ARTIFICIAL INTELLIGENCE

L T P C

BCSE306L - 3 0 0 3

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Dr. S M SATAPATHY

Associate Professor Sr.,
School of Computer Science and Engineering,
VIT Vellore, TN, India – 632 014.

Module – 2

PROBLEM SOLVING BASED ON SEARCHING

1. Problem Solving
2. Problem Space
3. Uninformed Search
4. Informed Search

INTRODUCTION

Problem?

- A matter or situation regarded as unwelcome or harmful and needing to be dealt with and overcome.

How the problems are solved ?

- Defining the problem.
- Generating alternatives.
- Evaluating and selecting alternatives.
- Implementing solutions

What are the approaches to deal with a problem

- Diagram
- Drill Down
- Plan do check act, Etc..
- Out of many, the best way is to divide the problem into smaller parts and find the link between them. May be a diagrammatic representation may help on this.

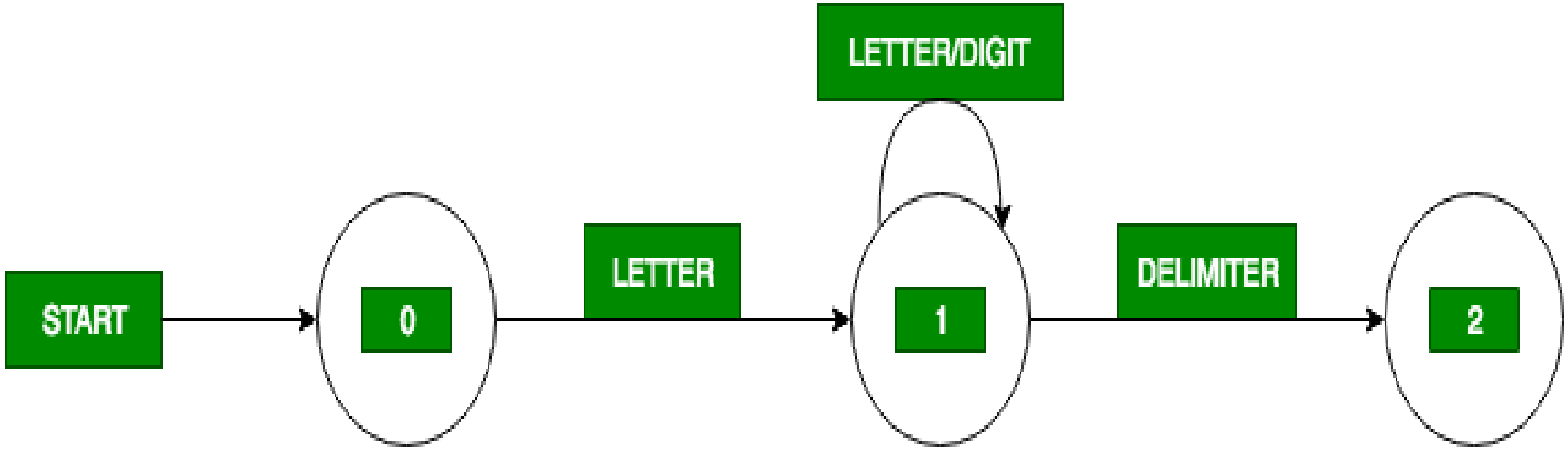
Problem Solving :: Definition

- According to psychology, “a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.”

- According to computer science, a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem.

Search Algorithm in AI

- Searching in AI is the process of navigating from a starting state to a goal state by transitioning through intermediate states.



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

Problem Solving Agents

- In Artificial Intelligence, Search techniques are universal problem-solving methods.

- Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result.

- Problem-solving agents are the goal-based agents and use atomic representation and focuses on satisfying the goal.

Problem Solving Agents :: Solving a Problem

- After formulating a goal and problem to solve the agent calls a search procedure to solve it. A problem can be defined by 5 components.

- The initial state: The state from which agent will start.
- The goal state: The state to be finally reached.
- The current state: The state at which the agent is present after starting from the initial state.
- Operator or Successor function: It is the description of possible actions and their outcomes.
- Path cost: It is a function that assigns a numeric cost to each path.

Problem Solving Agents :: Solving a Problem

State Space

- Initial state, actions, and transition model together define the state-space of the problem implicitly.
- 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.

Problem Solving Agents :: Solving a Problem

- Problem solving involves
  - Problem definition
  - Problem analysis
  - Knowledge representation
  - Problem solving (selection of best techniques)

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH( problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

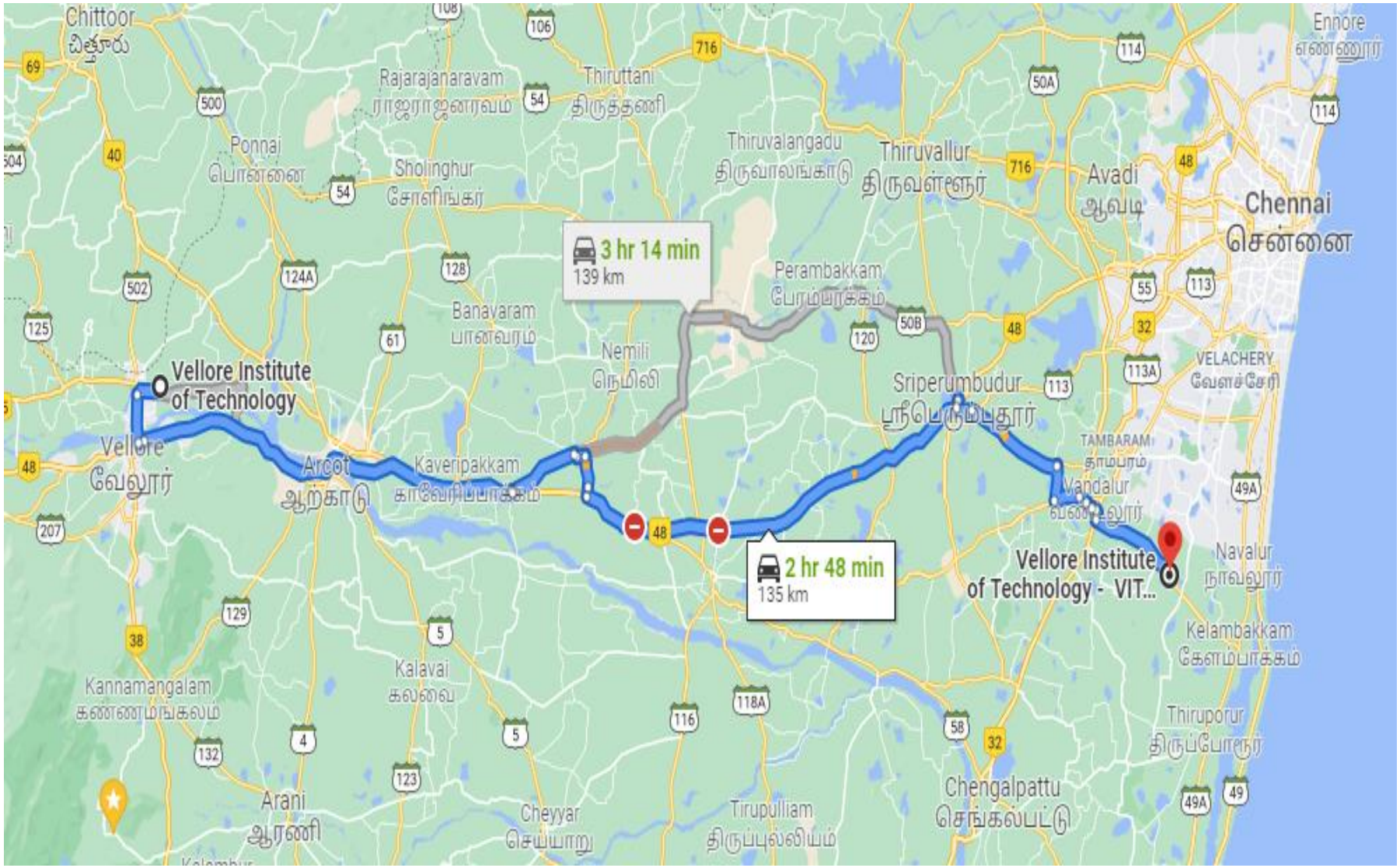
Problem Solving Agents :: Solving a Problem

Problem Approach

- 2 types
- Toy Problem: It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
- Real-world Problem: It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

Problem Solving Agents :: Solving a Problem - Example

- Shortest path between VIT Vellore Campus to VIT Chennai Campus



Problem Solving Agents :: Solving a Problem - Example

- To reach the campus in shortest route as well as cross the minimum number of Toll-gate's

Formulate goal:

- Shortest path

Formulate problem:

- states: various urban areas
- actions: drive between cities

Find solution:

- sequence of urban areas, e.g., Kancheepuram, Thambaram, Kelambakkam, Guindy

Problem Types

Single State problem (Fully observable, Deterministic, Static, Discrete)

- Complete world state knowledge
- Complete action knowledge
- The agent always knows its world state

Multiple State problem (Partially observable, Deterministic, Static, Discrete)

- Incomplete world state knowledge
- Incomplete action knowledge
- The agent only knows which group of world state it is in

Contingency problem (Partially observable, Non-Deterministic)

- It is impossible to define a complete sequence of actions that constitute a solution in advance because information about the intermediary state is unknown

Exploration problem (Unobservable)

- State space and effects of action unknown. Difficult!

Problem Types

Deterministic, fully observable ➜ single-state problem

- Agent knows exactly which state it will be in; solution is a sequence

Non-observable ➜ sensorless problem (conformant problem)

- Agent may have no idea where it is; solution is a sequence

Nondeterministic and/or partially observable ➜ contingency problem

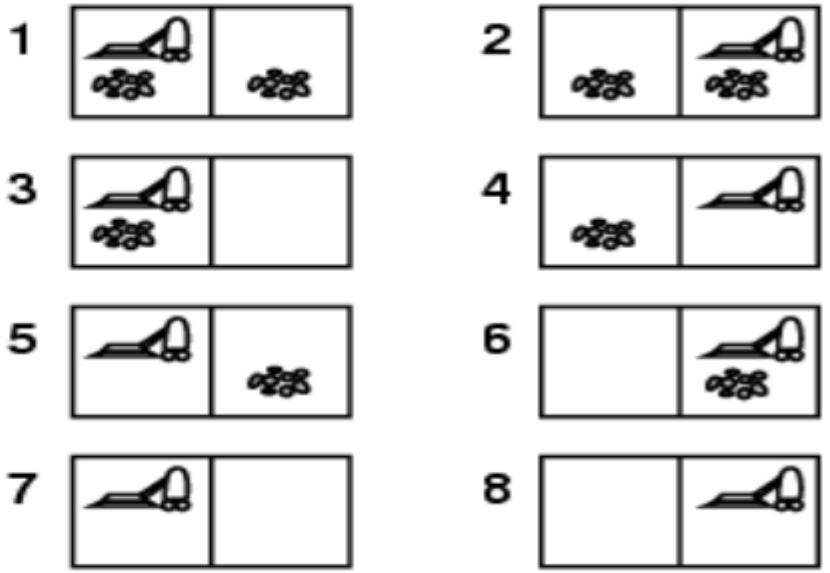
- percepts provide new information about current state
- often interleave, search, execution

Unknown state space ➜ exploration problem

Problem Types :: Vacuum World

Single-state, start in #5.

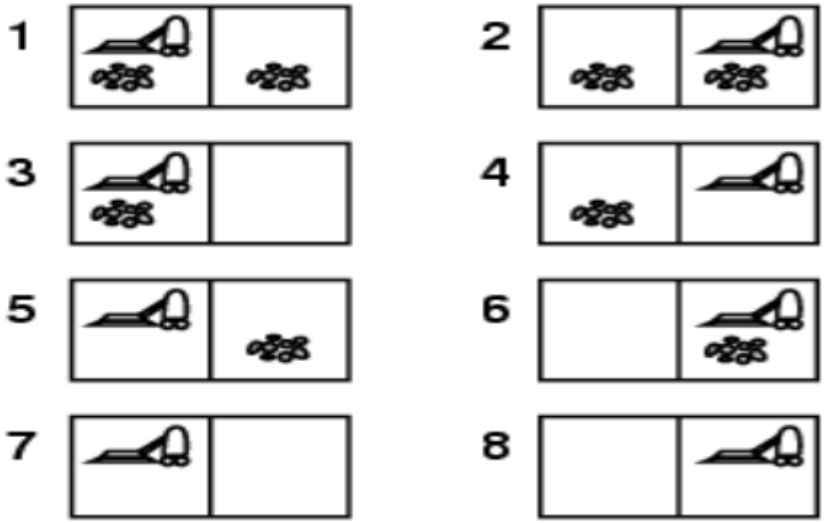
Solution?



Single-state, start in #5.

Solution?

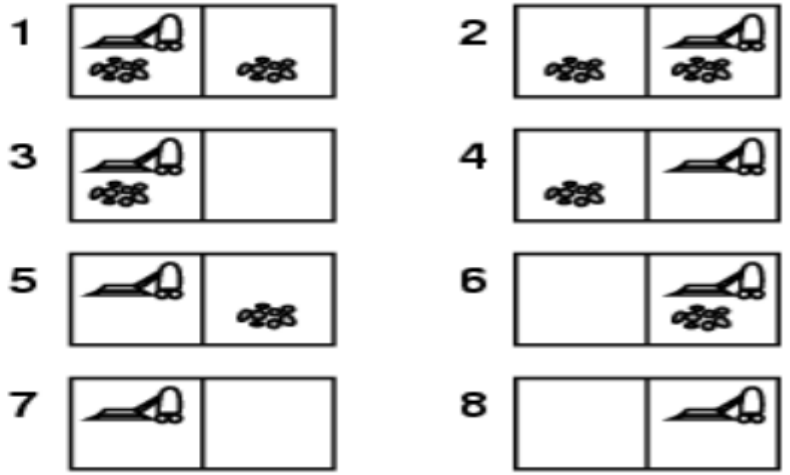
[Right, Suck]



Problem Types :: Vacuum World

Sensorless, start in $\{1,2,3,4,5,6,7,8\}$ e.g., *Right* goes to $\{2,4,6,8\}$

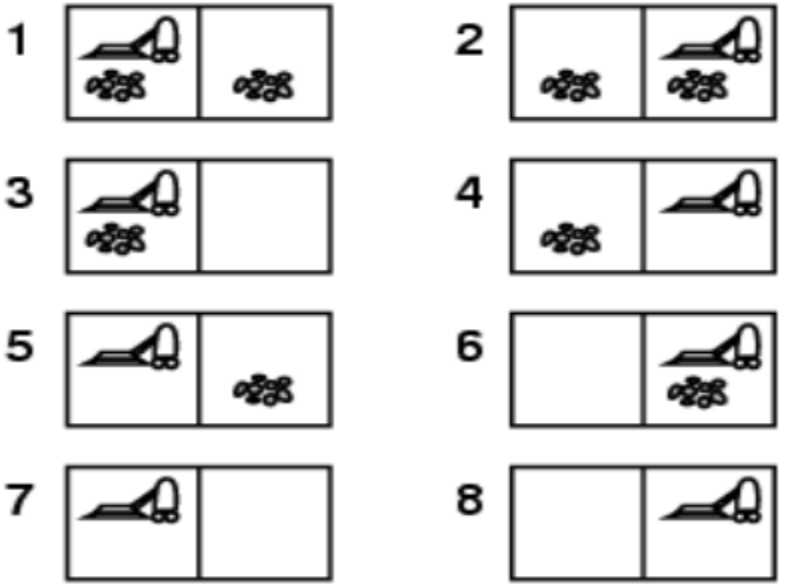
Solution?



Sensorless, start in $\{1,2,3,4,5,6,7,8\}$ e.g., *Right* goes to $\{2,4,6,8\}$

Solution?

[Right,Suck,Left,Suck]

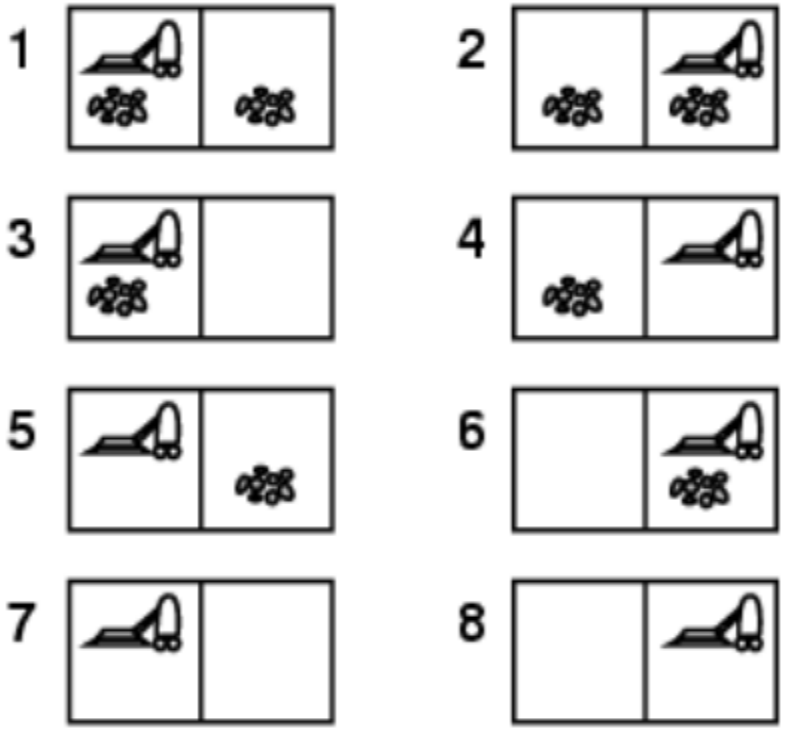


Problem Types :: Vacuum World

Contingency

- Nondeterministic: *Suck* may dirty a clean carpet [Murphy's law]
- Partially observable: location, dirt at current location.
- Percept: *[L, Clean]*, i.e., start in #5 or #7

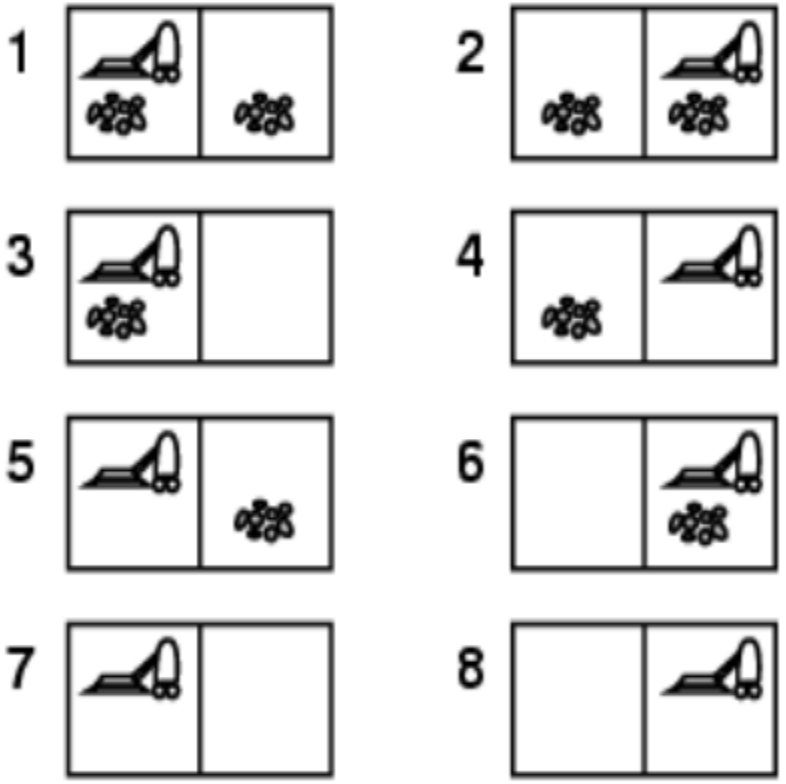
Solution?



