Problem Solving and Searching Techniques

Unit 9.2

Introduction

Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution.

Problem-solving methods divide into special purpose and general purpose.

A special-purpose method is tailor-made for a particular problem and often exploits very specific features of the situation in which the problem is embedded. In contrast, a general-purpose method is applicable to a wide variety of problems.

One general-purpose technique used in AI is means-end analysis—a step-by-step, or incremental, reduction of the difference between the current state and the final goal.

Four general steps in problem solving:

– Goal formulation

  – What are the successful world states

– Problem formulation

  – What actions and states to consider given the goal

– Search

  – Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.

– Execute

  – Give the solution perform the actions.

Problem formulation

A problem is defined by:

– An initial state: State from which agent start

– Successor function: Description of possible actions available to the agent.

– Goal test: Determine whether the given state is goal state or not

– Path cost: Sum of cost of each path from initial state to the given state.

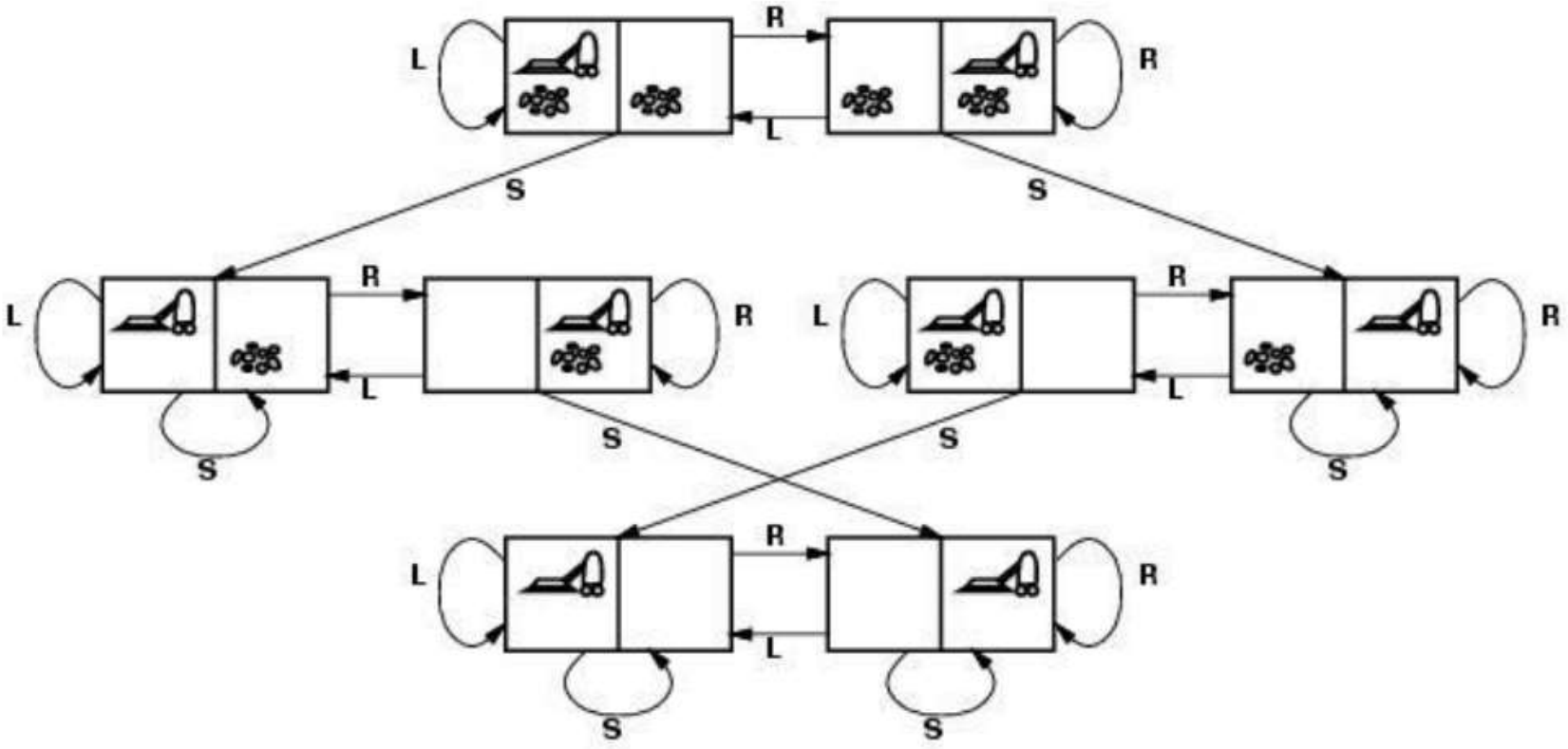
A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.

State Space representation

The state space is commonly defined as a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.

A solution is a path from the initial state to a goal state.

State Space representation of Vacuum World Problem:



States?? two locations with or without dirt: 2 x 2^2=8 states.

Initial state?? Any state can be initial

Actions?? {Left, Right, Suck}

Goal test?? Check whether squares are clean.

Path cost?? Number of actions to reach goal.

Water leakage problem

If hall _wet and kitchen_dry

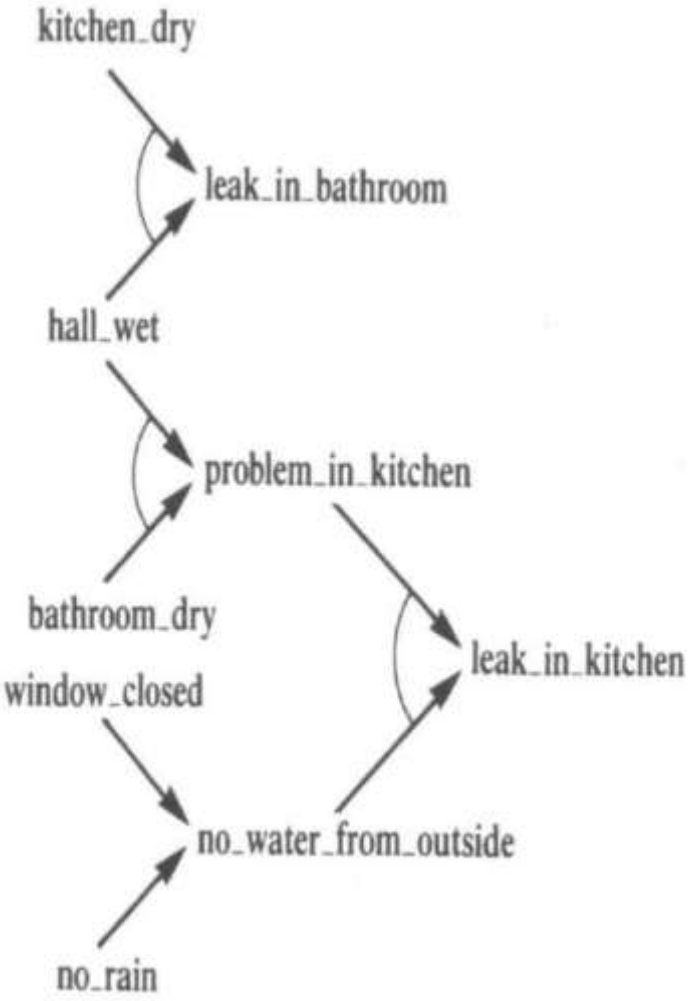
then leak_in_bathroom

If hall_wet and bathroom_dry

then problem_in_kitchen

If window_closed or no_rain

then no_water_from_outside



Production System

A production system (or production rule system) is a computer program typically used

to provide some form of artificial intelligence, which consists primarily of a set of rules

about behavior.

These rules, termed productions, are a basic representation found useful in automated planning, expert systems and action selection.

A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system.

Productions consist of two parts: a sensory precondition (or "IF" statement) and an action (or "THEN").

If a production's precondition matches the current state of the world, then the production is said to be triggered. If a production's action is executed, it is said to have fired.

A production system also contains a database, sometimes called working memory, which maintains data about current state or knowledge, and a rule interpreter.

The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

The underlying idea of production systems is to represent knowledge in the form of condition-action pairs called production rules:

If the condition C is satisfied then the action A is appropriate.

Types of production rules

Situation-action rules

If it is raining then open the umbrella.

Inference rules

If Cesar is a man then Cesar is a person

Production system is also called ruled-based system

Architecture of Production System:

Short Term Memory:

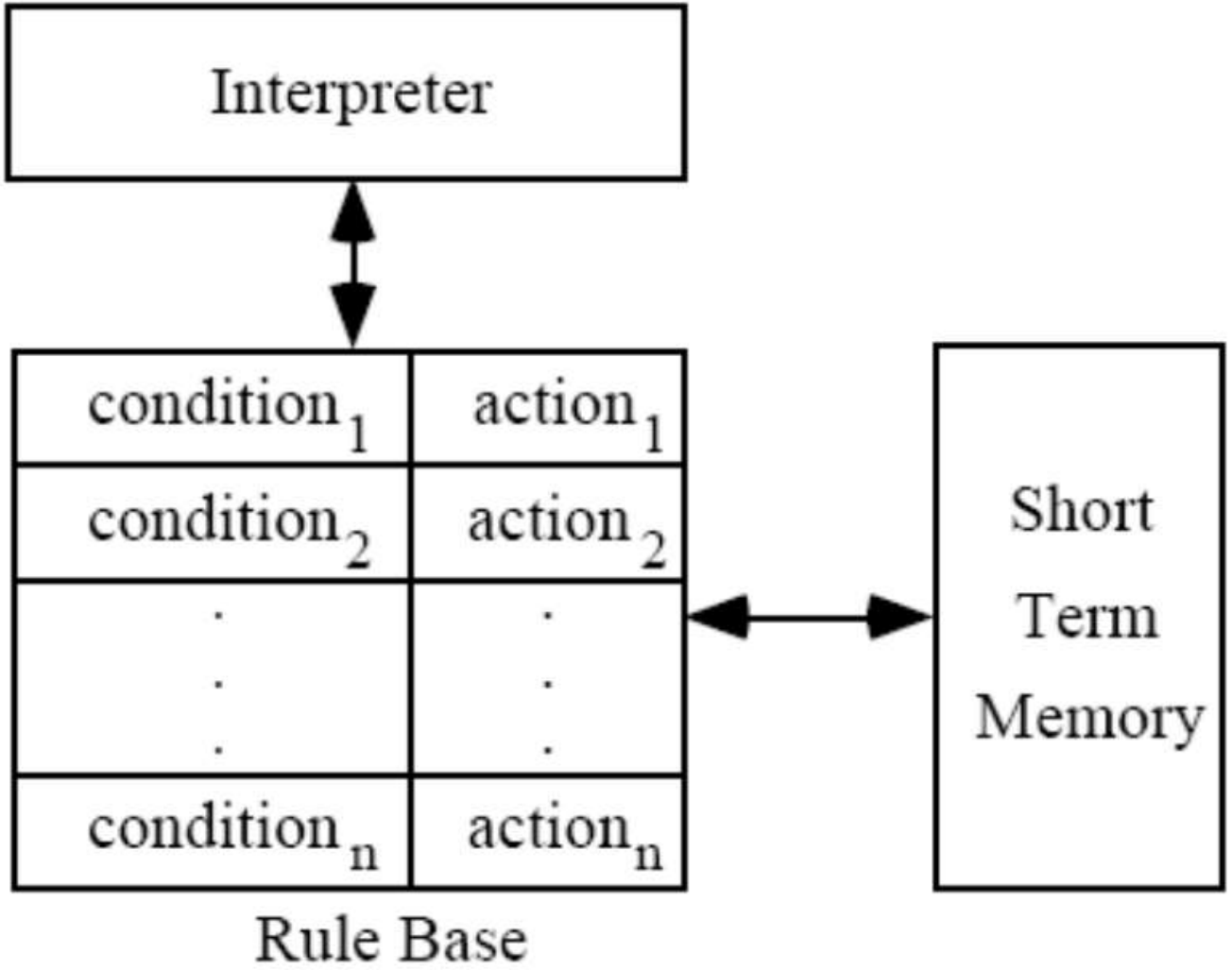
- Contains the description of the current state.

Set of Production Rules:

- Set of condition-action pairs and defines a single chunk of problem solving knowledge.

Interpreter:

- A mechanism to examine the short term memory and to determine which rules to fire (According to some strategies such as DFS, BFS, Priority, first-encounter etc)


Interpreter
condition 1
action 1
condition 2
action 2
condition n
action n
Short
Term
Memory
Rule Base

The execution of a production system can be defined as a series of recognize-act cycles:

Match –memory contain matched against condition of production rules, this produces a subset of production called conflict set.

Conflict resolution –one of the production in the conflict set is then selected, Apply the rule.

Consider an example:

Problem: Sorting a string composed of letters a, b & c.

Short Term Memory: cbaca

Production Set:

1. ba → ab
2. ca → ac.
3. cb → bc

Interpreter: Choose one rule according to some strategy.

Iteration #	Memory	Conflict Set	Rule fired
0	cbaca	1, 2, 3	1
1	cabca	2	2
2	acbca	2, 3	2
3	acbac	1, 3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	ø	halt

Production System: The water jug problem

Problem: There are two jugs, a 4-gallon one and a 3-gallon one. Neither jug has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly n (0, 1, 2, 3, 4) gallons of water into one of the two jugs ?

Solution Paradigm:

- build a simple production system for solving this problem.

- represent the problem by using the state space paradigm.

State = (x, y); where: x represents the number of gallons in the 4-gallon jug; y represents the number of gallons in the 3-gallon jug. x ε{0, 1, 2, 3, 4} and y ε{0, 1, 2, 3}.

The initial state represents the initial content of the two jugs. For instance, it may be (2, 3), meaning that the 4-gallon jug contains 2 gallons of water and the 3-gallon jug contains three gallons of water.

The goal state is the desired content of the two jugs.

The left hand side of a production rule indicates the state in which the rule is applicable and the right hand side indicates the state resulting after the application of the rule.

For instance;

(x, y) such that $x < 4 \rightarrow (4, y)$ represents the production
If the 4-gallon jug is not full then fill it from the pump.

The rule base contains the following production rules:

1. (x, y) such that $x < 4 \rightarrow (4, y)$; Fill the 4-gallon jug from pump
2. (x, y) such that $y < 3 \rightarrow (x, 3)$; Fill the 3-gallon jug from pump
3. (x, y) such that $x > 0 \rightarrow (0, y)$; Empty the 4-gallon jug on the ground
4. (x, y) such that $y > 0 \rightarrow (x, 0)$; Empty the 3-gallon jug on the ground
5. (x, y) such that $x + y \geq 4, x < 4, y > 0 \rightarrow (4, y - (4 - x))$
   ; Completely fill the 4-gallon jug from the 3-gallon jug
6. (x, y) such that $x + y \geq 3, x > 0, y < 3 \rightarrow (x - (3 - y), 3)$
   ; Completely fill the 3-gallon jug from the 4-gallon jug
7. (x, y) such that $x + y \leq 4, y > 0 \rightarrow (x+y, 0)$
   ; Empty the 3-gallon jug into the 4-gallon jug
8. (x, y) such that $x + y \leq 3, x > 0 \rightarrow (0, x + y)$
   ; Empty the 4-gallon jug into the 3-gallon jug

The short term memory contains the current state (x, y).

Let us consider the initial situation (0, 0) and the goal situation (n, 2)

short term memory : (0, 0)

1. Match: 1, 2. 2. Conflict resolution: select rule 2. 3. Apply the rule

short term memory becomes (0, 3)

1. Match: 1, 4, 7 2. Conflict resolution: select rule 7 3. Apply the rule

short term memory becomes (3, 0)

1. Match:1, 2, 3, 6, 8 2. Conflict resolution: select rule 2 3. Apply the rule

short term memory becomes (3, 3)

1. Match: 1, 3, 4, 5 2. Conflict resolution: select rule 5 3. Apply the rule

short term memory becomes (4, 2) Goal achieved

Solution:

The sequence of the applied rules:

Fill the 3-gallon jug from pump

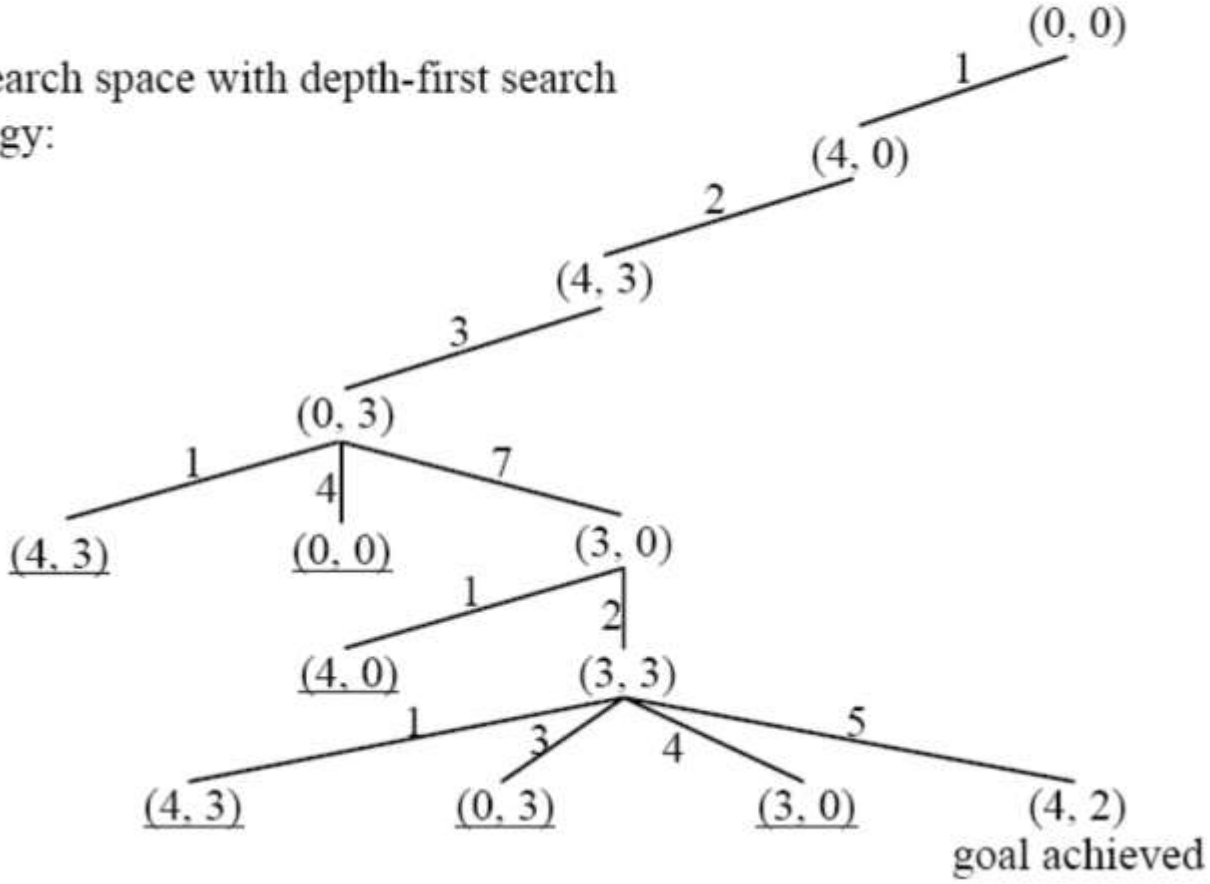
Empty the 3-gallon jug into the 4-gallon jug

Fill the 3-gallon jug from pump

Fill the 4-gallon jug from the 3-gallon jug

The Water Jug Problem: Representation

the search space with depth-first search strategy:



Constraint Satisfaction Problem

A Constraint Satisfaction Problem is characterized by:

- a set of variables {x1, x2, .., xn},
- for each variable xi a domain Di with the possible values for that variable, and
- a set of constraints, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied. Means that, we must find a value for each of the variables that satisfies all of the constraints.

