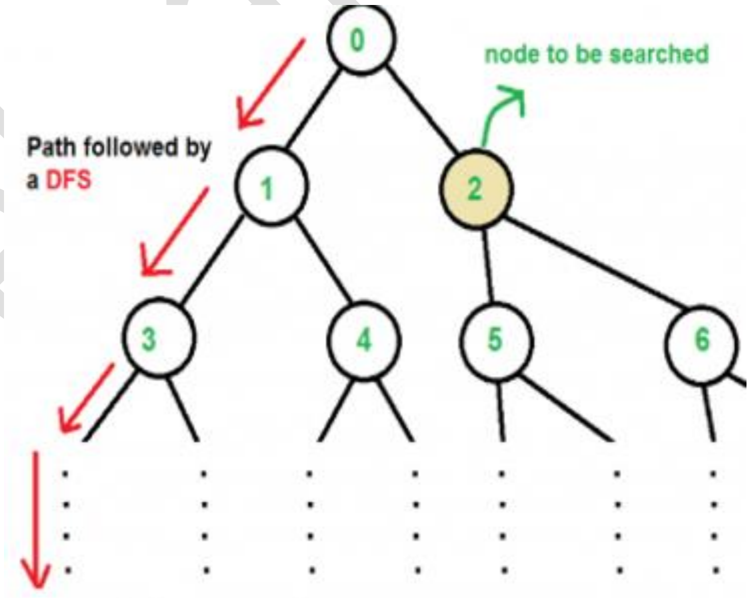


Figure: DFS

Thus, combining the advantages of BFS and DFS strategy, the IDS takes the completeness and optimality of BFS and the modest memory requirements of DFS.

IDS works by looking for the best search depth d, thus starting with depth limit 0 and make a BFS and if the search failed it increase the depth limit by 1 and try a BFS again with depth 1 and so on – first d = 0, then 1 then 2 and so on – until a depth d is reached where a goal is found.

Hence, the iterative deepening search algorithm repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.

<u>Performance Measure:</u>

1) Completeness:

IDS is like BFS, is complete when the branching factor b is finite.

2) Optimality:

IDS is also like BFS optimal when the steps are of the same cost.

3) Time Complexity:

Iterative deepening search may seem wasteful because states are generated multiple Times but actually it is not that costly compared to BFS, that is because most of the generated nodes are always in the deepest level reached,

In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level(depth d-1) are generated twice, the nodes at depth d-2 are generated 3 times and so on, until you reach depth 1 which is generated d times, we can view the total number of generated nodes in the worst case as:

$$\mathbf{N(IDS) = (b)d + (d - 1)b^2 + (d - 2)b^3 + \ldots + (2)b^{d-1} + (1)b^d = \mathit{O}(b^d)}$$

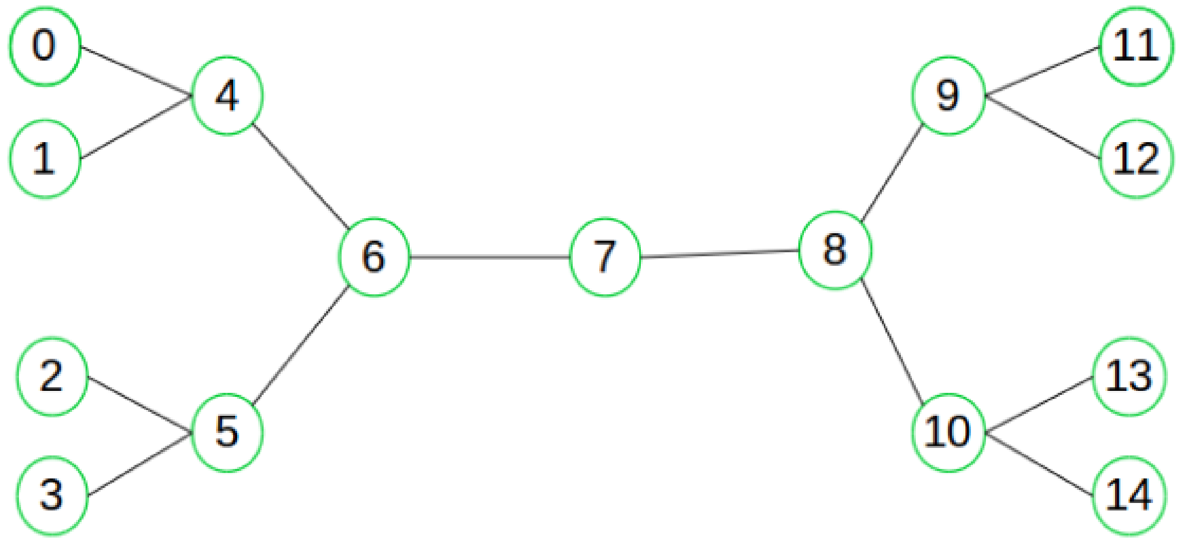
4) Space Complexity:

IDS is like DFS in its space complexity, taking *O*(bd) of memory.

<u>Bidirectional Search: -</u>

The idea of a bidirectional search is to reduce the search time by searching forward from the start and backward from the goal simultaneously. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start state through the frontier intersection to the goal.

Consider following simple example



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We can clearly see that we have successfully avoided unnecessary exploration.

A new problem arises during a bidirectional search, namely ensuring that the two search frontiers actually meet. For example, a depth-first search in both directions is not likely to work well because its small search frontiers are likely to pass each other by. Breadth-first search in both directions would be guaranteed to meet.

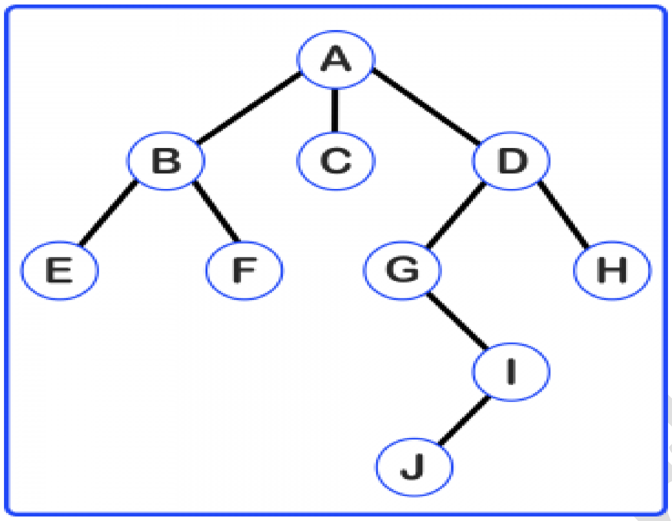
A combination of depth-first search in one direction and breadth-first search in the other would guarantee the required intersection of the search frontiers, but the choice of which to apply in which direction may be difficult. The decision depends on the cost of saving the breadth-first frontier and searching it to check when the depth-first method will intersect one of its elements.

In bi-directional search normally we use two simultaneous search techniques, which are:-

- Forward search from root node.
- Backward search from goal node.

EXAMPLE:

Consider a graph x, calculate the shortest path from A to J using bi-directional search, where length of each edge is given in the diagram.

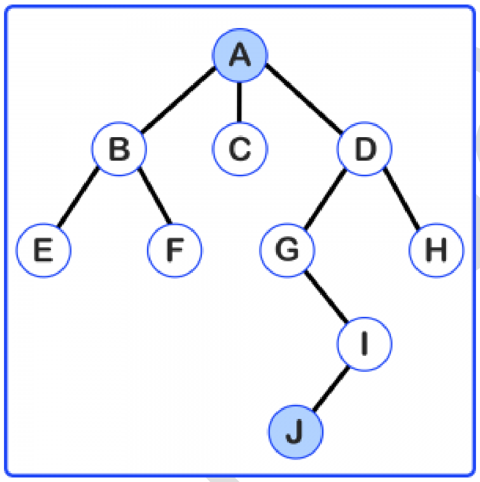


RULE:

Apply the Bidirectional Search Algorithm on The Given Search Tree. Here We Will Start Traversing from Root Node as Well as From the Goal Node Simultaneously.

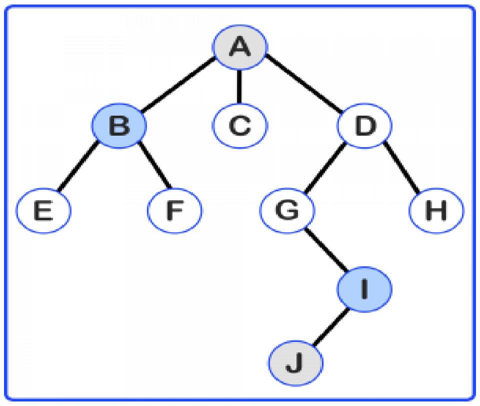
Step 1:

Initialization: {[A,0]} for forward search.
Initialization: {[J,0]} for backward search.

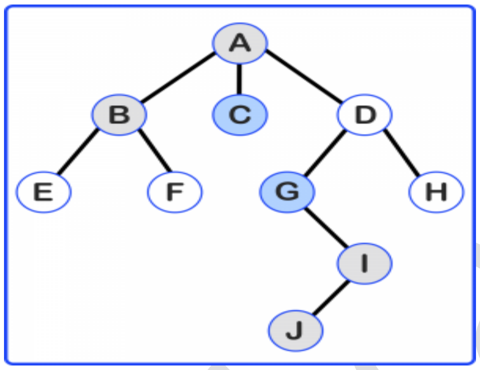


STEP 2:

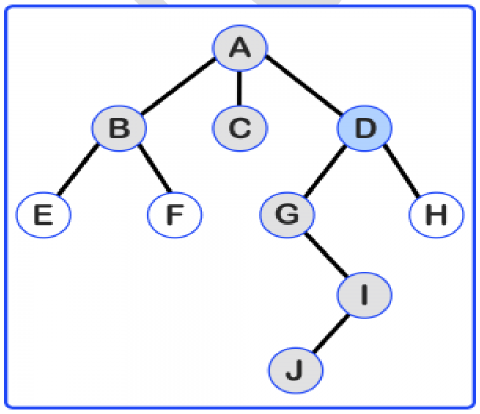
Applying BFS in forward search at level 1 we get : {[A->B]}
Applying DFS in Backward search at level 1 we get : {[J->I]}



STEP 3:
Applying BFS in forward search at level 2 we get : {[A->C]}
Applying DFS in Backward search at level 2 we get : {[I->G]}



STEP 4:
Applying BFS in forward search at level 3 we get : {[A->D]}
Applying DFS in Backward search at level 2 we get : {[G->D]}



Hence, the destination node through the forward search and backward search **have the same value** i.e; **D**. Here, we will **terminate** from the **searching procedure**. Thus, we got the shortest path from the root node to the goal node: **{[A->D->G->I->J]}.**

Hence, we use bidirectional search approach when-

- Both initial and goal states are unique and completely defined.
- The branching factor is exactly the same in both directions.

Performance measures

- **Completeness:** Bidirectional search is complete if BFS is used in both searches.
- **Optimality:** It is optimal if BFS is used for search and paths have uniform cost.
- **Time and Space Complexity:** Time and space complexity is **$O(b^{d/2})$** since each search need only proceed to half the solution depth. Since at least one of the searches must be breadth first in order to find a common state, the space complexity of bidirectional search is also $O(b^{d/2})$. As a result, bidirectional search is space bound in practice.

ADVANTAGES:

- Bidirectional search requires less memory.
- This searching technique is faster than any other techniques.

DIS-ADVANTAGES:

- In bi-directional search the goal node should be known from earlier.
- Implementation of bidirectional search is difficult.

Informed search

As we have studied 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 unlike uninformed search, informed search algorithm has the knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge helps the agent to explore the search space and find more efficiently to the goal node.

The informed search algorithm is more useful for large search space and it uses the idea of heuristic, so it is also called Heuristic search.

A heuristic is a method that might not always find the best solution but it guarantees to find a good solution in reasonable time and it increases efficiency by sacrificing completeness. It is Useful in solving tough problems which could not be solved by any other way and the solutions which takes an infinite time or very long time to compute.

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 agent is from the goal.

A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

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.

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.

Greedy Best First Search

Greedy best-first search algorithm always selects the path which appears best at that moment. 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)= h(n).

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

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

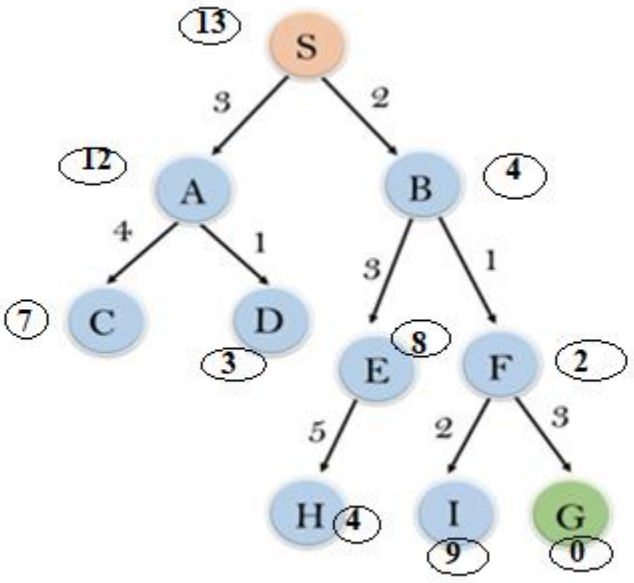
It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node.