Problem Types :: Vacuum World

Contingency

- Nondeterministic: *Suck* may dirty a clean carpet [Murphy's law]
- Partially observable: location, dirt at current location.
- Percept: *[L, Clean]*, i.e., start in #5 or #7
  Solution?
- *[Right, **if** dirt **then** Suck]*



Single State Problem Formulation

A problem is defined by four items:

1. Initial state e.g., "at Katpadi"
2. Actions or successor function S(x) = set of action–state pairs

   e.g., S(Katpadi) = {<Wallaja → Kancheepuram, Padappai>, … }
3. Goal test, can be

   explicit, e.g., x = "at VIT Chennai"

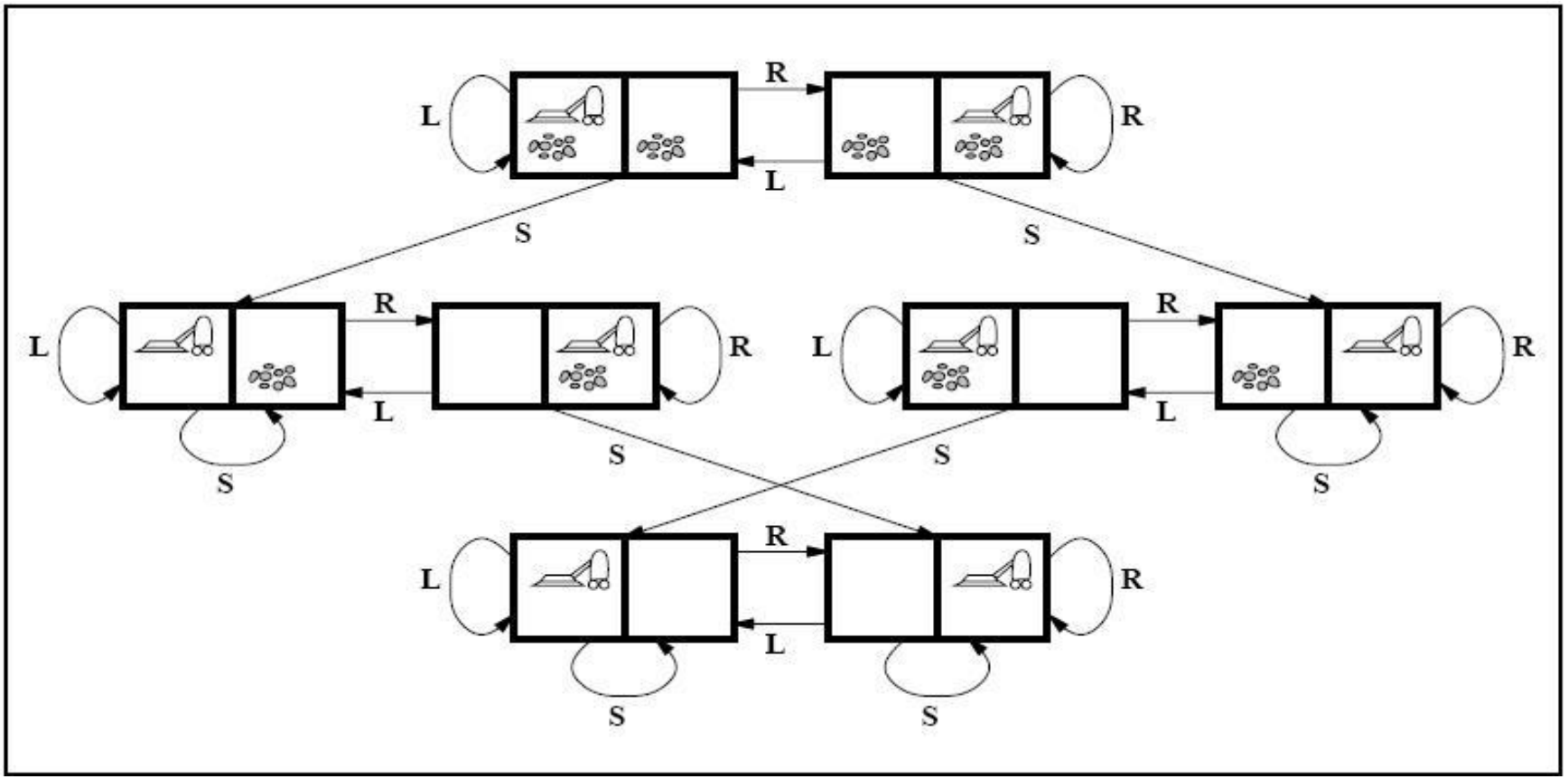
   implicit, e.g., Cross-Min no. of Toll-gate or Shortest path(x)
4. Path cost (additive)

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

   c(x,a,y) is the step cost, assumed to be ≥ 0

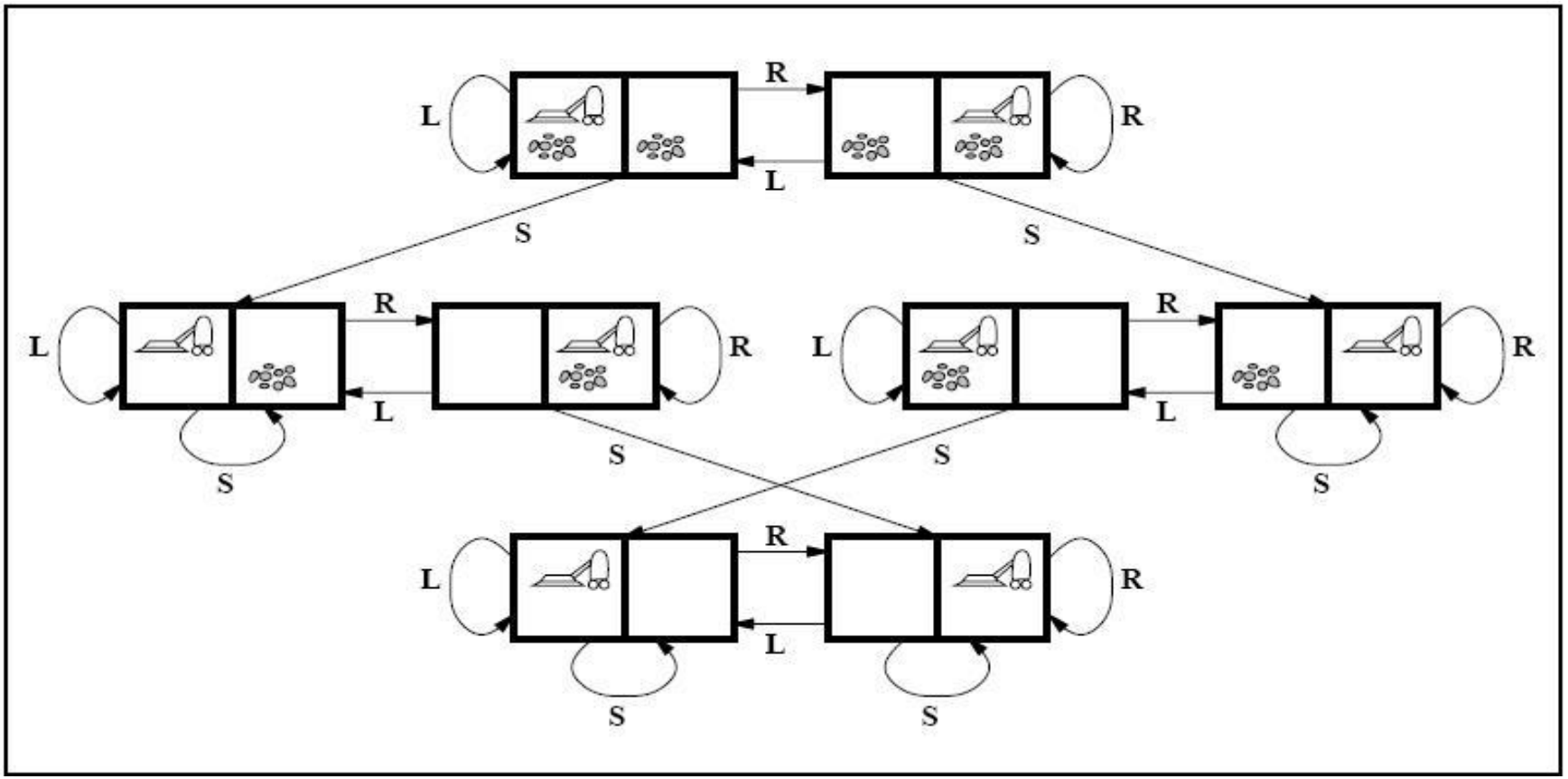
A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space :: Vacuum World Problem



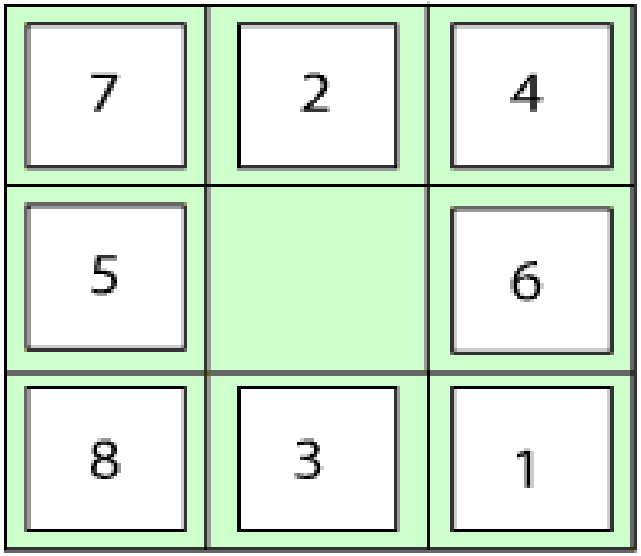
- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Vacuum World Problem

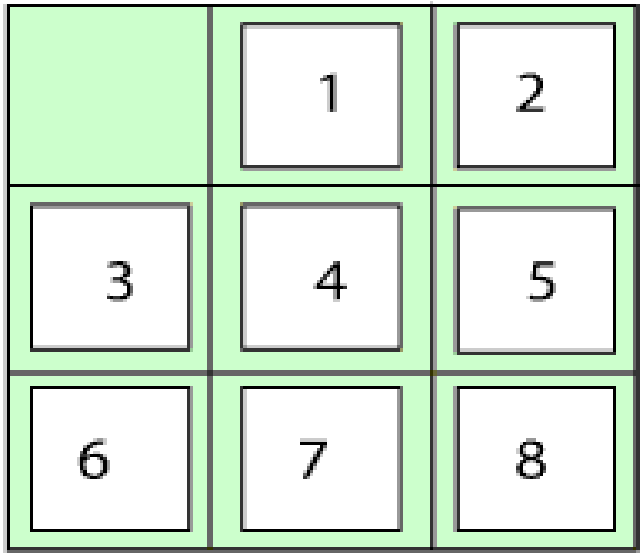


- states - integer dirt and robot location
- actions - Left, Right, Suck
- goal test - no dirt at all locations
- path cost - 1 per action

Selecting a state space :: 8-Puzzle Problem



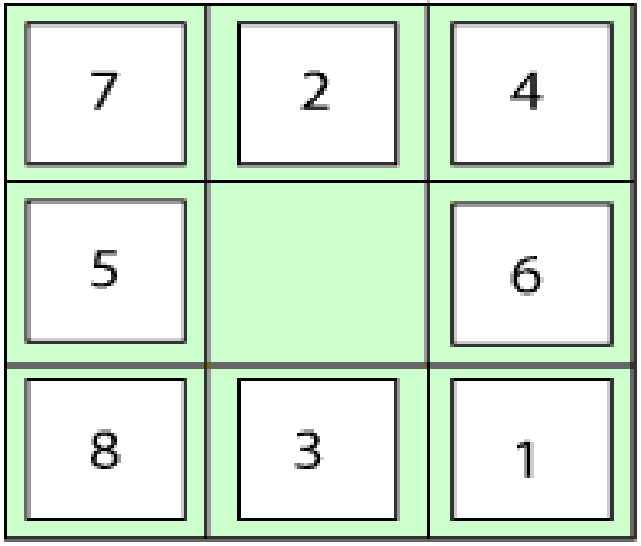
Start State



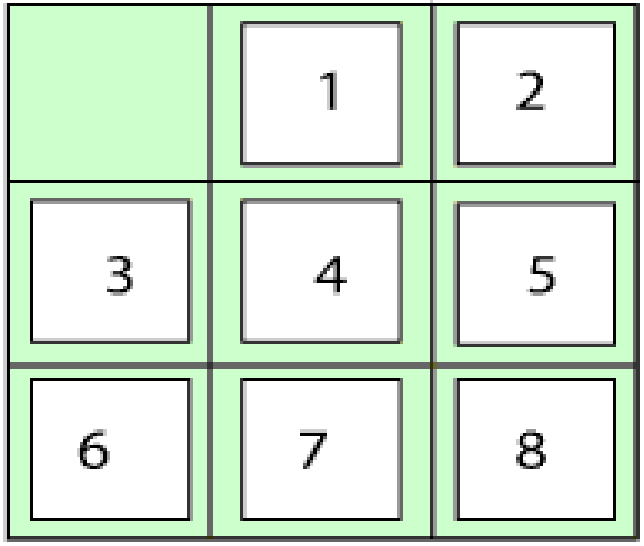
Goal State

- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Vacuum World Problem



Start State

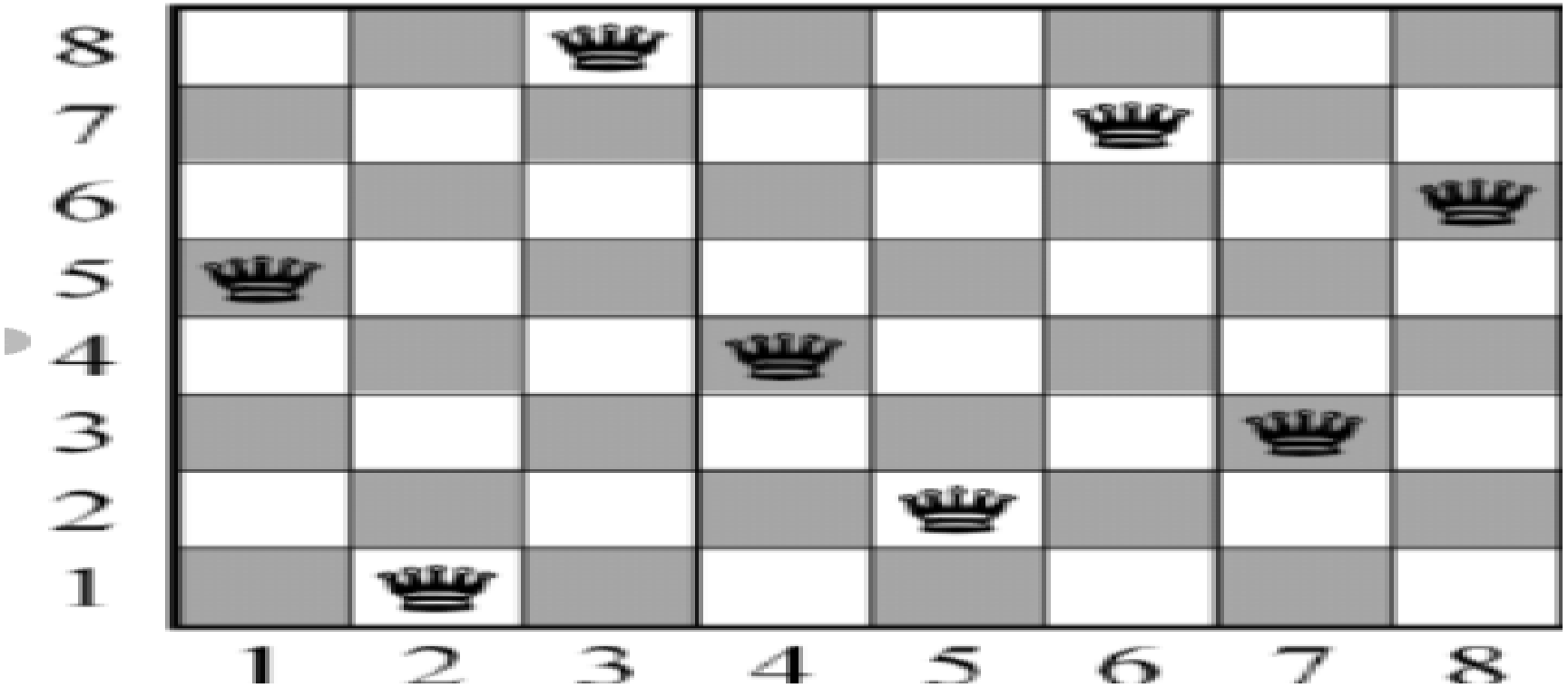


Goal State

- states - locations of tiles
- actions - move blank left, right, up, down
- goal test - goal state (given)
- path cost - 1 per move

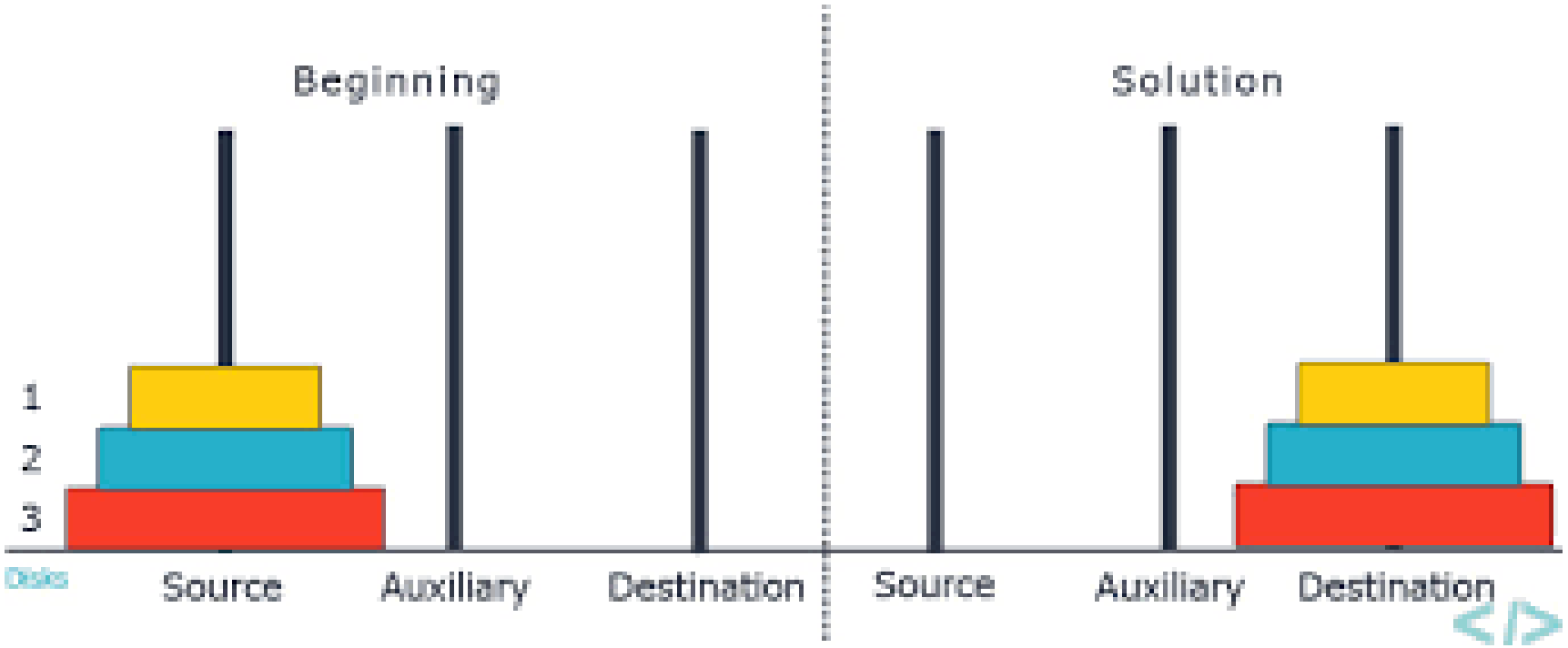
[Note: optimal solution of n-Puzzle family is NP-hard]

