Module - 2

Problem - Solving Methods

Problem Solving by search

→ Goal based agent are called as Go problem -Solving agent

A well defined problem

1] Initial State

→ Operator / Successor function - for any State x return S(x)

The set of states reachable from x with one action

→ State space : all states are reachable

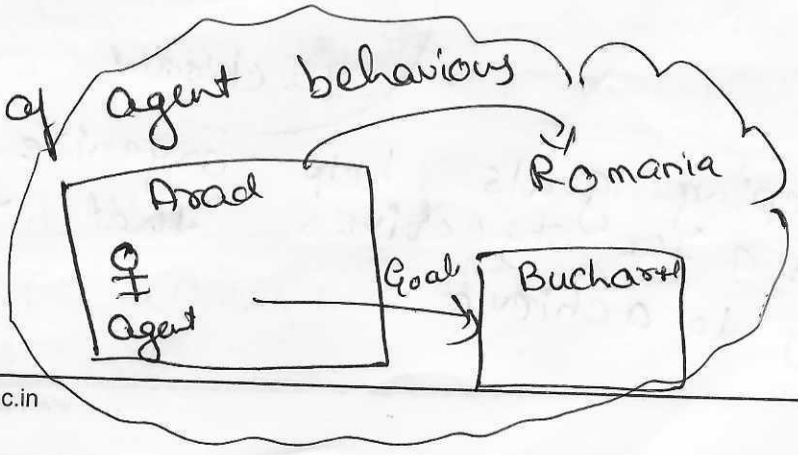
→ path - Sequence.

→ Goal test : test to determine if a goal state

Problem - Solving Agents

Intelligent agents are supposed to maximize their performance

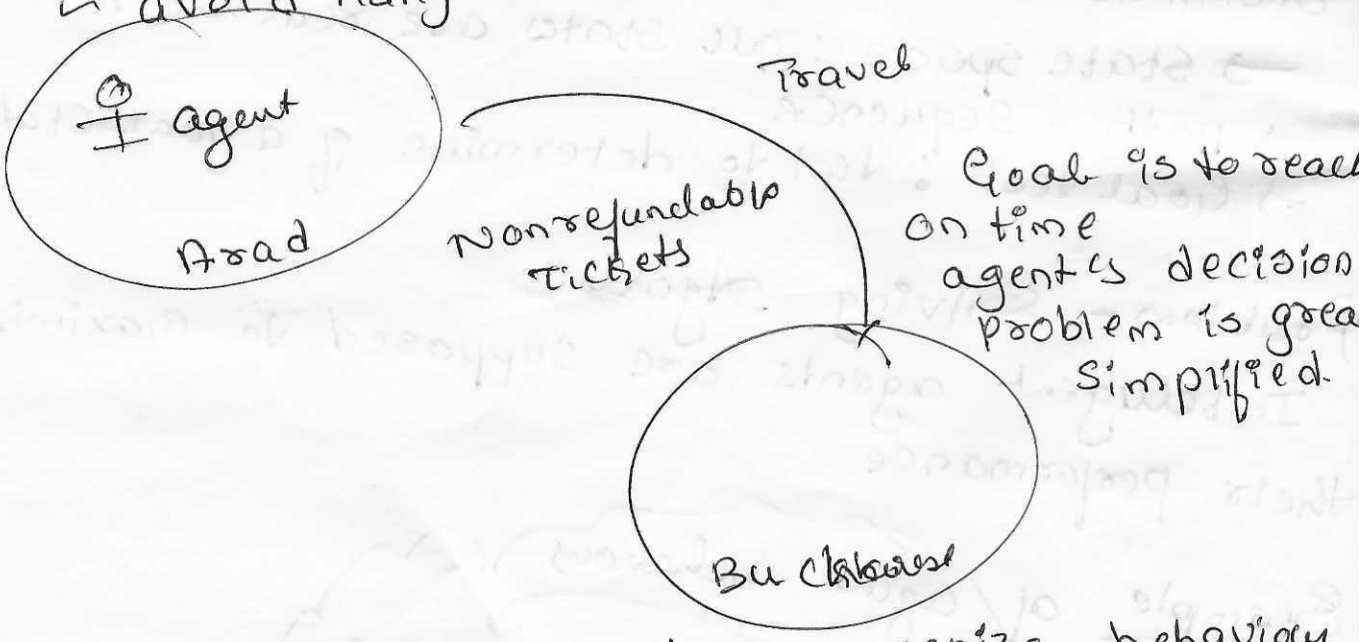
Example of agent behaviour



Agent is Enjoy the holiday

Goal formulation:

The agent performance measure contain many fac
- To improve suntan
- avoid hangover etc

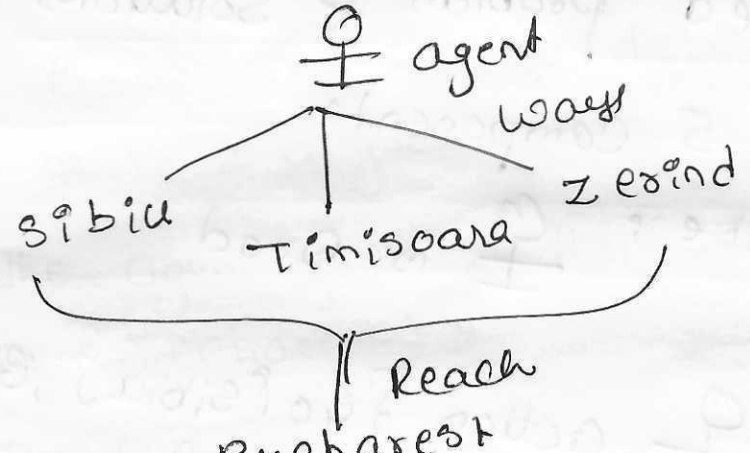


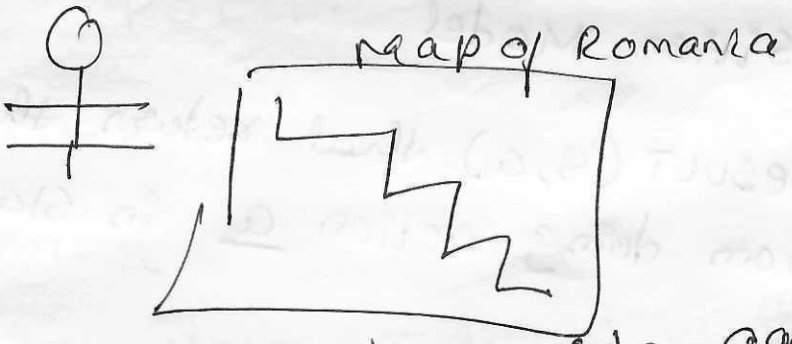
→ By keeping goals help organize behaviour by limiting the objectives that the agent is trying to achieve

Goal formulation: problem solving

problem formulation — what action & procedure to be followed.

Agent adopted goal to travel by road of driving to Bucharest & considering where to go from Arad





The point of map is to provide agent with information about states & then take an action

If found the Goal achieved

To carry out this Goal action is required

In general an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

Well-defined problem & Solutions

5 components

(1) Initial state : Arad

(2) Action — action {Go(Sibiu), Go(Timisoara), Go(Zerind)}

(3) Transition Model

RESULT(S, a) that returns the state that results from doing action a in state S

Successor ← refer to any state reachable from a given state by a single action.

Result (In (Arad), Go (Zerind)) = In (Zerind)

∴ Initial state, action & transition model

A path in the state space is a sequence of states connected by a sequence of actions

(4) The Goal Test → To determines whether a given state is a goal state.

The agent goal in Romania is the singleton set {In (Bucharest)}

(5) path cost

EXAMPLE PROBLEMS

(1) TOY PROBLEMS

Vaccum World Example.

(1) States: the state is defined by the agent's location & dirt locations.

In 2×2 grid example
↓ ↳ each location is either dirt / clean
agent in 2 locations

$2 \times 2^2 = 2 \times 4 = 8$ possible state

{

- Initial State : Any State can be designated as initial State.
- Actions → agent as Just 3 states