A CS problem can easily be stated as a sentence in first order logic, of the form: (exist x1)..(exist xn) (D1(x1) & .. Dn(xn) => C1..Cm)

Constraints

A constraint is a relation between a local collection of variables.

The constraint restricts the values that these variables can simultaneously have.

For example, all-diff(X1, X2, X3). This constraint says that X1, X2, and X3 must take on different values. Say that {1,2,3} is the set of values for each of these variables then:

X1=1, X2=2, X3=3 OK X1=1, X2=1,X3=3 NO

The constraints are the key component in expressing a problem as a CSP.

The constraints are determined by how the variables and the set of values are chosen.

Each constraint consists of;

1. A set of variables it is over.
2. A specification of the sets of assignments to those variables that satisfy the constraint.

The idea is that we break the problem up into a set of distinct conditions each of which have to be satisfied for the problem to be solved.

Example: In N-Queens: Place N queens on an N x N chess board so that queen can attack any other queen.

No queen can attack any other queen.

Given any two queens Qi and Qj they cannot attack each other.

Now we translate each of these individual conditions into a separate constraint.

Qi cannot attack Qj(i ≠j)

Qi is a queen to be placed in column i, Qj is a queen to be placed in column j.

The value of Qi and Qj are the rows the queens are to be placed in.

Note the translation is dependent on the representation we chose.

Queens can attack each other,

1. Vertically, if they are in the same column---this is impossible as Qi and Qj are placed in different columns.

2. Horizontally, if they are in the same row---we need the constraint Qi≠Qj.

3. Along a diagonal, they cannot be the same number of columns apart as they are rows apart: we need the constraint $|i-j| \neq |Qi-Qj|$ ($|\ |$ is absolute value)

Representing the Constraints;

1. Between every pair of variables (Qi,Qj) ($i \neq j$), we have a constraint Cij.

2. For each Cij, an assignment of values to the variables Qi= A and Qj= B, satisfies this constraint if and only if;

$A \neq B$

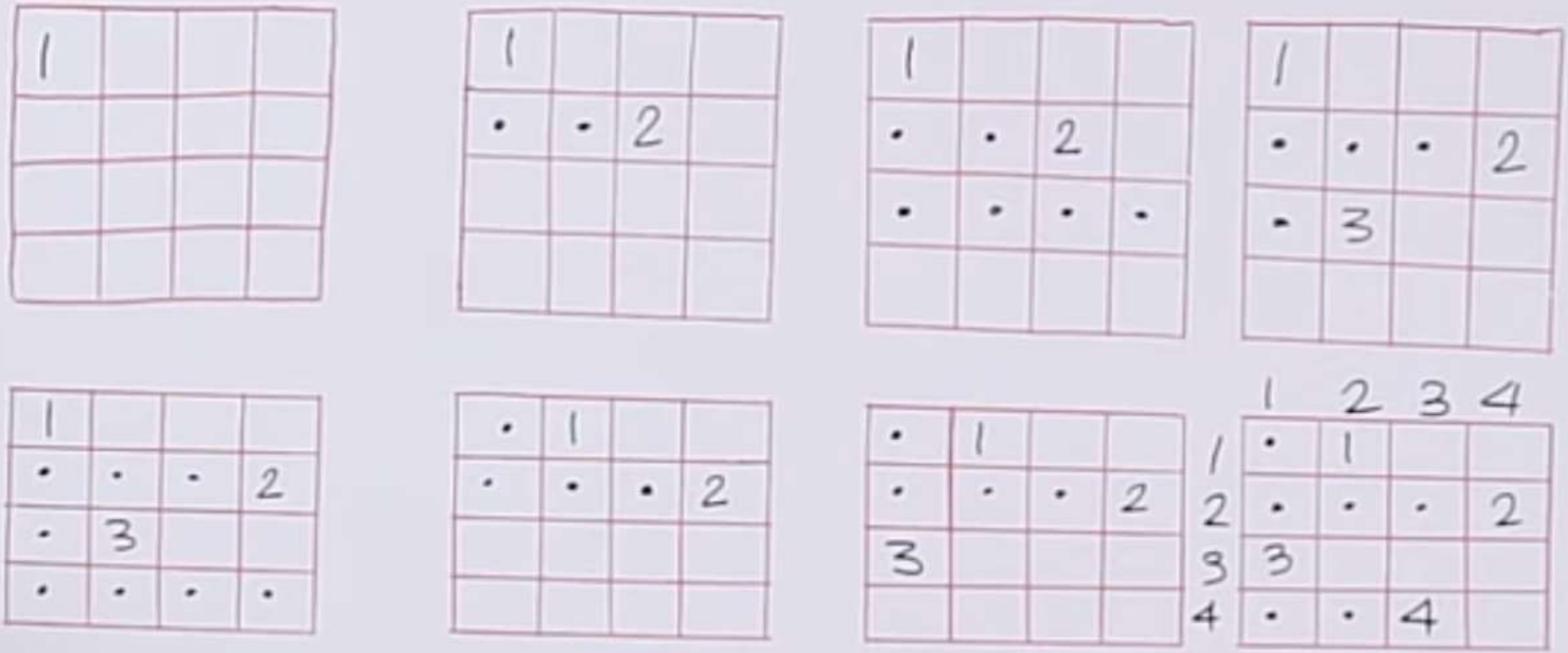
$|A-B| \neq |i-j|$

Solutions:

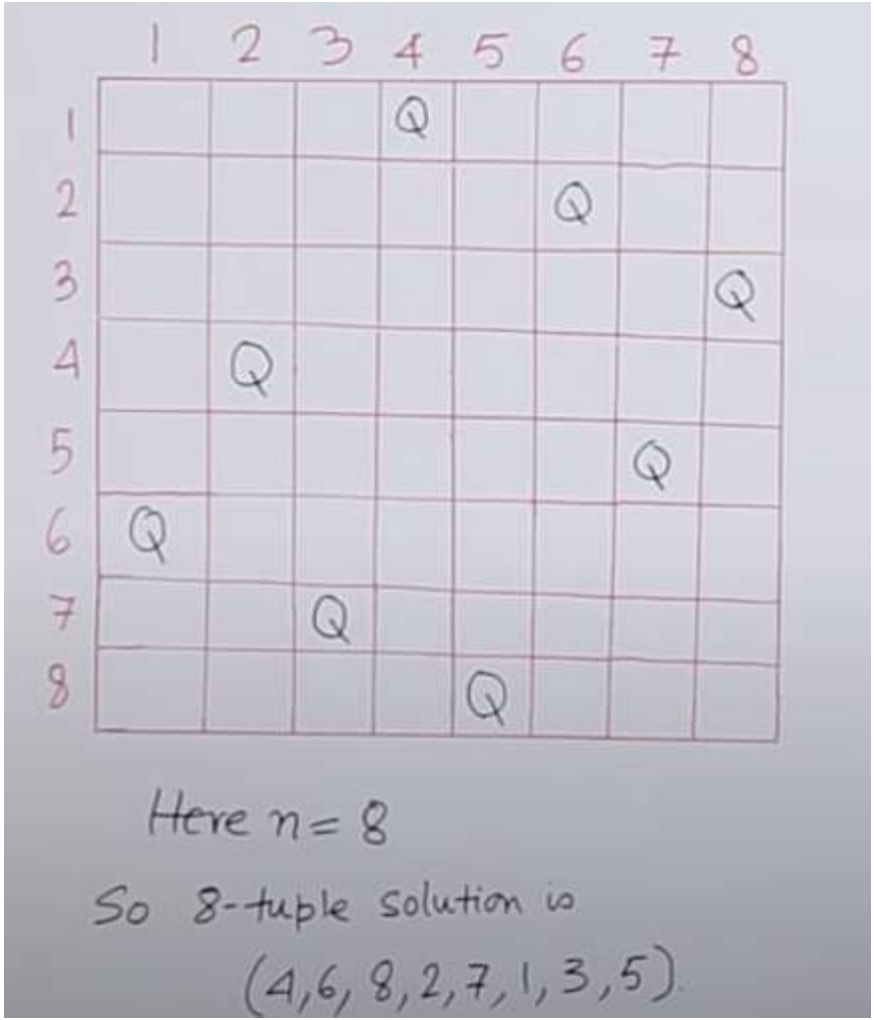
A solution to the N-Queens problem will be any assignment of values to the variables Q1,...,QN that satisfies all of the constraints.

Constraints can be over any collection of variables. In N-Queens we only need binary constraints---constraints over pairs of variables.

If $n=4$, then



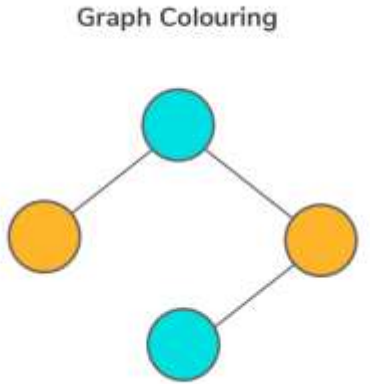
4-tuple answer will be $(2, 4, 1, 3)$.


1 2 3 4 5 6 7 8
1
2
3
4
5
6
7
8
Q
Here n = 8
So 8-tuple solution is
(4,6,8,2,7,1,3,5).

Graph Coloring Problem

Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such tha[illegible]me color is caller Graph Colouring.

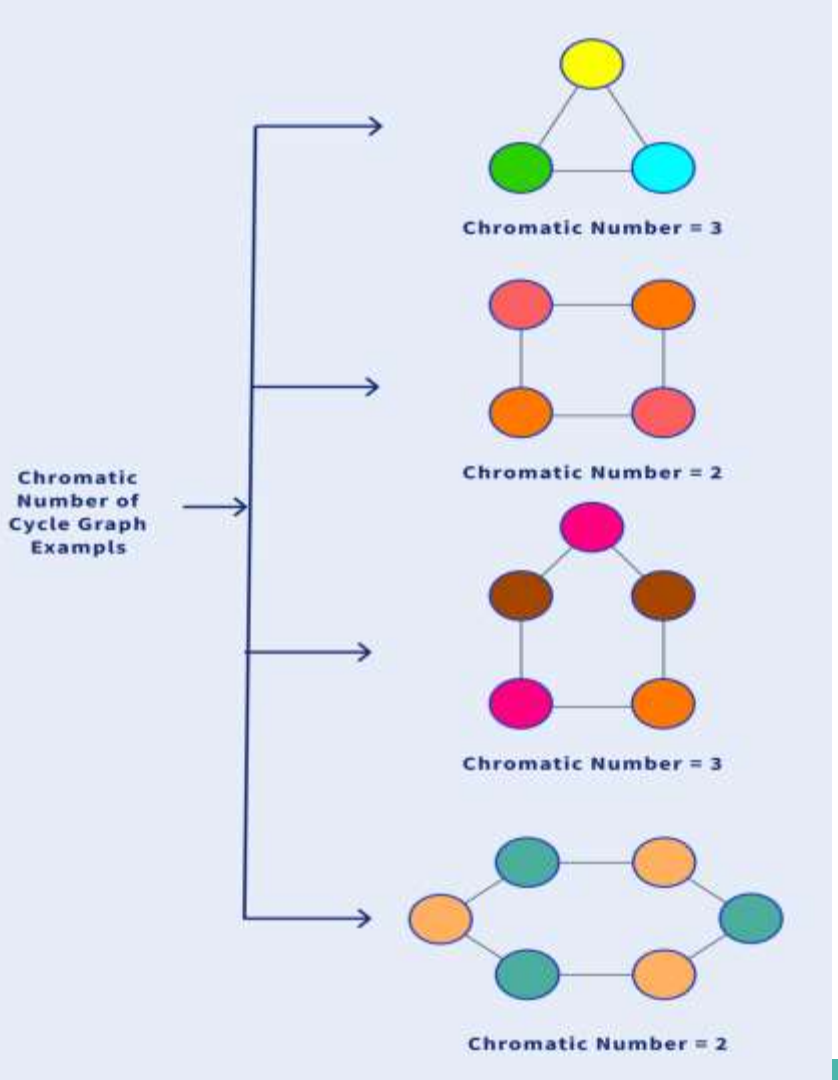
This is also known as vertex coloring.



Chromatic Number: The smallest number of colours needed to colour a graph G is called its chromatic number.

For example, in the above image, vertices can be coloured using a minimum of 2 colours.

Hence the chromatic number of the graph is 2.


Chromatic Number of Cycle Graph Exampls
Chromatic Number = 3
Chromatic Number = 2
Chromatic Number = 3
Chromatic Number = 2

Applications of Graph Colouring:

- Map Coloring
- Scheduling the tasks
- Preparing Time Table
- Assignment
- Conflict Resolution
- Sudoku

Approach 1: Brute Force

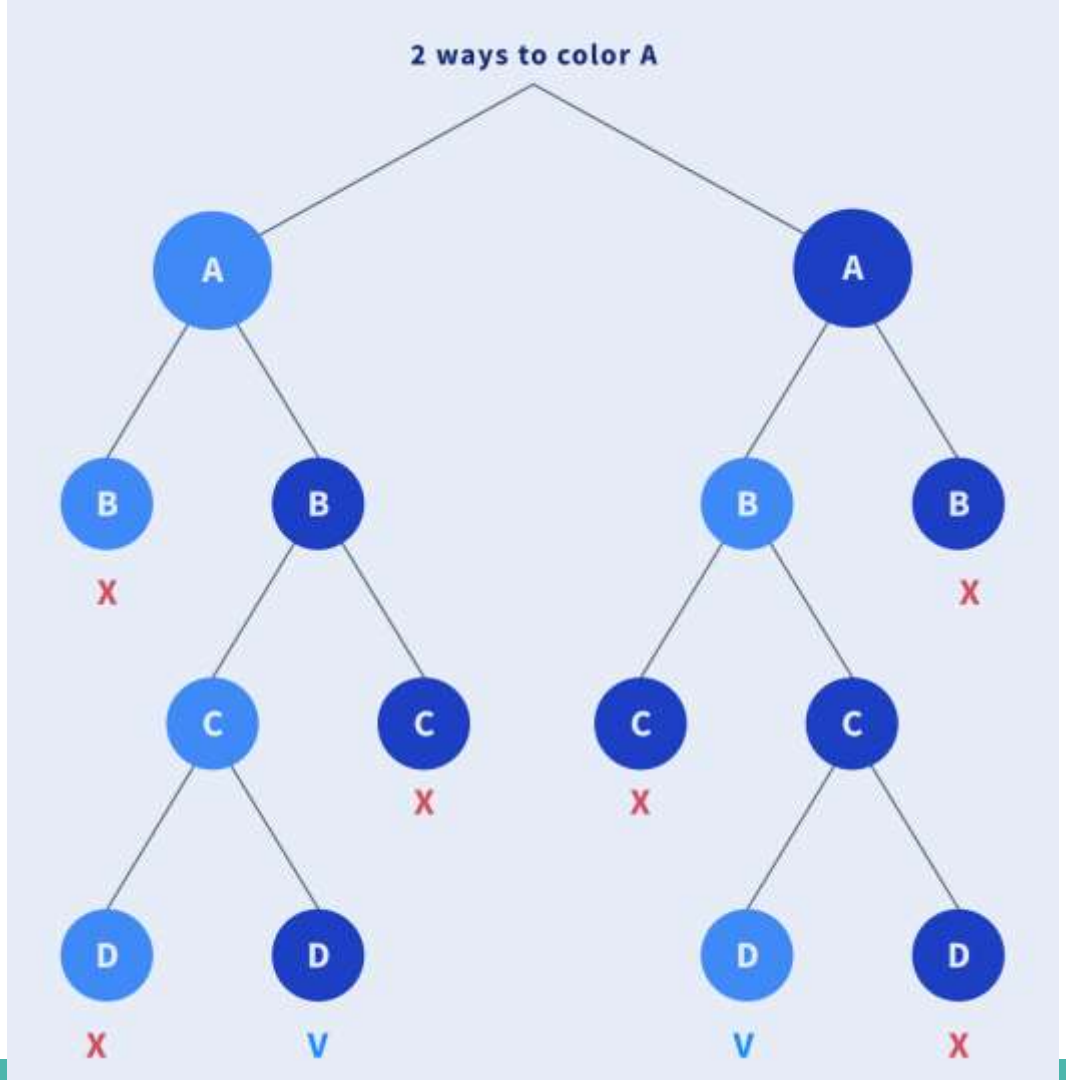
- The simplest approach to solve this problem would be to generate all possible combinations (or configurations) of colours.
- After generating a configuration, check if the adjacent vertices have the same colour or not. If the conditions are met, add the combination to the result and break the loop.
- Since each node can be coloured by using any of the M colours, the total number of possible colour configurations are mV. The complexity is exponential which is very huge.