Selecting a state space :: 8-Queens Problem



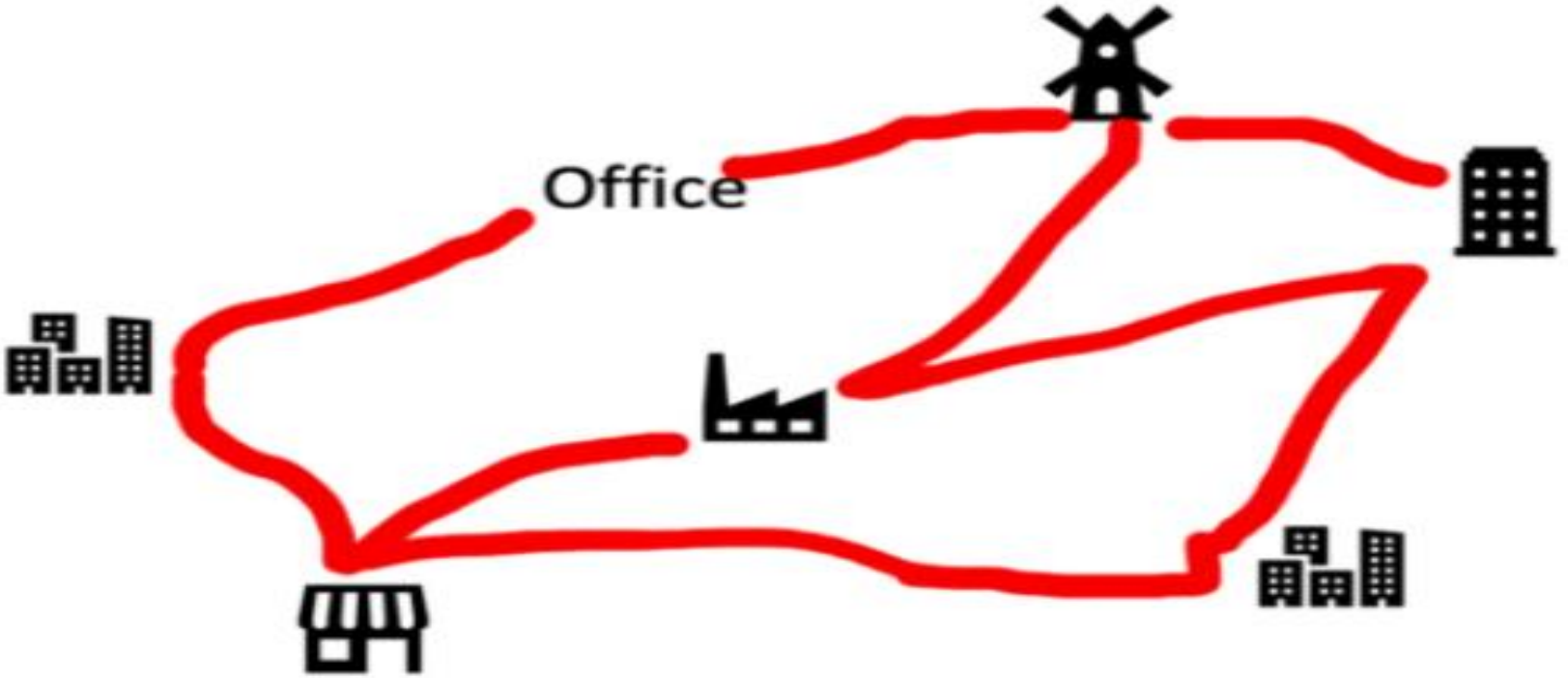
- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Towers of Hanoi Problem



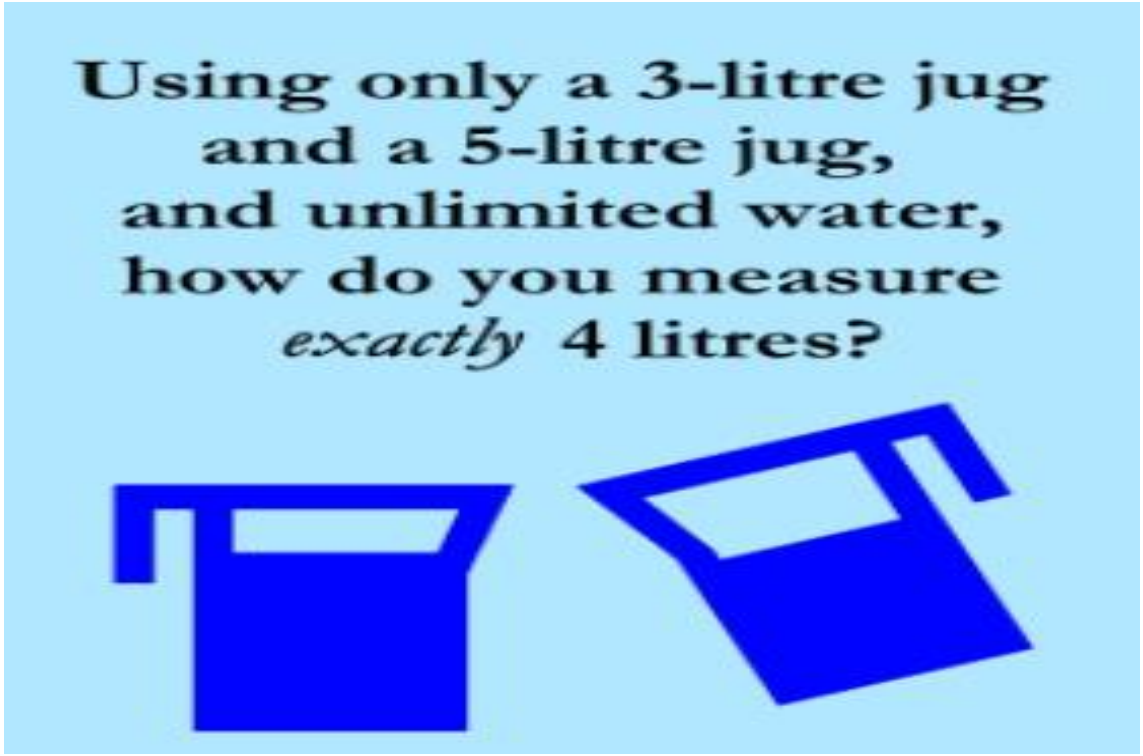
- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Travelling Salesman Problem



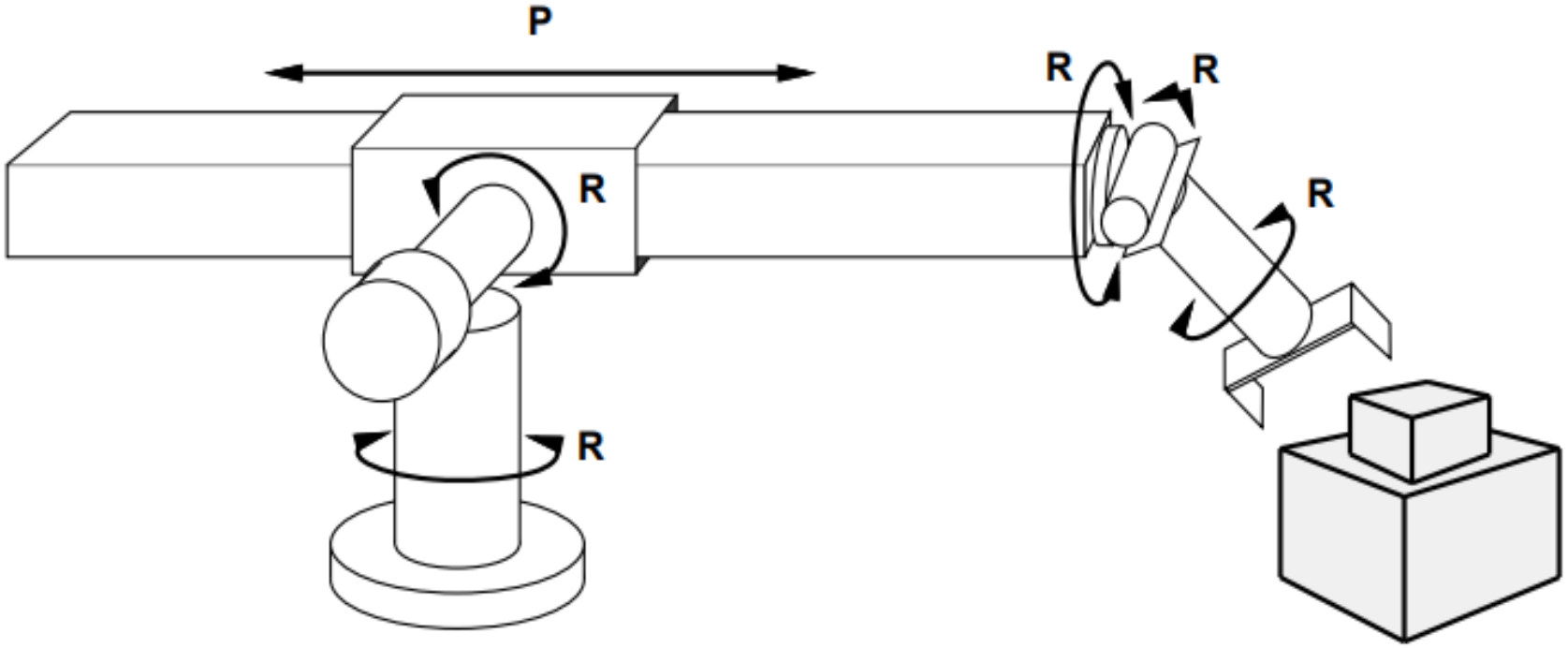
- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Water Jug Problem



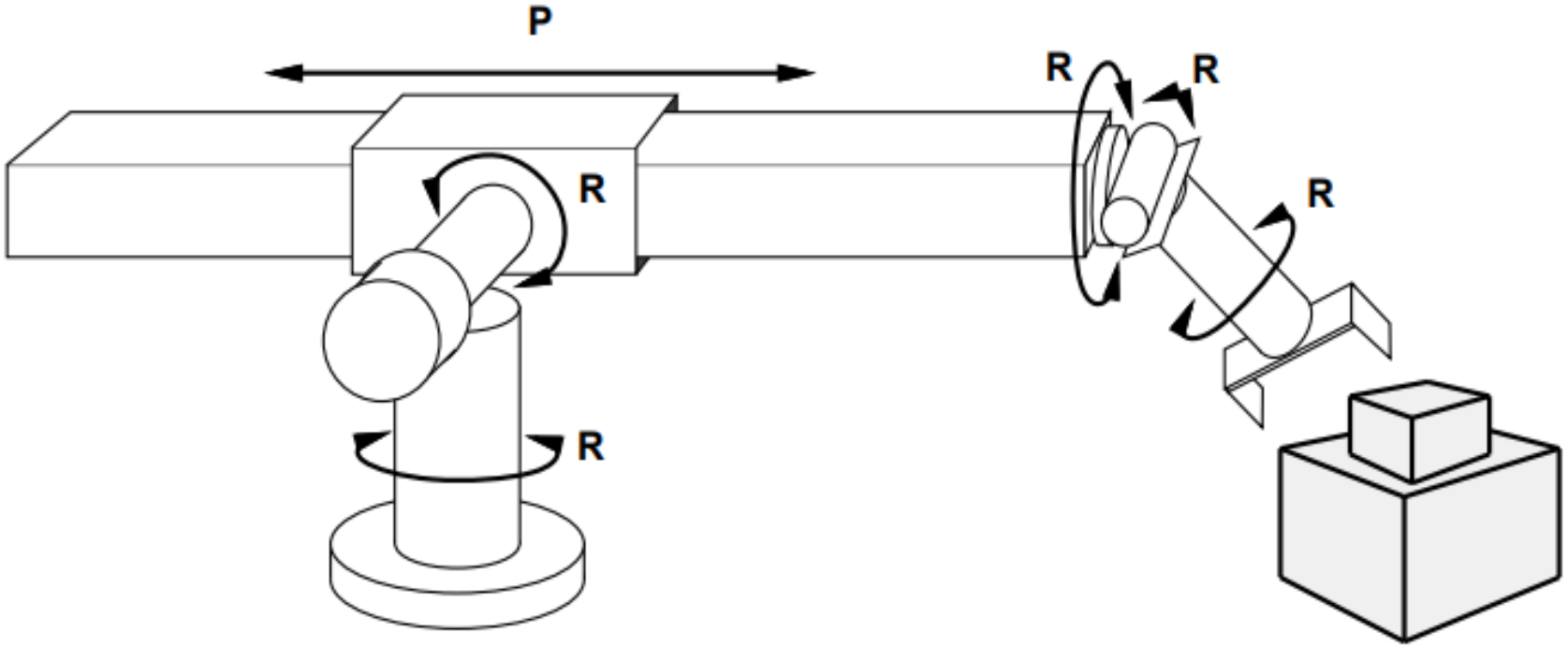
- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Robotic Assessmbly Problem



- states?
- actions?
- goal test?
- path cost?

Selecting a state space :: Robotic Assessmbly Problem



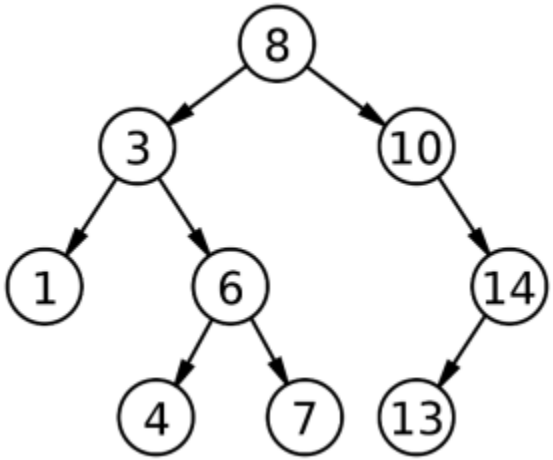
- states - real-valued coordinates of robot joint angles and parts of the object to be assembled
- actions - continuous motions of robot joints
- goal test - assembly complete?
- path cost - time to execute

Search Tree

- A search tree is a tree data structure used for locating specific keys from within a set.
- Search trees are often used to implement an associative array.

Basic Idea

- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)



Search Tree :: Search Algorithm Terminologies

- Search: Searching is a step by step procedure to solve a search-problem in a given search space.
- A search problem can have three main factors:
  1) Search Space: Search space represents a set of possible solutions, which a system may have.
  2) Start State: It is a state from where agent begins the search.
  3) Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.
- Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- Actions: It gives the description of all the available actions to the agent.
- Transition model: A description of what each action do, can be represented as a transition model.

Search Tree :: Search Algorithm Terminologies

- Path Cost: It is a function which assigns a numeric cost to each path.
- Solution: It is an action sequence which leads from the start node to the goal node.
- Optimal Solution: If a solution has the lowest cost among all solutions.

Properties of Search Algorithms

Measuring Problem Solving Performance

- We can evaluate a search algorithm's performance in four ways:
  - Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

    Is the algorithm guaranteed to find a solution when there is one?

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

    Does the strategy find the optimal solution?

Properties of Search Algorithms

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

How long does it take to find a solution?

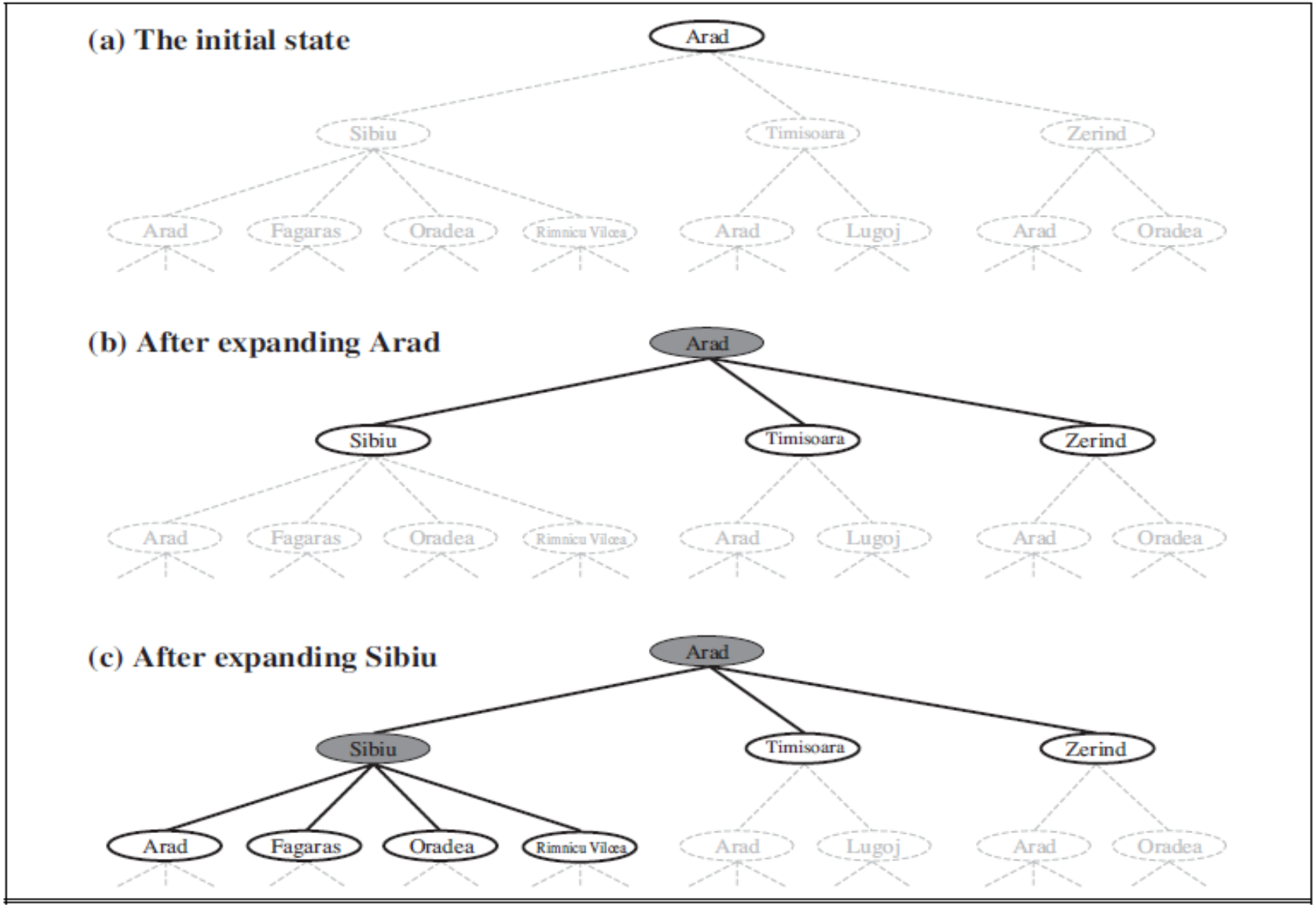
- Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

How much memory is needed to perform the search?

Tree Search Algorithms

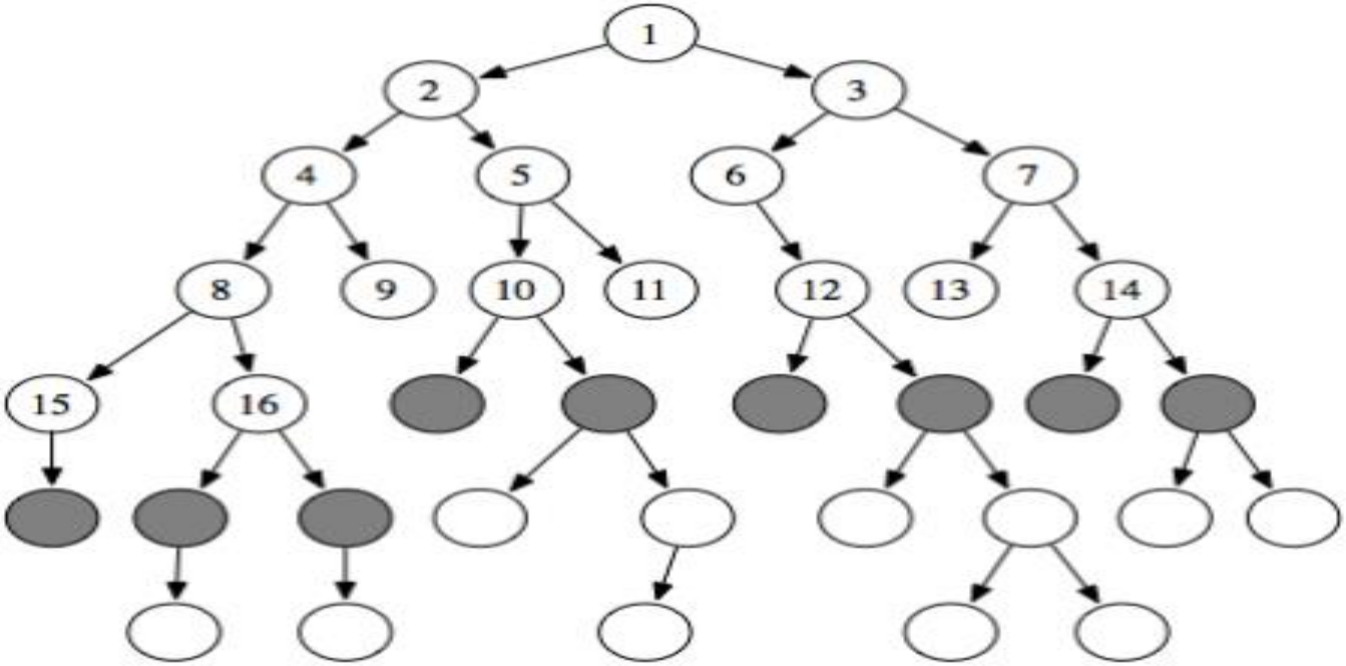
```
function Tree-Search(problem, fringe) returns a solution, or failure fringe ←
    Insert(Make-Node(Initial-State[problem]),fringe)  loop do
        if fringe is empty then return failure
        node ← Remove-Front(fringe)
        if Goal-Test(problem,State(node)) then return node fringe ←
        InsertAll(Expand(node,problem),fringe)

function Expand(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in Successor-Fn(problem,State[node]) do
        s ← a new Node
        Parent-Node[s]← node;  Action[s]← action;  State[s]← result
        Path-Cost[s] ← Path-Cost[node] + Step-Cost(node,action,s)
        Depth[s]← Depth[node] + 1  add s to
        successors
    return successors
```