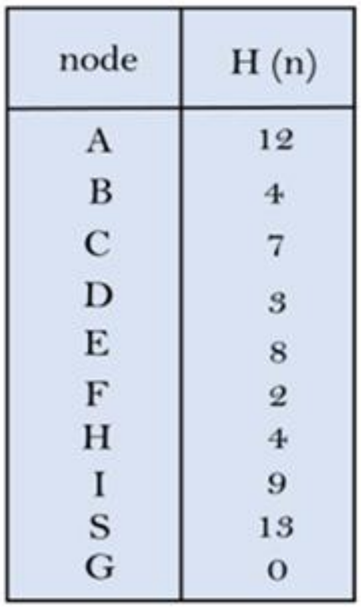
Greedy Best first search algorithm:

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 lists, then add it to the OPEN list.
Step 7: Return to Step 2.

Example1:

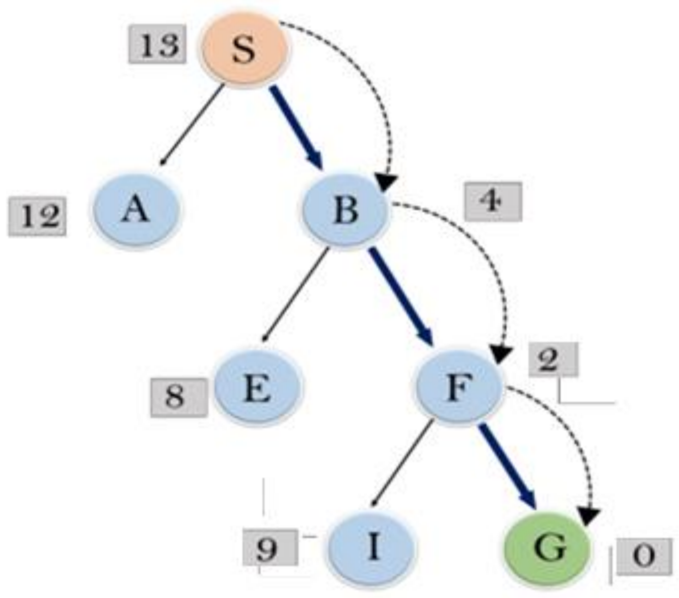
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





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 iteration 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**

Example2:

Let us see how this works for route-finding problems in Romania:

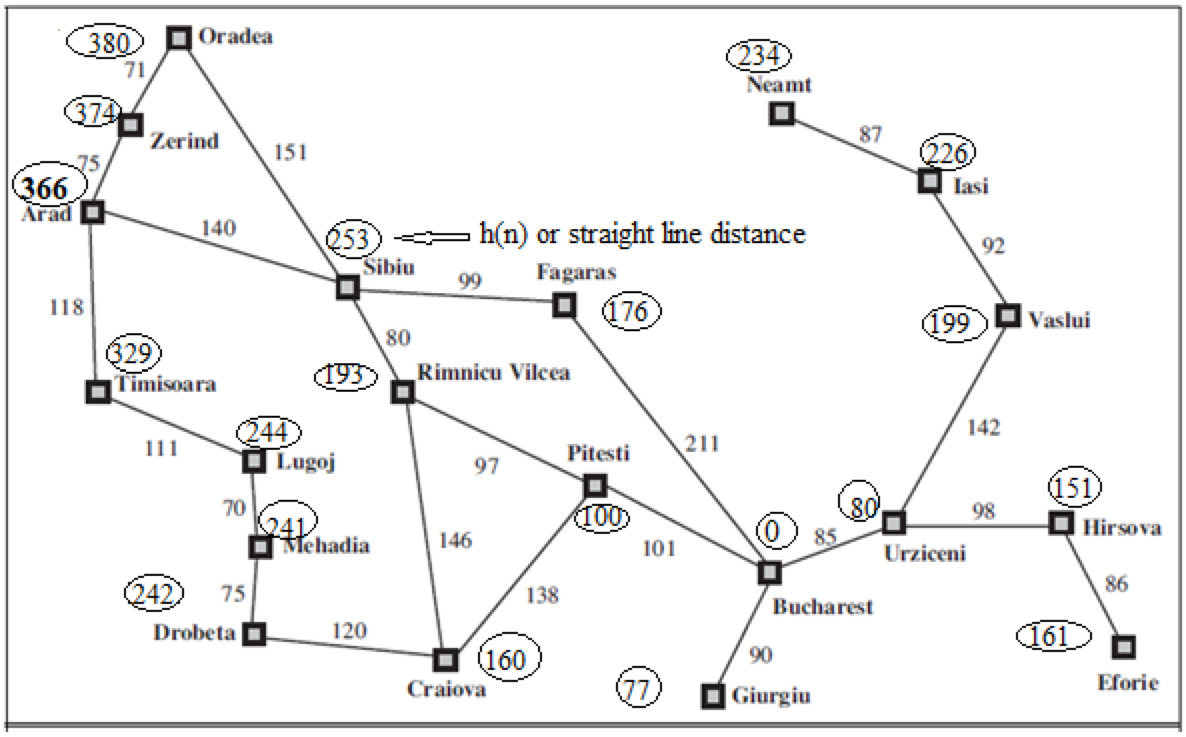
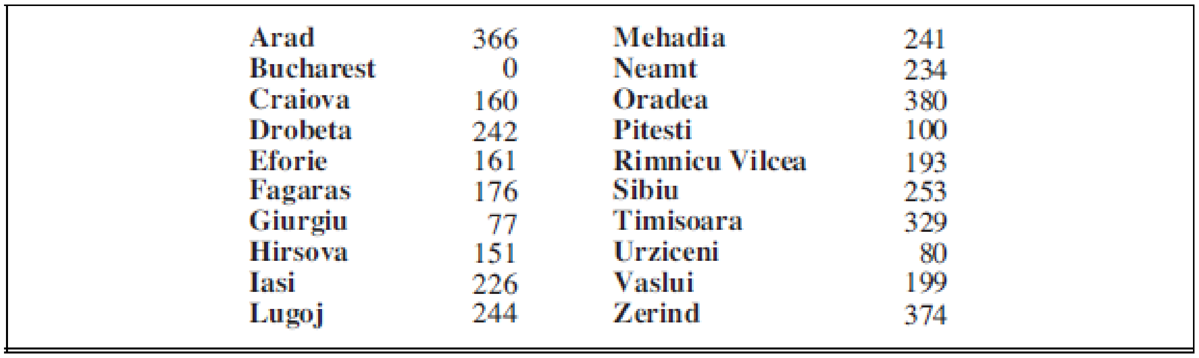


Figure: A simplified road map of part of Romania.

we use the **straight**-Line **distance** heuristic, which we will call **h**$_{\text{SLD}}$. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure below.



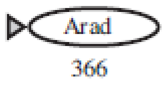
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure: Values of hSLD—straight-line distances to Bucharest.

For example,

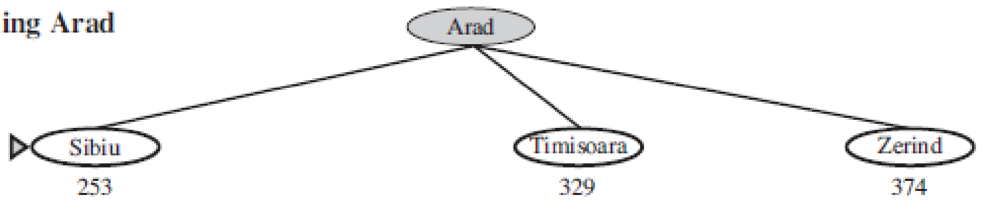
Figure below shows the progress of a greedy best-first search using $\mathbf{h}_{SLD}$ to find a path from Arad to Bucharest. For example, $\mathbf{h}_{SLD}$ (In(Arad))=366.

(a) The initial state



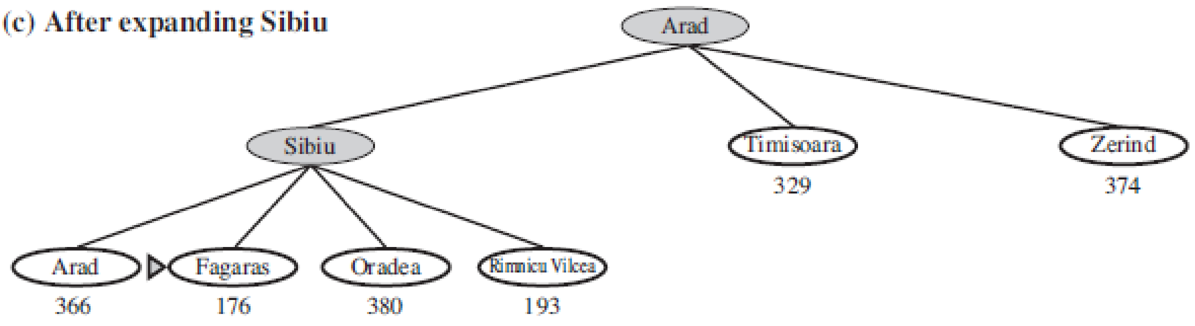
⇨ The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.

(b) After expanding Arad



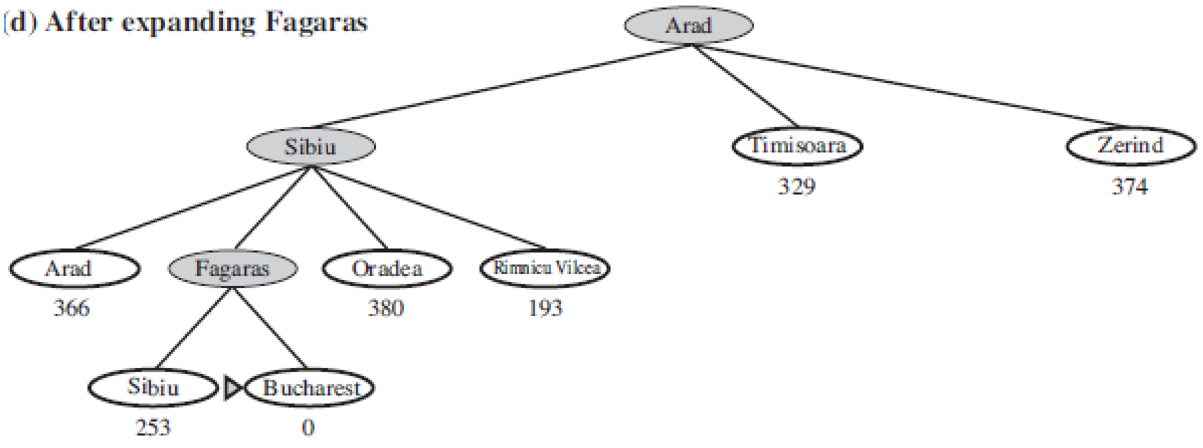
⇨ The next node to be expanded will be Fagaras because it is closest to Bucharest.

(c) After expanding Sibiu



⇨ Fagaras in turn generates Bucharest, which is the goal.

(d) After expanding Fagaras

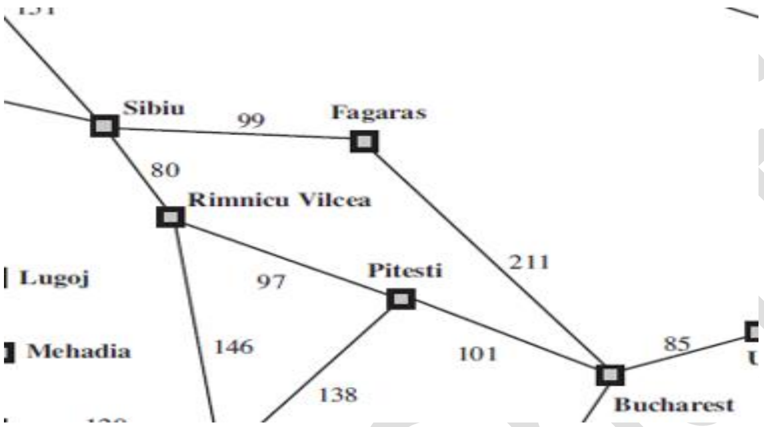


⇨ For this particular problem, greedy best-first search using $\mathbf{h}_{SLD}$ finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

The properties of Greedy Best First Search:

Optimality: Greedy best first search algorithm is not optimal.

However: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.



Path1=sibiu+fagaras=bucherest

=99+211=310

Path2=sibiu+rimnicu vilace+pitesti

=80+97+101

=278

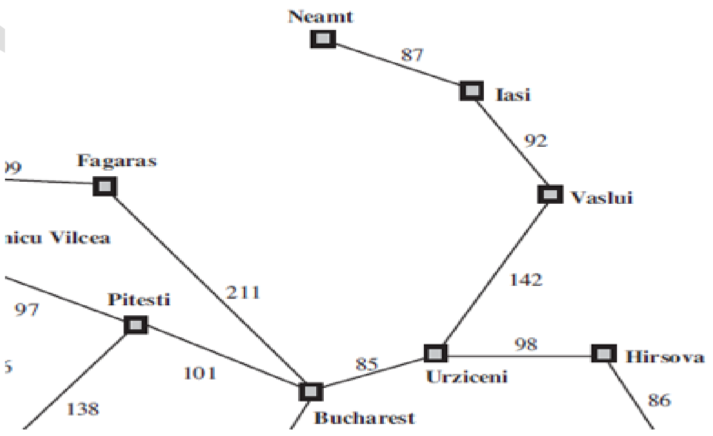
Diff=path1-path2

=310-278

= 32

Completeness: **Greedy best-first tree search is also incomplete**.

Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end.



The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras.

The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop**. Thus the graph search version *is* complete in finite spaces, but not in infinite ones.**

Time Complexity:

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

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.

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.

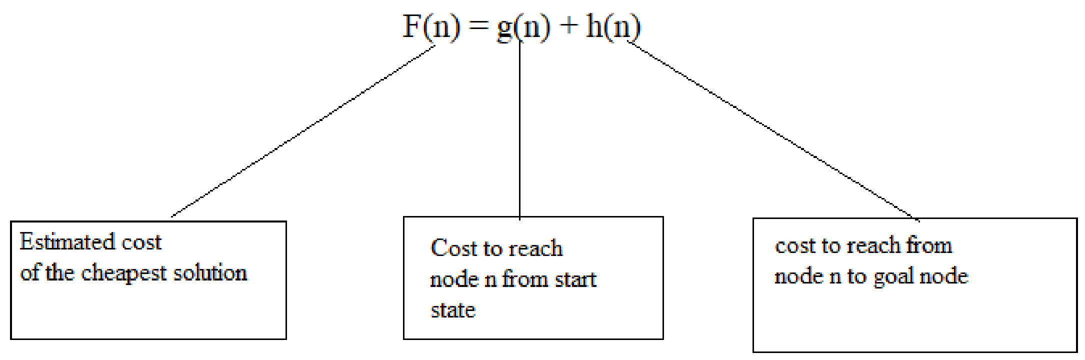
Disadvantages:

- 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.

A * Search

The most widely known form of best-first search is called A * search. A-star search is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.

It avoids expanding paths that are already expensive, but expands most promising paths first. It evaluates nodes by combining g(n), and h(n). Each time A* enters a state, it calculates the cost, f(n) to travel to all of the neighbouring nodes, and then enters the node with the lowest value of f(n).
These values are calculated with the following formula:


F(n) = g(n) + h(n)
Estimated cost of the cheapest solution
Cost to reach node n from start state
cost to reach from node n to goal node

f(n) = g(n) + h(n), where

- g(n) is the cost to reach the node n from start state
- h(n) is the estimated cost to get from the node n to the goal
- f(n) is the estimated total cost of path through n to goal node OR 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).

Algorithm of A* search:

Step-1:

- Define a list OPEN.
- Initially, OPEN consists start node S.

Step-2:

- If the list is empty, return failure and exit.

Step-3:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step-4:

- Expand node n.

Step-5:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-6.

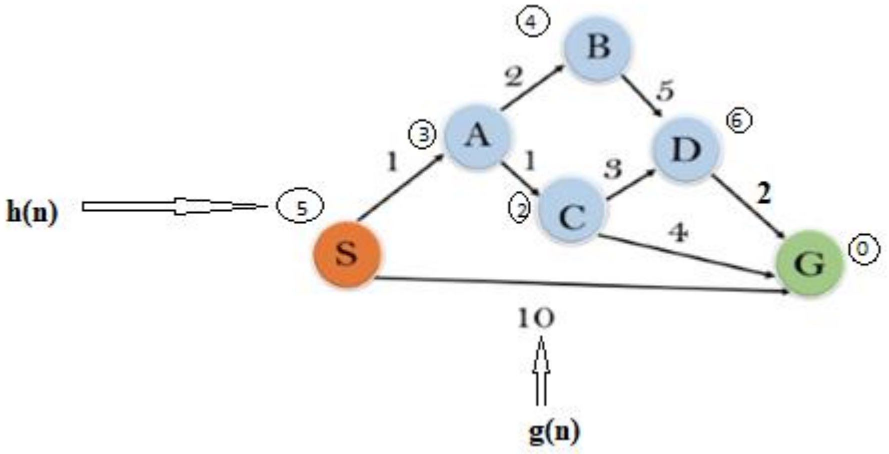
Step-6:

- For each successor node,
- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

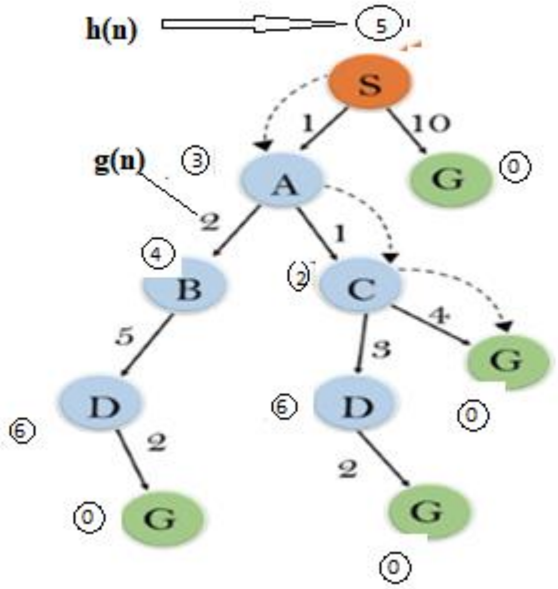
Step-7:

- Go back to Step-2.

Example 1: -



Solution:



Solution-

Step-01:

- We start with node s.
- Node A and Node G can be reached from node s.

A* Algorithm calculates f(B) and f(F).

- f(A) = 1 + 3 = 4
- f(G) = 10 + 0 = 10

Since f(A) < f(G), so it decides to go to node A.

Path- S→ A

Step-02:
Node B and Node C can be reached from node A.
A* Algorithm calculates f(B) and f(C).

- f(B) = (2+1) + 4 = 7
- f(C) = (1+1) + 2 = 4

Since f(C) < f(B), so it decides to go to node C.
Path- S→ A → C

Step-03:
Node D and Node G can be reached from node C.
A* Algorithm calculates f(D) and f(G).

- f(D) = (1+1+3) + 6 = 11
- f(G) = 1+1+4 + 0 = 6

Since f(G) < f(D), so it decides to go to node G.
Path- S → A → C → G

This is the required shortest path from node S to node G.

Example 2: -

In Figure below, we show the progress of an A∗ tree search for Bucharest.

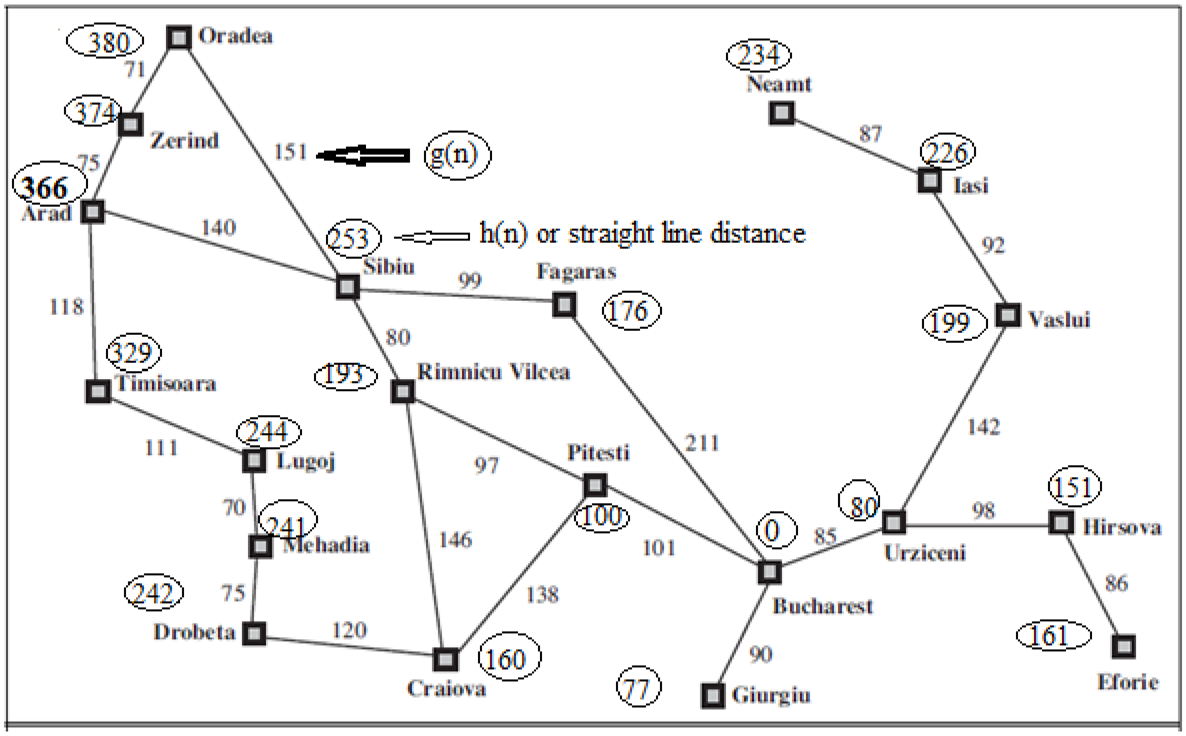
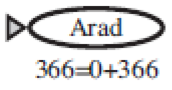


Figure: A simplified road map of part of Romania.

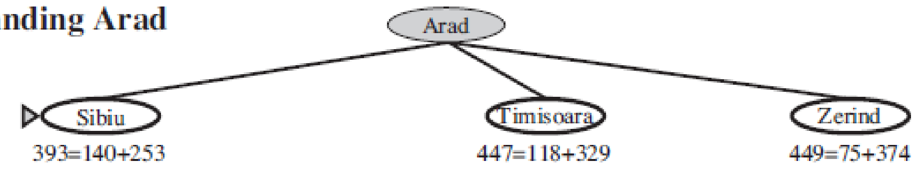
(a) The initial state



F(n)=g(n)+h(n)

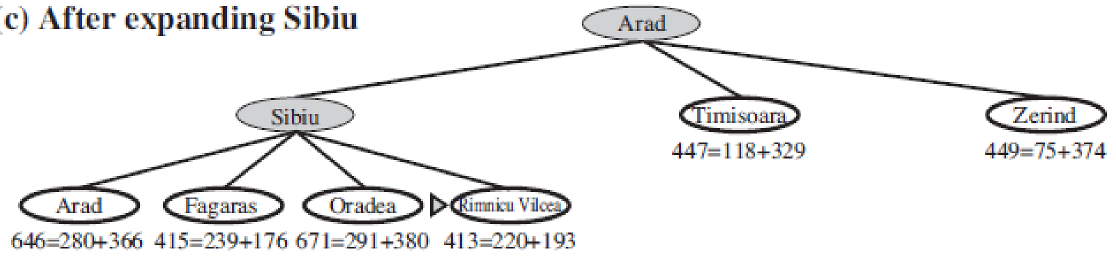
F(arad)=g(arad)+h(arad)
=0+366
=366

(b) After expanding Arad

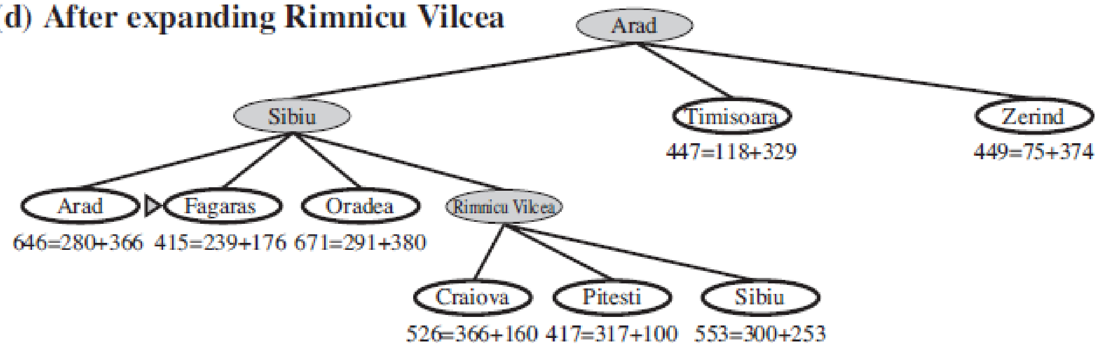


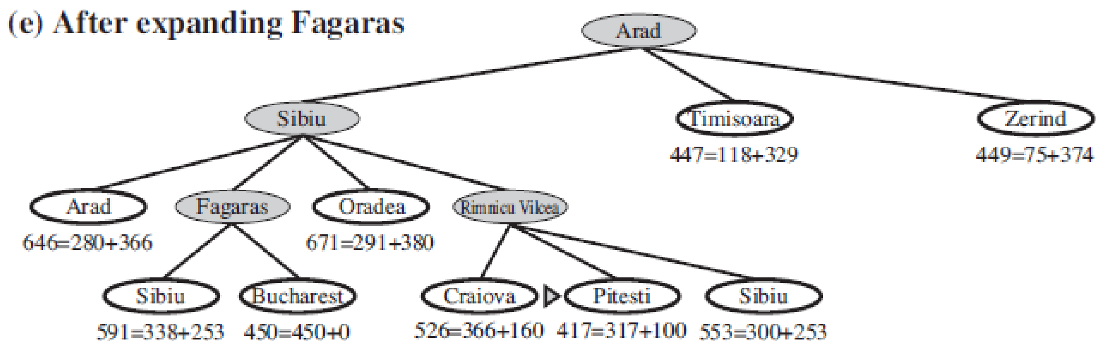
F(sibiu)=g(n)+h(n)
=140+253
=393
F(Timisoara)=g(n)+h(n)
=118+329
=447
F(zerind)=g(n)+h(n)
=75+374
=449