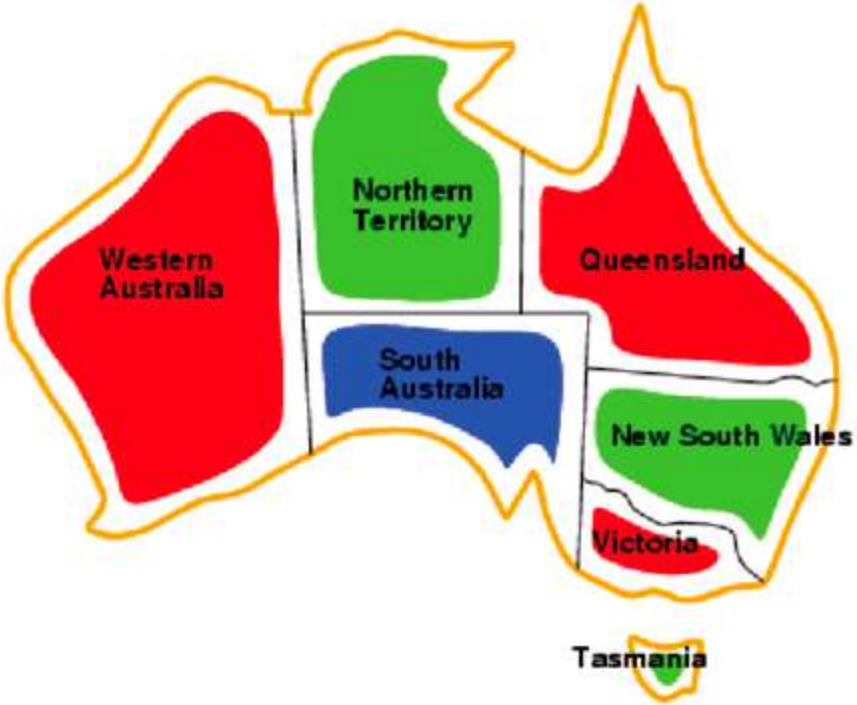
Approach 2: Backtracking

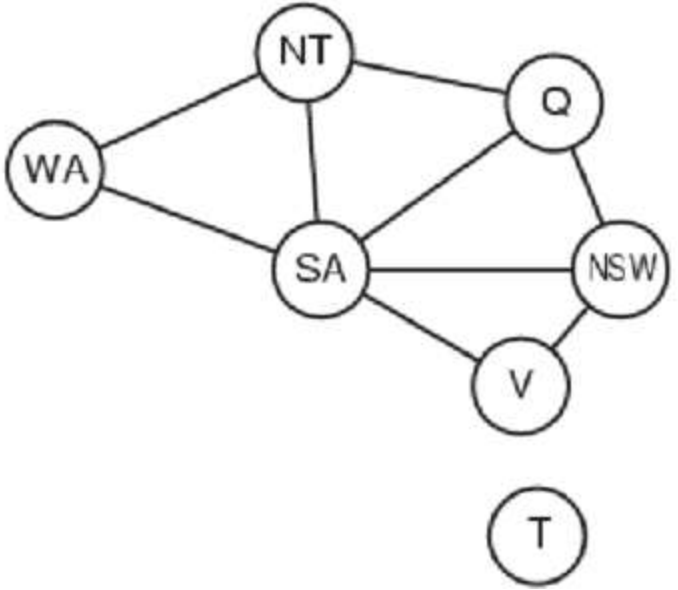
In the previous approach, trying and checking every possible combination was tedious and had an exponential time complexity.

Some of the permutation calculations were unnecessary but were calculated again and again. Therefore, the idea is to use a backtracking approach to solve the problem.

In this approach, the idea is to color a vertex and while coloring any adjacent vertex, choose a different color. Similarly, color every possible vertex following the restrictions, till any further vertex is left coloring. In any case, if all adjacent


2 ways to color A
A
A
B
B
B
B
X
X
C
C
C
C
X
X
D
D
D
D
X
V
V
X


Western Australia
Northern Territory
Queensland
South Australia
New South Wales
Victoria
Tasmania


NT
Q
WA
SA
NSW
V
T

Searching

Figure below contains a representation of a map. The nodes represent cities, and the links represent direct road connections between cities. The number associated to a link represents the length of the corresponding road.

The search problem is to find a path from a city S to a city G

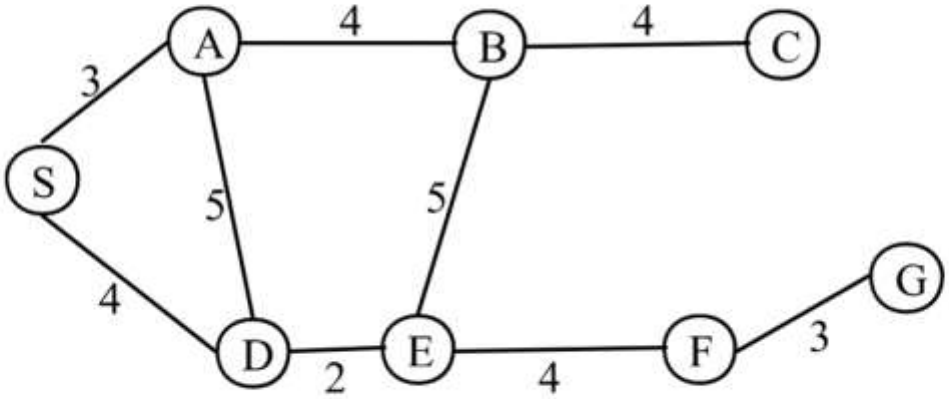


Figure : A graph representation of a map

This problem will be used to illustrate some search methods.

Search problems are part of a large number of real world applications:

- VLSI layout
- Path planning
- Robot navigation etc.

There are two broad classes of search methods:

- **uninformed (or blind) search methods;**
- **heuristically informed search methods.**

In the case of the uninformed search methods, the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie.

In the case of the heuristically informed search methods, one uses domain-dependent (heuristic) information in order to search the space more efficiently.

Measuring problem Solving Performance

The performance of a search algorithm can be evaluated in four ways:

- Completeness: An algorithm is said to be complete if it definitely finds solution to the problem, if exist.
- Time Complexity: How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes explained.
- Space Complexity: How much space is used by the algorithm? Usually measured in terms of the maximum number of nodes in memory at a time.
- Optimality/ Admissibility: If a solution is found, is it guaranteed to be a optimal one? For example, is it the one with the minimum cost?

Time and space complexity is measured in terms of

b - maximum branching factor (number of successor of any node) of the search tree

d - depth of least cost solution

m - maximum length of any path in the space

State space search

In Artificial Intelligence a state space consists of the following elements,

1. A (possibly infinite) set of states

1.1. Out of the possible states, one state represents the start state that is the initial state of the problem.

1.2. Each state represents some configuration reachable from the start state

1.3. Out of the possible states, some states may be goal states (solutions)

2. A set of rules,

2.1. Applying a rule to the current state, transforms it to another or a new state in the state space

2.2 All operators may not be applicable to all states in the state space

State spaces are used extensively in Artificial Intelligence (AI) to represent and solve problems.

Uninformed Search Strategies

Uninformed search (or **blind search**)
– Strategies have no additional information about states beyond that provided in the problem definition

Informed (or **heuristic**) search
– Search strategies know whether one state is more promising than another

Uninformed strategies (defined by order in which nodes are expanded):

– Breadth-first search
– Uniform-cost search
– Depth-first search
– Depth-limited search
– Iterative deepening search – Bi-directional search

State space search examples

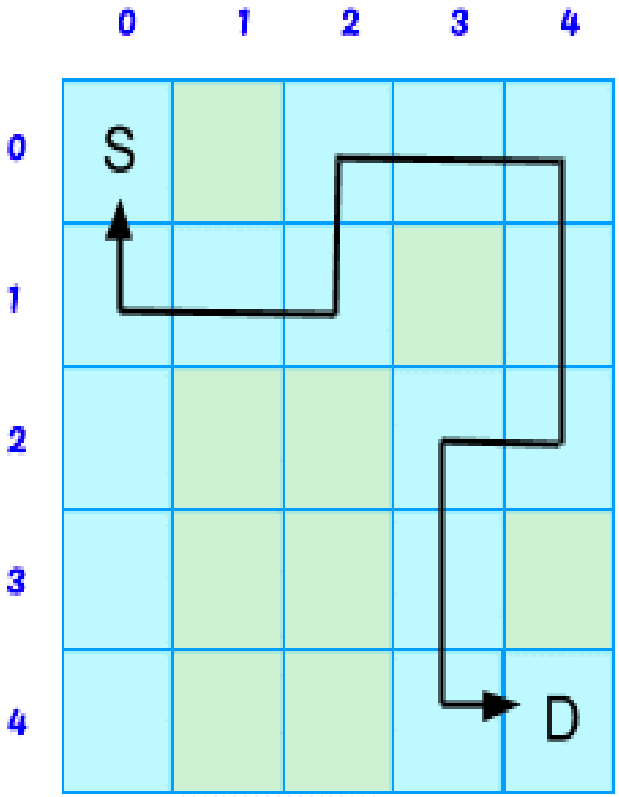
Example: Maze

A maze problem can be represented as a state-space

- Each state represents “where you are” that is the current position in the maze
- The start state or initial state represents your starting position
- The goal state represents the exit from the maze

Rules (for a rectangular maze) are: move north, move south, move east, and move west

- Each rule takes you to a new state (maze location)
- Rules may not always apply, because of walls in the maze



Example 2. The 15 Puzzle

Start state:

3	10	13	7
9	14	6	1
4		15	2
11	8	5	12

Goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The start state is some (almost) random configuration of the tiles and the goal state is as shown.

State Space Search Rules are

- Move empty space up
- Move empty space down
- Move empty space right
- Move empty space left

These Rules apply if empty space is not against the edge.

General State space search problem

1. First, select some way to represent states in the given problem in an unambiguous way.

2. Next, formulate all actions or operators that can be performed in states, including their preconditions and effects.

- Actions or operates are called PRODUCTION RULES

3. Represent the initial state or states of the problem.

4. Formulate precisely when a state satisfies the goal of our problem.

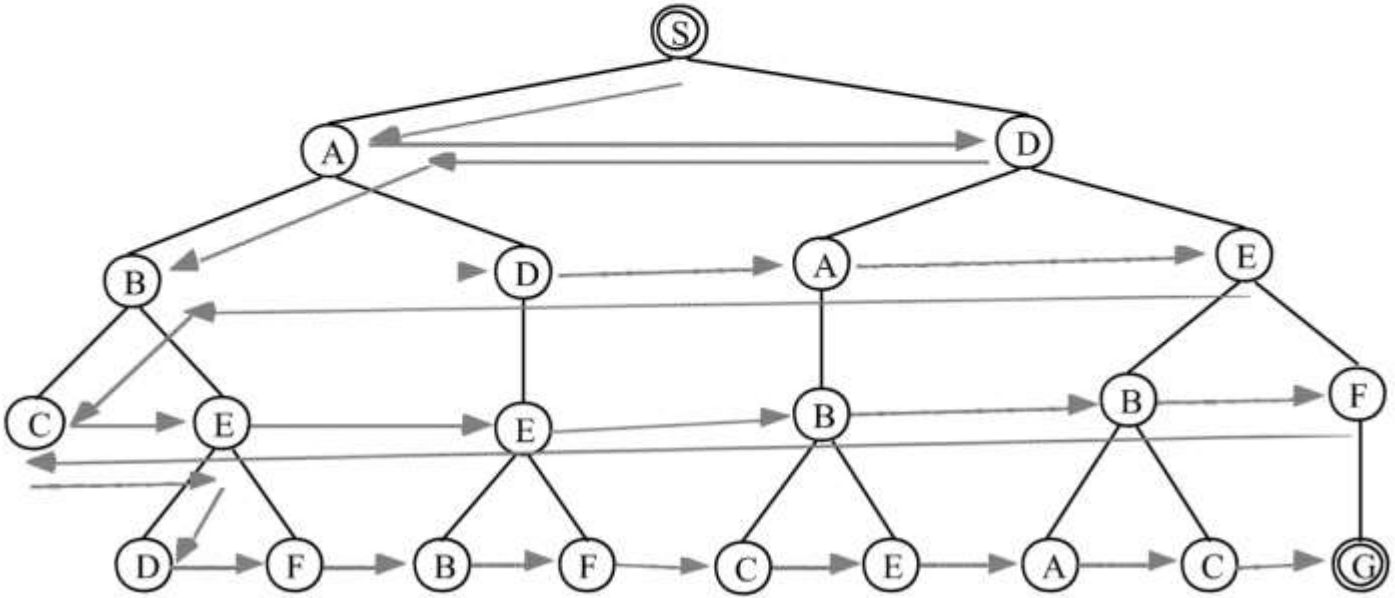
5. Activate the production rules on the initial state and its descendants, until a goal state is reached

Breadth First Search

All nodes are expended at a given depth in the search tree before any nodes at the next level are expanded until the goal reached.

Expand *shallowest* unexpanded node. *fringe* is implemented as a FIFO queue

Constraint: Do not generate as child node if the node is already parent to avoid more loop.



Fringe: it is the collection of nodes that have been generated but not yet expanded.

The set of all leaf nodes available for expansion at any given point is called the frontier.

Implementation: Fringe / Frontier is a FIFO queue.

BFS Evaluation:

Completeness:

- *Does it always find a solution if one exists?*
- YES
  - If shallowest goal node is at some finite depth d and If b is finite

Time complexity:

- Assume a state space where every state has b successors.

- root has b successors, each node at the next level has again b successors (total b^2), …
- Assume solution is at depth d
- Worst case; expand all except the last node at depth d
- Total no. of nodes generated:

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

Space complexity:

- Each node that is generated must remain in memory
- Total no. of nodes in memory:

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

Optimal (i.e., admissible):

- if all paths have the same cost. Otherwise, not optimal but finds solution with shortest path length (shallowest solution). If each path does not have same path cost shallowest solution may not be optimal

Two lessons:

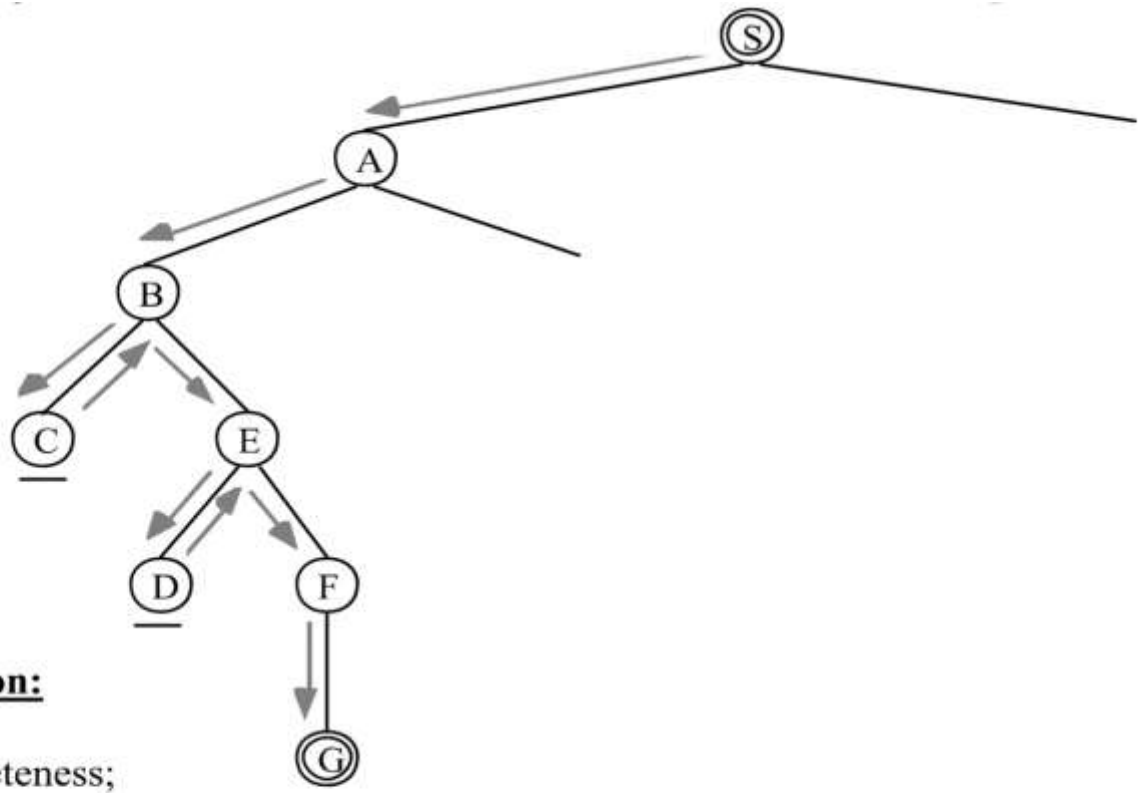
- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH2	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	107	19 minutes	10 gigabytes
8	109	31 hours	1 terabyte
10	1011	129 days	101 terabytes
12	1013	35 years	10 petabytes
14	1015	3523 years	1 exabyte

Depth First Search

Looks for the goal node among all the children of the current node before using the sibling of this node i.e. expand deepest unexpanded node.

Here Fringe is implemented as Stack or (LIFO queue)



DFS Evaluation:

Completeness;

- *Does it always find a solution if one exists?*
- NO
  - If search space is infinite and search space contains loops then DFS may not find solution.

Time complexity;

- Let m is the maximum depth of the search tree. In the worst case Solution may exist at depth m.
- root has *b* successors, each node at the next level has again b successors (total b^2), …
- Worst case; expand all except the last node at depth *m*
- Total no. of nodes generated:

$$b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots b^m = \mathbf{O(b^m)}$$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:

$$1 + b + b + b + \ldots\ldots\ldots\ldots\ldots\ldots \text{ b m times} = \mathbf{O(bm)}$$

Optimal (i.e., admissible):

- DFS expand deepest node first, if expands entire let sub-tree even if right sub-tree contains goal nodes at levels 2 or 3. Thus we can say DFS may not always give optimal solution.

Uniform Cost Search:

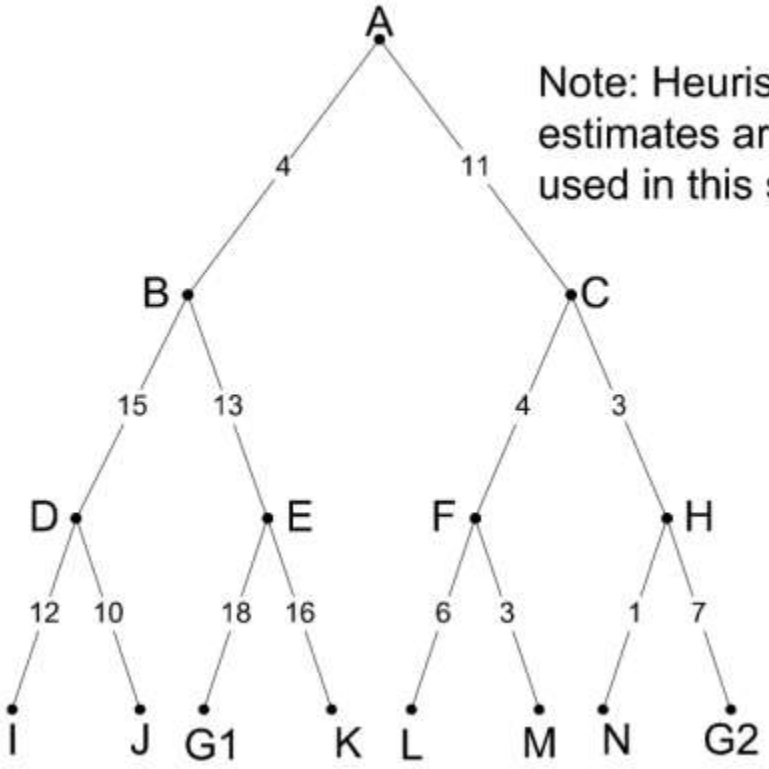
Uniform-cost search (**UCS**) is modified version of BFS to make optimal. It is basically a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node. The uniform-cost search is **complete** and **optimal** if the cost of each step exceeds some positive bound ε.

Does not care about the number of steps, only care about total cost.