Tree Search :: Example



Fringe / Frontier / Border

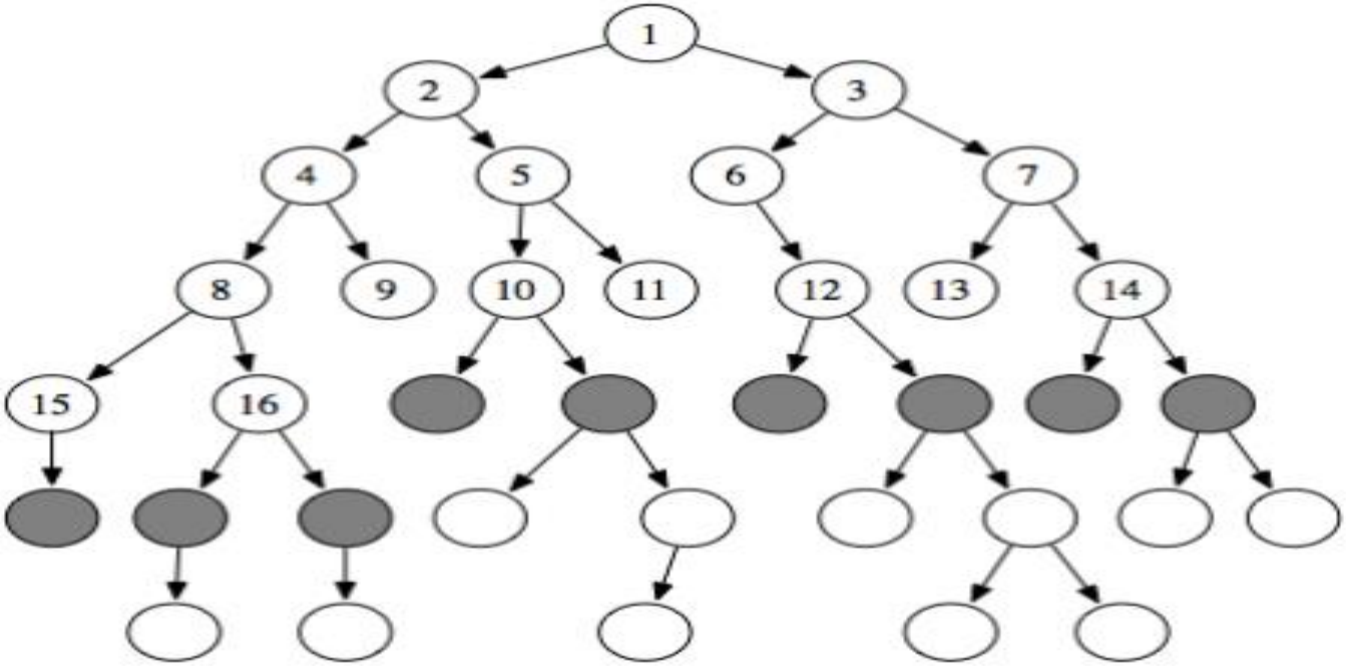
- If you're performing a tree (or graph) search, then the set of all nodes at the end of all visited paths is called the fringe, frontier or border.
- A fringe, which is a data structure used to store all the possible states (nodes) that you can go from the current states.



- The grey nodes (the lastly visited nodes of each path) form the fringe.

Fringe / Frontier / Border

- If you're performing a tree (or graph) search, then the set of all nodes at the end of all visited paths is called the fringe, frontier or border.
- A fringe, which is a data structure used to store all the possible states (nodes) that you can go from the current states.



- The grey nodes (the lastly visited nodes of each path) form the fringe.

Search Strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - b: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be ∞)

Search Strategies :: Search Algorithm Types

Uninformed Search (Blind)

- No domain knowledge
- Only knows how to traverse to find the leaf and goal nodes.
- No proper way to reach the goal in the search space
- Blind search

Informed Search

- Use domain knowledge
- Problem definition guides to solve the problem
- More efficient since they are guided
- Heuristic search
- Heuristic search guarantees good solution with reasonable time. (May not be best solution)
- Used to solve complex problem by various alternative ways

Search Strategies :: Search Algorithm Types

Uninformed Search (Blind)

- Uninformed search algorithms have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and length of actions.

Informed Search

- Informed Search algorithms have information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state.

Search Strategies :: Search Algorithm Types

Types of Uninformed Search algorithms

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

Types of Informed Search

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

UNINFORMED SEARCH

Search Algorithm Types :: Breadth-first Search

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

https://www.youtube.com/watch?v=QRq6p9s8NVg

Search Algorithm Types :: Breadth-first Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

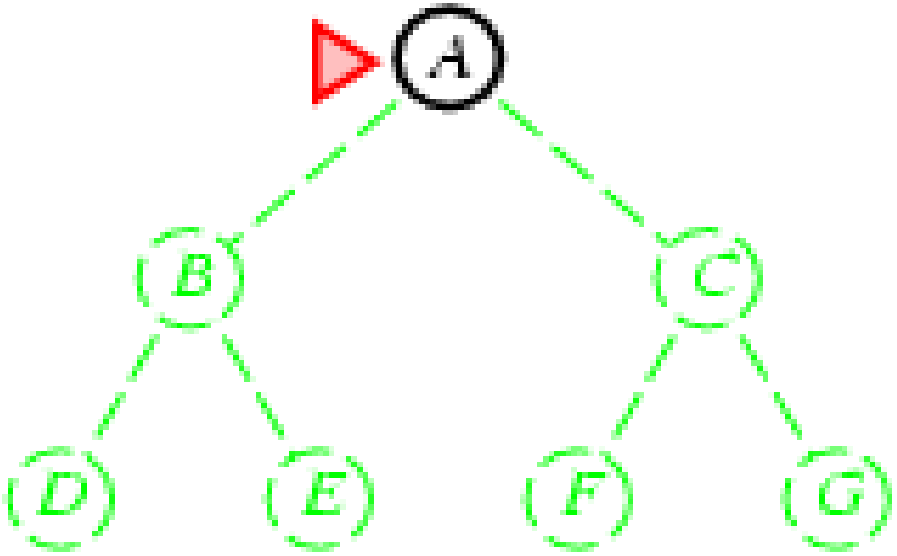
Search Algorithm Types :: Breadth-first Search

Idea

- Expand shallowest unexpanded node

Implementation

- fringe is a FIFO queue, i.e. successors go in at the end of the queue



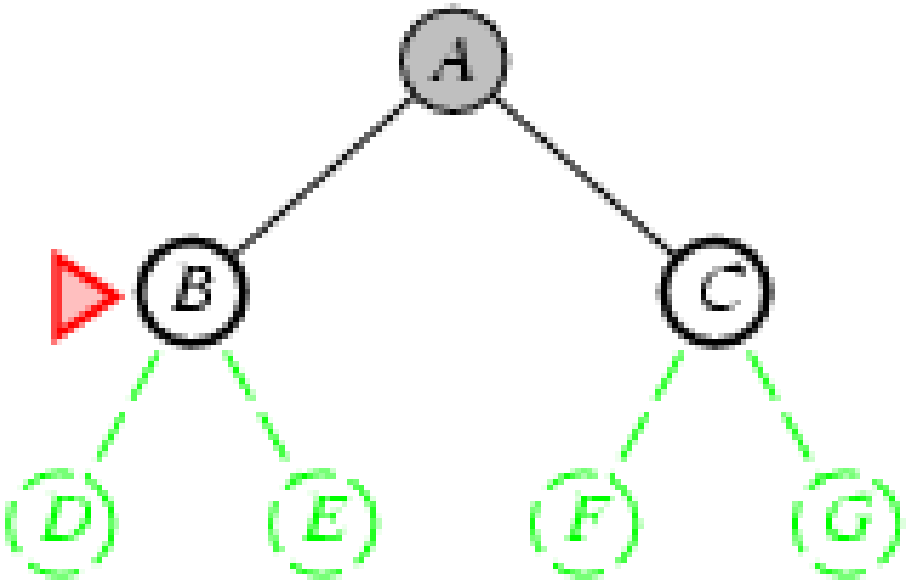
Search Algorithm Types :: Breadth-first Search

Idea

- Expand shallowest unexpanded node

Implementation

- fringe is a FIFO queue, i.e. successors go in at the end of the queue



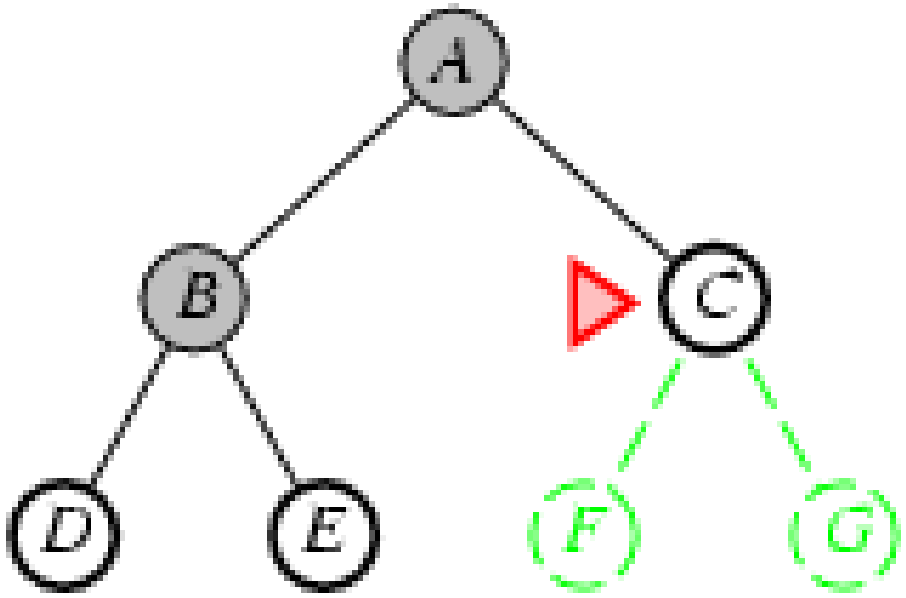
Search Algorithm Types :: Breadth-first Search

Idea

- Expand shallowest unexpanded node

Implementation

- fringe is a FIFO queue, i.e. successors go in at the end of the queue



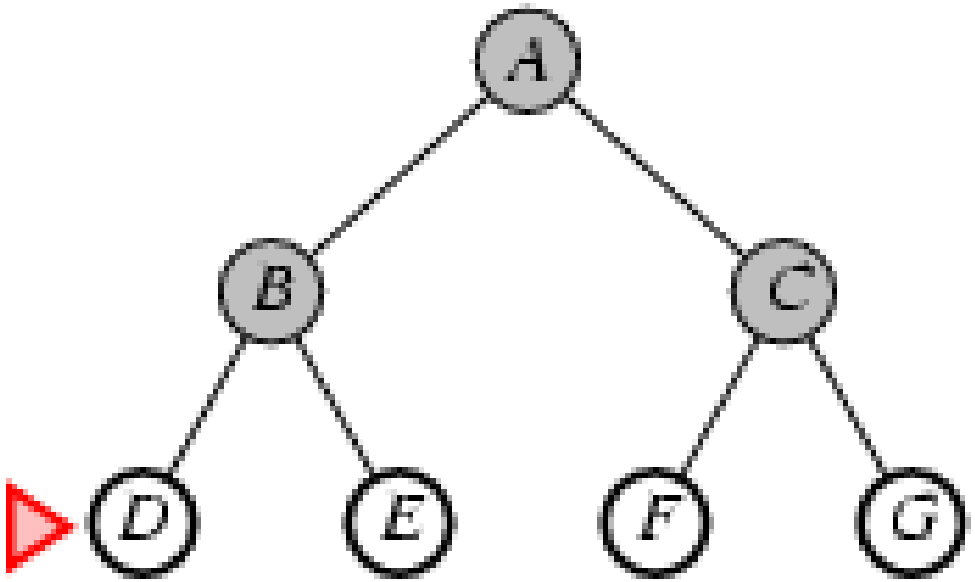
Search Algorithm Types :: Breadth-first Search

Idea

- Expand shallowest unexpanded node

Implementation

- fringe is a FIFO queue, i.e. successors go in at the end of the queue



Search Algorithm Types :: Breadth-first Search

Time Complexity: $O(b^d)$

- Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node.

Where d = depth of shallowest solution and b is the number of nodes in every level, then $T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$

Space Complexity: $O(b^d)$

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

Completeness: Yes (if b is finite)

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

Optimality: Yes (if cost = 1 per step)

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

Search Algorithm Types :: Breadth-first Search

Advantages:

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

Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.
- Space is the bigger problem (more than time)

Search Algorithm Types :: Depth-first Search

- 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.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

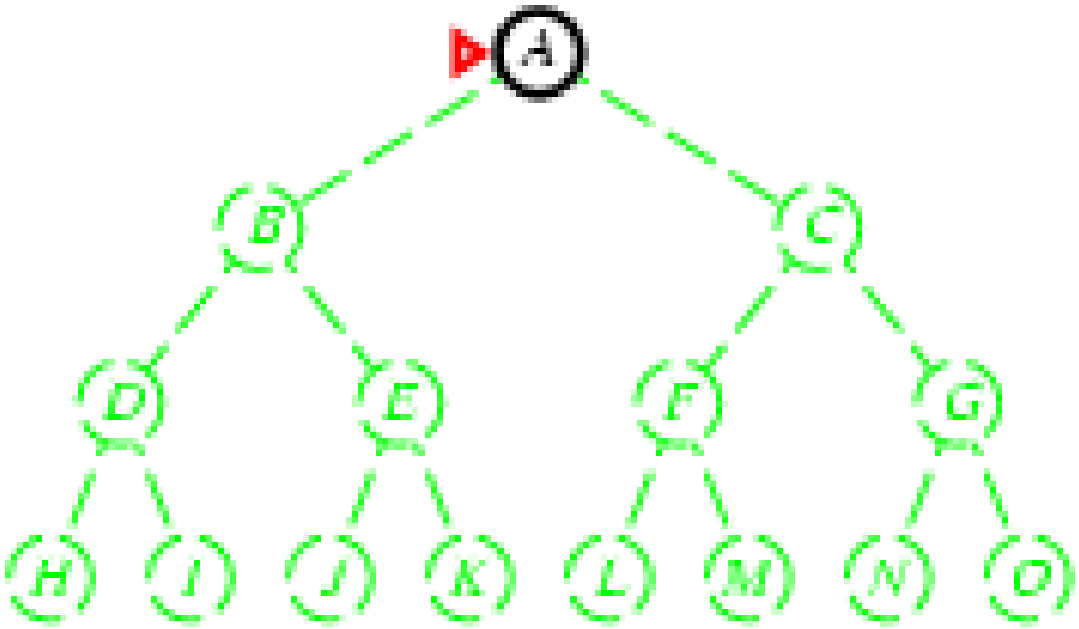
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



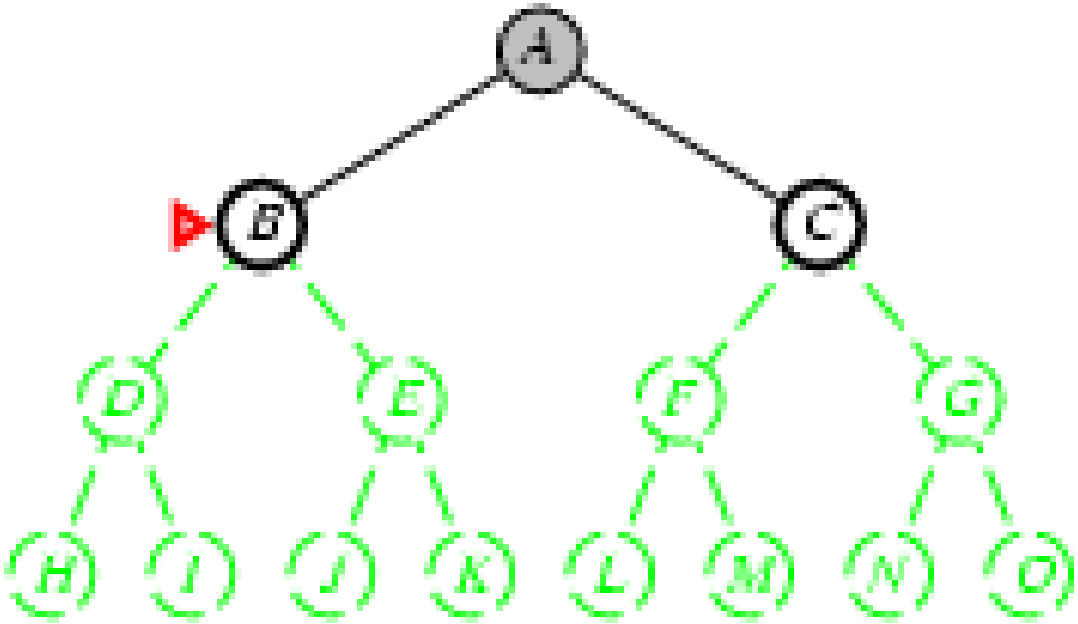
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



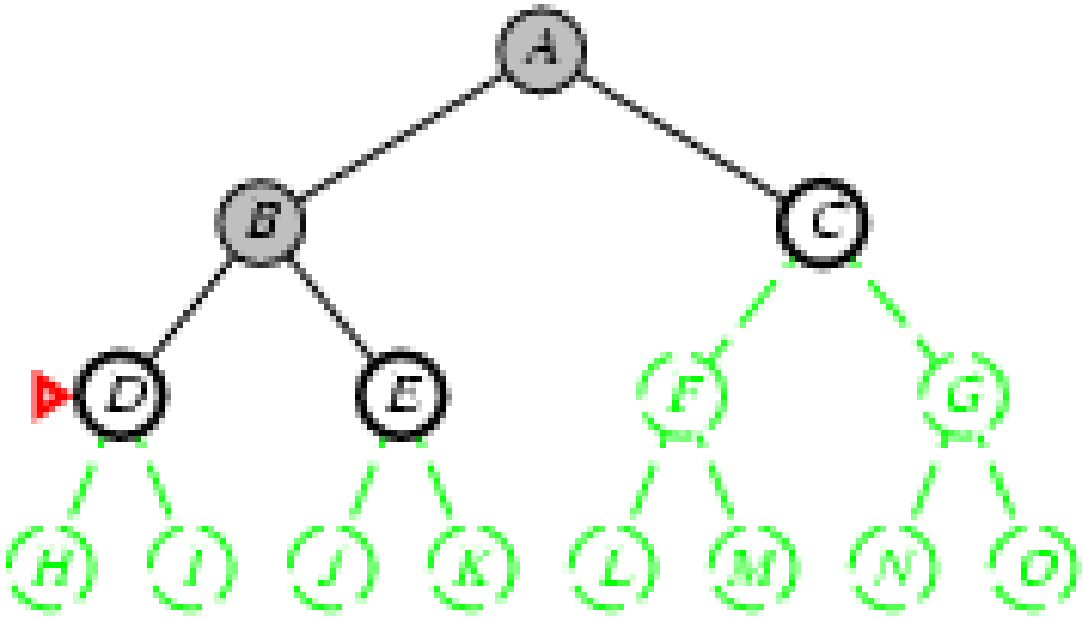
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



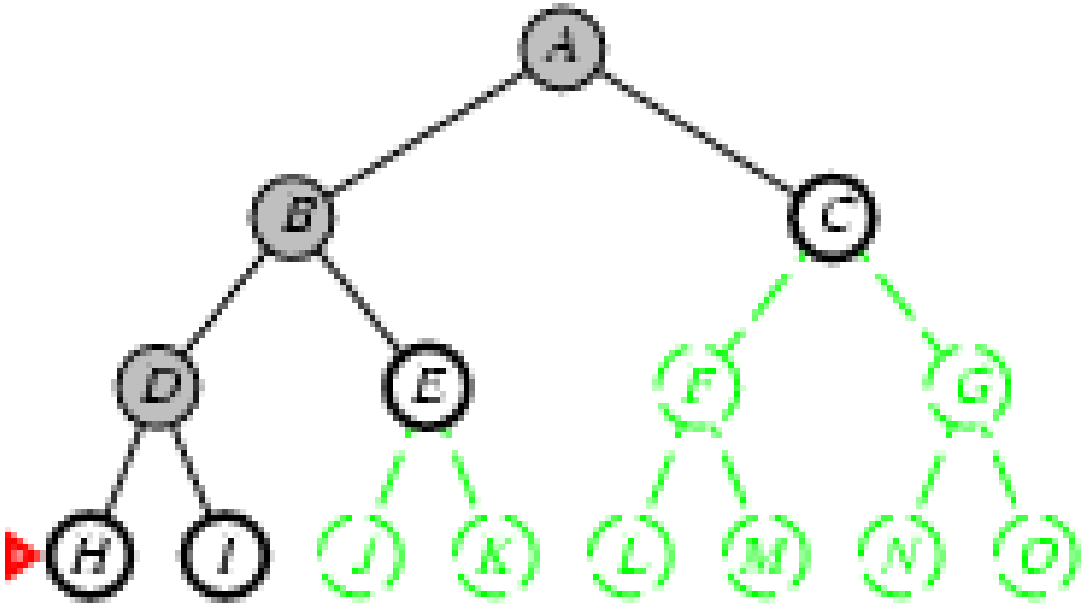
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