(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea





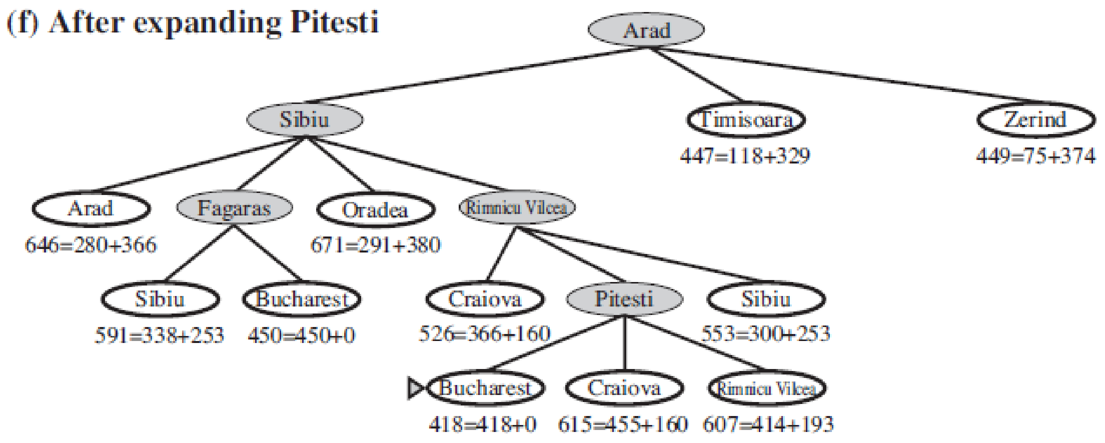


Figure: - Stages in an A∗ search for Bucharest.

Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

The properties of A * Search:

Optimality:

When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution, i.e. shortest path to the finish state. A* search algorithm is optimal if it follows below two conditions:

1. **Admissible**:

A given heuristic function h(n) is ***admissible*** if it never overestimates the real distance between *n* and the goal node. Therefore, for every node *n* the following formula applies:

$$h(n) \leq h*(n)$$

h*(n) is the real distance between *n* and the goal node. Hence, If the heuristic function is admissible, then A* search will always find the least cost path.

2. **Consistency:**

A given heuristic function h(n) is *consistent* if the estimate is always less than or equal to the estimated distance between the goal *n* and any given neighbour, plus the estimated cost of reaching that neighbour:

$$\mathbf{h(n) \leq c(n,m)+h(m)}$$

c(n,m) is the distance between nodes n and m. Additionally, if h(n) is consistent, then we know the optimal path to any node that has been already inspected. This means that this function is optimal.

Completeness: -

When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it. A* algorithm is complete as long as:

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

Time Complexity:

The time complexity of A* 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)**

Advantages:

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

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* 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.

Hill Climbing:

- Hill Climbing is a form of heuristic search algorithm which is used in solving optimization related problems in Artificial Intelligence domain.

- The algorithm starts with a non-optimal state and iteratively improves its state until some predefined condition is met. The condition to be met is based on the heuristic function.

- The aim of the algorithm is to reach an optimal state which is better than its current state. The starting point which is the non-optimal state is referred to as the base of the hill and it tries to constantly iterate (climb) until it reaches the peak value, that is why it is called Hill Climbing Algorithm.

- Hill climbing algorithm continuously moves in the direction of increasing value to find the best solution to the problem (peak of the mountain). It terminates when it reaches a peak value where no neighbour has a higher value.

- It is a technique which is used for optimizing the mathematical problems. **mathematical optimization problems** imply that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance travelled by the salesman.

- It is also called greedy local search as it only looks to its good immediate neighbour 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

1. Variant of generate and test algorithm:

Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution. Hill Climbing is a variant of generate and test algorithm. The generate and test algorithm is as follows:

1. Generate a possible solution.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

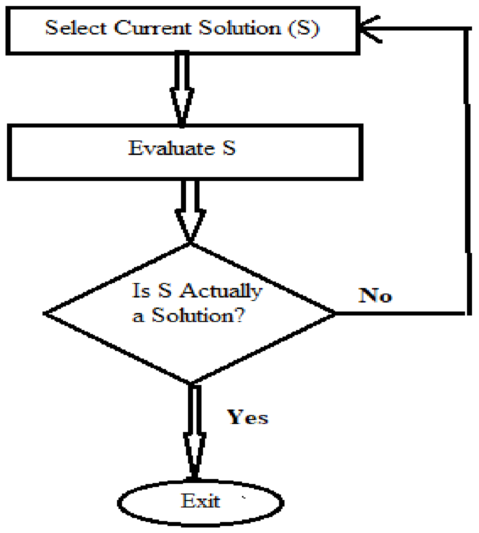
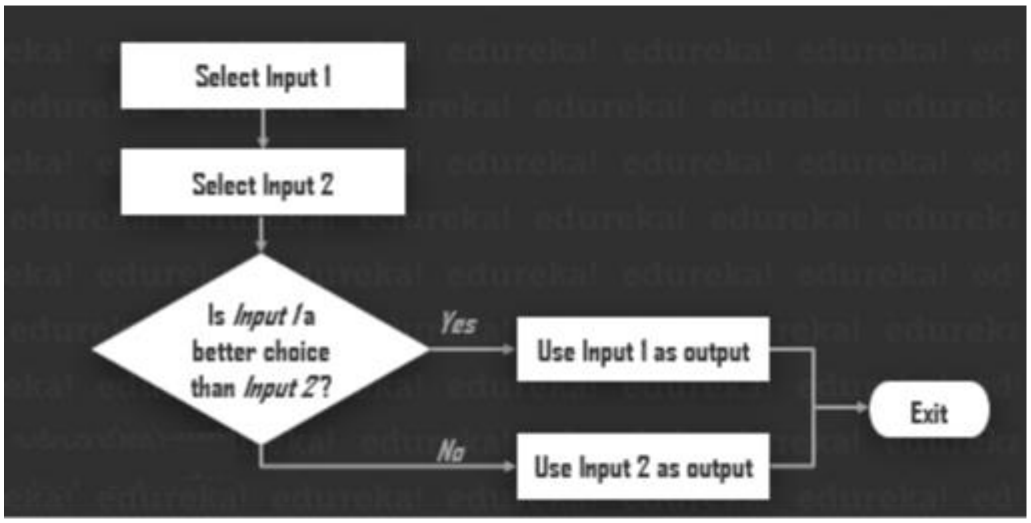


Figure: Generate and Test

The solutions that need to be generated vary depending on the kinds of problems. For some problems the possible solutions may be particular points in the problem space and for some problems, paths from the start state. Hence, we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

2. **Uses the Greedy approach :**

At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.



3. **No backtracking:**
   A hill-climbing algorithm only works on the current state and succeeding states (future). It does not look at the previous states.
4. **Feedback Mechanism:** The feedback from the previous computation helps in deciding the next course of action i.e. whether to move up or down the slope

State-space Diagram for Hill Climbing:

- The state-space diagram is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function, and on the x-axis have taken state-space.
- If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

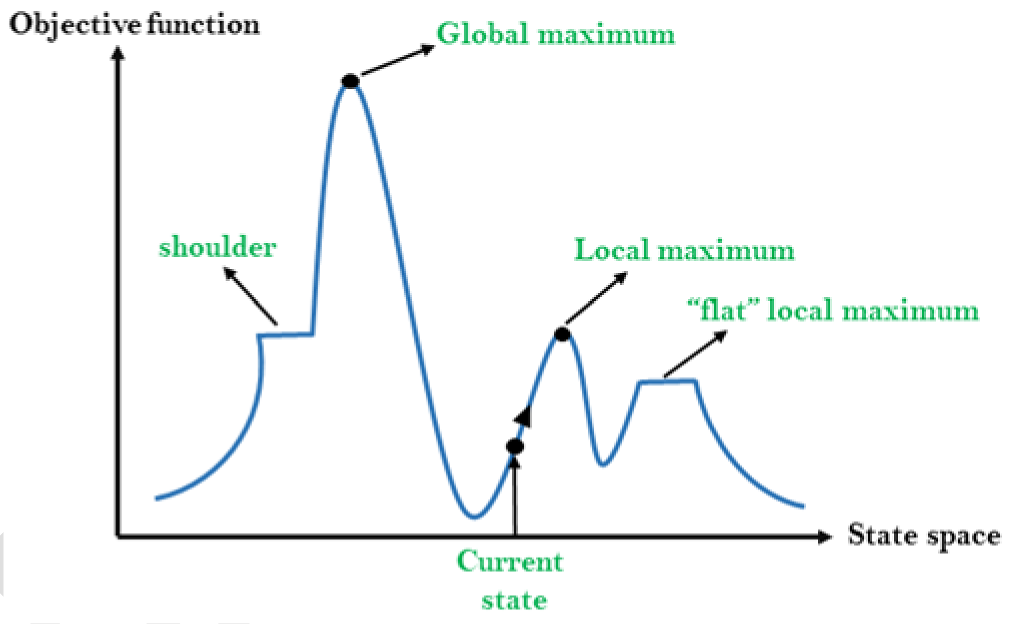


Figure: A one-dimensional state-space landscape

1. **Local Maximum:** Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it.
2. **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
3. **Current state:** It is a state in a landscape diagram where an agent is currently present.
4. **Flat local maximum:** It is a flat space in the landscape where all the neighbour states of current states have the same value.
5. **Shoulder:** It is a region having an edge upwards.

Types of Hill Climbing Algorithm:

1. Simple hill Climbing:
2. Steepest-Ascent hill-climbing:
3. Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set 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 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:

- If it is goal state, then return success and quit.
- Else if it is better than the current state then assigns new state as a current state.
- 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 simple hill climbing algorithm. This algorithm examines all the neighbouring nodes of the current state and selects one neighbour node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbours.

Algorithm for Steepest-Ascent hill climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and stop, otherwise continue with the initial state as the current state

Step 2: Loop until a solution is found or Until a complete iteration no change on current state

- Let SUCC be a state such that any possible successor of the current state will be better than SUCC.
- For each operator that applies to the current state do:
    a. Apply the new operator and generate a new state.
    b. Evaluate the new state.
    c. If it is goal state, then return it and quit, else compare it to the SUCC.
    d. If it is better than SUCC, then set new state as SUCC.
    e. If the SUCC is better than the current state, then set current state to SUCC.

Step 3: Exit.

3. Stochastic hill climbing:

This algorithm is different from the other two algorithms, as it selects neighbour nodes randomly and makes a decision to move or choose another randomly. This algorithm is very less used compared to the other two algorithms.

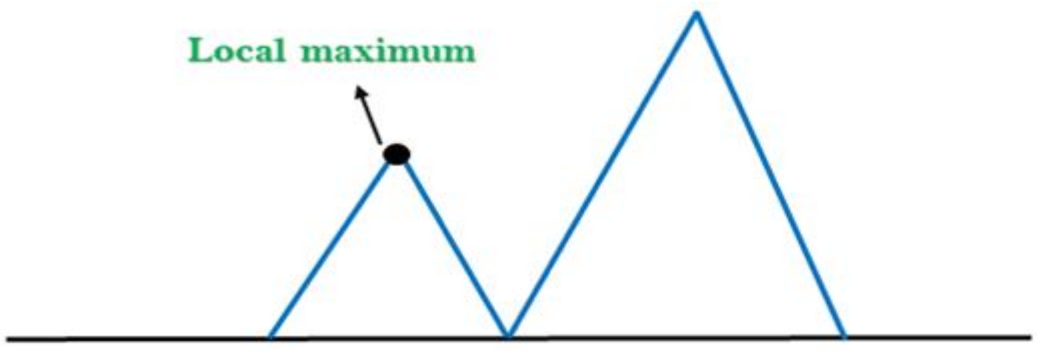
The features of this algorithm are given below:

- It uses a greedy approach as it goes on finding those states which are capable of reducing the cost function irrespective of any direction.
- It is considered as a variant in generating expected solutions and the test algorithm. It first tries to generate solutions that are optimal and evaluates whether it is expected or not. If it is found the same as expected, it stops; else it again goes to find a solution.
- It does not perform a backtracking approach because it does not contain a memory to remember the previous space.

<u>Problems in Hill Climbing Algorithm:</u>

Both simple and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state or getting to a state from which no better state can be generated. this will happen if the program has reached either a local maximum, a plateau or a ridge.