- Complete? Yes, if step cost $\geq \varepsilon$ (small positive number).
- Time? Maximum as of BFS
- Space? Maximum as of BFS.
- Optimal? Yes

Consider an example:



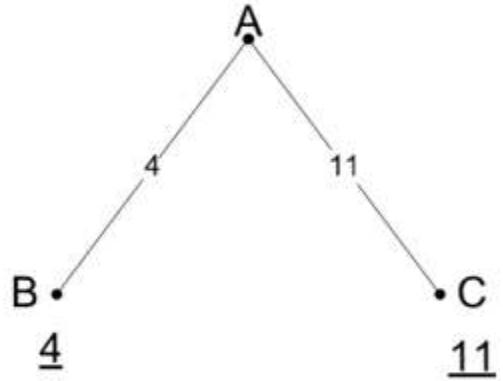
Note: Heurisitic estimates are not used in this search!

- Start with root node A.

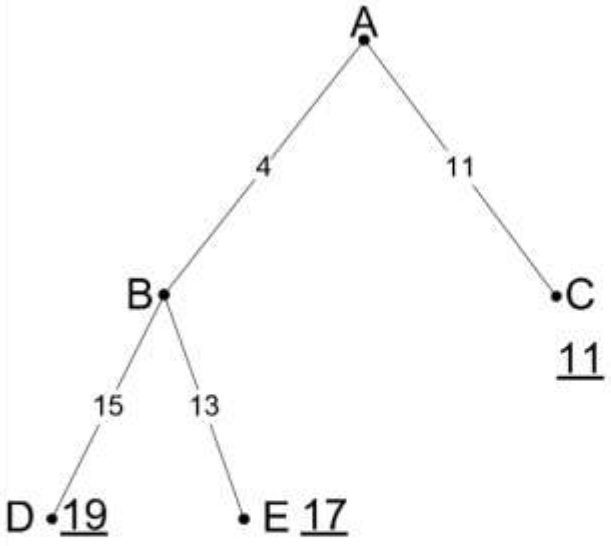
A

0

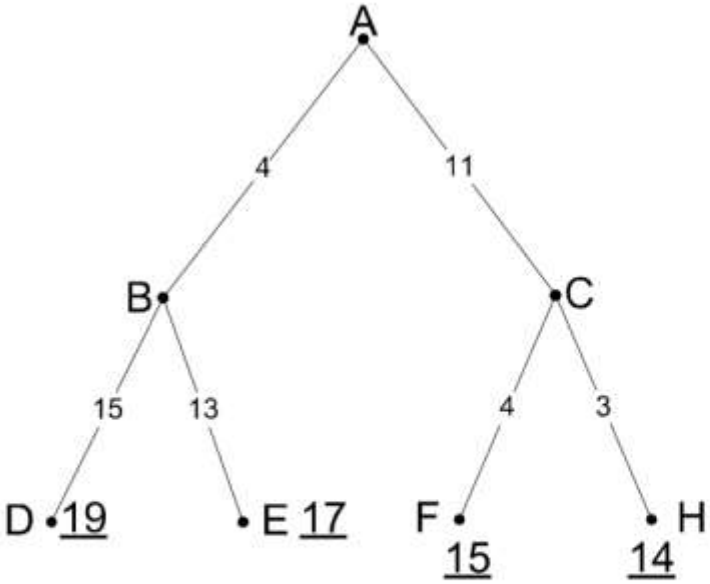
Paths from root are generated.



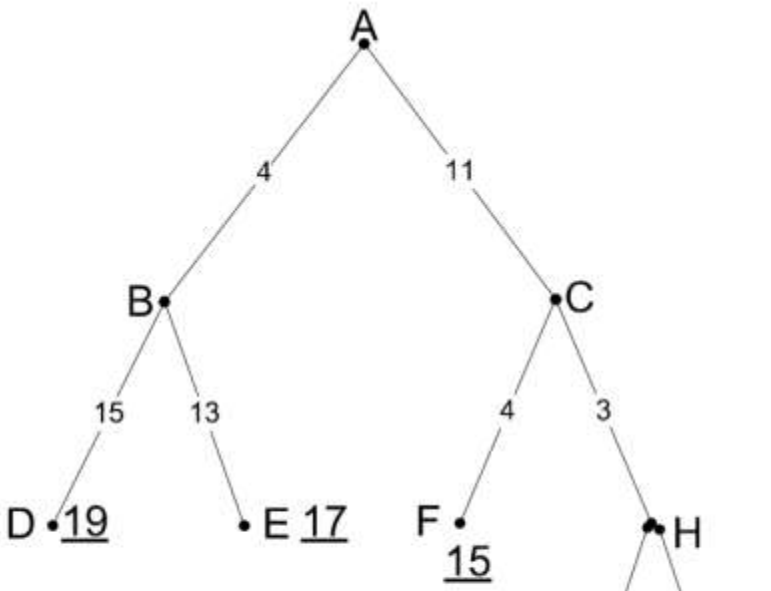
Since B has the least cost, we expand it.



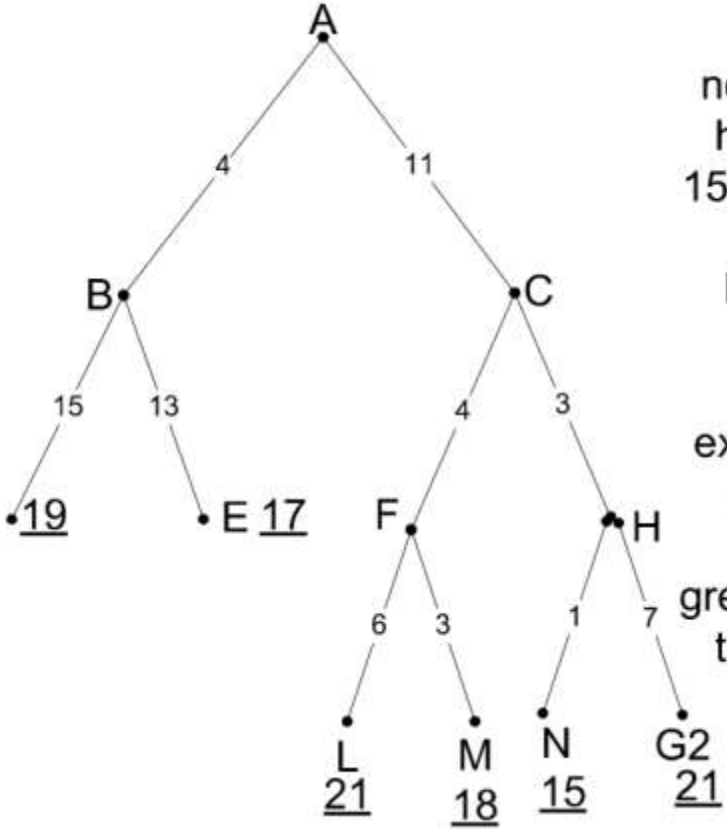
Of our 3 choices, C has the least cost so we'll expand it.



Node H has the least cost thus far, so we expand it.



We have a goal, G2 but need to expand other branches to see if there is another goal with less distance.



Note: Both nodes F and N have a cost of 15, we chose to expand the leftmost node first. We continue expanding until all remaining paths are greater than 21, the cost of G2

Depth Limited Search:

The problem of unbounded trees can be solve by supplying depth-first search with a determined depth limit (nodes at depth are treated as they have no successors) –**Depth limited search. Depth-limited search** is an algorithm to explore the vertices of a graph. It is a modification of depth-first search and is used for example in the iterative deepening depth-first search algorithm.

Like the normal depth-first search, depth-limited search is an uninformed search. It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search. Even if the search could still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles. Therefore depth-limited search will find a solution if it is within the depth limit, which guarantees at least completeness on all graphs.

It solves the infinite-path problem of DFS. Yet it introduces another source of problem if we are unable to find good guess of l. Let d is the depth of shallowest solution.

If $l < d$ then incompleteness results.
If $l > d$ then not optimal.

Time complexity: $O(b^l)$
Space complexity: $O(bl)$

Iterative Deepening Depth First Search:

In this strategy, depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state. On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, assuming no pruning, is effectively breadth-first.

IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness (when the branching factor is finite). It is optimal when the path cost is a non-decreasing function of the depth of the node.

The technique of *iterative deepening* is based on this idea. *Iterative deepening* is depth-first search to a fixed depth in the tree being searched. If no solution is found up to this depth then the depth to be searched is increased and the whole `bounded' depth-first search begun again.

IIt works by setting a depth of search -say, depth 1- and doing depth-first search to that depth. If a solution is found then the process stops -otherwise, increase the depth by, say, 1 and repeat until a solution is found. Note that every time we start up a new bounded depth search *we start from scratch* - i.e. we throw away any results from the previous search.

Now *iterative deepening* is a popular method of search. We explain why this is so.

Depth-first search can be implemented to be much cheaper than breadth-first search in terms of memory usage -but it is not guaranteed to find a solution even where one is guaranteed.

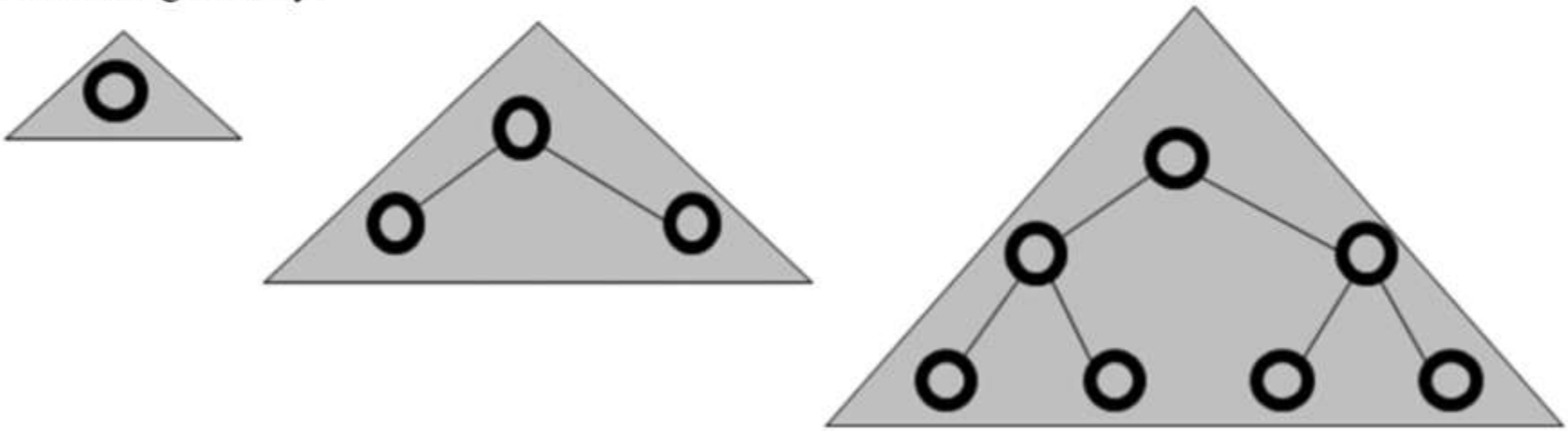
On the other hand, breadth-first search can be guaranteed to terminate if there is a winning state to be found and will always find the `quickest' solution (in terms of how many steps need to be taken from the root node). It is, however, a very expensive method in terms of memory usage.

Iterative deepening is liked because it is an effective compromise between the two other methods of search. It is a form of depth-first search with a lower bound on how deep the search can go. Iterative deepening terminates if there is a solution. It can produce the same solution that breadth-first search would produce but does not require the same memory usage (as for breadth-first search).

Note that depth-first search achieves its efficiency by generating the next node to explore only when this needed. The breadth-first search algorithm has to grow all the search paths available until a solution is found -and this takes up memory. Iterative deepening achieves its memory saving in the same way that depth-first search does -at the expense of redoing some computations again and again (a time cost rather than a memory one). In the search illustrated, we had to visit node d three times in all!

- Complete (like BFS)
- Has linear memory requirements (like DFS)
- Classical time-space tradeoff.
- This is the preferred method for large state spaces, where the solution path length is unknown.

The overall idea goes as follows until the goal node is not found i.e. the depth limit is increased gradually.



Iterative Deepening search evaluation:

Completeness:

- YES (no infinite paths)

Time complexity:

- Algorithm seems costly due to repeated generation of certain states.
- Node generation:

  level d : once
  level d-1: 2
  level d-2: 3
  …
  level 2: d-1
  level 1: d

- Total no. of nodes generated:

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

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:

  $1 + b + b + b + \ldots\ldots\ldots\ldots\ldots\ldots$ b d times = **O(bd)**

Optimality:

- YES if path cost is non-decreasing function of the depth of the node.

Notice that BFS generates some nodes at depth d+1, whereas IDS does not. The result is that IDS is actually faster than BFS, despite the repeated generation of node.

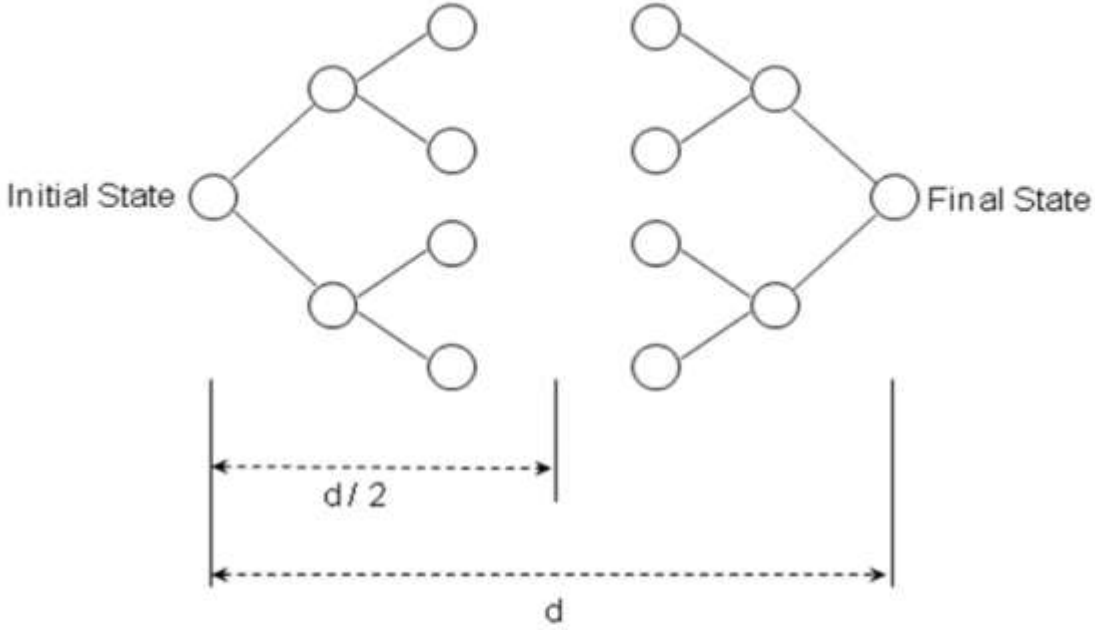
Example: Number of nodes generated for b=10 and d=5 solution at far right

$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$

$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$

Bidirectional Search:

This is a search algorithm which replaces a single search graph, which is likely to with two smaller graphs -- one starting from the initial state and one starting from the goal state. It then, expands nodes from the start and goal state simultaneously. Check at each stage if the nodes of one have been generated by the other, i.e, they meet in the middle. If so, the path concatenation is the solution.



- Completeness: yes
- Optimality: yes (If done with correct strategy- e.g. breadth first)
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$

Problems: generate predecessors; many goal states; efficient check for node already visited by other half of the search; and, what kind of search.

Drawbacks of uninformed search

- Criterion to choose next node to expand depends only on a global criterion: level
- Does not exploit the structure of the problem
- One may prefer to use a more flexible rule, that takes advantage of what is being discovered on the way, and hunches about what can be a good move
- Very often, we can select which rule to apply by comparing the current state and the desired state.

Heuristic Search (Informed Search)

Heuristic Search Uses domain-dependent (heuristic) information in order to search the space more efficiently.

Ways of using heuristic information:

- Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
- Deciding that certain nodes should be discarded, or *pruned*, from the search space.

Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method

Informed Search uses domain specific information to improve the search pattern

- Define a heuristic function, h(n), that estimates the "goodness" of a node n.
- Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.

Best-First Search

Idea: use an *evaluation function f(n)* that gives an indication of which node to expand next for each node.

- usually gives an estimate to the goal.
- the node with the lowest value is expanded first.

A key component of *f(n)* is a heuristic function, *h(n),* which is a additional knowledge of the problem.

There is a whole family of best-first search strategies, each with a different evaluation function.

Typically, strategies use estimates of the cost of reaching the goal and try to minimize it.

Special cases: based on the evaluation function.

- Greedy best-first search
- A*search

Greedy Best First Search

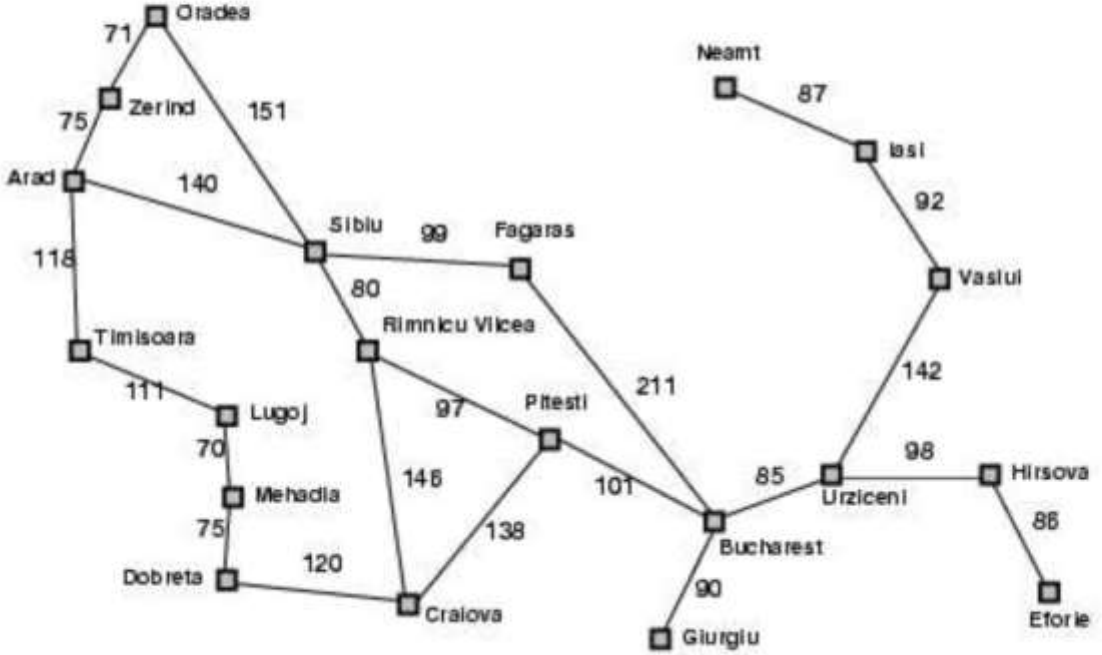
The best-first search part of the name means that it uses an evaluation function to select which node is to be expanded next. The node with the lowest evaluation is selected for expansion because that is the *best* node, since it supposedly has the closest path to the goal (if the heuristic is good). Unlike A* which uses both the link costs and a heuristic of the cost to the goal, greedy best-first search uses only the heuristic, and not any link costs. A disadvantage of this approach is that if the heuristic is not accurate, it can go down paths with high link cost since there might be a low heuristic for the connecting node.

Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from n to *goal*.

e.g., $h_{SLD}(n)$ = straight-line distance from n to goal

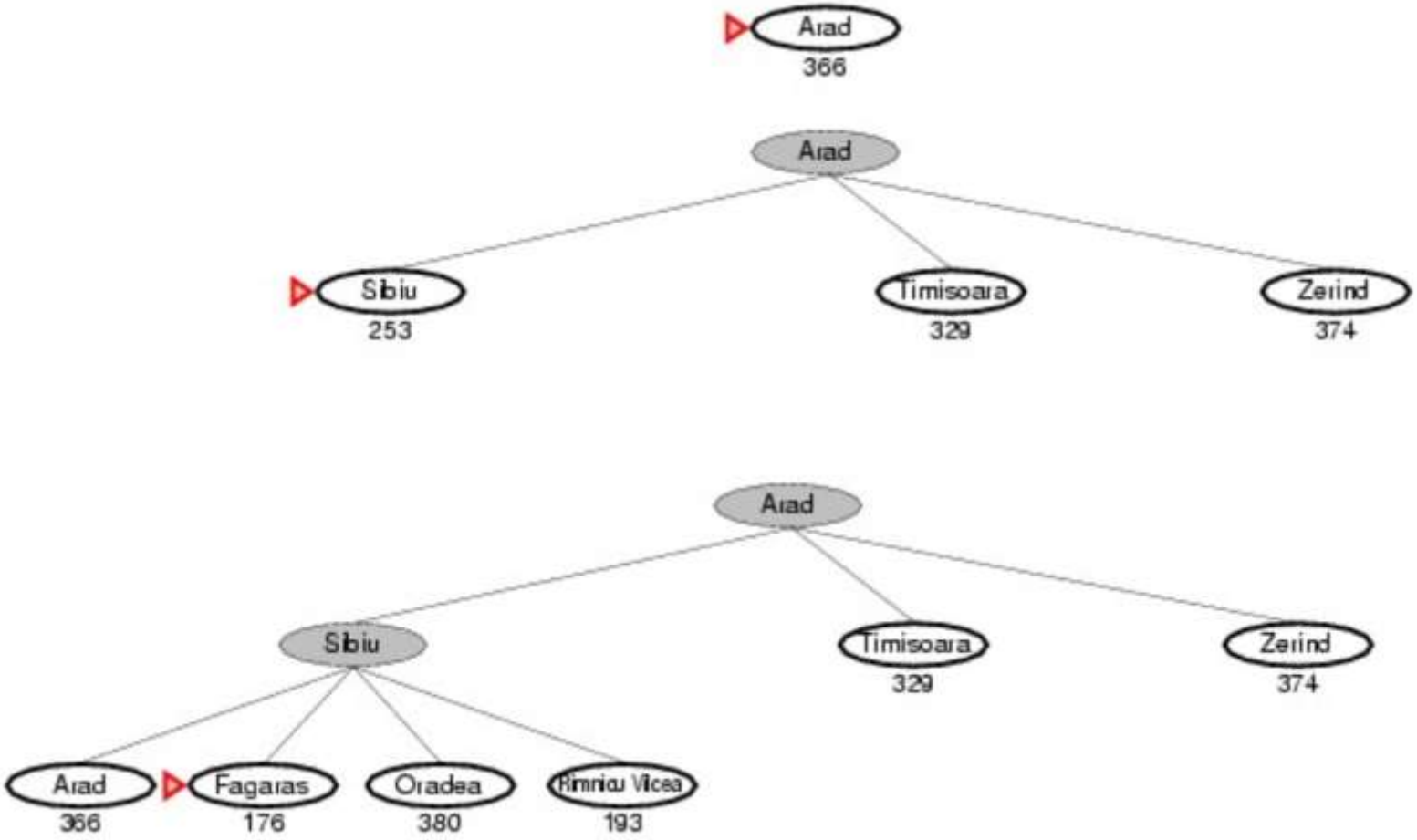
Greedy best-first search expands the node that appears to be closest to goal. The greedy best-first search algorithm is $O(b^m)$ in terms of space and time complexity. (Where b is the average branching factor (the average number of successors from a state), and m is the maximum depth of the search tree.)

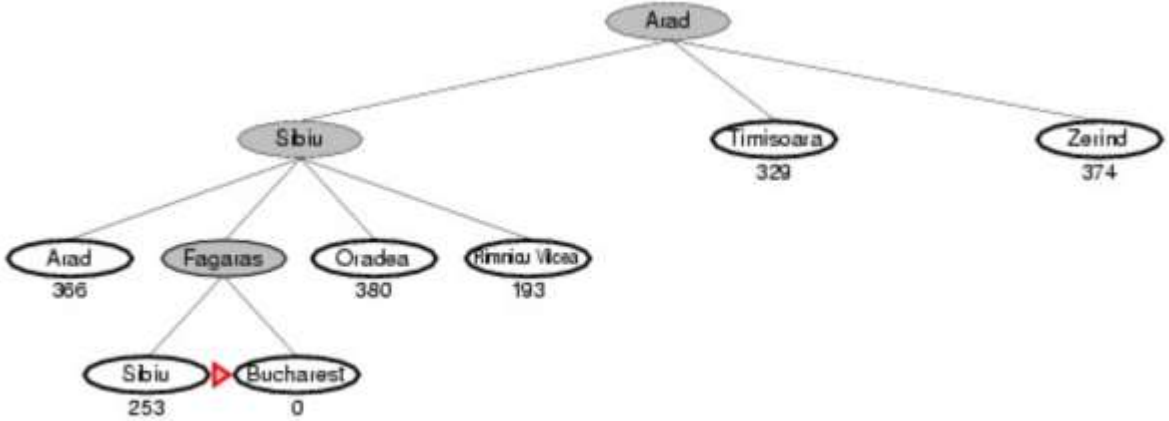
Example: Given following graph of cities, starting at Arad city, problem is to reach to the Bucharest.



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Solution using greedy best first can be as below:


Arad
366
Arad
Sibiu
253
Timisoara
329
Zerind
374
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193



Greedy Best-first search

- minimizes estimated cost h(n) from current node n to goal;
- is informed but (almost always) suboptimal and incomplete.

Admissible Heuristic:

A heuristic function is said to be **admissible** if it is no more than the lowest-cost path to the goal. In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal. An admissible heuristic is also known as an **optimistic heuristic**.

An admissible heuristic is used to estimate the cost of reaching the goal state in an informed search algorithm. In order for a heuristic to be admissible to the search problem, the estimated cost must always be lower than the actual cost of reaching the goal state. The search algorithm uses the admissible heuristic to find an estimated optimal path to the goal

state from the current node. For example, in A* search the evaluation function (where n is the current node) is: $f(n) = g(n) + h(n)$

where;

$f(n)$ = the evaluation function.
$g(n)$ = the cost from the start node to the current node
$h(n)$ = estimated cost from current node to goal.

$h(n)$ is calculated using the heuristic function. *With a non-admissible heuristic, the A* algorithm would overlook the optimal solution to a search problem due to an overestimation in f(n).*

It is obvious that the SLD heuristic function is admissible as we can never find a shorter distance between any two towns.

A* Search:

A* is a best first, informed graph search algorithm. A* is different from other best first search algorithms in that it uses a heuristic function h(x) as well as the path cost to the node g(x), in computing the cost f(x) = h(x) + g(x) for the node. The $h(x)$ part of the $f(x)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

It finds a minimal cost-path joining the start node and a goal node for node n.
Evaluation function: f(n) = g(n) + h(n)

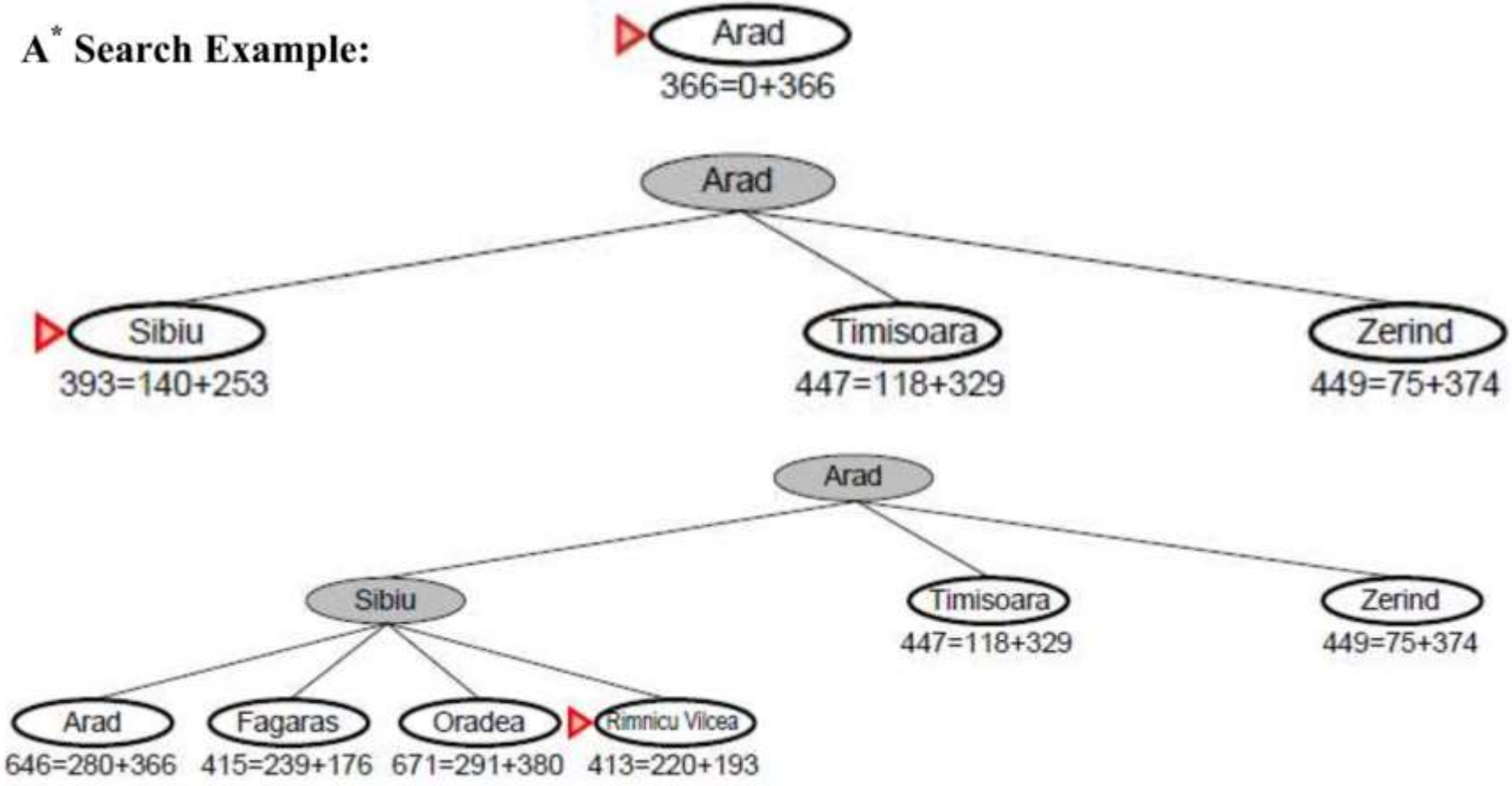
Where,

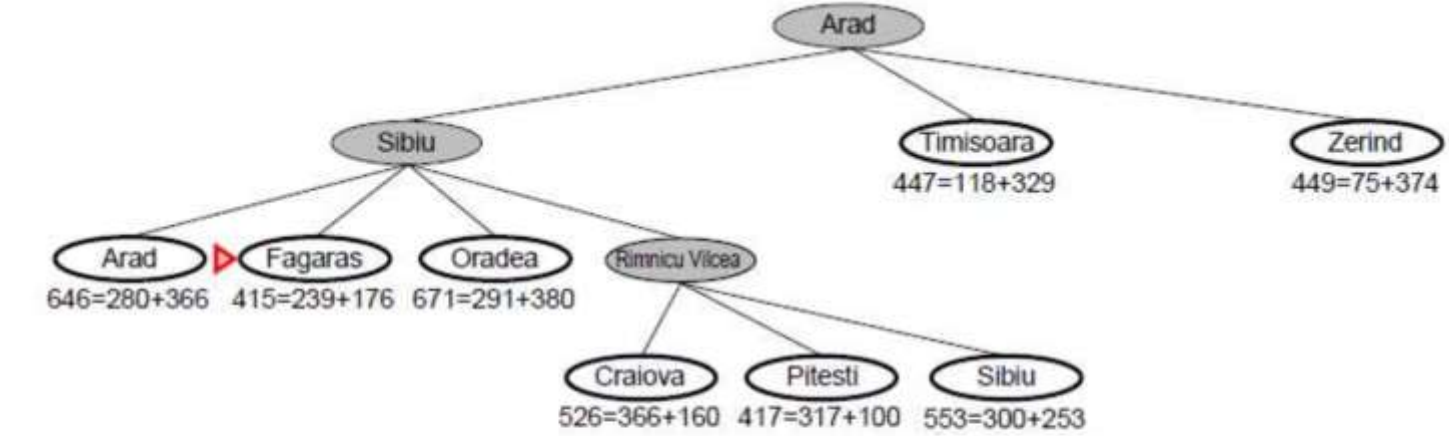
g(n) = cost so far to reach n from root
h(n) = estimated cost to goal from n
f(n) = estimated total cost of path through n to goal

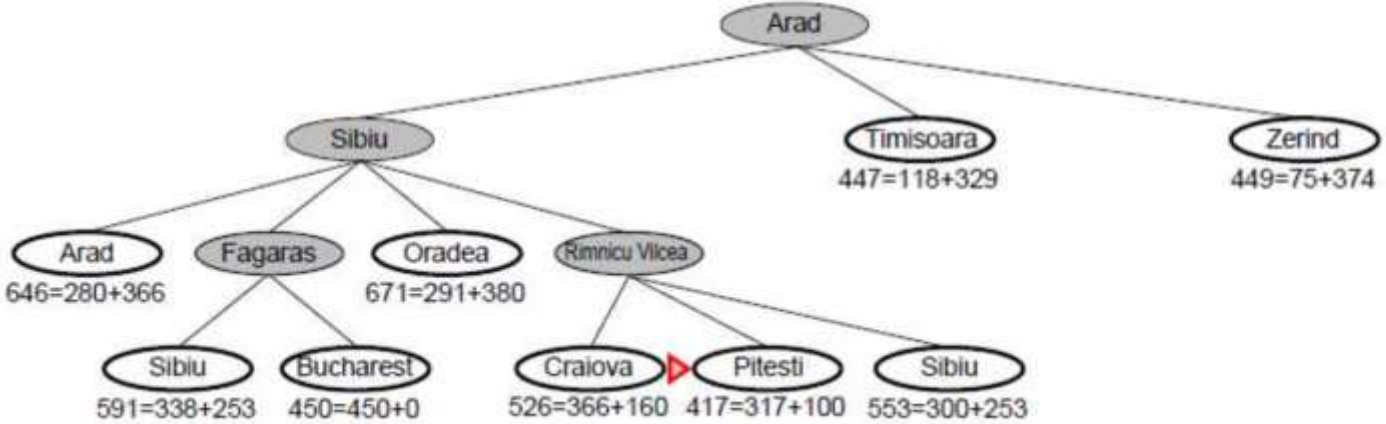
- combines the two by minimizing f(n) = g(n) + h(n);
- is informed and, *under reasonable assumptions*, optimal and complete.

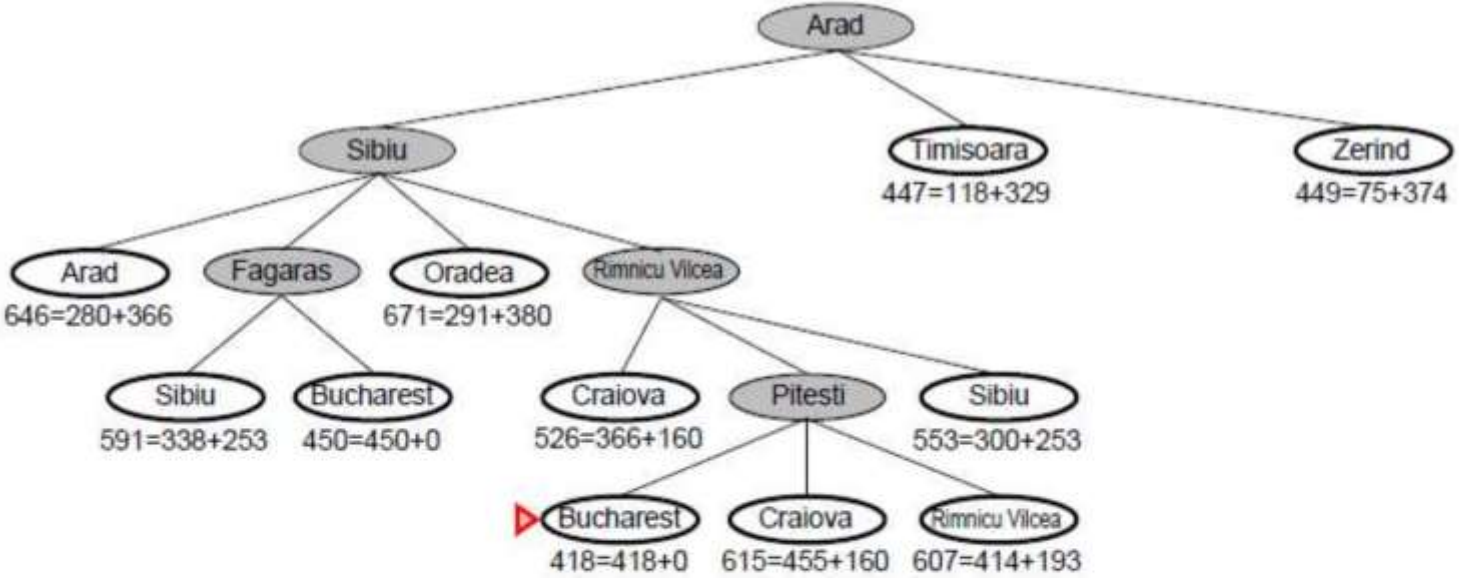
As A* traverses the graph, it follows a path of the lowest *known* path, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.

A* Search Example:




Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253



Admissibility and Optimality:

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

Hill Climbing Algorithm

- A search method of selecting the best local choice at each step in hopes of finding an optimal solution.
- Does not consider how optimal the current solution is.
- Does not consider steps past the next immediate choice.
- Steps:
  1. Define an evaluation function f(x) to determine the value of a state.
  2. From the current state, determine the search space (actions) for one step ahead.
  3. Select the action from the search space that returns the highest value.