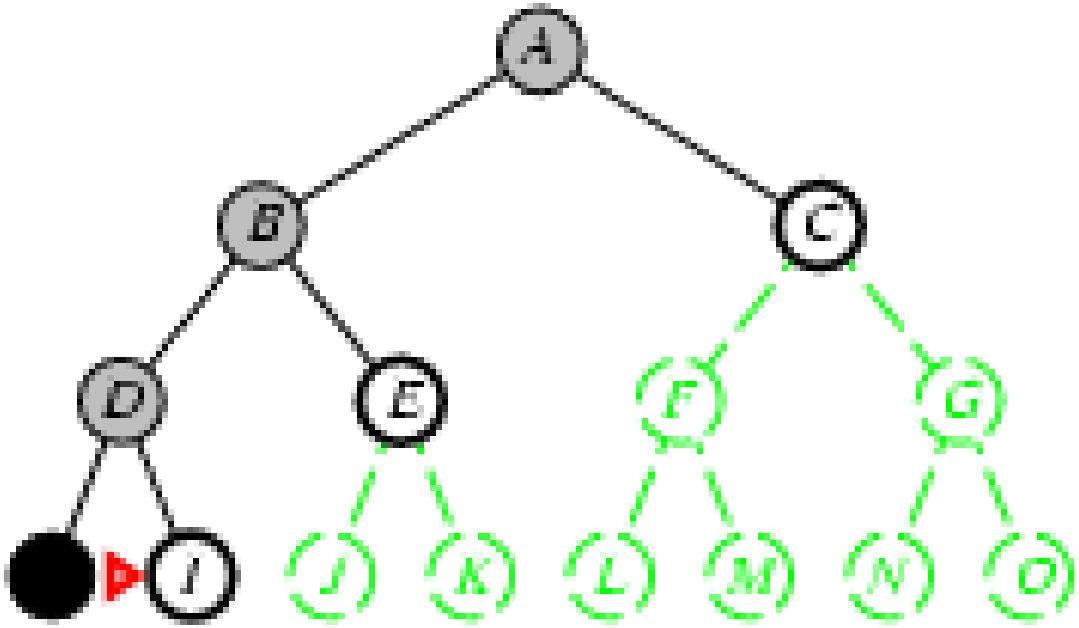
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



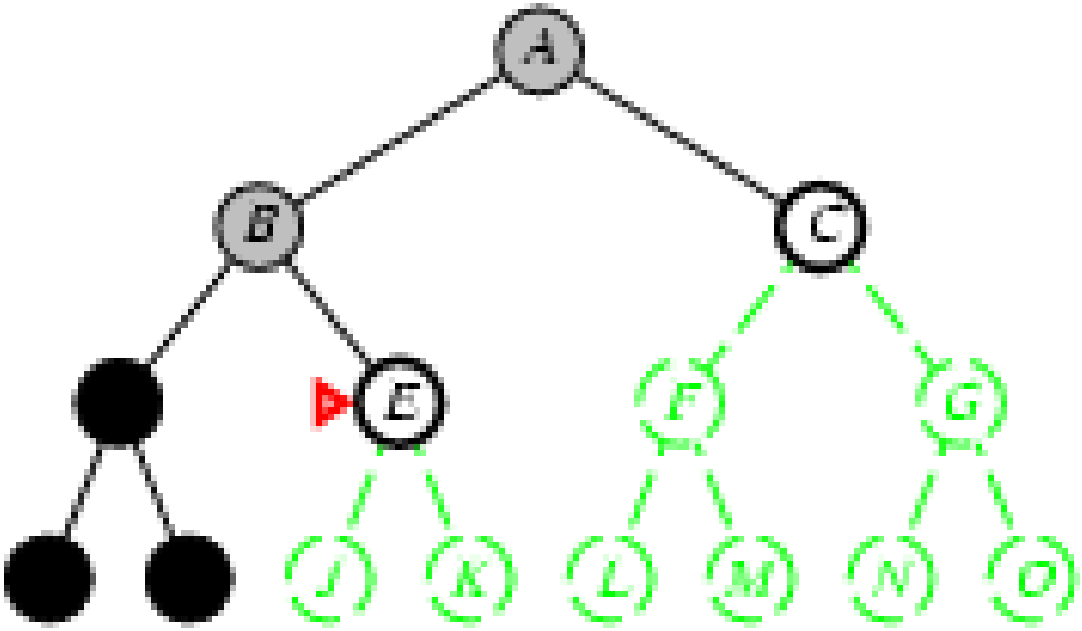
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



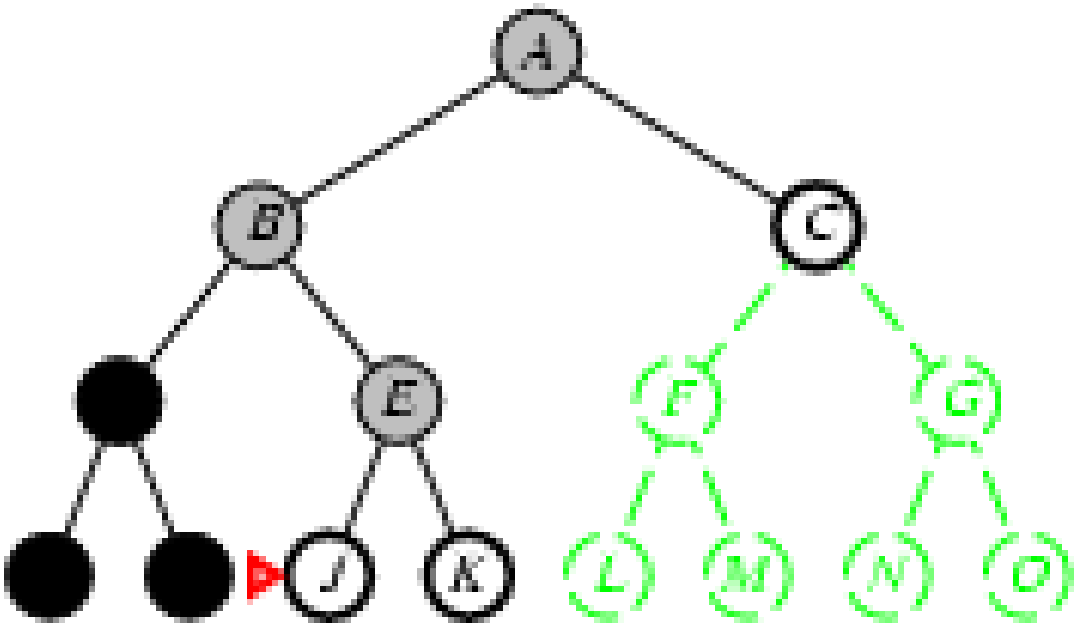
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



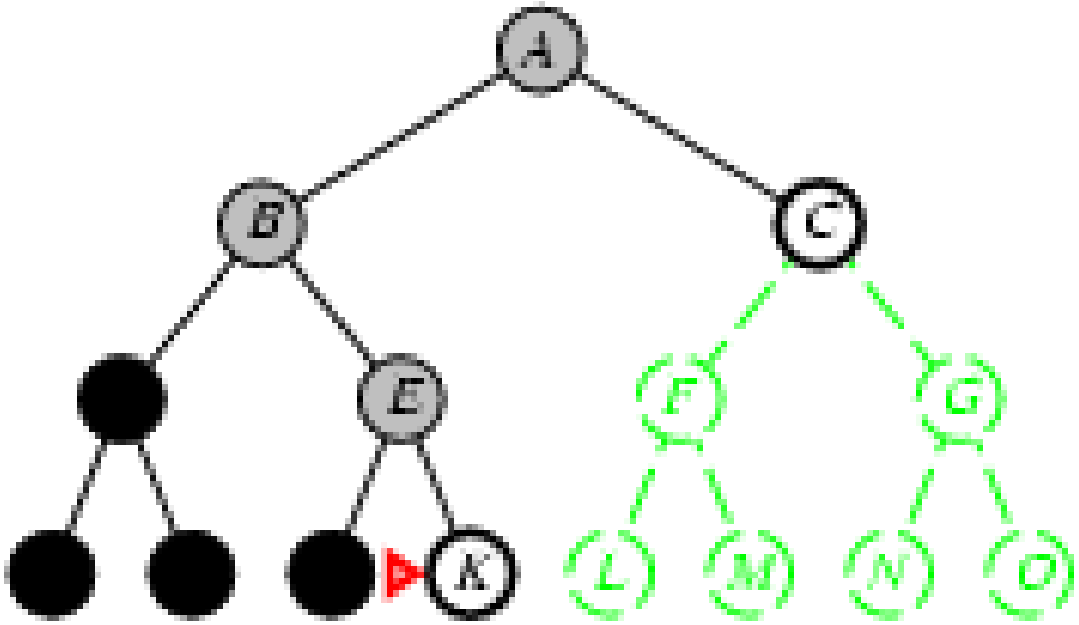
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



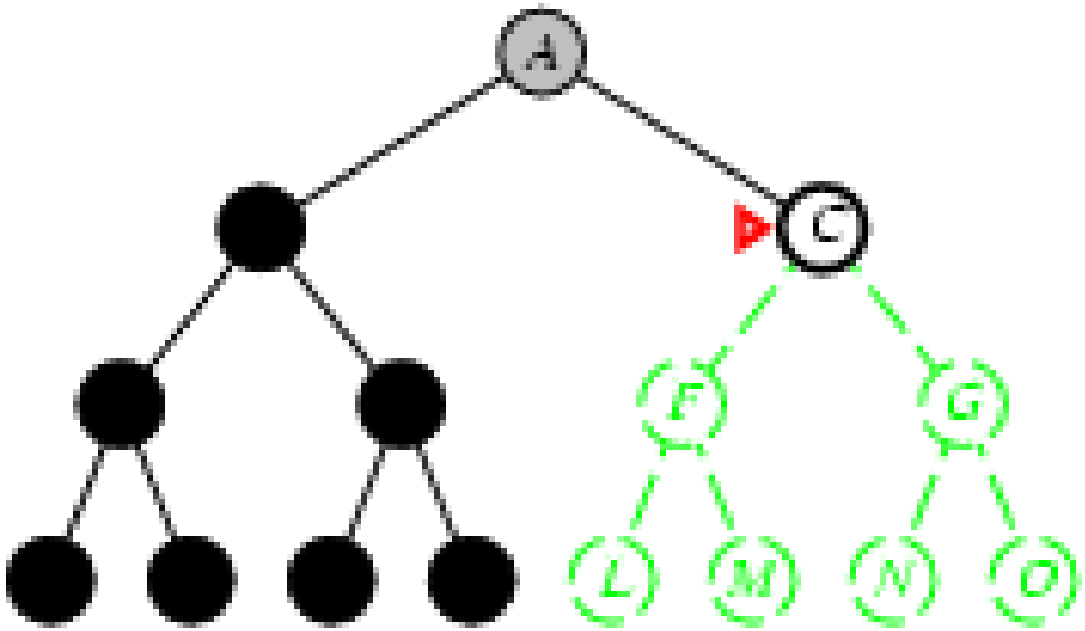
Search Algorithm Types :: Depth-first Search

Idea

- Expand deepest unexpanded node

Implementation

- fringe is a LIFO queue, i.e. put successors at front



Search Algorithm Types :: Depth-first Search

Time Complexity: $O(n^m)$, terrible if m is much larger than d

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

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$, Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: $O(n \times m)$ i.e., linear space!

- Space complexity of DFS algorithm is $O(n \times m)$.

Completeness: No: fails in infinite-depth spaces, spaces with loops

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

Optimality: No

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

Search Algorithm Types :: Depth-first Search

Advantages:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path). If solutions are dense, may be much faster than breadth-first.

Disadvantages:

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

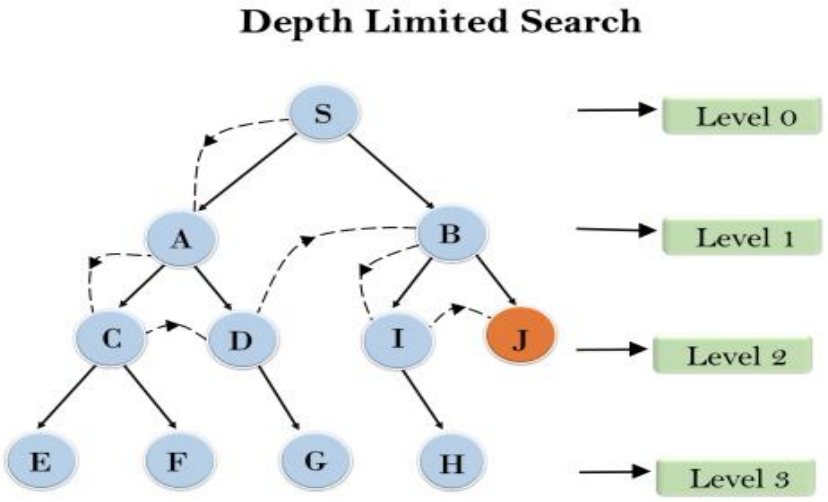
Search Algorithm Types :: Depth-limited Search

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit.
- Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will be treated as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
  - Standard failure value: It indicates that problem does not have any solution.
  - Cutoff failure value: It defines no solution for the problem within a given depth limit
- It follows depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors.

Search Algorithm Types :: Depth-limited Search

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Process: If depth is fixed to 2, DLS carries out depth first search till second level in the search tree.



Search Algorithm Types :: Depth-limited Search

Time Complexity:

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

Space Complexity:

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

Completeness:

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

Optimality:

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

Search Algorithm Types :: Depth-limited Search

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.

Search Algorithm Types :: Iterative Deepening Depth-first Search

- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don’t expand beyond depth L.

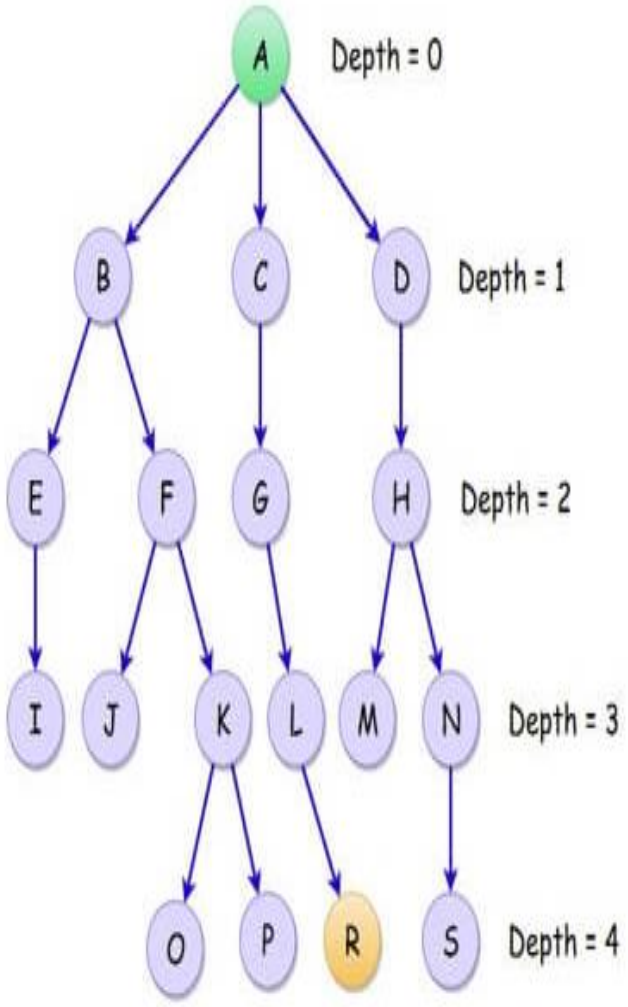
Depth-Limited Search

- What if solution is deeper than L? -> Increase L iteratively.

Iterative Deepening Search

- This inherits the memory advantage of Depth-First search, and is better in terms of time complexity than Breadth first search.
- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Search Algorithm Types :: Iterative Deepening Depth-first Search



DEPTH LIMITS	IDDFS
0	A
1	ABCD
2	ABEFCGDH
3	ABEIFJKCGLDHMN
4	ABEIFJKOPCGLRDHMNS

Search Algorithm Types :: Iterative Deepening Depth-first Search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Search Algorithm Types :: Iterative Deepening Depth-first Search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b* = *10*, *d* = *5*,

$$N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$$

$$N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Search Algorithm Types :: Iterative Deepening Depth-first Search

Time Complexity:

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

Space Complexity:

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

Completeness:

- This algorithm is complete if the branching factor is finite.

Optimality:

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

Search Algorithm Types :: Iterative Deepening Depth-first Search

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

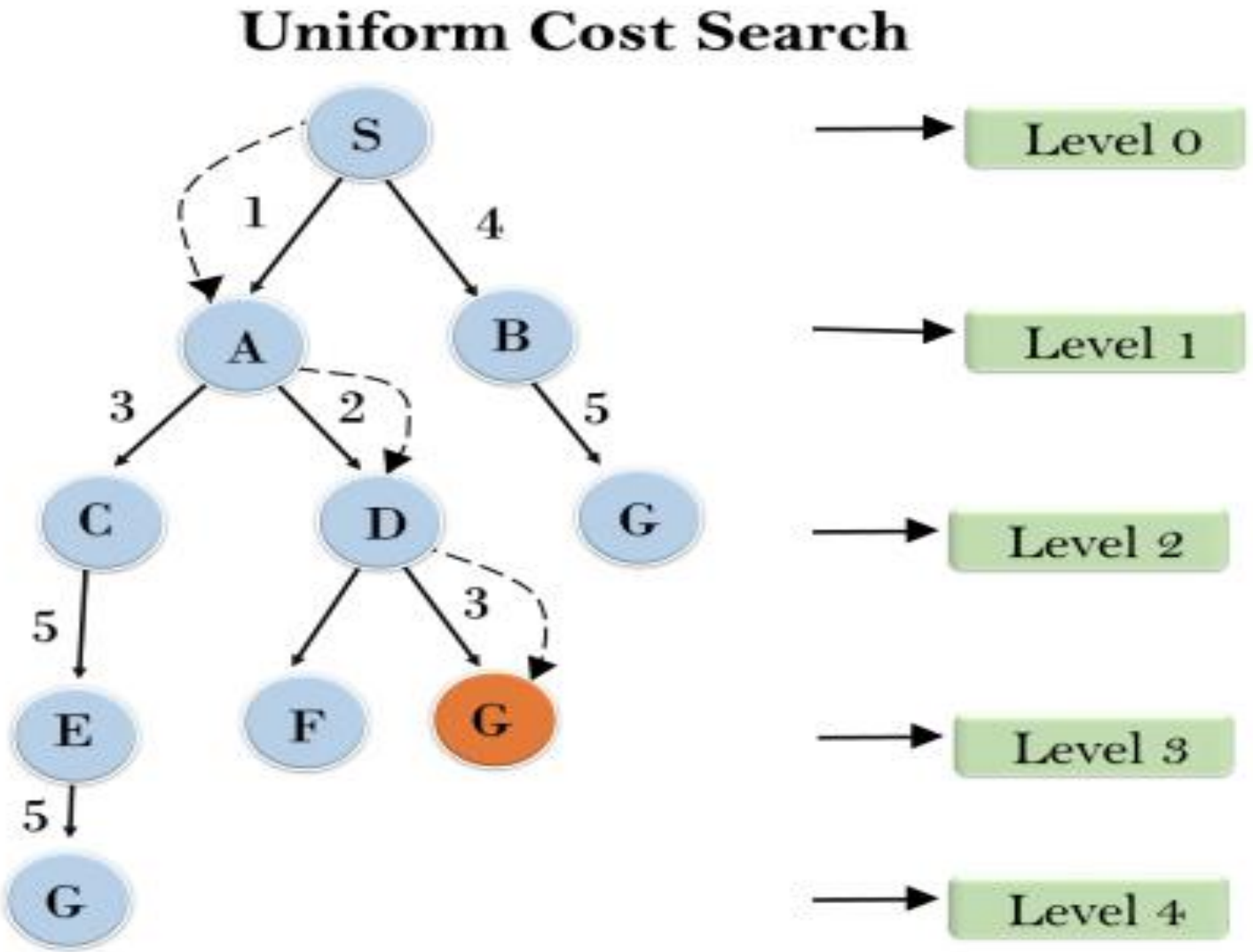
Disadvantages:

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

Search Algorithm Types :: Uniform Cost Search

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

Search Algorithm Types :: Uniform Cost Search



Search Algorithm Types :: Uniform Cost Search

Time Complexity:

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

$$\mathbf{O(b^{1 + [C^*/\varepsilon]})}.$$

Space Complexity:

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

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

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

Search Algorithm Types :: Uniform Cost Search

Advantages:

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

Disadvantages:

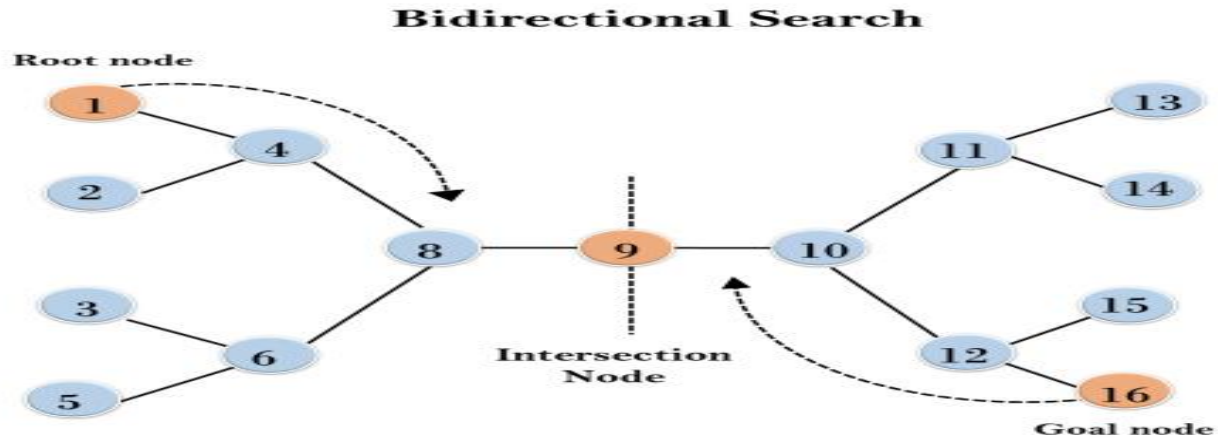
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Search Algorithm Types :: Bidirectional Search

- Heuristic refers to the concept of finding the shortest path from the current node in the graph to the goal node.
- A heuristic, or a heuristic technique, is any approach to problem solving that uses a practical method or various shortcuts in order to produce solutions that may not be optimal but are sufficient given a limited timeframe or deadline
- The search always takes the shortest path to the goal node. This principle is used in a bidirectional heuristic search.
- The only difference being the two simultaneous searches from the initial point and from goal vertex.
- The main idea behind bidirectional searches is to reduce the time taken for search drastically.