1. **The Local Maximum / The foothills Problem:**
   A local maximum is a peak state that is better than all its neighbours but not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are practically frustrating because they often occur almost within sight of a solution. In this case, they are called **foothills.**

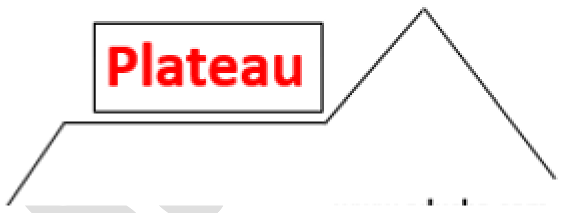


Solution:

Backtracking to earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

2. The Plateau problem:

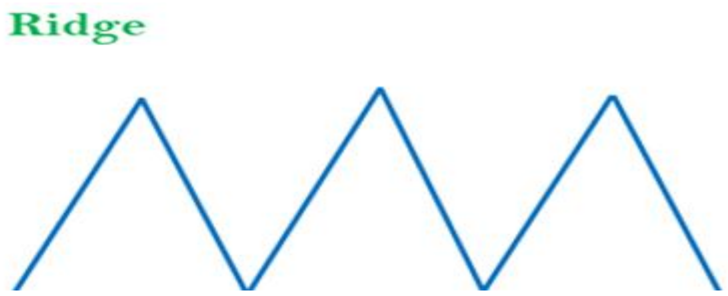
A plateau is the flat area of the search space in which all the neighbour states of the current state contain the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.



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 non-plateau region.

3. The Ridge problem:

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. It occurs when there are multiple peaks and all have the same value or in other words, there are multiple local maxima which are same as global maxima

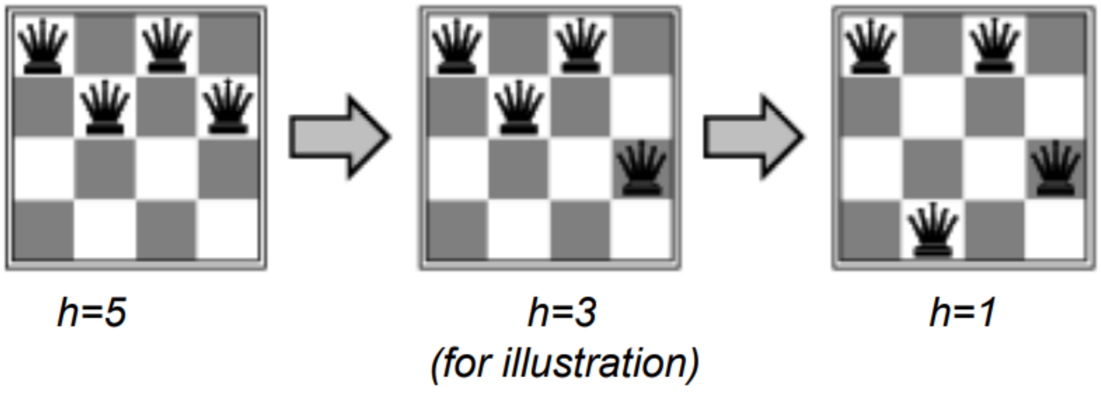


Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem. Ridge obstacle can be solved by moving in several directions at the same time

Hill-climbing Example: n-queens

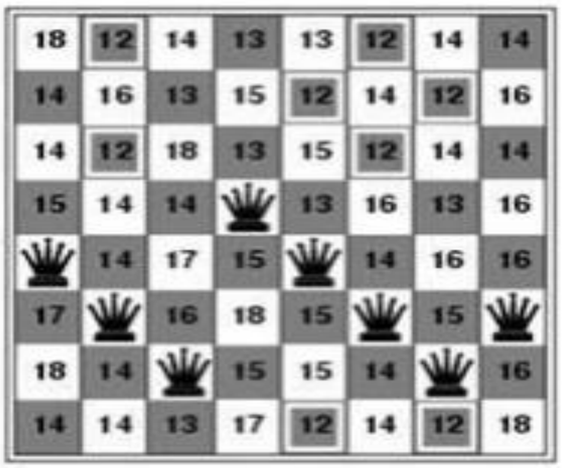
1. n-queens problem: Put n queens on an n × n board with no two queens on the same row, column, or diagonal

2. Good heuristic: h = number of pairs of queens that are attacking each other

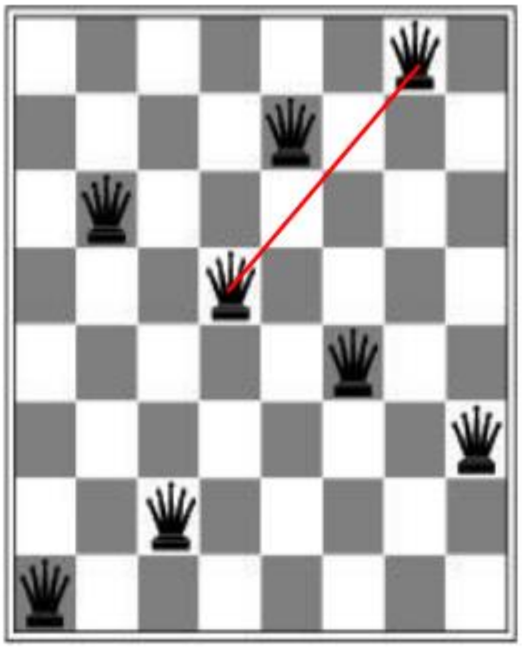


Hill-climbing Example: 8-queens

h = number of pairs of queens that are attacking each other



A state with h=17 and the h-value for each possible successor



A local minimum of h in the 8-queens state space (h=1).

Simulated Annealing

Simulated annealing actually has its origins in metallurgy. In metallurgy, annealing refers to the process of heating metal to a high temperature and then slowly cooling it in a controlled environment. The high heat gives the atoms in the metal the freedom to move around wildly. If the metal were to be cooled quickly, the atoms would suddenly come to rest wherever they were when the metal was hot, resulting in a random arrangement and a poor-quality result. But when the metal is cooled slowly, the atoms have time to gradually find the best possible orientation, and align themselves into a nice crystal lattice. This is preferable to slow cooling; it makes the metal more ductile, reduces defects, and makes it significantly stronger. In general, annealing gradually reduces energy and movement to allow a substance's components to settle into their most stable, simplest order. With this in mind Simulated Annealing Algorithm is come into picture.

Hence,

Simulated Annealing is a stochastic global search optimization algorithm. The algorithm is inspired by annealing in metallurgy where metal is heated to a high temperature quickly, then cooled slowly, which increases its strength and makes it easier to work with.

The annealing process works by first exciting the atoms in the material at a high temperature, allowing the atoms to move around a lot, then decreasing their excitement slowly, allowing the atoms to fall into a new, more stable configuration.

When hot, the atoms in the material are freer to move around, and, through random motion, tend to settle into better positions. A slow cooling brings the material to an ordered, crystalline state.

The simulated annealing optimization algorithm can be thought of as a modified version of stochastic hill climbing.

Stochastic hill climbing maintains a single candidate solution and takes steps of a random but constrained size from the candidate in the search space. If the new point is better than the current point, then the current point is replaced with the new point. This process continues for a fixed number of iterations.

Simulated annealing executes the search in the same way. The main difference between stochastic hill-climbing and simulated annealing is that in stochastic hill-climbing steps are taken at random and the current point is replaced with a new point provided the new point is an improvement to the previous point.

Whereas in simulated annealing, the search works the same way but sometimes the worse points are also accepted to allow the algorithm to learn answers that are eventually better.

The worse points are accepted based on some probability. The probability of accepting a worse state is given by the equation as follows…

P=exponential(-c/t)>r

Where,

c= The change in the evaluation function

t= the current value

r= a random number between 0 and 1

1. If the Random Number > Acceptance Probability then the new feature set is Rejected and the previous feature set will be continued to be used.
2. If the Random Number < Acceptance Probability then the new feature set is Accepted.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random nearby solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter t (called the temperature), that is gradually decreased during the process.

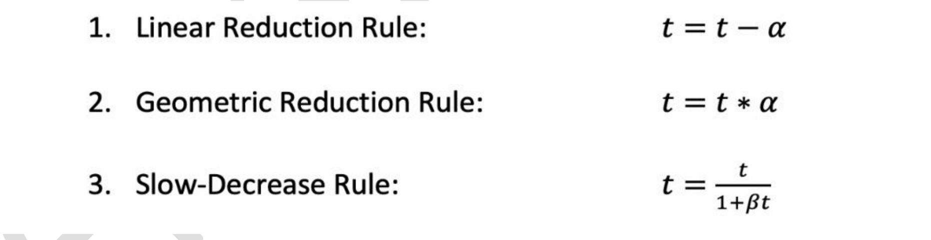
Algorithm

Step 1:

We first start with an initial solution $s = S_0$. This can be any solution that fits the criteria for an acceptable solution. We also start with an initial temperature $t = t_0$.

Step 2:

Setup a temperature reduction function alpha. There are usually 3 main types of temperature reduction rules:



1. Linear Reduction Rule:	$t = t - \alpha$
2. Geometric Reduction Rule:	$t = t * \alpha$
3. Slow-Decrease Rule:	$t = \frac{t}{1+\beta t}$

Each reduction rule reduces the temperature at a different rate and each method is better at optimizing a different type of model. For the 3rd rule, beta is an arbitrary constant.

Step 3:

Starting at the initial temperature, loop through n iterations of Step 4 and then decrease the temperature according to alpha. Stop this loop until the termination conditions are reached. The termination conditions could be reaching some end temperature, reaching some acceptable threshold of performance for a given set of parameters, etc. The mapping of time to temperature and how fast the temperature decreases is called the Annealing Schedule.

Step 4:

Given the neighbourhood of solutions N(s), pick one of the solutions and calculate the difference in cost between the old solution and the new neighbour solution. The neighbourhood of a solution are all solutions that are close to the solution.

Step 5:

If the difference in cost between the old and new solution is greater than 0 (the new solution is better), then accept the new solution. If the difference in cost is less than 0 (the old solution is better), then generate a random number between 0 and 1 and accept it if it's under the value calculated from the Energy Magnitude equation from before.

In the Simulated Annealing case, the equation has been altered to the following:

$$P = \begin{cases} 1 & if \Delta c \le 0 \\ e^{-\Delta c / t} & if \Delta c > 0 \end{cases}$$

Where the delta c is the change in cost and the t is the current temperature.
The P calculated in this case is the probability that we should accept the new solution.

Some Example of Problems to Optimize with SA

- Travelling Salesman Problem (TSP)
- Scheduling Problems
- Task Allocations
- Graph colouring and partitioning
- Non-linear function optimizations

Advantages vs. Disadvantages of SA

Advantages

- Easy to implement and use
- Provides optimal solutions to a wide range of problems

Disadvantages

- Can take a long time to run if the annealing schedule is very long
- There are a lot of tuneable parameters in this algorithm

Game playing and AI: -

Game Playing is an important domain of artificial intelligence. Game don't require much knowledge; the only knowledge we need provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So, both of them try to make the best move possible at each turn.

Searching techniques like BFS (Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improves-

- Generate procedure so that only good moves are generated
- Test procedure so that the best move can be explored first.

Games are a form of multi-agent environment.

- What do other agents do and how do they affect our success?
- Cooperative vs. competitive multi-agent environments
- Competitive multi-agent environments give rise to adversarial search often known as games

A game playing can be formally defined as a kind of search problem with the following elements:

- **S0:** The initial state, which specifies how the game is set up at the start.
- **PLAYER(s):** Defines which player has the move in a state.
- **ACTIONS(s):** Returns the set of legal moves in a state.
- **RESULT (s, a):** The transition model, which defines the result of a move.
- **TERMINAL-TEST(s):** A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- **UTILITY (s, p):** A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state 's' for a player 'p'.

In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/ 2. Some games have a zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or 1/2+ 1/2.

Adversarial search techniques

- Adversarial search is a method applied to a situation where you are planning while another actor prepares against you. Your plans, therefore, could be affected by your opponent’s actions. The term “search” in the context of adversarial search refers to games in the area of artificial intelligence (AI).

- Adversarial search problems typically exist in two-player games where the players’ actions alternate. Quick examples that come to mind are chess, checkers, and tic-tac-toe.

- It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players.

- The game for instance, teaches three different opening strategies. Your next moves, however, would entirely depend on your opponent’s moves. The other player’s countermoves, on the other hand, would also be dependent on your opening moves, and so on.

Games-adversary

- Solution is strategy (strategy specifies move for every possible opponent reply),
- Time limits force an approximate solution
- Evaluation function: evaluate "goodness" of game position
- Examples: chess, checkers, and tic-tac-toe etc.

Difference between the search space of a game and the search space of a problem: In the first case it represents the moves of two (or more) players, whereas in the latter case it represents the "moves" of a single problem-solving agent.

Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line. The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X). The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.

Terminal states are those representing a win for X, loss for X, or a draw Each path from the root node to a terminal node gives a different complete play of the game.

Figure given below shows the initial search space of Tic-Tac-Toe.