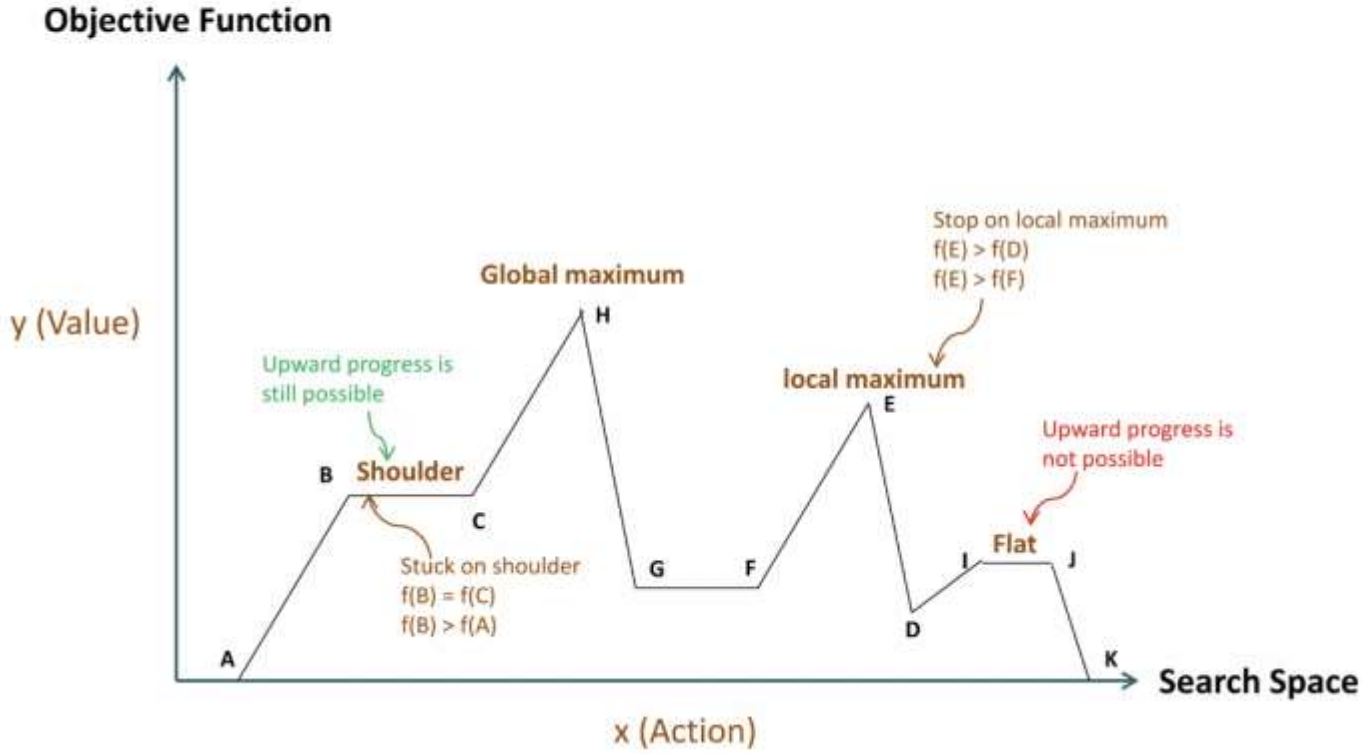
Hill Climbing

Hill climbing can be used to solve problems that have many solutions, some of which are better than others.

It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little.

When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

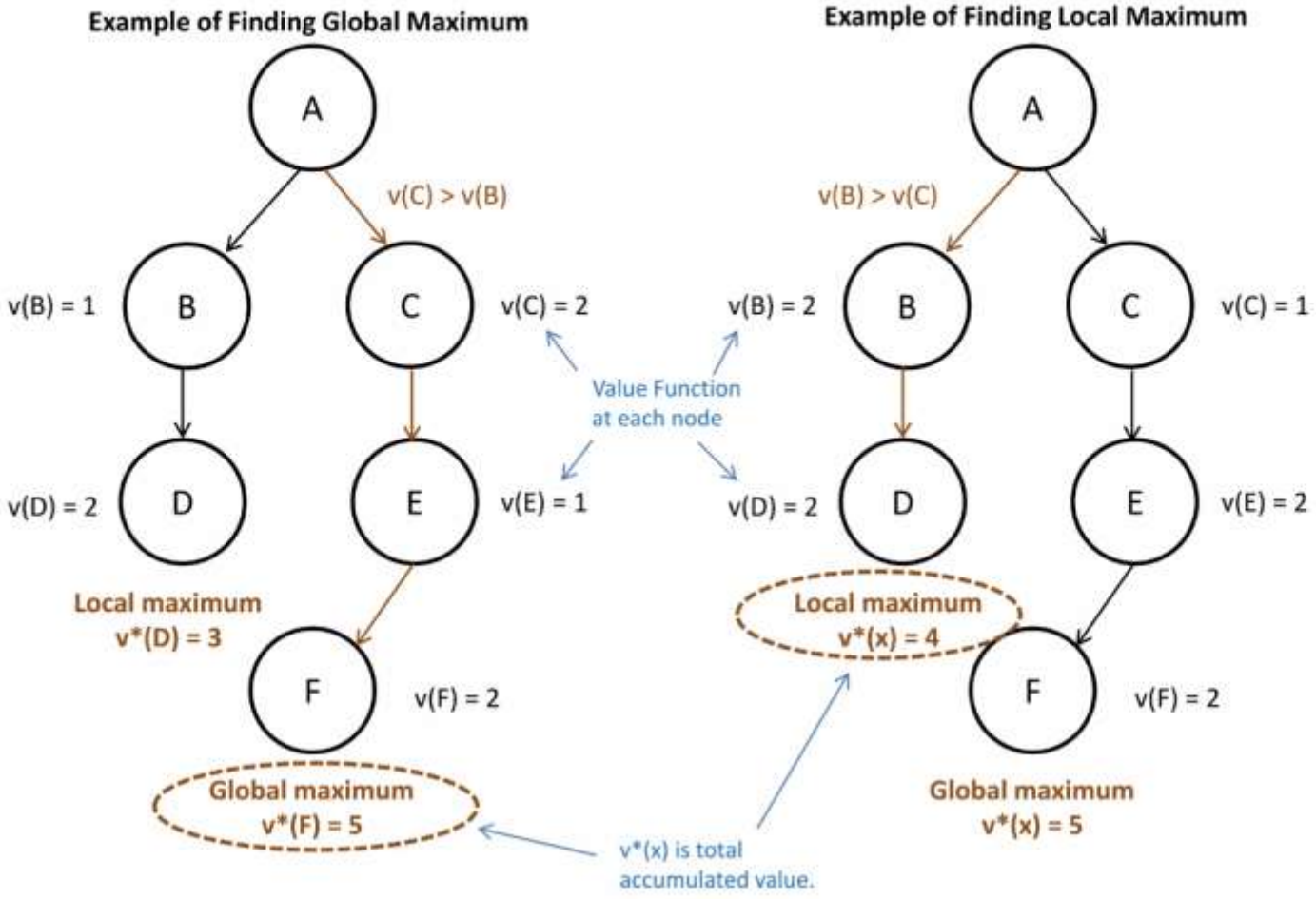
Hill Climbing – Objective Function



The hill climbing can be described as follows:

- Start with *current-state* = initial-state.
- Until *current-state* = goal-state OR there is no change in *current-state* do:
  - Get the successors of the current state and use the evaluation function to assign a score to each successor.
  - If one of the successors has a better score than the current-state then set the new current-state to be the successor with the best score.

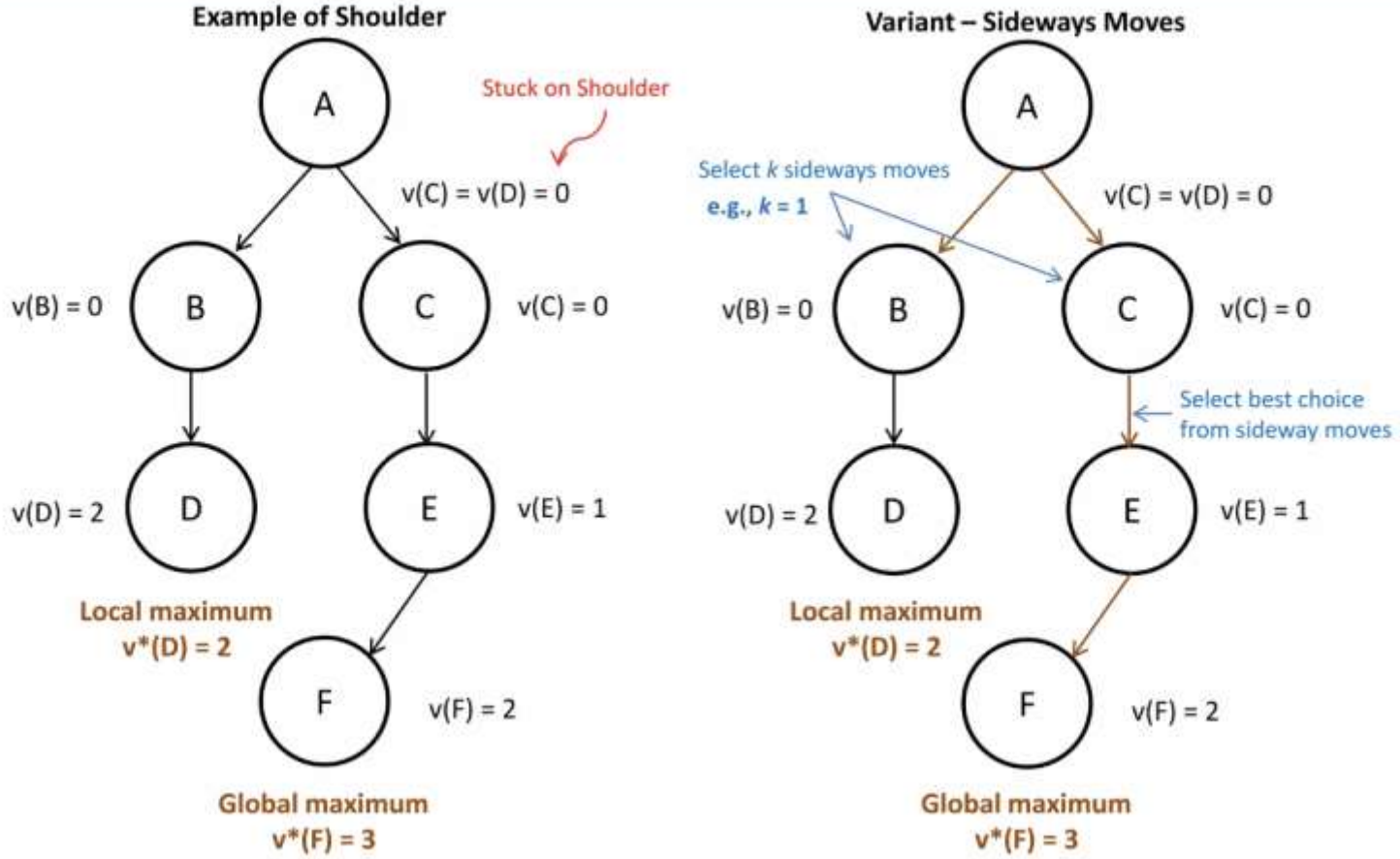
Value Function



Algorithm

```
function HillClimb( initial )

        initialize node with initial                    # set the current node to the starting point

        while forever                                   # continue until you cannot climb higher
                initialize max  to minus infinity       # minimum value
                for each child (neighbor) of the node
                        if v(child) >  max              # find neighbor with max value
                                max = v(child)
                                next = child
                if max <= 0                             # cannot climb higher
                        return node

                node = next                             # climb to the next node
```

Sideways Moves – Escape Shoulder



Algorithm – Sideways Moves

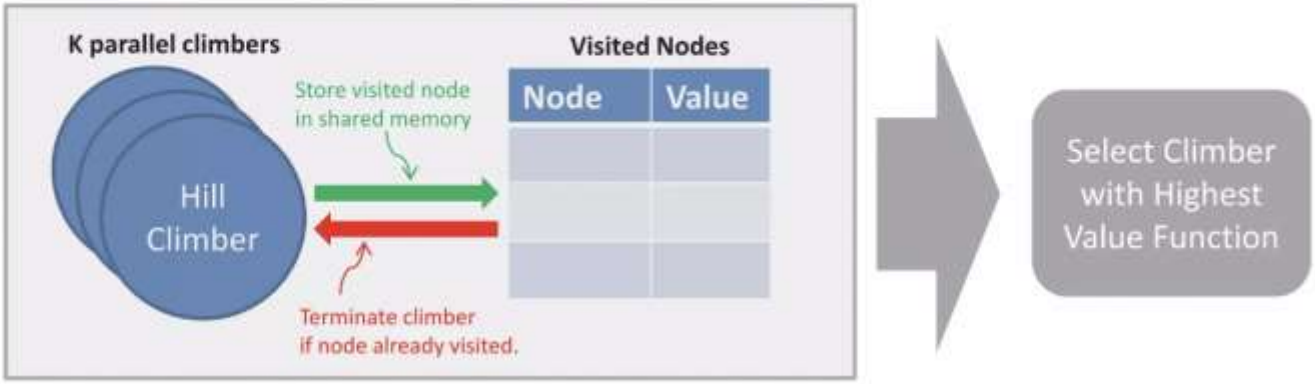
```
function HillClimb( initial, k )        ← Add parameter for number of sideways moves

        initialize node with initial                    # set the current node to the starting point

        while forever                                   # continue until you cannot climb higher
                initialize max  to minus infinity       # minimum value
                for each child (neighbor) of the node
                        if v(child) >  max              # find neighbor with max value
                                max = v(child)
                                next = child
                if max <= 0                             # cannot climb higher
No sideways moves ←             if k == 0
                                return node
                        for each child (neighbor) of the node
                                value = HillClimb( child, k-1)   ← Decrease k by one
Recursively call the HillClimb          if value > max
algorithm on each child, while                  max = value
decrementing the number of                      next = child
sideways moves (depth)          if max <= 0
                                return node
                node = next                             # climb to the next node
```

Variant - Stochastic

- **Stochastic – choose node at random**
  - **If none of the child (neighbor) nodes has a value > 0, then choose one of the nodes at random.**
  - **There needs to be a cutoff (k) of random choices, or the method will go into an infinite loop (e.g., you are at the global maximum).**

Variant - Local Beam Search

- Search with multiple (k) Hill Climbers in parallel.
  - Select k different initial states (nodes).
  - Communicate between climbers which states (nodes) have been visited.
  - Terminate Hill Climber if state (node) already visited by another Hill Climber – eliminates duplication in search.
  - When all Hill Climbers reach terminal state (node), select path with the highest value function.

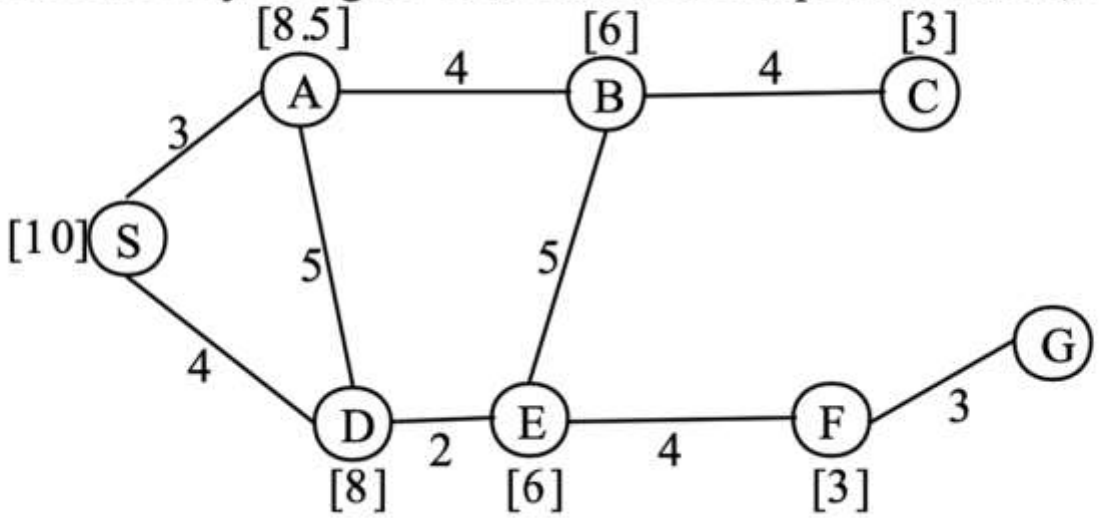


Problems with Hill Climbing

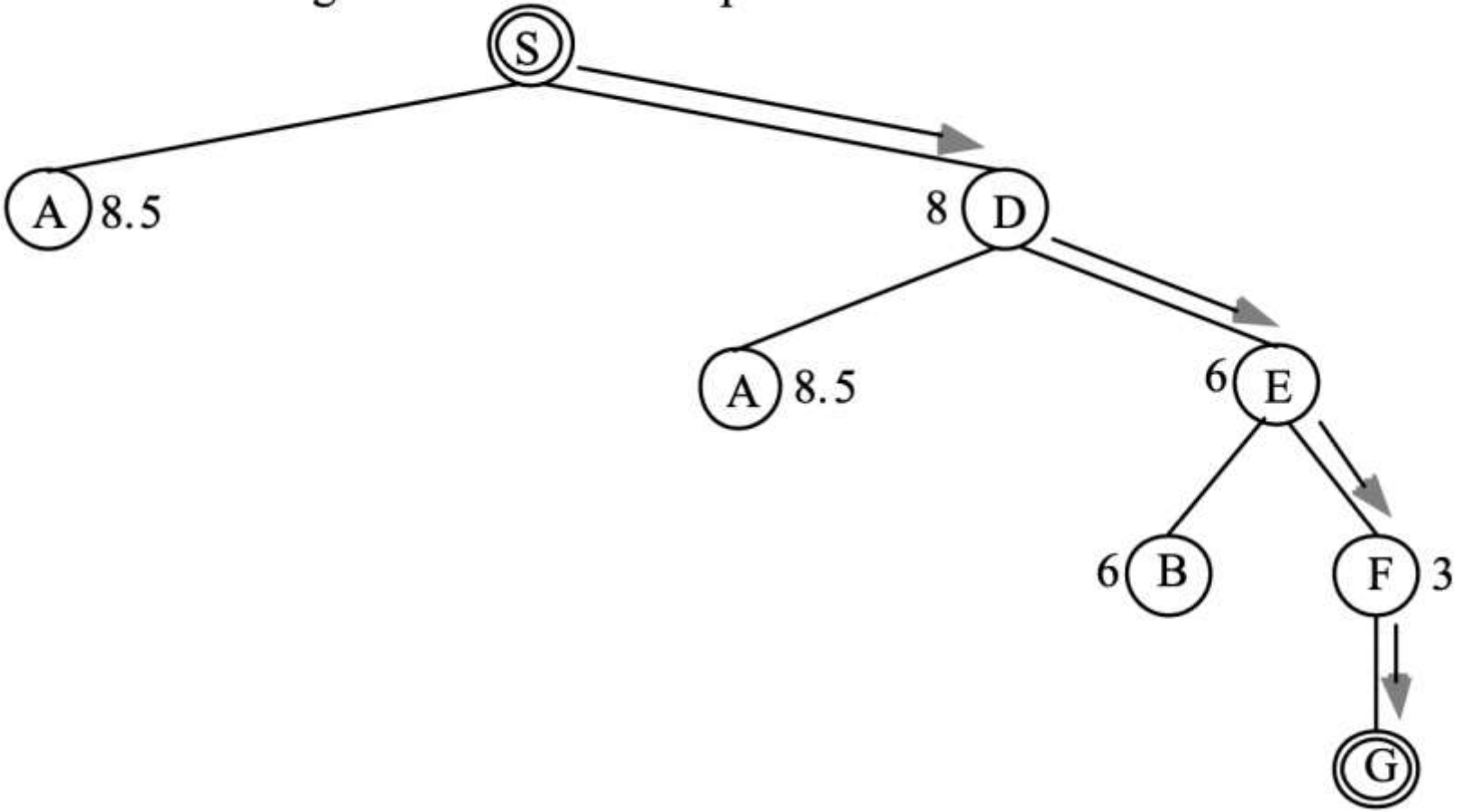
Gets stuck at local minima when we reach a position where there are no better neighbors, it is not a guarantee that we have found the best solution. Ridge is a sequence of local maxima.