Search Algorithm Types :: Bidirectional Search

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



Search Algorithm Types :: Bidirectional Search

Why Bidirectional Search?

- Because in many cases it is faster, it dramatically reduce the amount of required exploration.
- Suppose if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity would be $O(b^d)$.
- On the other hand, if we execute two search operation then the complexity would be $O(b^{d/2})$ for each search and total complexity would be $O(b^{d/2} + b^{d/2})$ which is far less than $O(b^d)$.

Search Algorithm Types :: Bidirectional Search

When to use bidirectional approach?

- We can consider bidirectional 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 $O(b^{d/2})$.

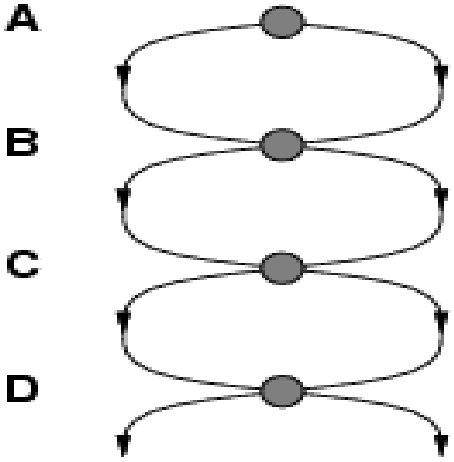
Search Algorithm Types :: Bidirectional Search

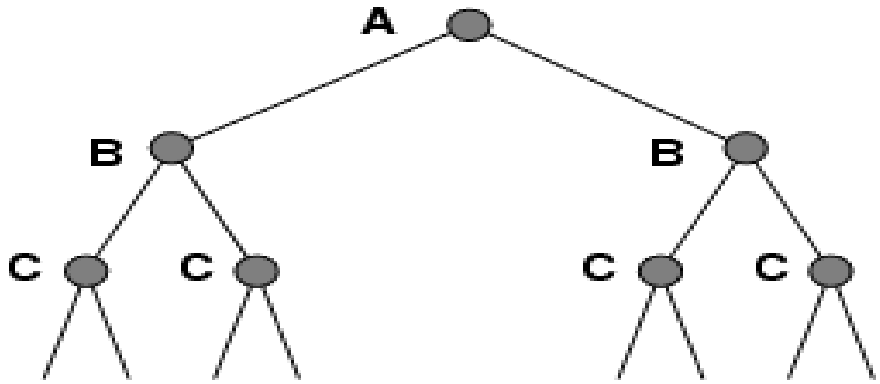
Advantages:

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

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.
- Repeated State: Failure to detect repeated states can turn a linear problem into an exponential one!





Search Algorithm Types :: Summary of Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

INFORMED SEARCH

Search Algorithm Types :: Informed Search

- 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 informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Search Algorithm Types :: Heuristic 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.
- 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.

Search Algorithm Types :: Heuristic Function

Admissibility of the heuristic function

- It 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.

Non-admissibility of the heuristic function

- It 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 more than the estimated cost.

Search Algorithm Types :: Pure-Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n).
- It maintains two lists, OPEN and CLOSED list.
- In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list.
- The algorithm continues until a goal state is found.

Search Algorithm Types :: Best-first Search

- The general approach we consider is called best first search.
- Best-first search is an instance of the general Tree-Search or Graph-Search algorithm in which a node is selected for expansion based on an evaluation function, f(n).
- The evaluation function is construed as a cost-estimate, so the node with the lowest evaluation is expanded first.
- The implementation of best first graph search is identical to that for Uniform-Cost Search, except for the use of f instead of g to order the priority queue.

Search Algorithm Types :: Best-first Search

Idea

- use an evaluation function f(n) for each node
- estimate of "desirability"
- Expand most desirable unexpanded node

Implementation

- Order the nodes in fringe in decreasing order of desirability

Special cases:

- greedy best-first search
- A* search

Search Algorithm Types :: Best-first Search

- 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.
- At each step, uses a heuristic to estimate the distance (cost) of each local choice from the goal.

Steps:

1) Define a heuristic function h(x) to estimate the distance to the goal from any state.
2) From the current state, determine the search space (actions) for one step ahead.
3) Select the action from the search space that minimizes the heuristic function.

Search Algorithm Types :: Best-first Search

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

Step 4: Expand the node n, and generate the successors of node n.

Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

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

Step 7: Return to Step 2.

Search Algorithm Types :: Best-first Search

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.

Search Algorithm Types :: Greedy Best-first Search

- 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 state/place

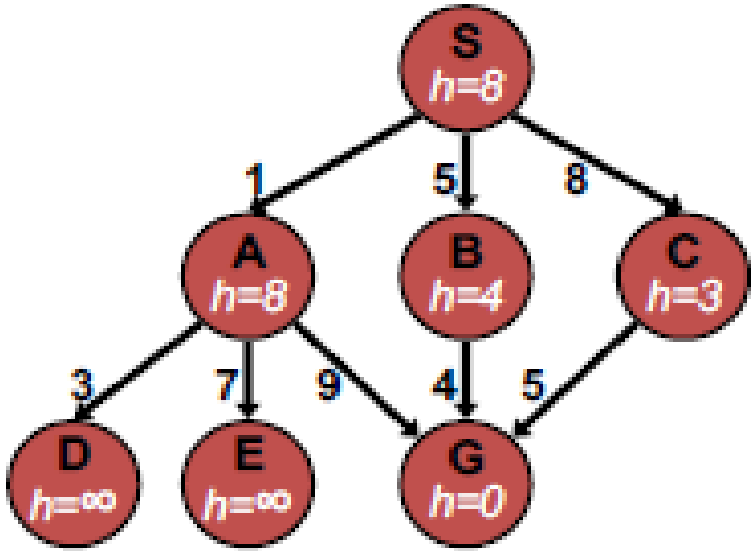
- Greedy best-first search expands the node that appears to be closest to goal

Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 0, expanded: 0

expnd. node	Frontier
	{S:8}

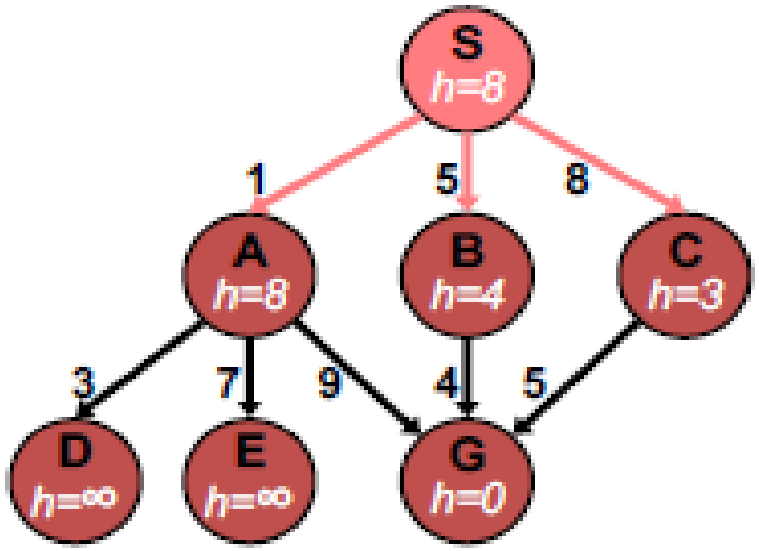


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 1, expanded: 1

expnd. node	Frontier
	{S:8}
S not goal	{C:3,B:4,A:8}

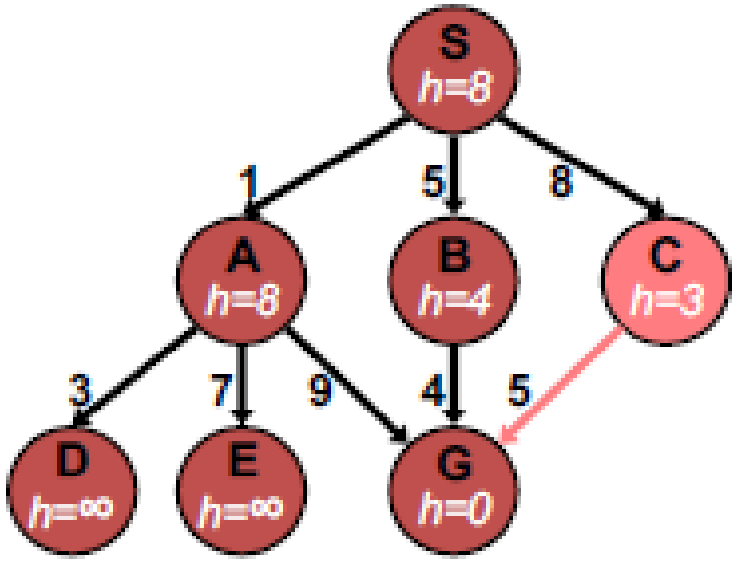


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 2, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C not goal	{G:0,B:4,A:8}

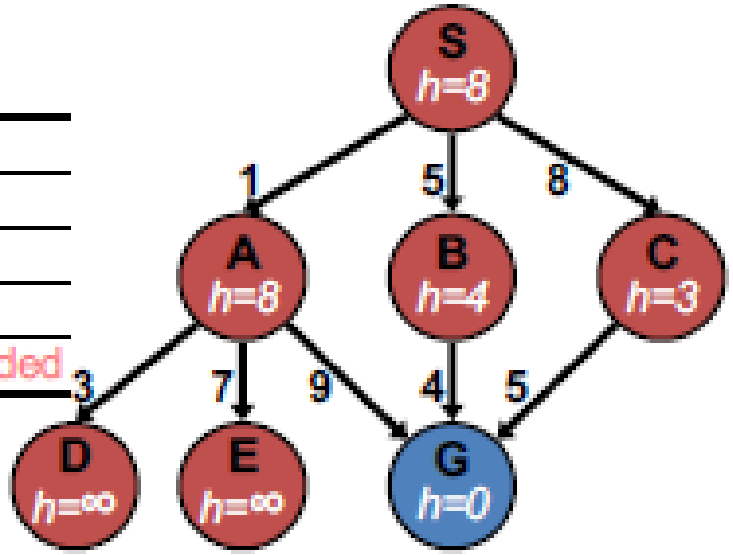


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G goal	{B:4, A:8} not expanded

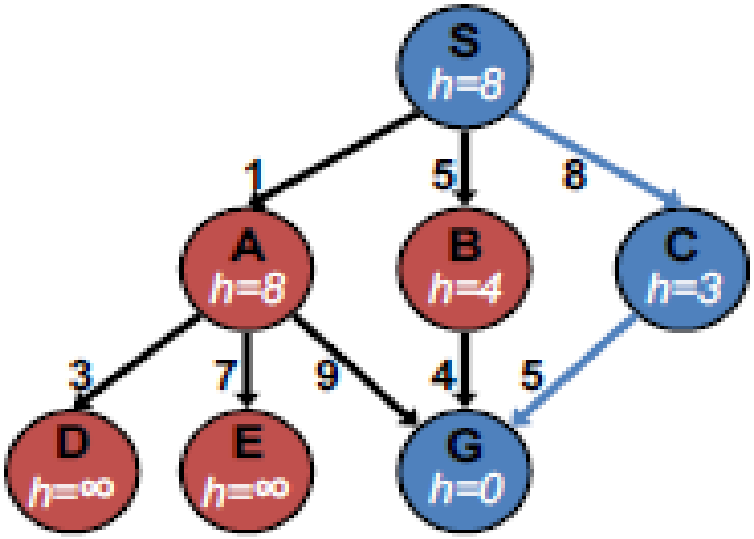


Greedy Best-First Search

$f(n) = h(n)$

# of nodes tested: 3, expanded: 2

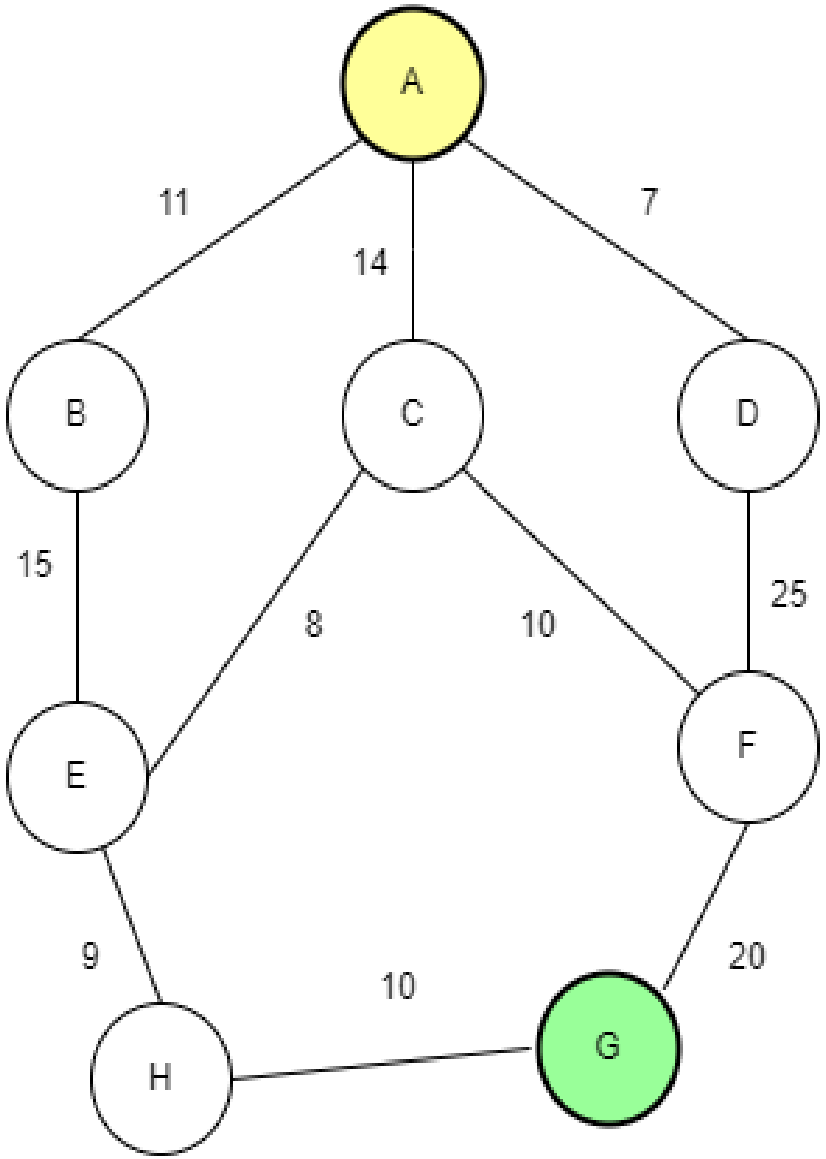
expnd. node	Frontier
	{S:8}
S	{C:3,B:4,A:8}
C	{G:0,B:4, A:8}
G	{B:4, A:8}



- *Fast but not optimal*

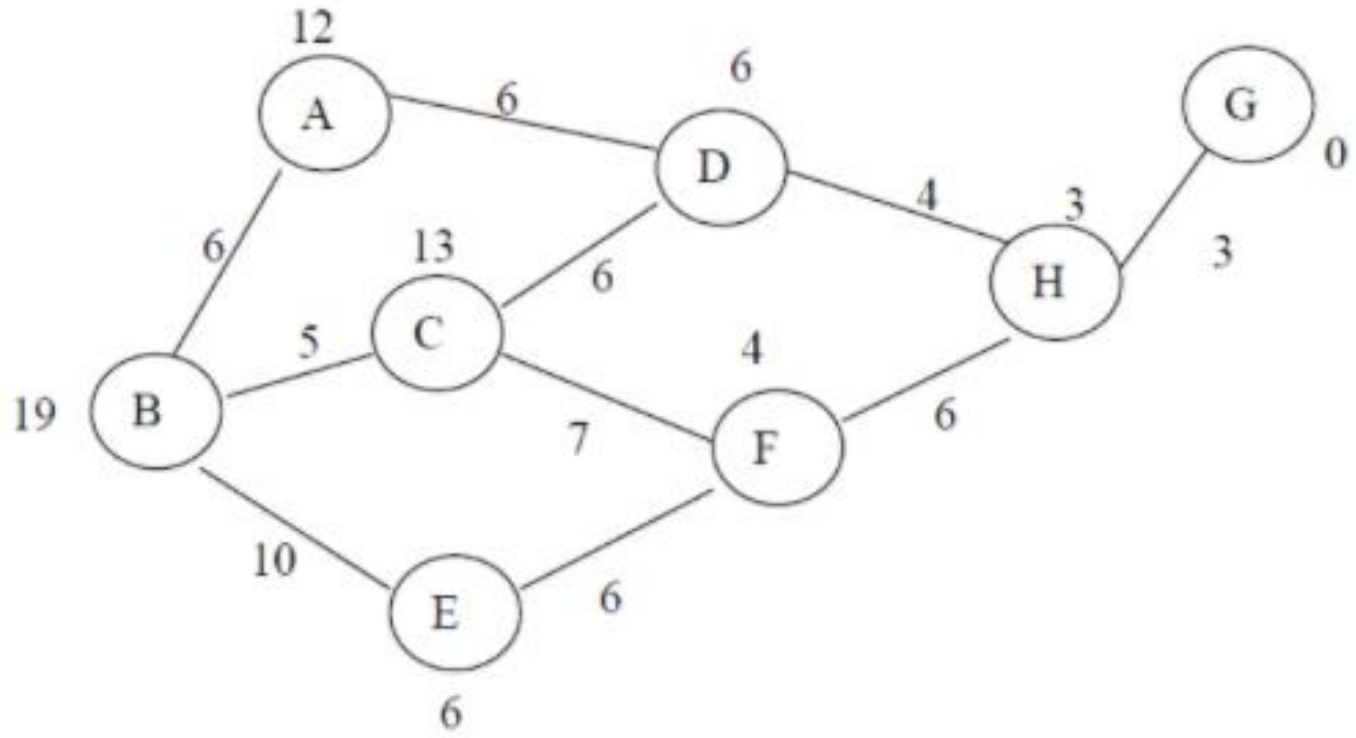
path: S,C,G
cost: 13

Search Algorithm Types :: Greedy Best-first Search



Path	$h_{SLD}(n)$
A → G	40
B → G	32
C → G	25
D → G	35
E → G	19
F → G	17
G → G	0
H → G	10

Search Algorithm Types :: Greedy Best-first Search



Search Algorithm Types :: Best-first Search

Time Complexity:

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

Space Complexity:

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

Completeness:

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

Optimality:

- Greedy best first search algorithm is not optimal.