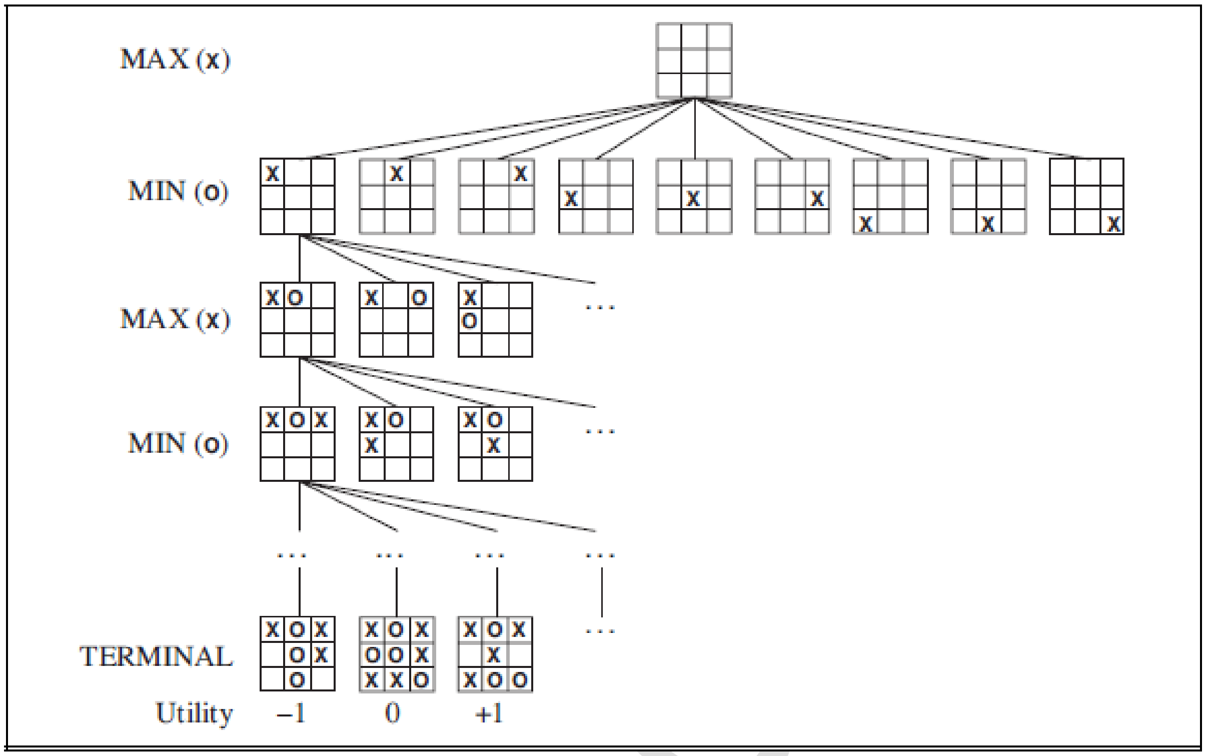


Figure: Partial game tree for Tic-Tac-Toe

A game can be formally defined as a kind of search problem as below

- **Initial state:** It includes the board position and identifies the players to move.
- **Successor function:** It gives a list of (move, state) pairs each indicating a legal move and resulting state
- **Terminal test:** This determines when the game is over. States where the game is ended are called terminal states
- **Utility function:** It gives numerical value of terminal states Eg win (1), loose (-1) and draw (0). Some games have a wider variety of possible outcomes eg. ranging from +2 to-192

Game Tree and Minmax Evaluation

We first consider games with two players, whom we call MAX and MIN. MAX moves first, and then another takes turn moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser.

Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS (Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time.

The initial state, ACTIONS function, and RESULT function define the game tree for the game—a tree where the nodes are game states and the edges are moves. Figure below shows part of the game tree for tic-tac-toe.

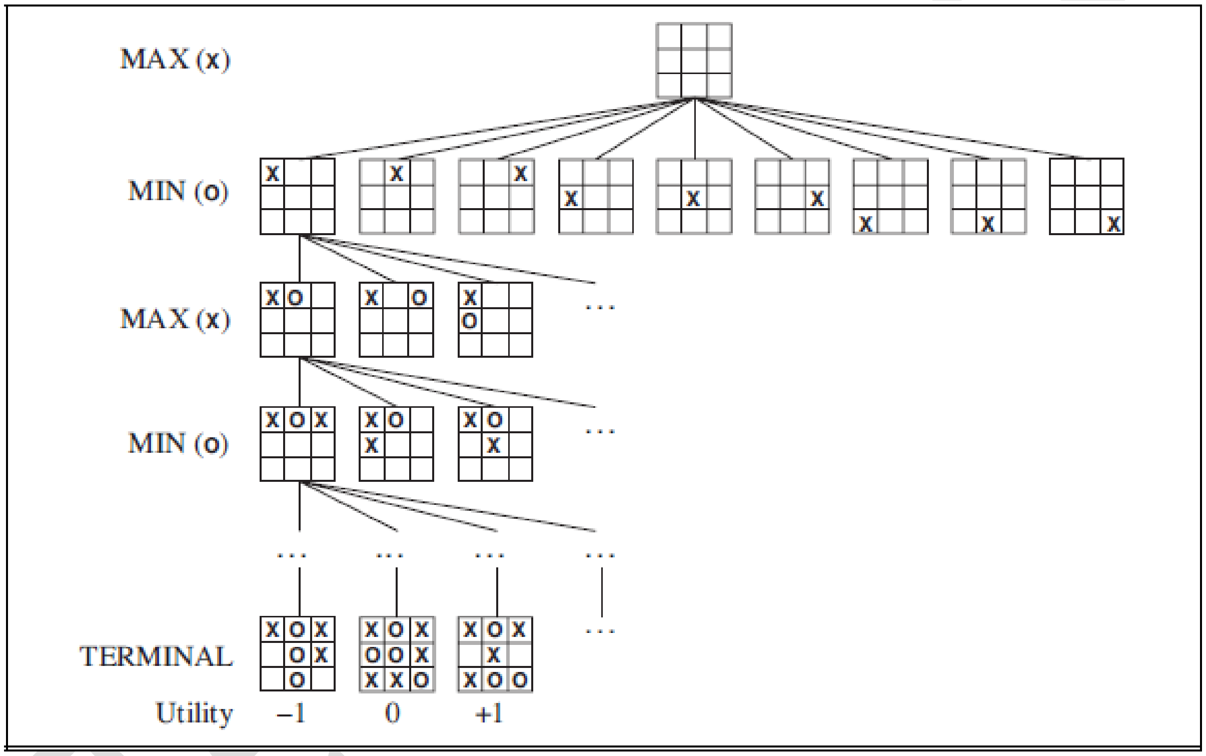


Figure : A (partial) game tree for the game of tic-tac-toe.

The top node is the initial state, MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure below. The possible moves for MAX at the root node are labelled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, and so on. This particular game ends after one move each by MAX and MIN.
The utilities of the terminal states in this game range from 2 to 14.

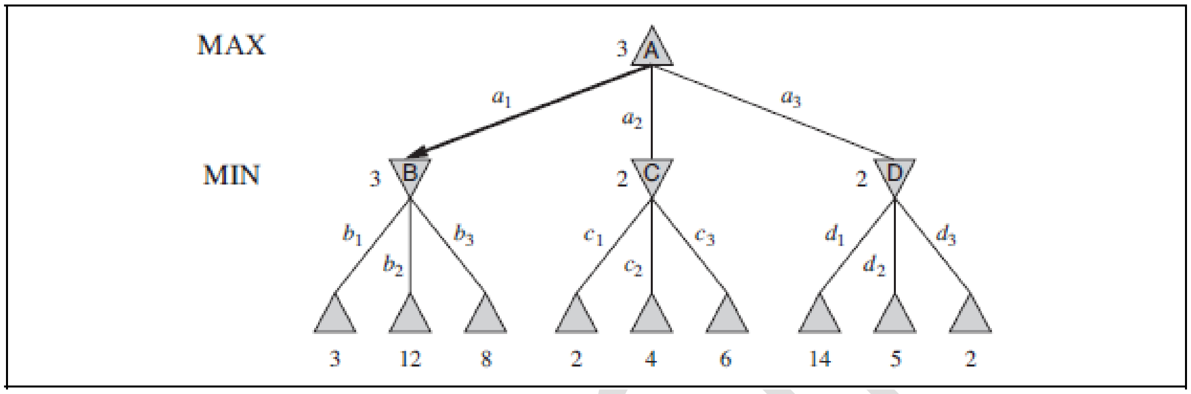


Figure: A two-ply game tree.

Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as MINIMAX(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

As shown in Figure above, the terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labelled B, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so, it has a minimax value of 3. We can also identify the minimax decision at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game; one is called MAX and other is called MIN.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of Min-Max Algorithm:

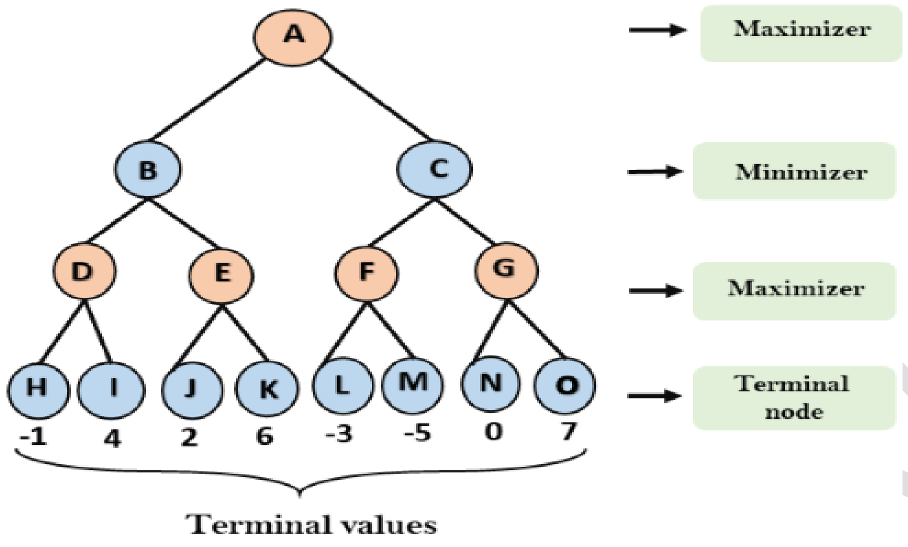
The working of the minimax algorithm can be easily described using an example. Here we have taken an example of game-tree which is representing the two-player game. In this example, there are two players one is called Maximiser and other is called Minimizer. Maximiser will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.

This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes. At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs.

Following are the main steps involved in solving the two-player game tree:

Step-1:

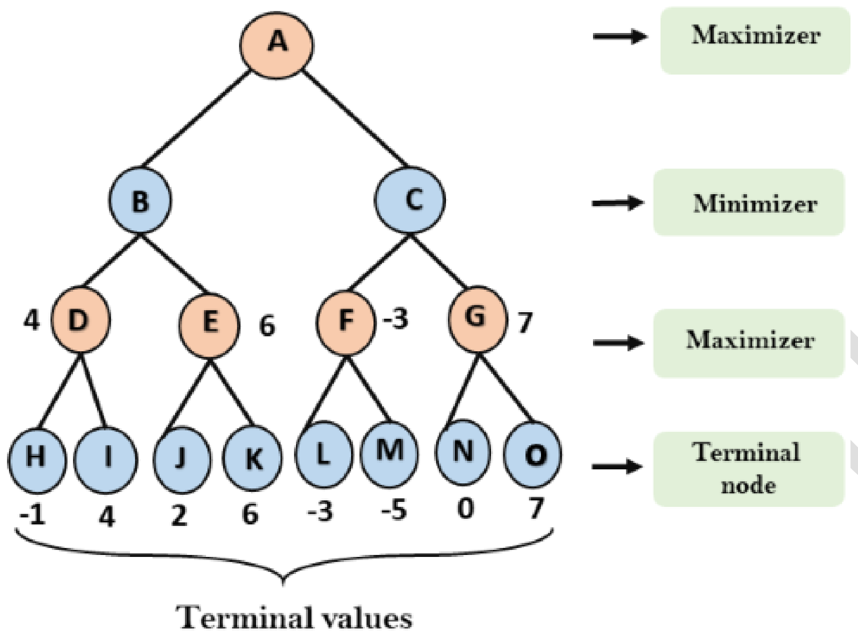
In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximiser takes first turn which has worst-case initial value = - infinity, and minimizer will take next turn which has worst-case initial value = + infinity.



Step 2:

Now, first we find the utilities value for the Maximiser, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximiser and determines the higher nodes values. It will find the maximum among the all.