Another type of problem we may find with hill climbing searches is finding a ***plateau.*** This is an area where the search space is flat so that all neighbors return the same evaluation

For instance, consider that the most promising successor of a node is the one that has the shortest straight-line distance to the goal node G. In figure below, the straight line distances between each city and goal G is indicated in square brackets, i.e. the heuristic.

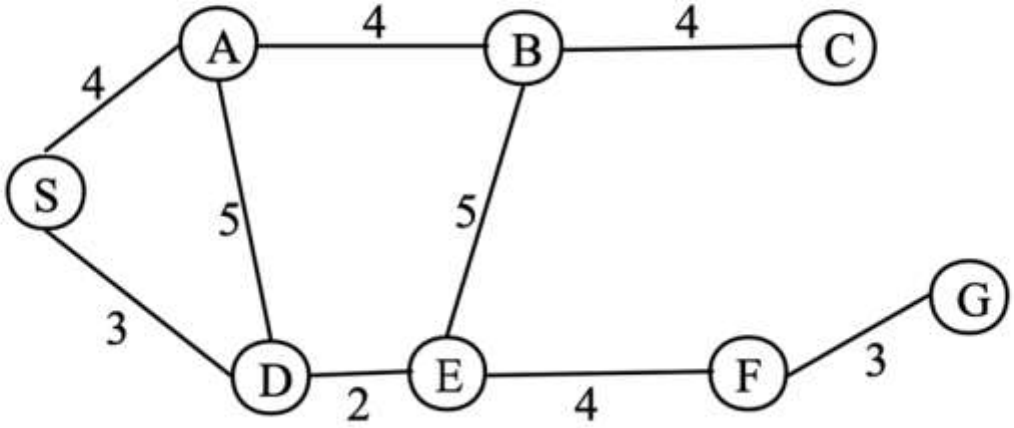


The hill climbing search from S to G proceeds as follows:



Exercise:

Apply the hill climbing algorithm to find a path from S to G, considering that the most promising successor of a node is its closest neighbor.



Hill Climbing: Local Heuristic function

Blocks World

Local Heuristic: +1 for each block that is resting on the thing it is supposed to be resting on

-1 for each block that is resting on a wrong thing

Global Heuristic

For each block that has the correct support structure: +1 to every block in the support system

For each block that has a wrong support structure : -1 to every block in the support system

Simulated Annealing

It is motivated by the physical annealing process in which material is heated and slowly cooled into a uniform structure.

Compared to hill climbing the main difference is that SA allows downwards steps.

Simulated annealing also differs from hill climbing in that a move is selected at random and then decides whether to accept it.

If the move is better than its current position then simulated annealing will always take it.

If the move is worse (i.e. lesser quality) then it will be accepted based on some probability.

The probability of accepting a worse state is given by the equation

$$P = \text{exponential}(-c / t) > r$$

Where

c	=	the change in the evaluation function
t	=	the current value
r	=	a random number between 0 and 1

The probability of accepting a worse state is a function of both the current value and the change in the cost function. The most common way of implementing an SA algorithm is to implement hill climbing with an accept function and modify it for SA

Game playing

Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to 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 improve –

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

The most common search technique in game playing is Minimax search procedure. It is depth-first depth-limited search procedure. It is used for games like chess and 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 player 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. E.g. win (+1), loose (-1) and draw (0). Some games have a wider variety of possible outcomes eg. ranging from +92 to -192.

MinMax Algorithm

Let us assign the following values for the game: 1 for win by X, 0 for draw, -1 for loss by X.

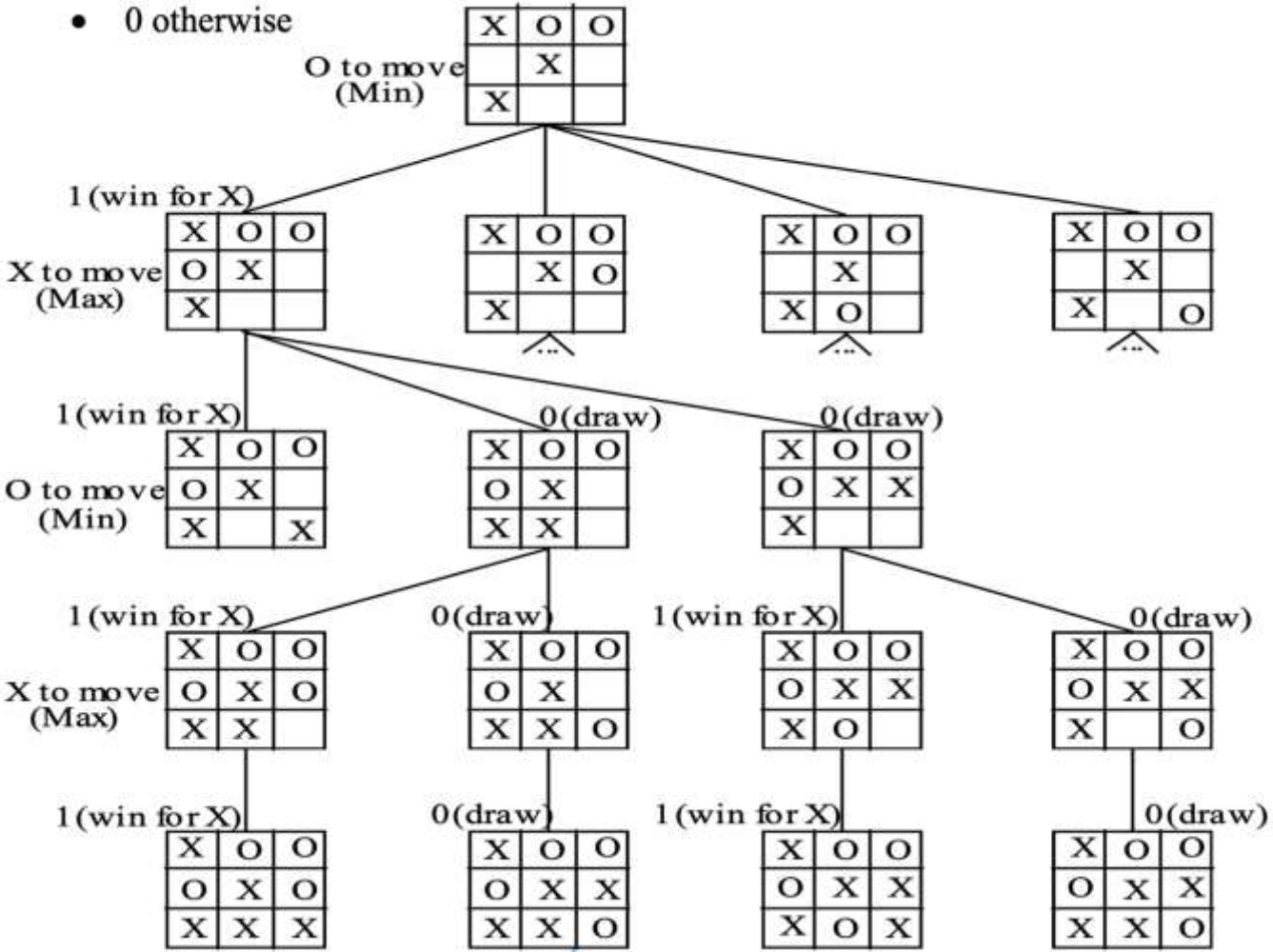
Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:

1. the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);

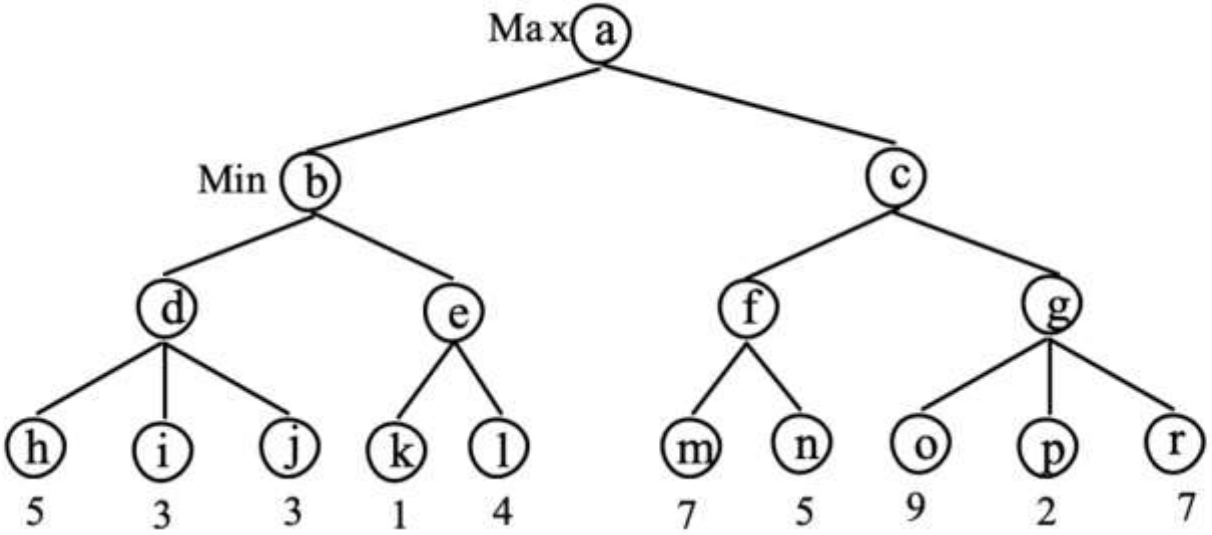
2. the value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome of X).

Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree. The values of the leaves of the tree are given by the rules of the game

- 1 if there are three X in a row, column or diagonal;
- -1 if there are three O in a row, column or diagonal;
- 0 otherwise



Consider the following game tree (drawn from the point of view of the Maximizing player):



Show what moves should be chosen by the two players, assuming that both are using the mini-max procedure.

Alpha beta pruning

The problem with minimax search is that the number if game states it has examined is exponential in the number of moves. Unfortunately, we can't eliminate the exponent, but we can effectively cut it in half.

The idea is to compute the correct minimax decision without looking at every node in the game tree, which is the concept behind pruning.

Here the idea is to eliminate large parts of the tree from consideration.

The particular technique for pruning that we will discuss here is Alpha-Beta Pruning.

When this approach is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

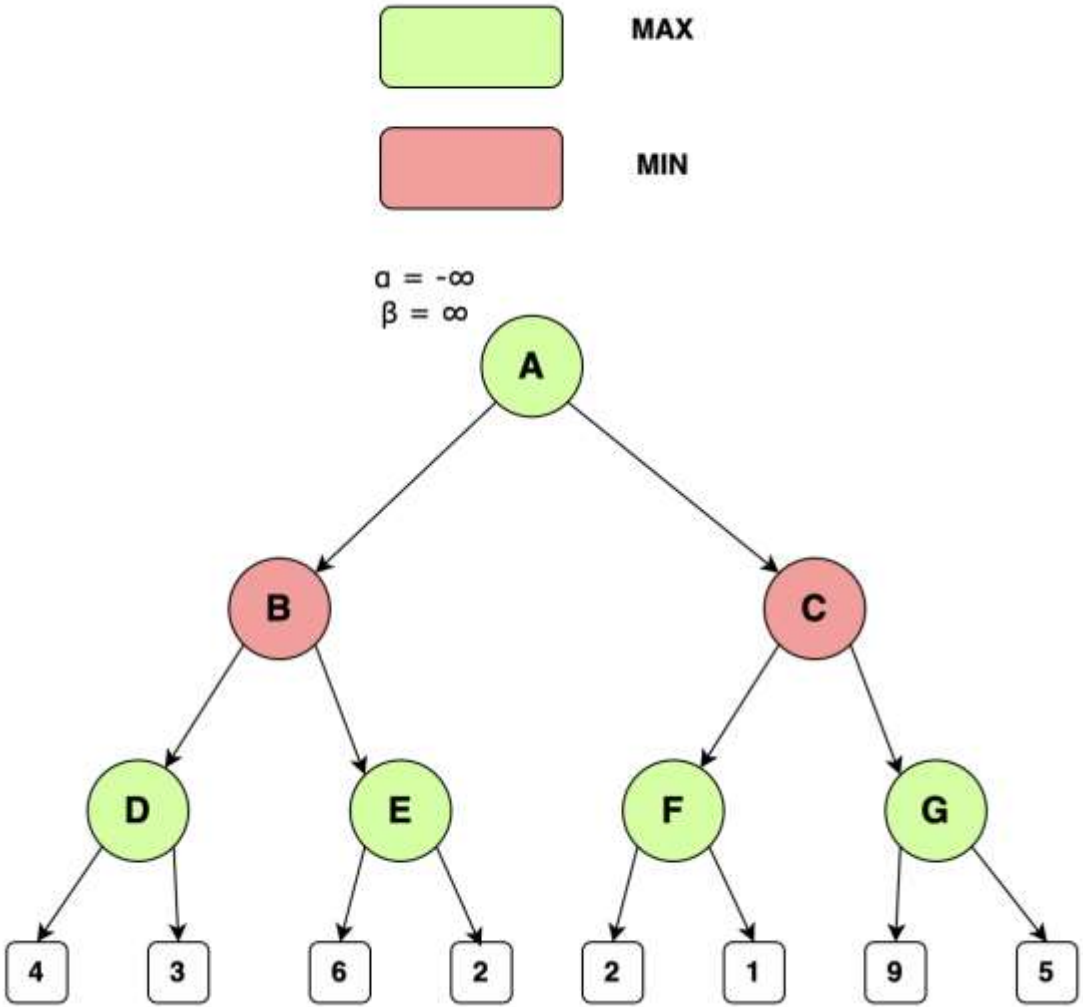
Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub-trees rather than just leaves.

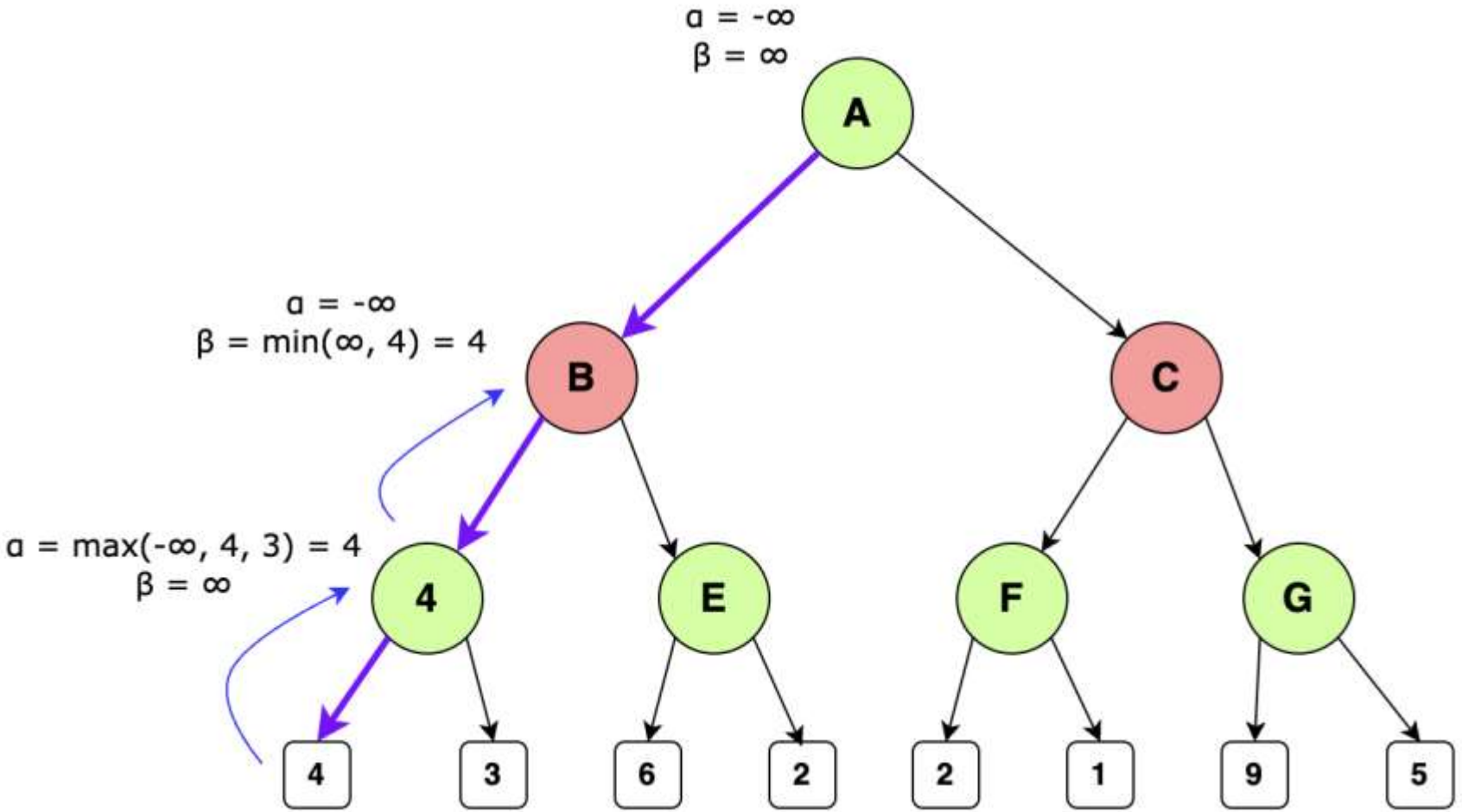
Alpha-beta pruning is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. It uses two parameters, alpha and beta.