Search Algorithm Types :: A* Search

- A* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
- A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
- It is essentially a best first search algorithm.
- A* Algorithm extends the path that minimizes the following function-

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

Here,

- ‘n’ is the last node on the path
- g(n) is the cost of the path from start node to node ‘n’
- h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

Search Algorithm Types :: A* Search

Idea

- avoid expanding paths that are already expensive

Implementation

- Evaluation function f(n) = g(n) + h(n)

  g(n) = cost so far to reach n

  h(n) = estimated cost from n to goal

  f(n) = estimated total cost of path through n to goal

- A* Algorithm is one of the best path finding algorithms.
- But it does not produce the shortest path always.
- This is because it heavily depends on heuristics.

Search Algorithm Types :: A* Search

Working

1) The algorithm maintains two sets
   - OPEN list: The OPEN list keeps track of those nodes that need to be examined.
   - CLOSED list: The CLOSED list keeps track of nodes that have already been examined.
2) Initially, the OPEN list contains just the initial node, and the CLOSED list is empty
   - g(n) = the cost of getting from the initial node to n
   - h(n) = the estimate, according to the heuristic function, of the cost from n to goal node
   - f(n) = g(n)+h(n); intuitively, this is the estimate of the best solution that goes through n

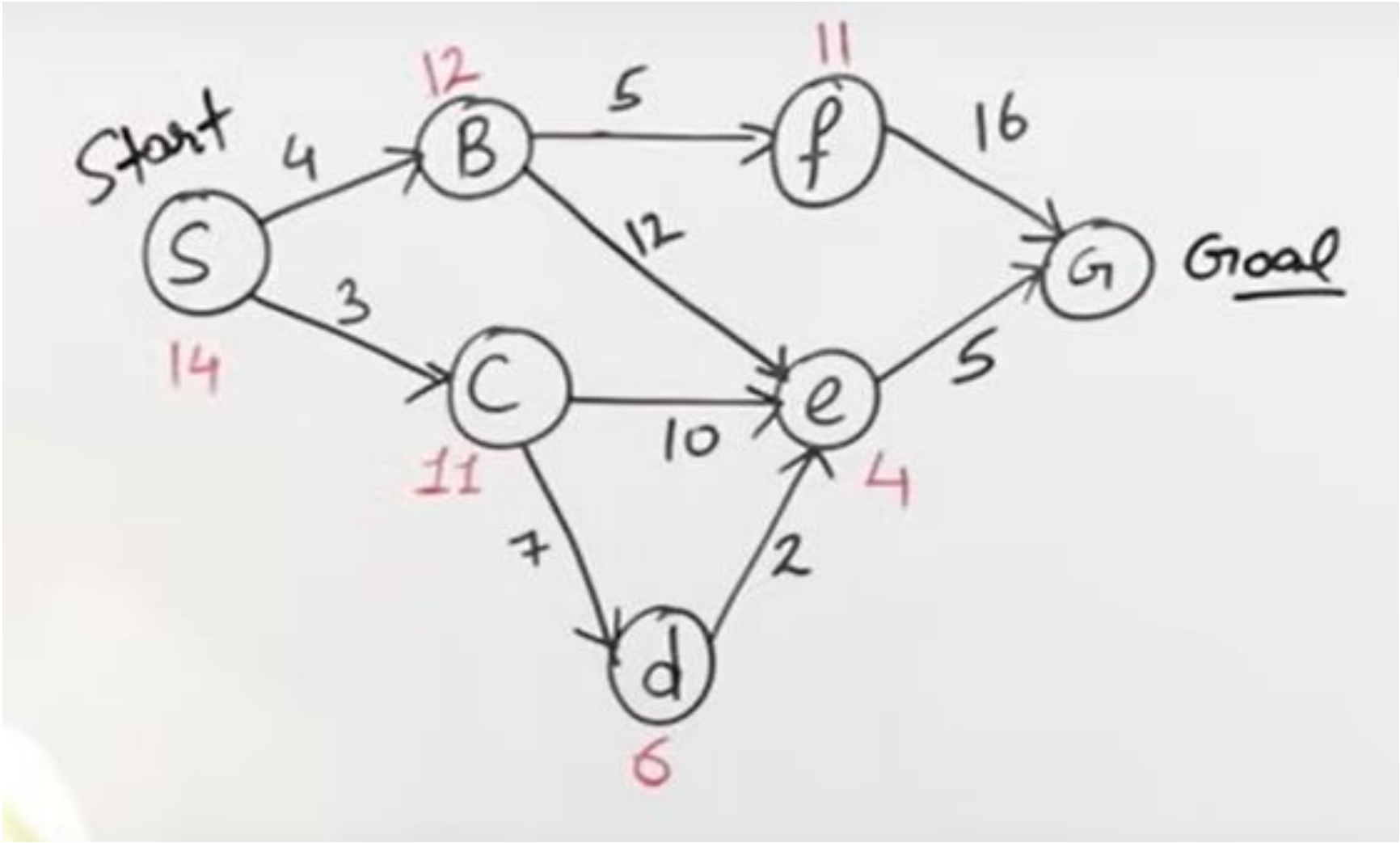
Search Algorithm Types :: A* Search

Working

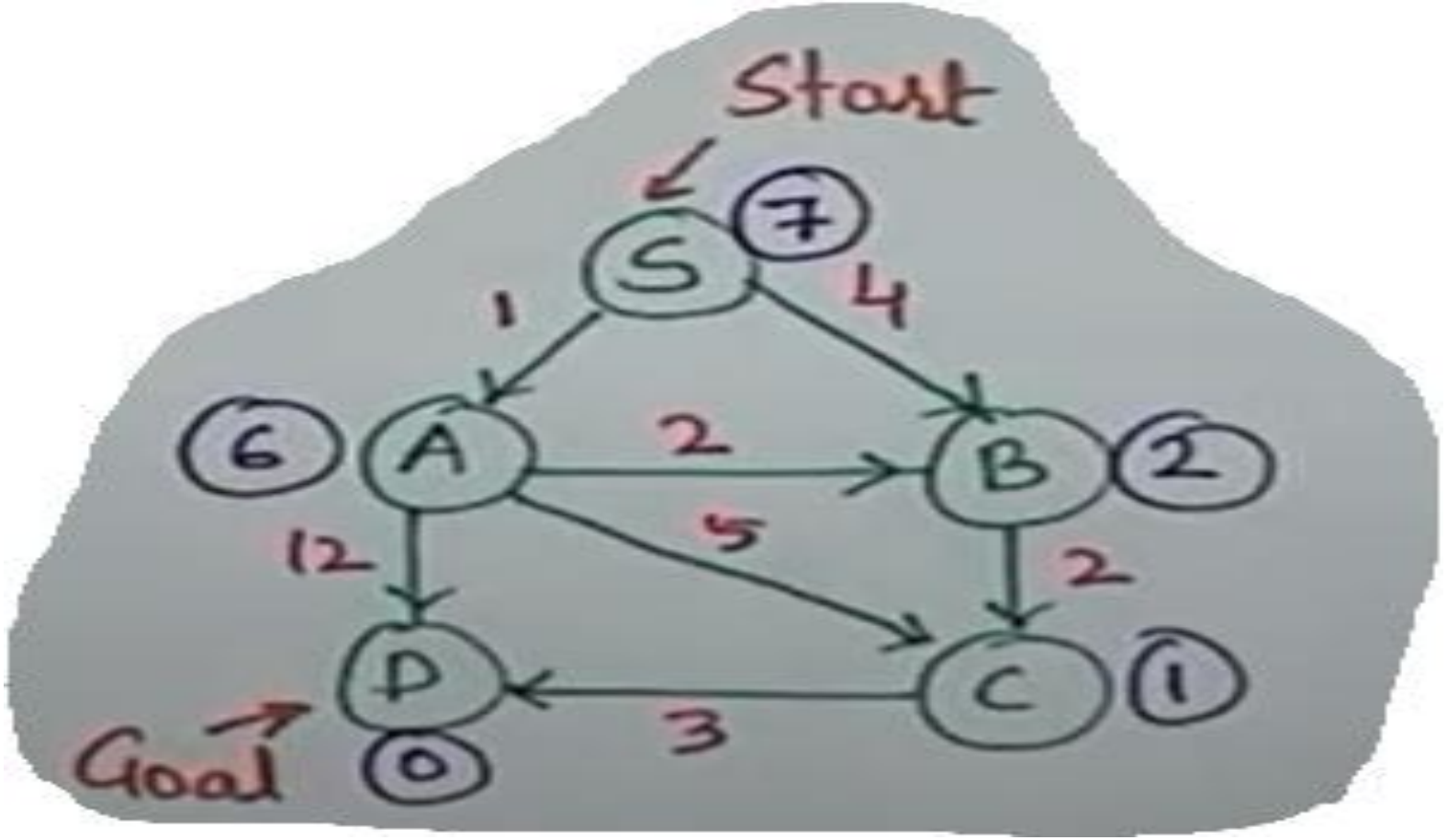
3) Each node also maintains a pointer to its parent, so that the best solution if found can be retrieved.
   - ➢ It has main loop that repeatedly gets the node, call it n with lowest f(n) value from OPEN list.
   - ➢ If n is goal node then stop (done) otherwise, n is removed from OPEN list & added to the CLOSED list.
   - ➢ Next all the possible successor nodes of n are generated.

4) For each successor node n, if it is already in the CLOSED list and the copy there has an equal or lower f estimate, and then we can safely discard the newly generated n and move on.
   - ➢ Similarly, if n is already in the OPEN list & the copy there has an equal or lower f estimate, we can discard the newly generated n and move on.

Search Algorithm Types :: A* Search



Search Algorithm Types :: A* Search



Search Algorithm Types :: A* Search

Given an initial state of a 8-puzzle problem and final state to be reached-

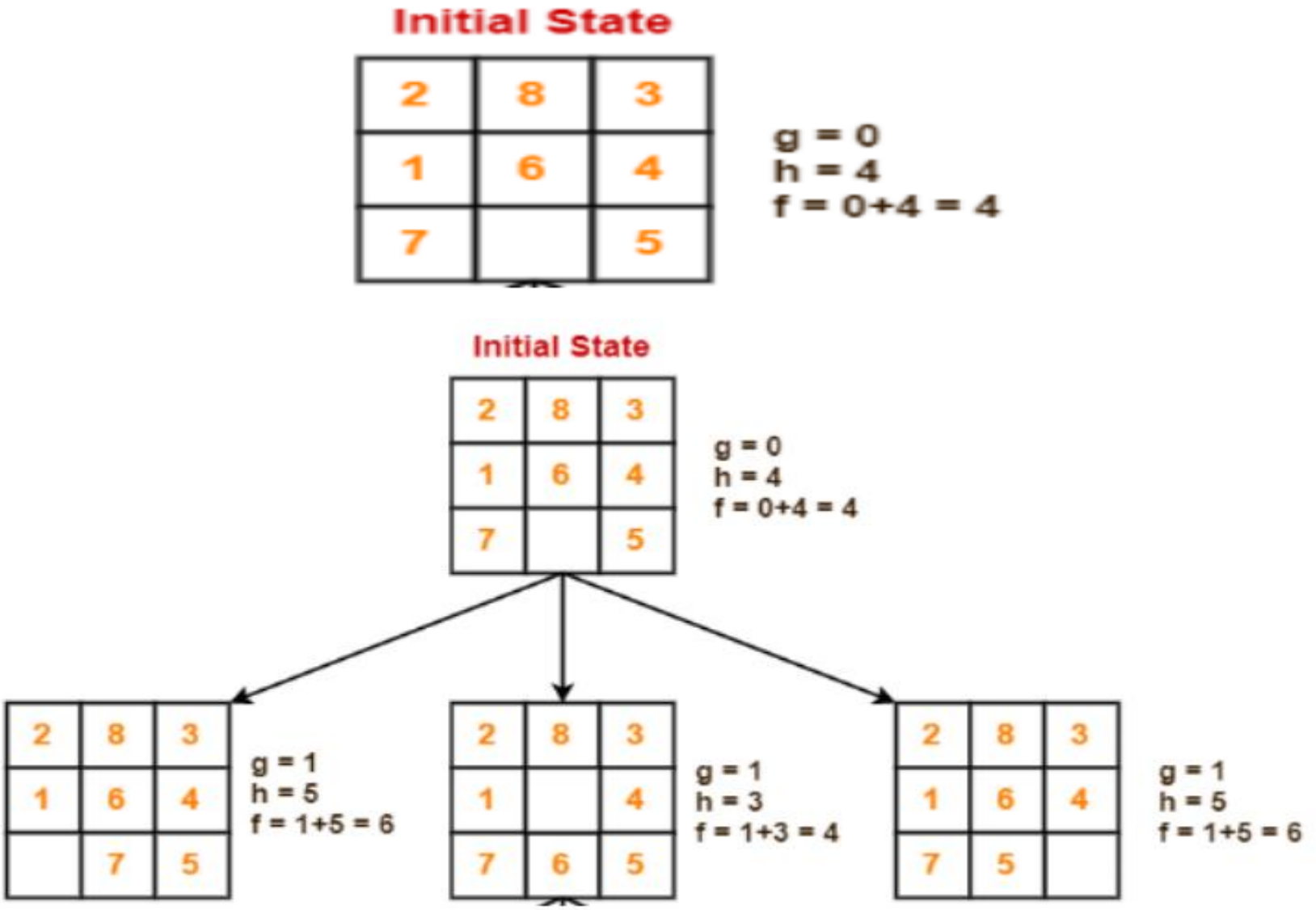
Initial State

Final State

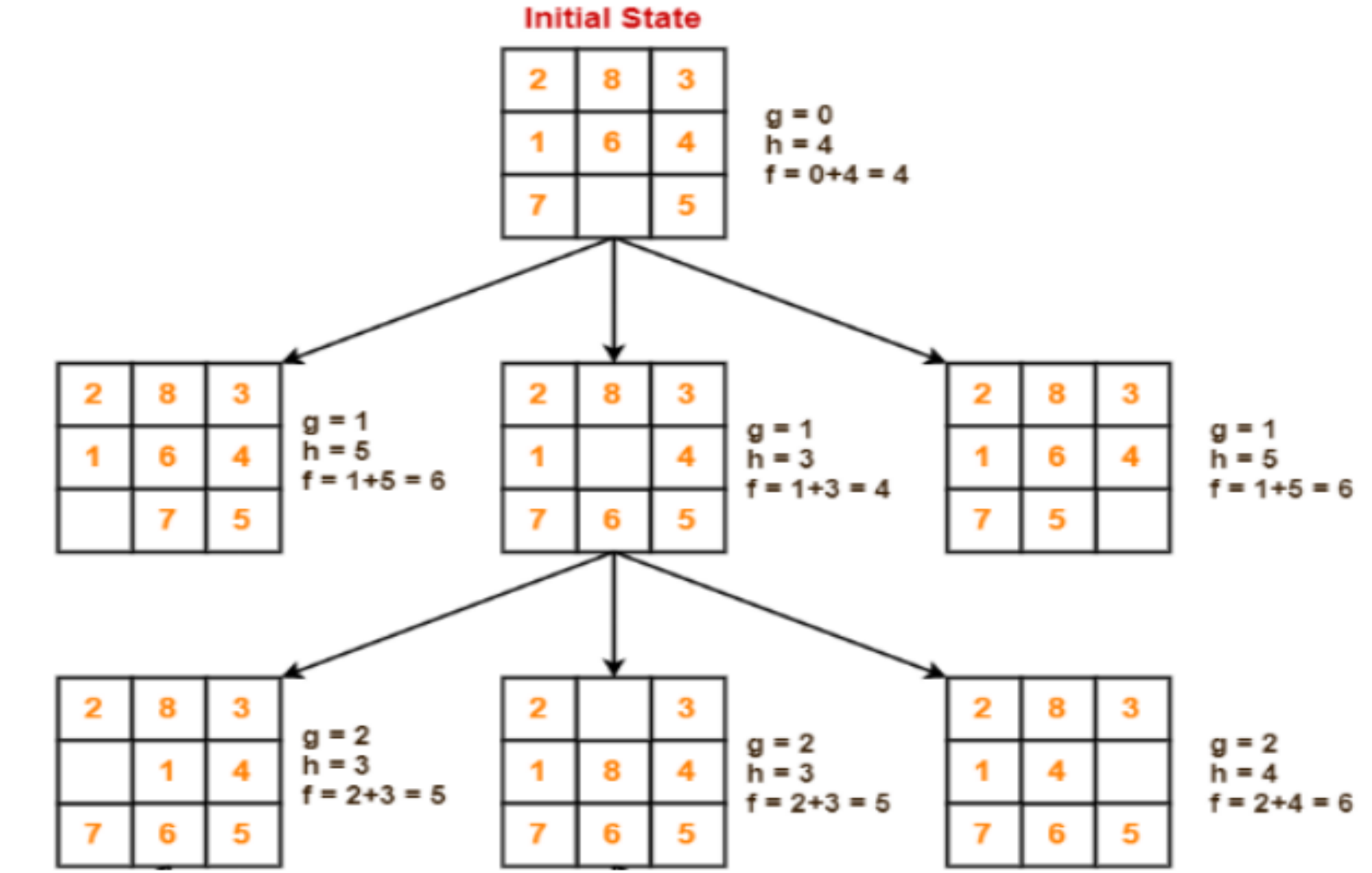
Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