For node D	max(-1, -∞) => max(-1,4)= 4
For Node E	max(2, -∞) => max(2, 6)= 6
For Node F	max(-3, -∞) => max(-3,-5) = -3
For node G	max(0, -∞) = max(0, 7) = 7

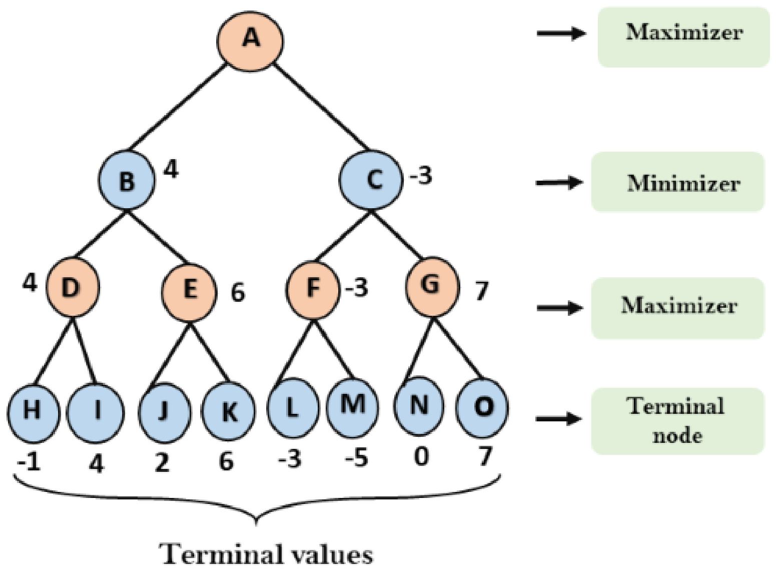


Step 3:

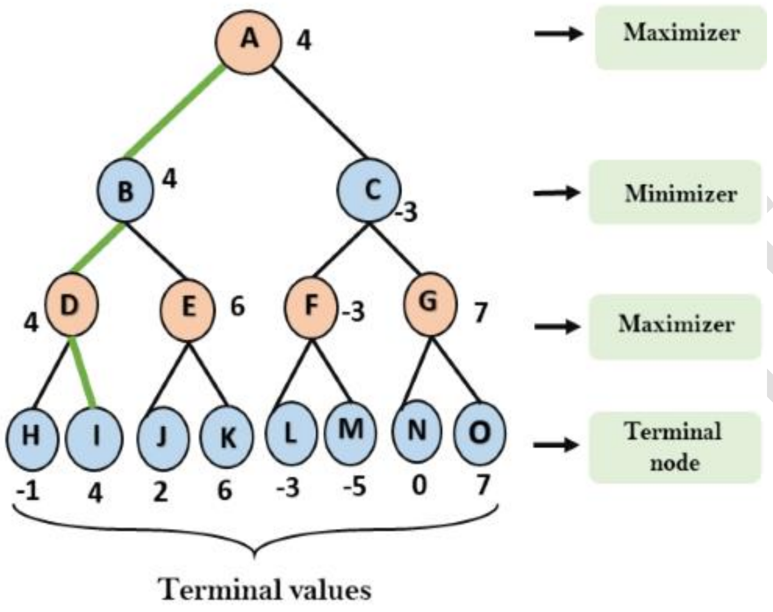
In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

For node B= min(4,6) = 4

For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximiser, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
For node A max(4, -3)= 4



This is the complete workflow of the minimax two player game.

<u>Properties of Mini-Max algorithm:</u>

- **Completeness-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimality-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **O(b^d)**, where b is branching factor of the game-tree, and d is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bd)**.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

Min-Max with Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm. Pruning allows us to ignore portions of the search tree that make no difference to the final choice.

- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.

- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prunes the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximiser. The initial value of alpha is -∞.
  b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

1. **$\alpha >= \beta$**

Key points about alpha-beta pruning:

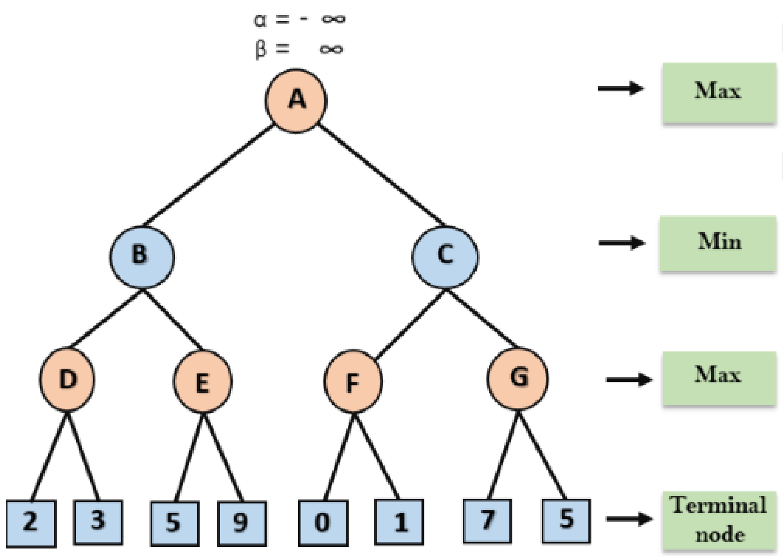
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Working of Alpha-Beta Pruning:

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1:

At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to **node B** where again α= -∞ and β= +∞, and **Node B** passes the same value to its child D.

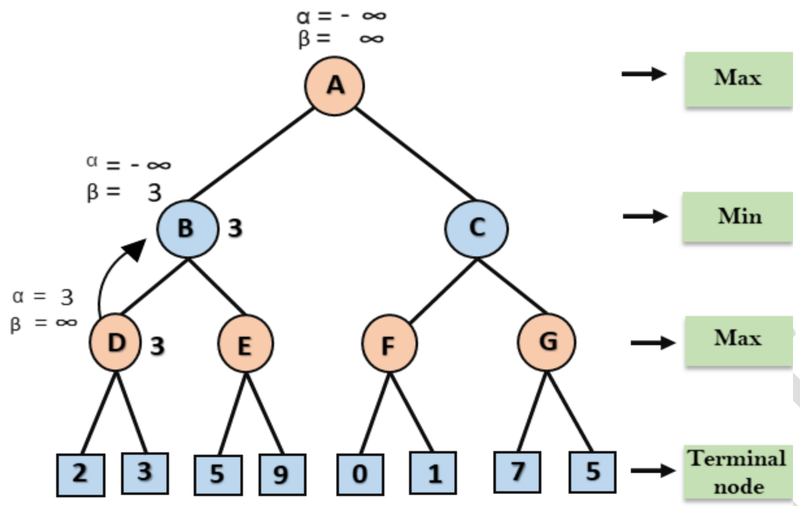


Step 2:

At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3:

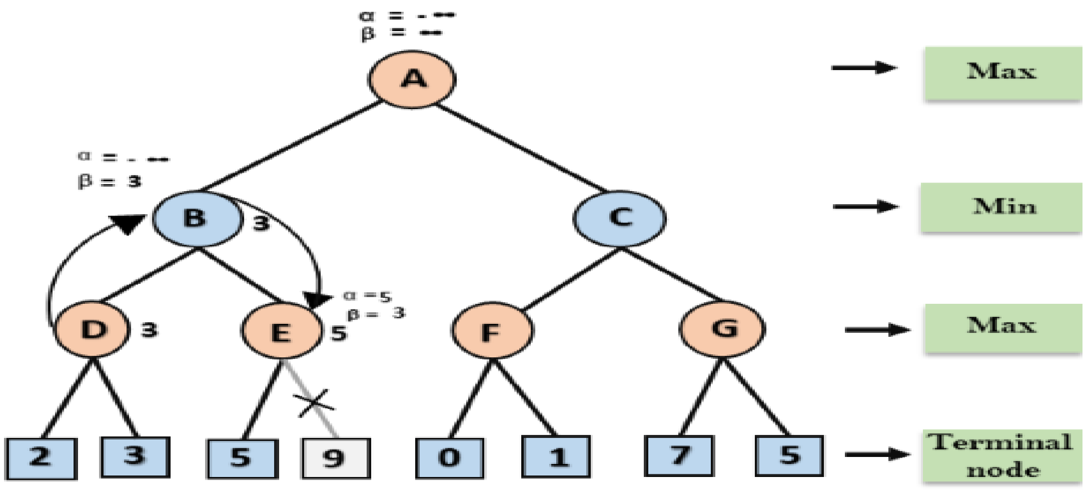
Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

Step 4:

At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

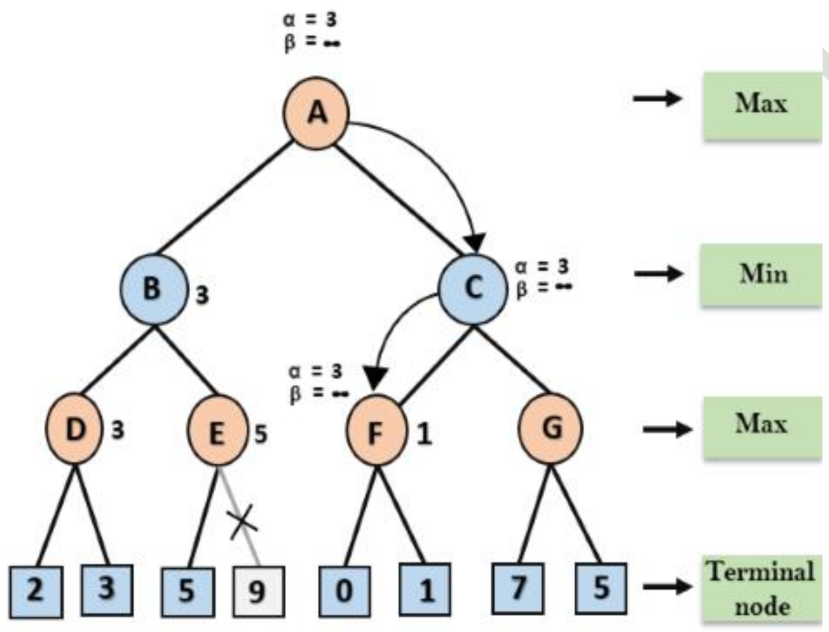


Step 5:

At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values will be passed on to node F.

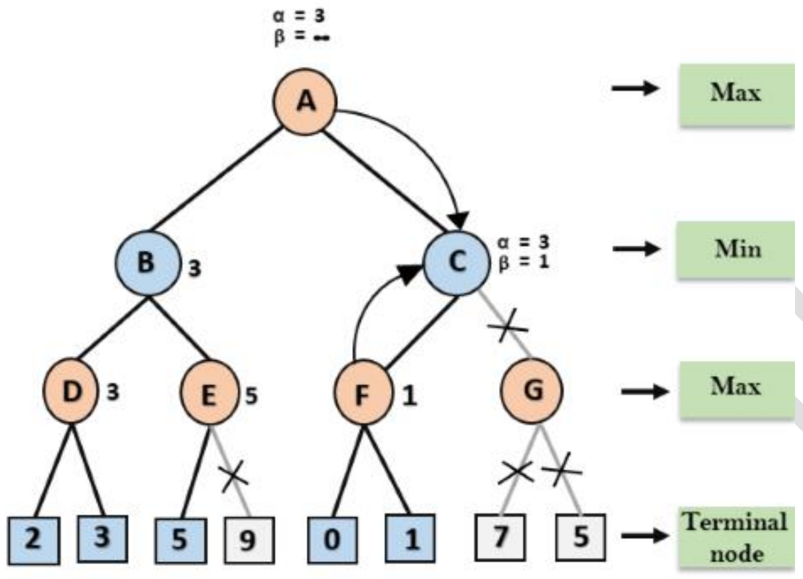
Step 6:

At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



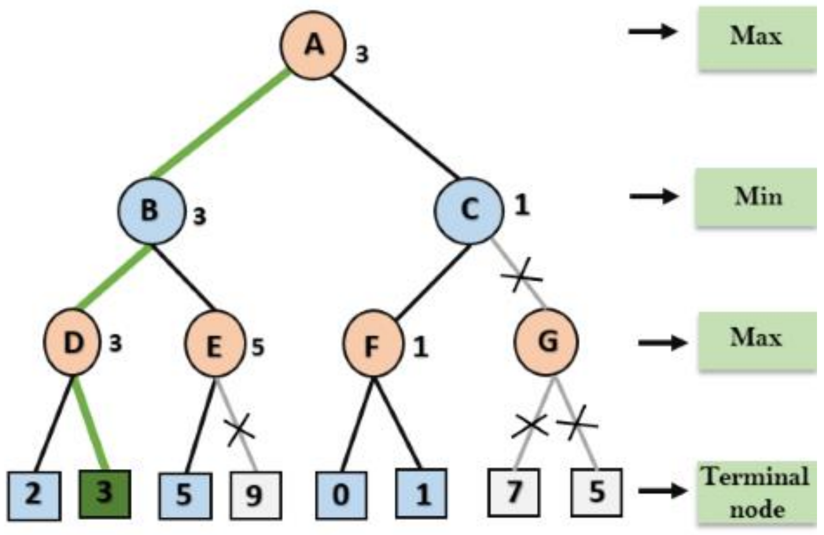
Step 7:

Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8:

C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.
It can be of two types:

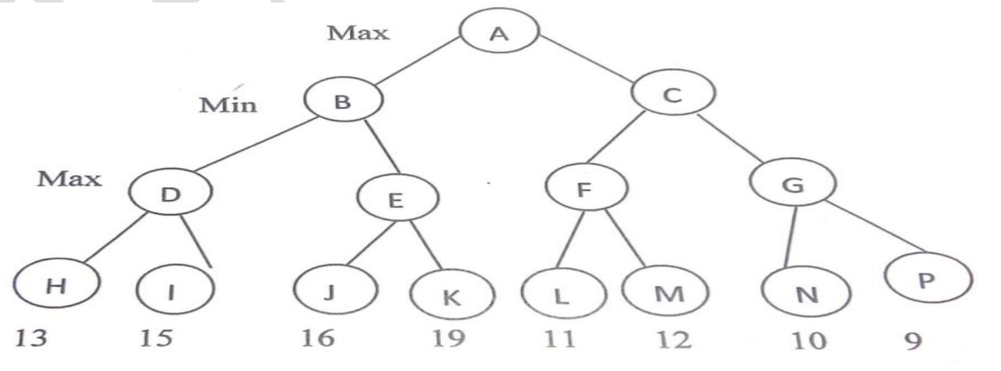
1. **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(b^d).