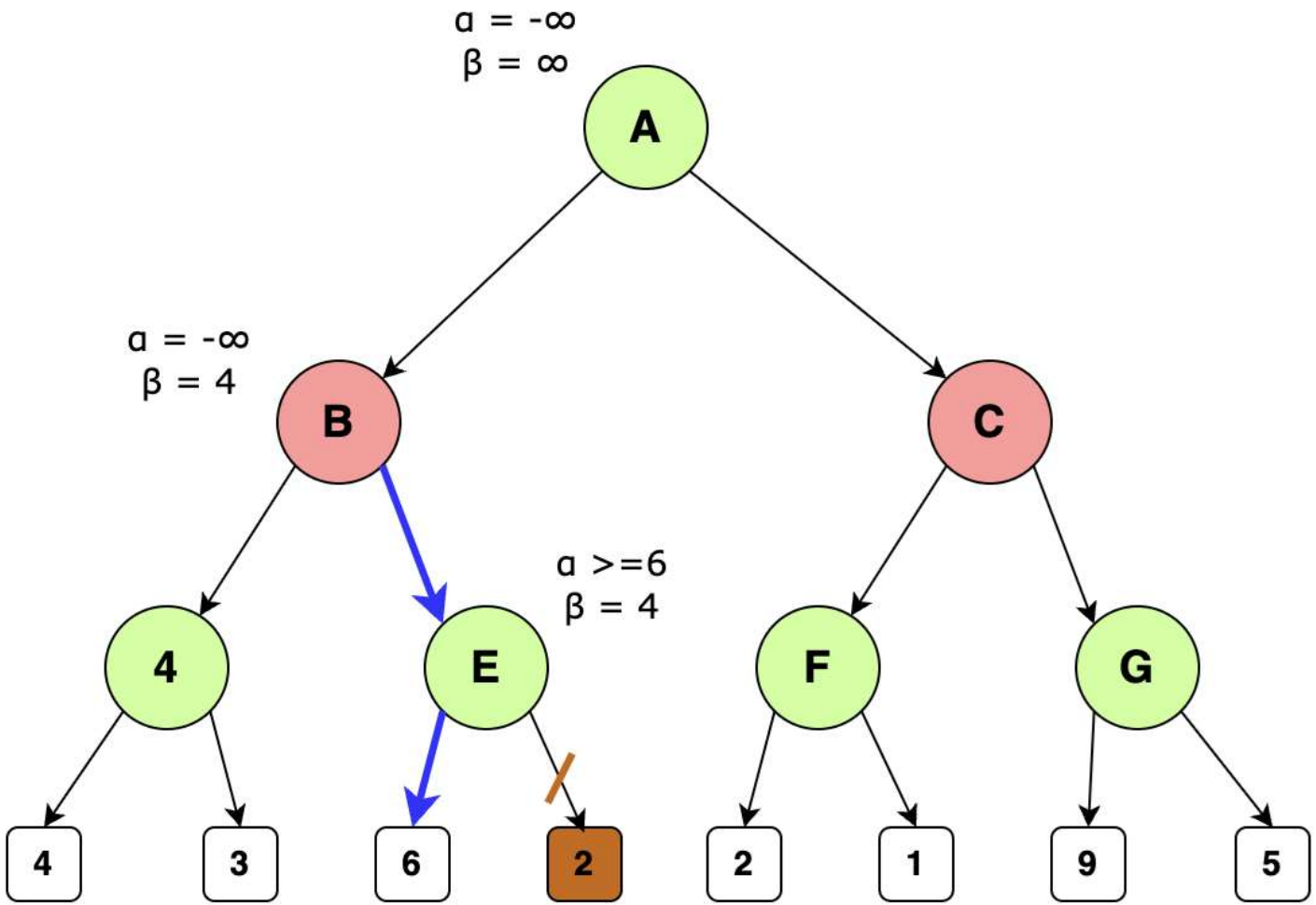
Alpha: is the value of the best (i.e. highest value) choice we have found so far at any choice point along the path for MAX.

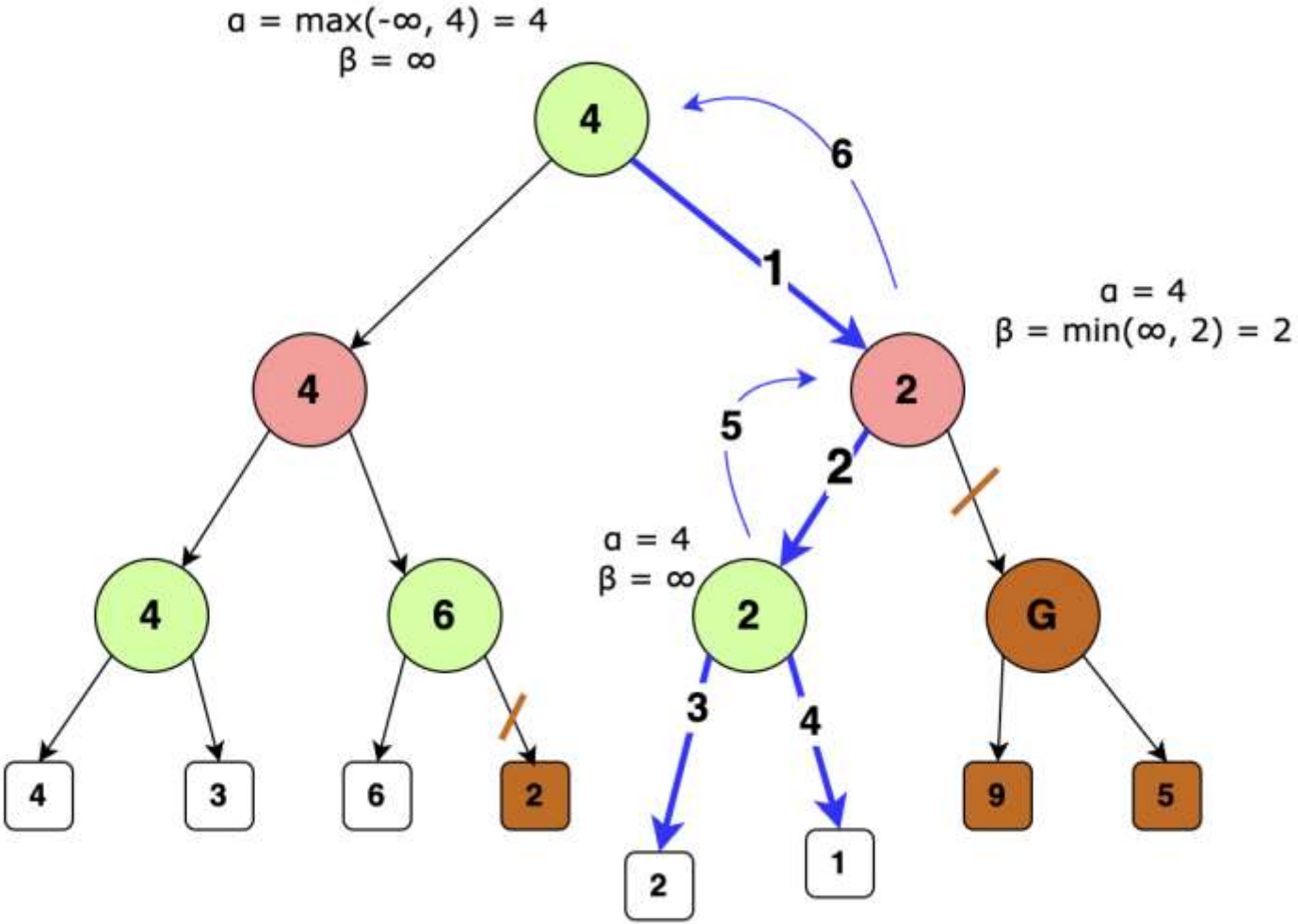
Beta: is the value of the best (i.e. lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search updates the values of alpha and beta as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current alpha or beta for MAX or MIN respectively.


MAX
MIN
α = -∞
β = ∞
A
B
C
D
E
F
G
4
3
6
2
2
1
9
5


α = -∞
β = ∞
A
α = -∞
β = min(∞, 4) = 4
B
C
α = max(-∞, 4, 3) = 4
β = ∞
4
E
F
G
4
3
6
2
2
1
9
5


α = -∞
β = ∞
A
α = -∞
β = 4
B
C
α >=6
β = 4
4
E
F
G
4
3
6
2
2
1
9
5


α = max(-∞, 4) = 4
β = ∞
4
6
1
α = 4
β = min(∞, 2) = 2
4
2
5
2
α = 4
β = ∞
4
6
2
G
3
4
4
3
6
2
9
5
2
1

Example of 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.

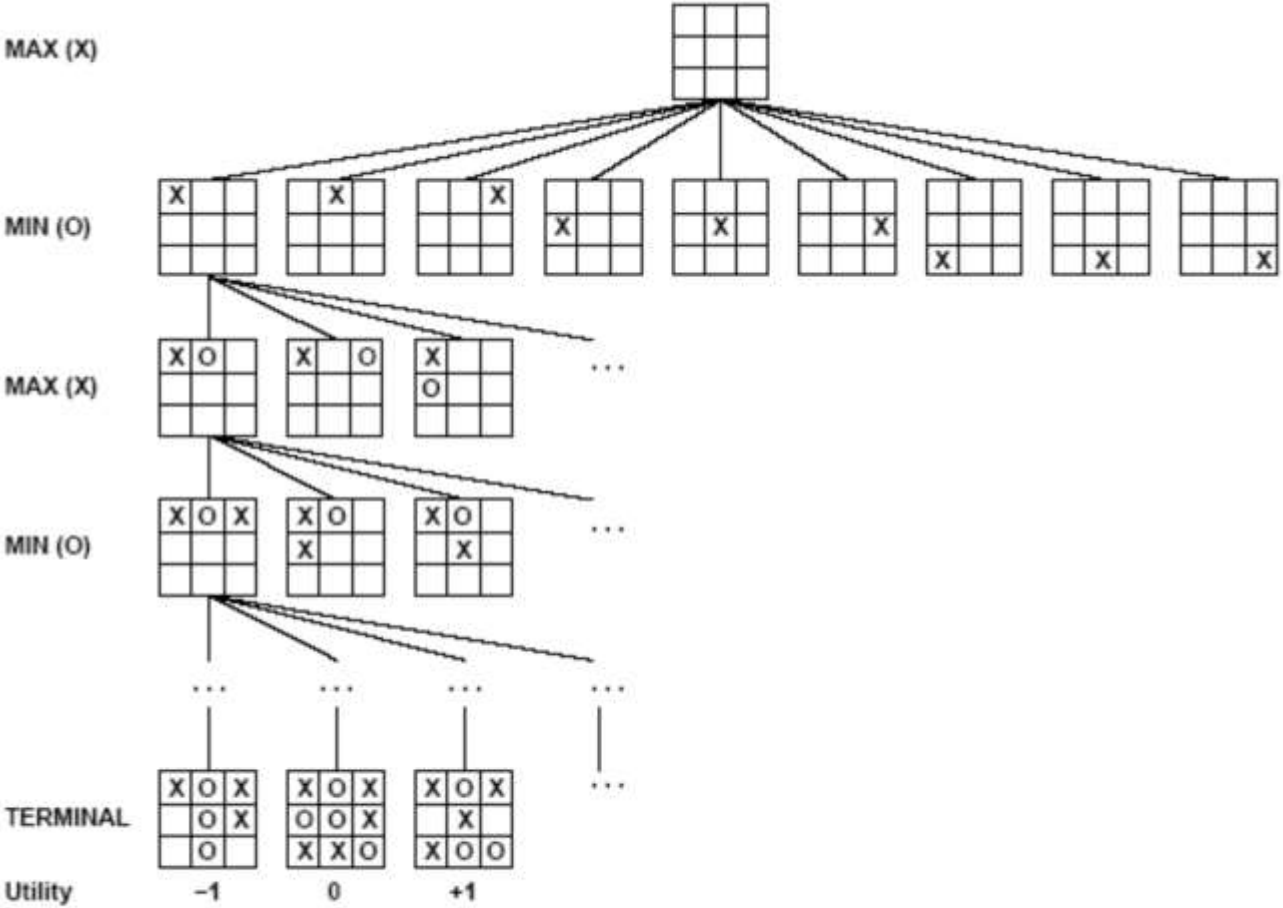


Fig: Partial game tree for Tic-Tac-Toe

MCQ

1. What is the main task of a problem-solving agent?

a) Solve the given problem and reach to goal

b) To find out which sequence of action will get it to the goal state

c) All of the mentioned

d) None of the mentioned

Answer: c

2. What is state space?

a) The whole problem

b) Your Definition to a problem

c) Problem you design

d) Representing your problem with variable and parameter

Answer: d

3. A search algorithm takes _____ as an input and returns _____ as an output.

a) Input, output

b) Problem, solution

c) Solution, problem

d) Parameters, sequence of actions

Answer: b

4. A problem in a search space is defined by one of these state.

a) Initial state

b) Last state

c) Intermediate state

d) All of the mentioned

Answer: a

5. The process of removing detail from a given state representation is called _____

a) Extraction

b) Abstraction

c) Information Retrieval

d) Mining of data

Answer: b

6. The _____ is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.

a) Finding shortest path between a source and a destination

b) Travelling Salesman problem

c) Map coloring problem

d) Depth first search traversal on a given map represented as a graph

Answer: b

7. Web Crawler is a/an _____

a) Intelligent goal-based agent

b) Problem-solving agent

c) Simple reflex agent

d) Model based agent

Answer: a

8. What is the major component/components for measuring the performance of problem solving?

a) Completeness

b) Optimality

c) Time and Space complexity

d) All of the mentioned

Answer: d

9. . A production rule consists of _____

a) A set of Rule

b) A sequence of steps

c) Set of Rule & sequence of steps

d) Arbitrary representation to problem

Answer: c

10. Which search method takes less memory?

a) Depth-First Search

b) Breadth-First search

c) Linear Search

d) Optimal search

Answer: a

11. Which is the best way to go for Game playing problem?

a) Linear approach

b) Heuristic approach (Some knowledge is stored)

c) Random approach

d) An Optimal approach

Answer: b

12. A* algorithm is based on _____

a) Breadth-First-Search

b) Depth-First –Search

c) Best-First-Search

d) Hill climbing

Answer: c

13. The search strategy the uses a problem specific knowledge is known as _____

a) Informed Search

b) Best First Search

c) Heuristic Search

d) All of the mentioned

Answer: d

14. Best-First search can be implemented using the following data structure.

a) Queue

b) Stack

c) Priority Queue

d) Circular Queue

Answer: c

15. Heuristic function h(n) is _____

a) Lowest path cost

b) Cheapest path from root to goal node

c) Estimated cost of cheapest path from root to goal node

d) Average path cost

Answer: c

16 What is the evaluation function in A* approach?

a) Heuristic function

b) Path cost from start node to current node

c) Path cost from start node to current node + Heuristic cost

d) Average of Path cost from start node to current node and Heuristic cost

Answer: c

17. Which search strategy is also called as blind search?

a) Uninformed search

b) Informed search

c) Simple reflex search

d) All of the mentioned

Answer: a

18. Which search is implemented with an empty first-in-first-out queue?

a) Depth-first search

b) Breadth-first search

c) Bidirectional search

d) None of the mentioned

Answer: b

19. When is breadth-first search is optimal?

a) When there is less number of nodes

b) When all step costs are equal

c) When all step costs are unequal

d) None of the mentioned

Answer: b

20. What is the space complexity of Depth-first search?

a) O(b)

b) O(bl)

c) O(m)

d) O(bm)

Answer: d

21. How many parts does a problem consists of?

a) 1

b) 2

c) 3

d) 4

Answer: d

22. Which search implements stack operation for searching the states?

a) Depth-limited search

b) Depth-first search

c) Breadth-first search

d) None of the mentioned

Answer: b

23. _____ Is an algorithm, a loop that continually moves in the direction of increasing value – that is uphill.

a) Up-Hill Search

b) Hill-Climbing

c) Hill algorithm

d) Reverse-Down-Hill search

Answer: b

24. When will Hill-Climbing algorithm terminate?

a) Stopping criterion met

b) Global Min/Max is achieved

c) No neighbor has higher value

d) All of the mentioned

Answer: c

25. Hill climbing sometimes called _____ because it grabs a good neighbor state without thinking ahead about where to go next.

a) Needy local search

b) Heuristic local search

c) Greedy local search

d) Optimal local search

Answer: c

26. Searching using query on Internet is, use of _____ type of agent.

a) Offline agent

b) Online agent

c) Both Offline & Online agent

d) Goal Based & Online agent

Answer: d

27. Which of the Following problems can be modeled as CSP?

a) 8-Puzzle problem

b) 8-Queen problem

c) Map coloring problem

d) All of the mentioned

Answer: d

27. What among the following constitutes to the incremental formulation of CSP?

a) Path cost

b) Goal cost

c) Successor function

d) All of the mentioned

Answer: d

28. The term _____ is used for a depth-first search that chooses values for one variable at a time and returns when a variable has no legal values left to assign.

a) Forward search

b) Backtrack search

c) Hill algorithm

d) Reverse-Down-Hill search

Answer: b

29. Consider a problem of preparing a schedule for a class of student. What type of problem is this?

a) Search Problem

b) Backtrack Problem

c) CSP

d) Planning Problem

Answer: c

30. Language/Languages used for programming Constraint Programming includes _____

a) Prolog

b) C#

c) C

d) Fortran

Answer: a

31. Which of the following algorithm is generally used CSP search algorithm?

a) Breadth-first search algorithm

b) Depth-first search algorithm

c) Hill-climbing search algorithm

d) None of the mentioned

Answer: b

32. Which is the most straightforward approach for planning algorithm?

a) Best-first search

b) State-space search

c) Depth-first search

d) Hill-climbing search

Answer: b

33. The initial state and the legal moves for each side define the _____ for the game.

a) Search Tree

b) Game Tree

c) State Space Search

d) Forest

Answer: b

34. General algorithm applied on game tree for making decision of win/lose is _____

a) DFS/BFS Search Algorithms

b) Heuristic Search Algorithms

c) Greedy Search Algorithms

d) MIN/MAX Algorithms

Answer: d

35. What is the complexity of minimax algorithm?

a) Same as of DFS

b) Space – bm and time – bm

c) Time – bm and space – bm

d) Same as BFS

Answer: a

36. Which search is equal to minimax search but eliminates the branches that can’t influence the final decision?

a) Depth-first search

b) Breadth-first search

c) Alpha-beta pruning

d) None of the mentioned

Answer: c

37. Which search is similar to minimax search?

a) Hill-climbing search

b) Depth-first search

c) Breadth-first search

d) All of the mentioned

Answer: b

38. Which value is assigned to alpha and beta in the alpha-beta pruning?

a) Alpha = max

b) Beta = min

c) Beta = max

d) Both Alpha = max & Beta = min

Answer: d

39. What is the primary purpose of the Minimax algorithm in game theory?

a) To find the best move for both players

b) To calculate the possible scores of all moves

c) To maximize the minimum gain for a player

d) All of the above

Answer: d

40. Which of the following statements about Alpha-Beta Pruning is true?

a) It guarantees an optimal solution

b) It always finds the best move faster than Minimax

c) It reduces the number of nodes evaluated in the search tree

d) It requires more memory than Minimax

Answer: c