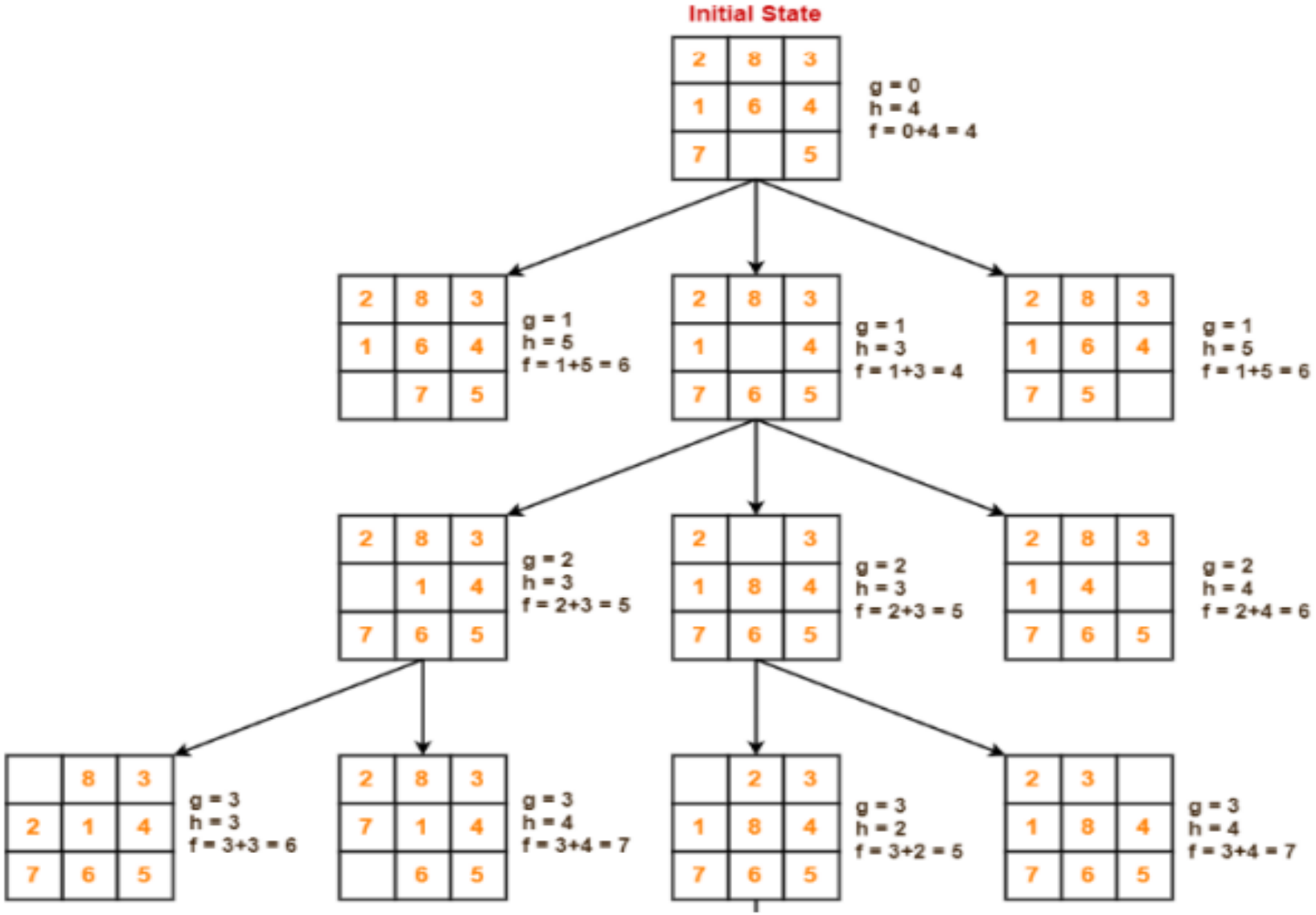
Search Algorithm Types :: A* Search



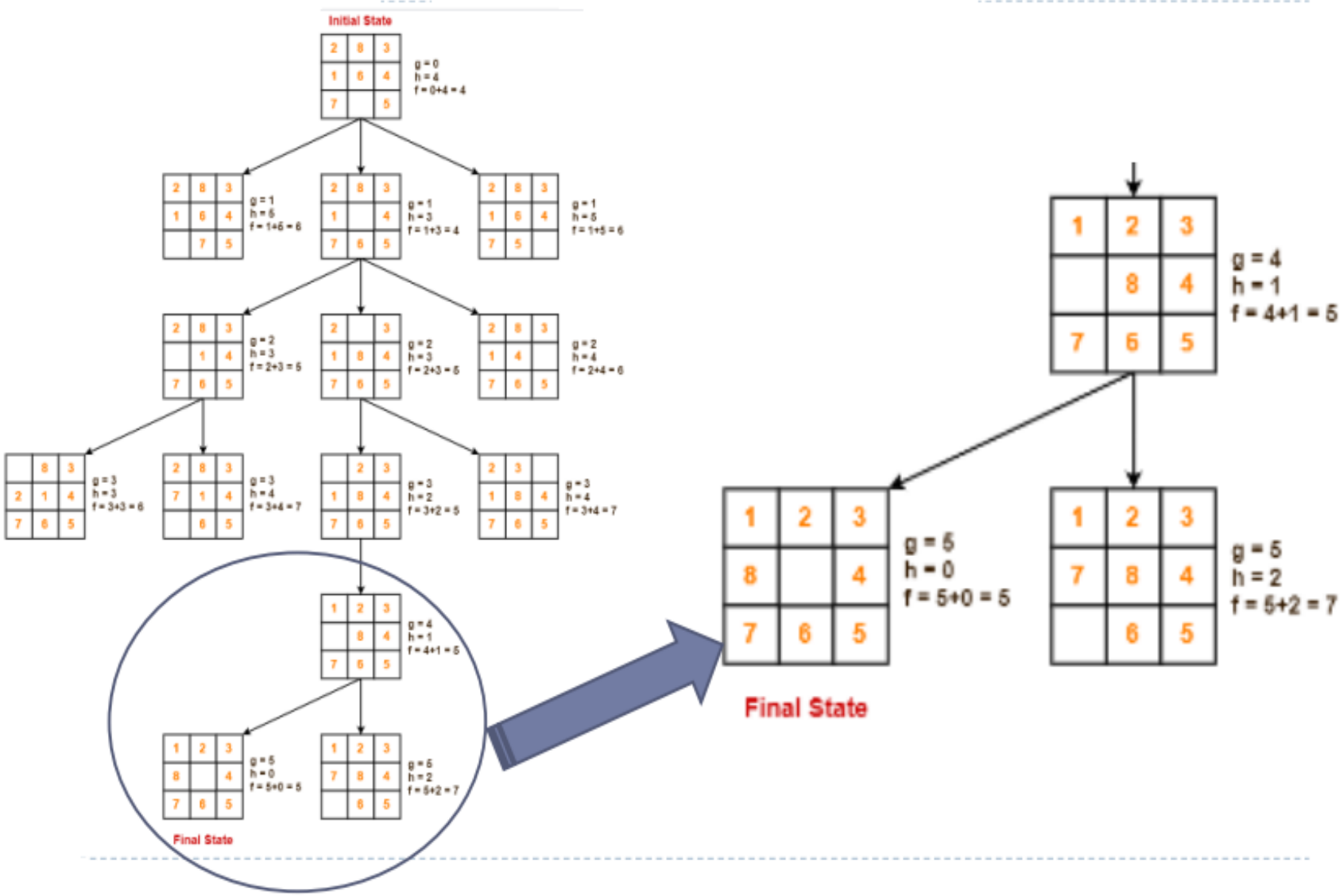
Search Algorithm Types :: A* Search



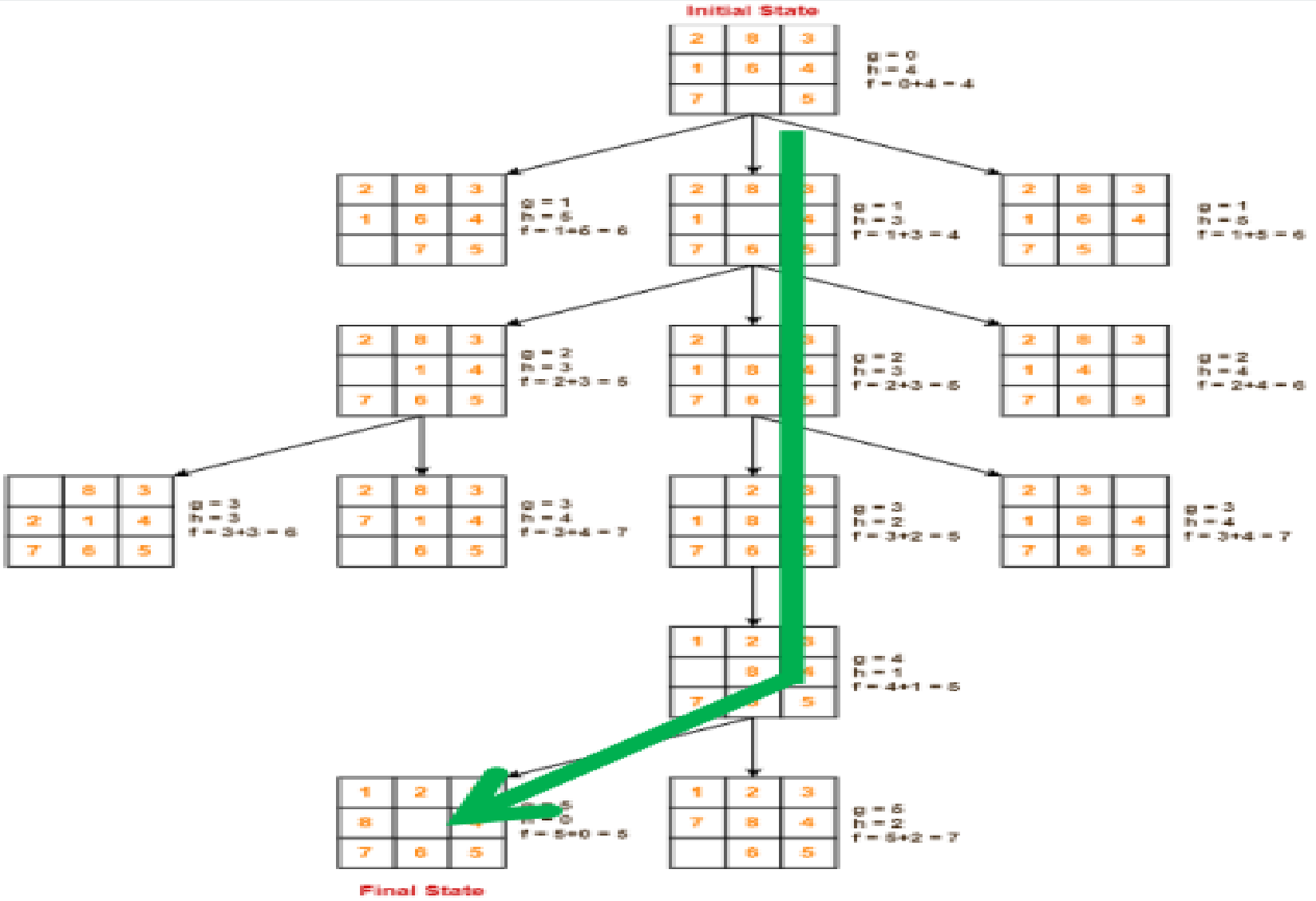
Search Algorithm Types :: A* Search



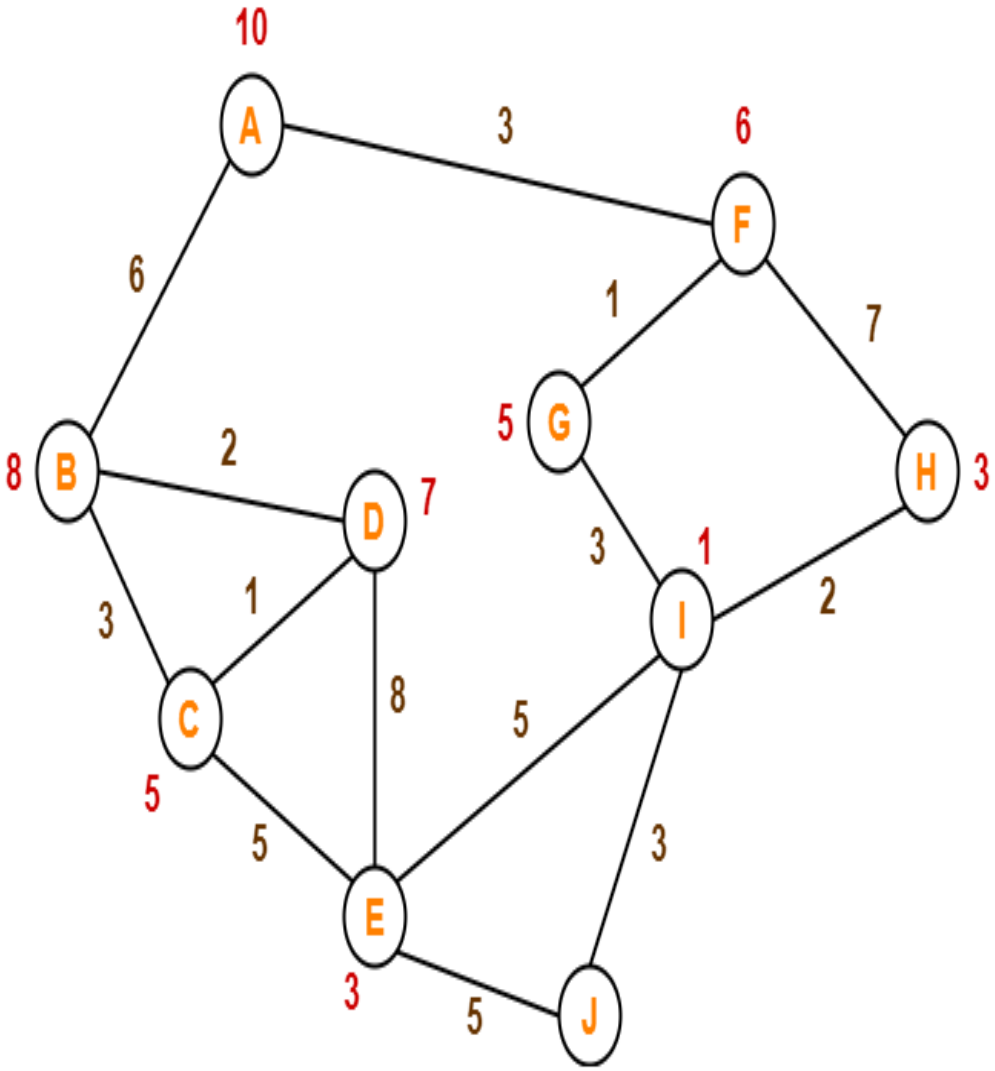
Search Algorithm Types :: A* Search



Search Algorithm Types :: A* Search



Search Algorithm Types :: A* Search



The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

Search Algorithm Types :: A* Search

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.

Search Algorithm Types :: A* Search

Time Complexity:

- Exponential - $O(b^m)$.

Space Complexity:

- Keeps all nodes in memory - $O(b^m)$. Where, m is the maximum depth of the search space.

Completeness:

- Yes (unless there are infinitely many nodes with $f \leq f(G)$).

Optimality:

- Yes

Search Algorithm Types :: A* Search

Properties:

- Admissible
- Optimality
- Consistent

Admissible Property:

- A heuristic h(n) is admissible if for every node n,

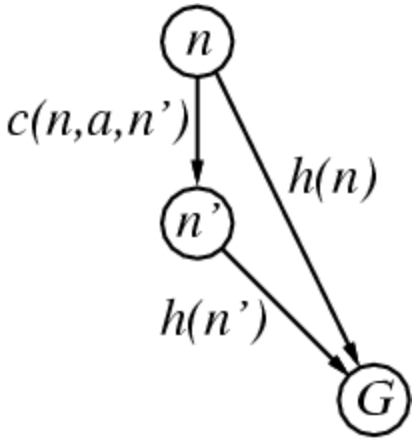
  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If h(n) is admissible, A* using TREE-SEARCH is optimal

Search Algorithm Types :: A* Search

Consistent Property:

- A heuristic is consistent if for every node n, every successor n' of n generated by any action a,

  $h(n) \leq c(n,a,n') + h(n')$



- If h is consistent, we have

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

  $= g(n) + c(n,a,n') + h(n')$

  $\geq g(n) + h(n)$

  $= f(n)$

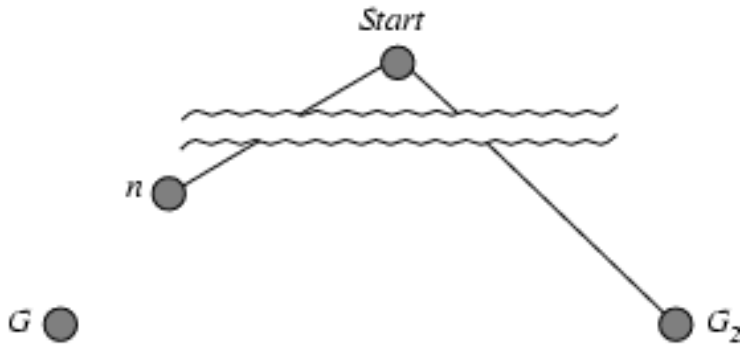
  i.e., f(n) is non-decreasing along any path.

- Theorem: If h(n) is consistent, A* using GRAPH-SEARCH is optimal

Search Algorithm Types :: A* Search

Optimality of A* (Proof):

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

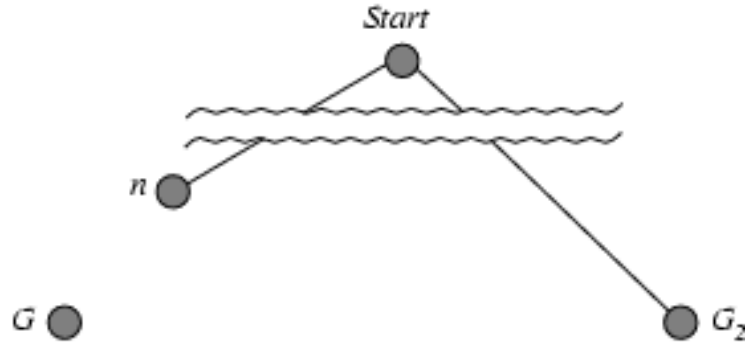


- f(G2) = g(G2) since h(G2) = 0
- g(G2) > g(G) since G2 is suboptimal
- f(G) = g(G) since h(G) = 0
- f(G2) > f(G) from above

Search Algorithm Types :: A* Search

Optimality of A* (Proof):

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



- f(G2) > f(G) from above
- h(n) ≤ h*(n) since h is admissible
- g(n) + h(n) ≤ g(n) + h*(n)
- f(n) ≤ f(G)
- Hence f(G2) > f(n), and A* will never select G2 for expansion

URL: https://www.youtube.com/watch?v=xgkkzeFX_fI

Search Algorithm Types :: Iterative-Deepening A*

- The iterative deepening are applied in the context of heuristic search result in the iterative-deepening A* (IDA*) algorithm
- It reduce memory requirement for A*
- The cut off value f-cost (g+h) is differentiate between IDA* and standard iterative-deepening.

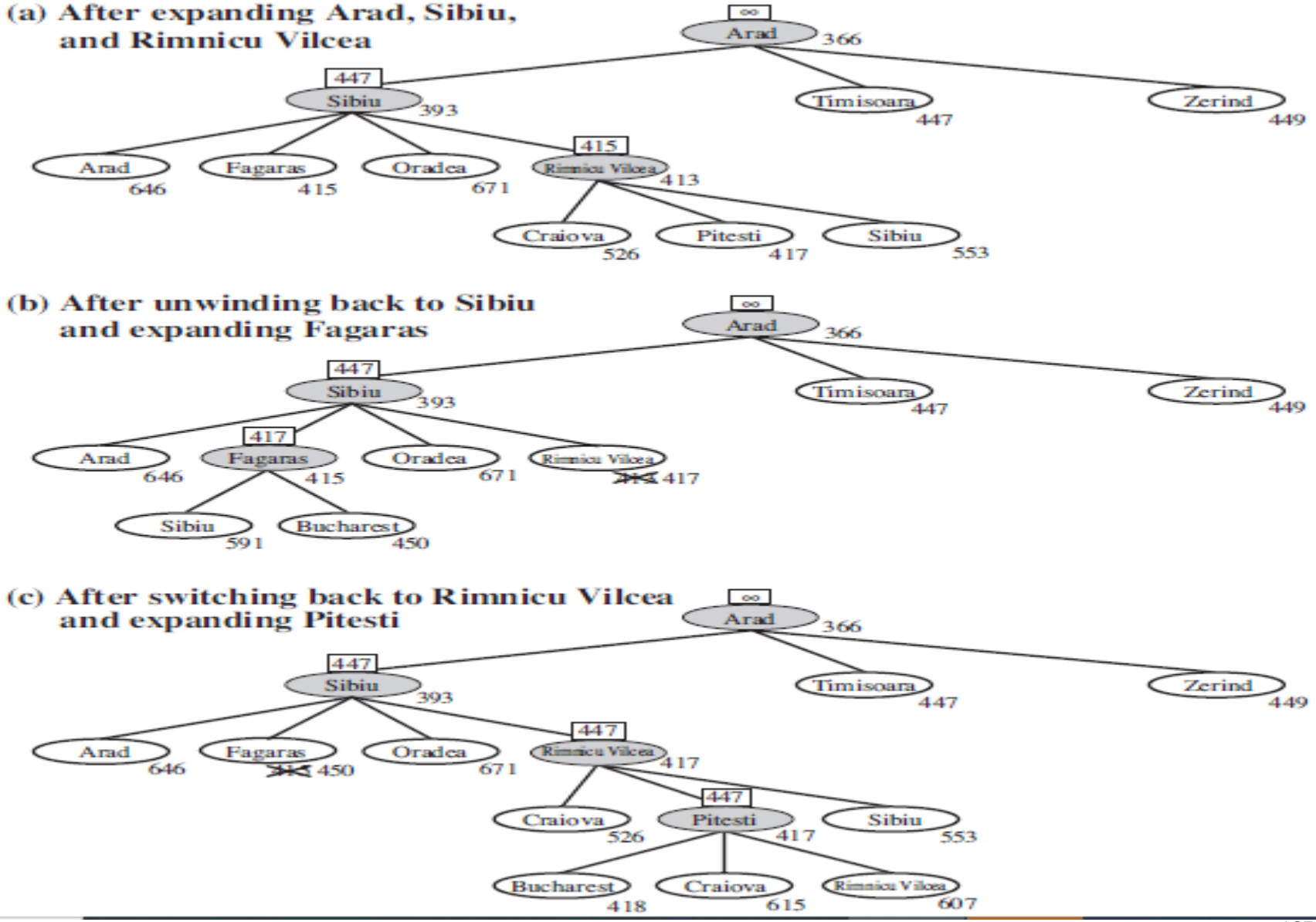
Memory-Bound Heuristic Search

- Iterative-Deepening A* falls under Memory-Bound Heuristic Search
- There are two other memory-bounded algorithm
  - RBFS (Recursive Best-First Search)
  - MA* (Memory-bounded A*)

Search Algorithm Types :: Recursive Best-First Search

- Idea: mimic the operation of standard best-first search, but use only linear space
- Runs similar to recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children.
- In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time.

Search Algorithm Types :: Recursive Best-First Search



Search Algorithm Types :: Simplified Memory-bounded A* (SMA)

- SMA is very simple and small it performs like A* expanding the best leaf until memory is full
- It first drop on an old node and then add a new node to the search tree
- The node with highest f-value i.e. worst leaf node is always dropped in SMA
- Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree.
- With this information, SMA∗ regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten.
- Another way of saying this is that, if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

Disclaimer

The material for the presentation has been compiled from various sources such as prescribed text books by Russell and Norvig and other tutorials and lecture notes. The information contained in this lecture/ presentation is for educational purpose only.

Thank You *for* Your Attention !