2. **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is O($b^{d/2}$).

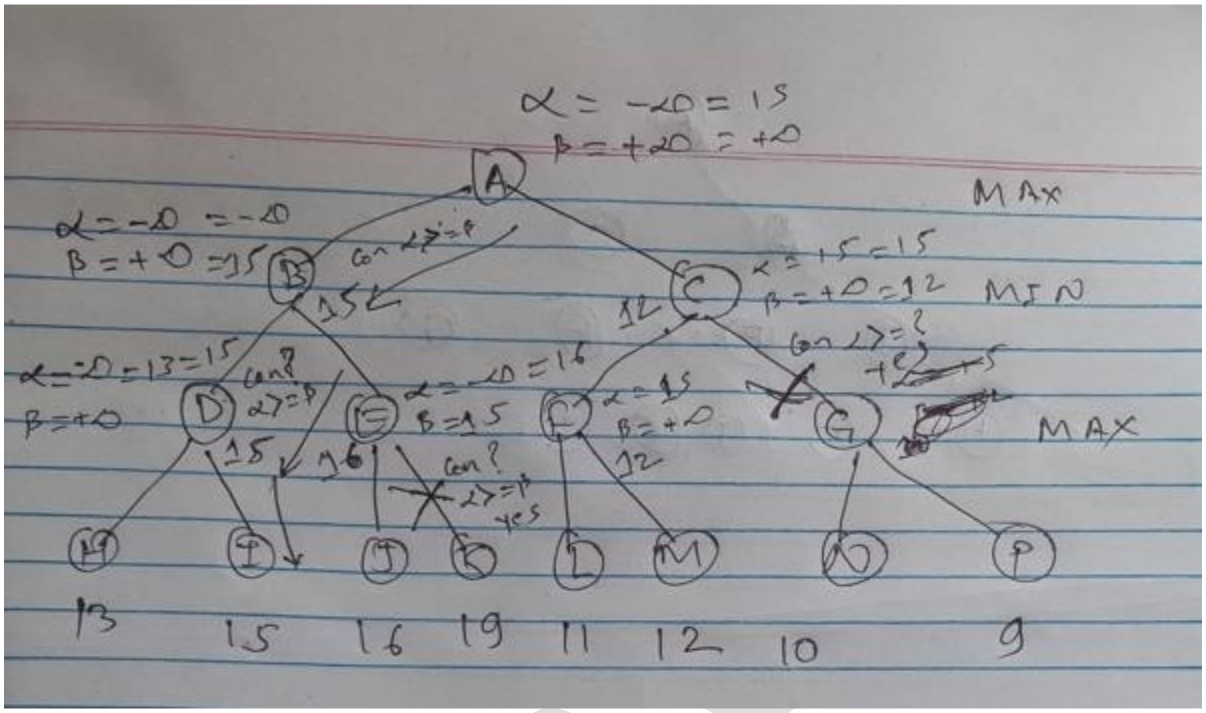
Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

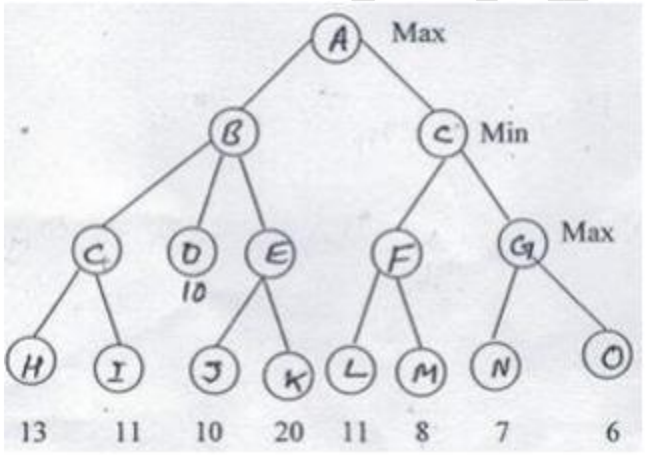
Example:
2076: Q9. Consider a following state space representing a game. Use minimax search to find solution and perform alpha-beta pruning, if exists.



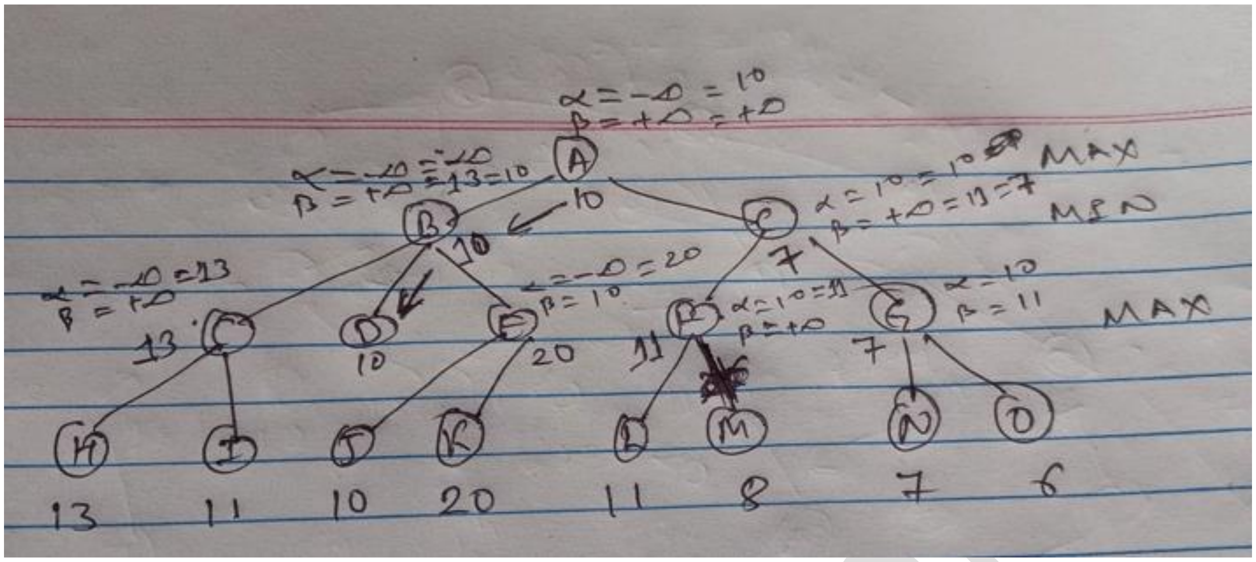
Solution:



2074: . What is the need of alphabeta pruning in game search? Given following search space with utility, perform mini-max search and identify alpha-beta cutoff if any. Play from perspective of max player first



Solution:



Constraint satisfaction problem: -

A constraint satisfaction problem (CSP) is a problem that requires its solution within some limitations/conditions also known as constraints. It consists of three variables V, D, C:

1) V is finite set of variables which stores the solution.
   i.e. V = {V1, V2, V3, ... , Vn}
2) D is a set of discrete values known as domain from which the solution is picked.
   i.e. D = {D1, D2, D3,.....,Dn}) for each variable.
3) C is a finite set of constraints that specifies allowable combination of values.
   i.e. Ci=(scope, Rel)

Each domain Di consists of a set of allowable values, {v1, . . . , vn} for variable Vi. Each constraint Ci consists of a pair (scope, rel), where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on Or A relation can be represented as an explicit list of all tuples of values that satisfy the constraint.

For example, if V1 and V2 two variables having the domain {A,B} respectively, then the constraint saying the two variables must have different values can be written as (V1,V2), C1= [(V1,V2), (V1 ≠ V2)] or as

Converting problems to CSPs

A problem to be converted to CSP requires the following steps:

- **Step 1: Create a variable set.**
- **Step 2: Create a domain set.**
- **Step 3: Create a constraint set with variables and domains (if possible) after considering the constraints.**
- **Step 4: Find an optimal solution.**

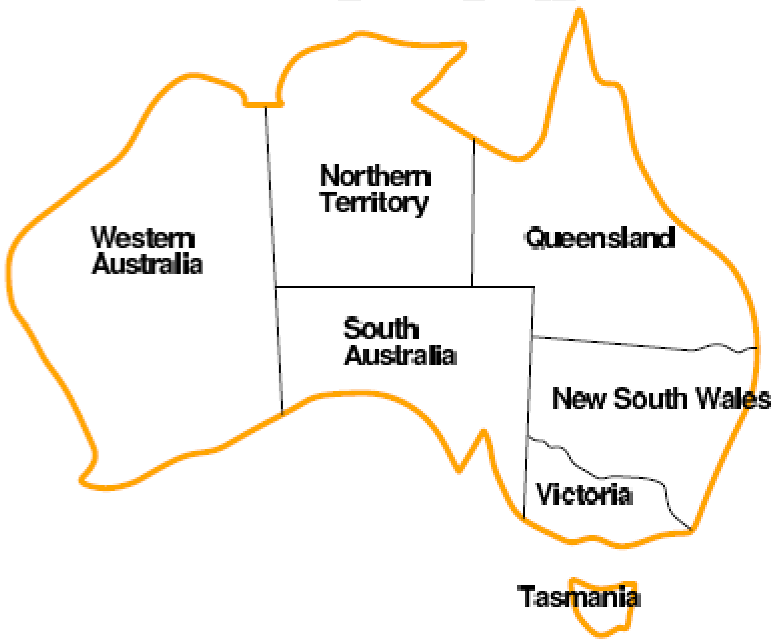
Popular Problems of CSP

The following problems are some of the popular problems that can be solved using CSP:

- **CryptArithmetic** (Coding alphabets to numbers.)
- **n-Queen** (In an n-queen problem, n queens should be placed in a nXn matrix such that no queen shares the same row, column or diagonal.)
- **Map Colouring** (Colouring different regions of map ensuring no adjacent regions have the same colour.)
- **Crossword** (Everyday puzzles appearing in newspapers.)
- **Sudoku** (A number grid.)
- **Latin Square Problem**

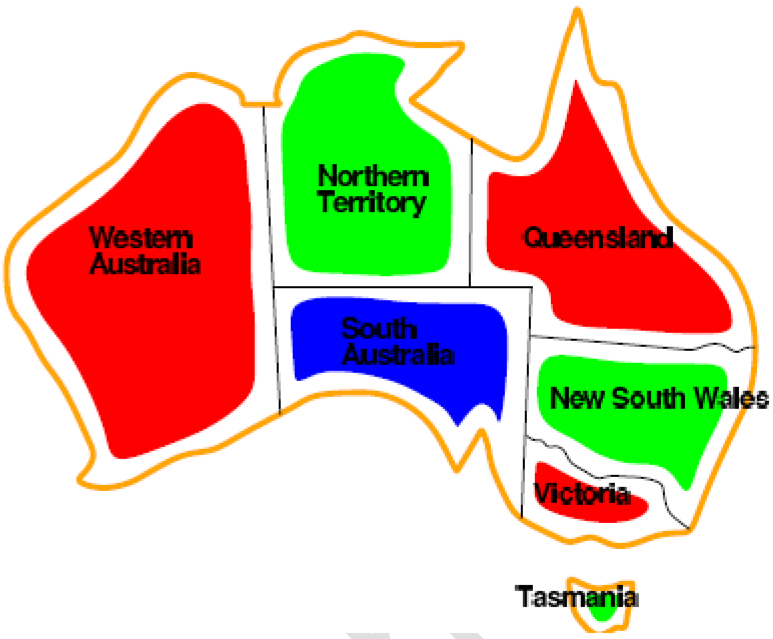
Example Problem (Map Colouring)

Consider a map colouring problem below. A map of Australia is given with its states and territories, as in Figure below, and that we are given the task of colouring each region either red, green, or blue in such a way that **no neighbouring regions have the same colour**.



Declare Variable Set, Domain Set and Constraint set

Variables WA, NT, SA, Q, NSW, V, T
Domains Di = {red, green, blue} for all variables
Constraints: adjacent regions must have different colours
e.g., WA ≠ NT —So (WA, NT) must be in {(red, green),(red, blue),(green, red), …}



Solutions are assignments satisfying all constraints, e.g.,
- e.g., {WA = red, NT = green, SA = blue, Q = red, NSW = green, V = red, T = green}

========================== End Unit 3===============================

Unit III: Problem Solving by Searching (9 Hrs.)

3.1. Definition, Problem as a state space search, Problem formulation, Well-defined problems,
3.2. Solving Problems by Searching, Search Strategies, Performance evaluation of search techniques
3.3. Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Bidirectional Search
3.4. Informed Search: Greedy Best first search, A* search, Hill Climbing, Simulated Annealing
3.5. Game playing, Adversarial search techniques, Mini-max Search, Alpha-Beta Pruning.
3.6. Constraint Satisfaction Problems

===