  Left → move left

  Right → move right

  Suck → clean the current Square

In Larges environment might include up & Dow

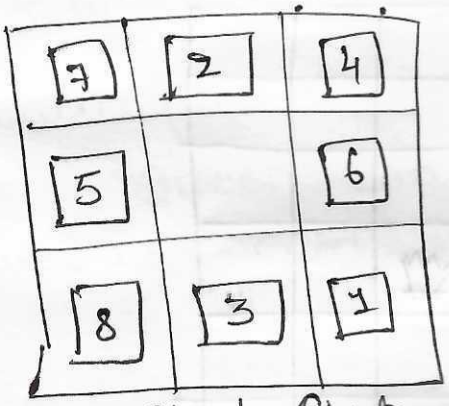
- Transition Model

  ~~No~~ moving left
  Right moving
  } changes the agent location

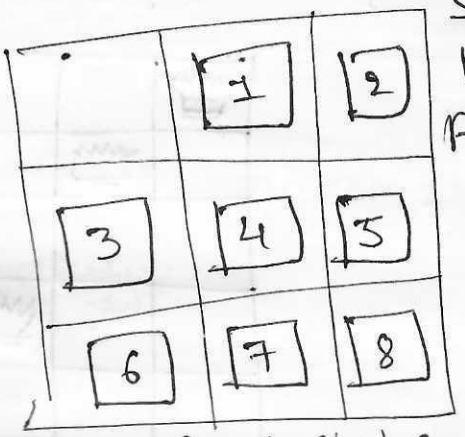
  Sucking clean the current location

- Goal ~~path~~ Test : The goal is achieved when all Squares are clean
- Path cost : Each step costs 1, so the path cost is the number of steps in the path.

4 8 - PUZZLE / belongs to family of sliding block puzzles



Start State



Goal State

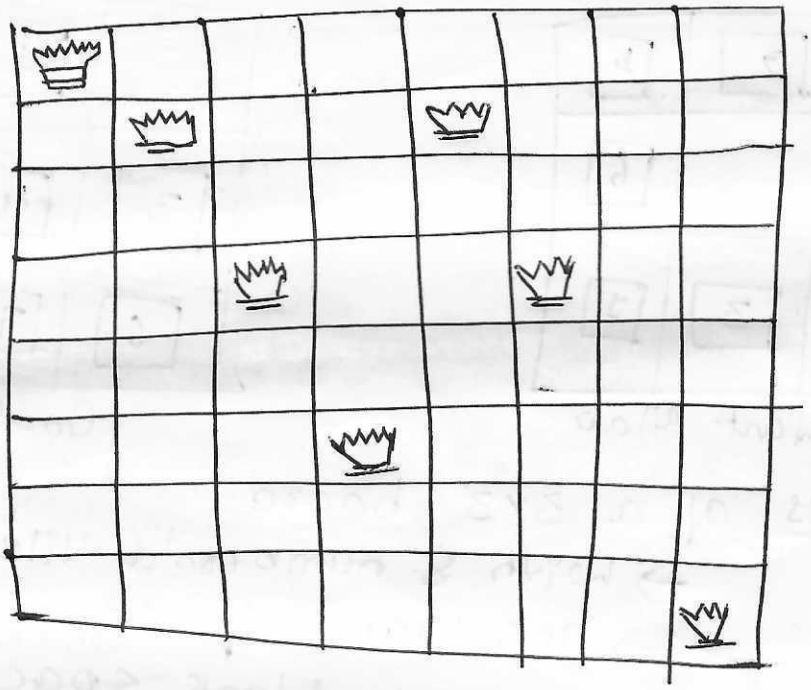
Consists of a 3x3 board
→ with 8 numbered tiles.
→ one blank space.

→ A tile adjacent to blank space can slide into space.

- State → The puzzle is represented by 8 numbered tiles & 1 blank space on 3x3 board
- Initial State: any configuration of the tiles can be the starting point.
- Actions: movement of the blank space left, down up & Right
- Transition Model: The goal is to reach a specified configuration as shown in figure
- Path Cost: Each step costs 1, so the path cost is number of steps in the path.

8 - Queens Problem [N - Queens problem]

- States :



State : Any arrangement of 0 to 8 queen (N = 8)

Initial State : No queens are placed initially

Actions : A queens are placed initially is added to any empty square

Transition Model : The system returns the update board after placing a queen.

Goal Test : All 8 queen should be on board & non as to attack each other

Formulation

Incremental formulation : Add queens one at a time.

Complete – State formulation : place all queens & move them around.

- Challenges :
  A Queen can attack another queen if they are in same row, column or diagonal

---

Understanding the 8-Queen
- 8 queens on a 8×8 Chessboard
- no two queens attack each other
- a queen can attack in any row, column or diagonal

Board size – 8 rows × 8 columns

State formulation – 64 possible squares [1st queen, 2nd queen, ...]

Once the queen take place, for next queen the possible will reduce

Total possible sequence = $64 \times 63 \times 62 \times \ldots \ldots 57$

$$\frac{64!}{(64-8)!}$$ which is large number

1.8×10^{14} ← Total number of possible ways to place 8 queens on a chessboard

REAL-WORLD PROBLEMS

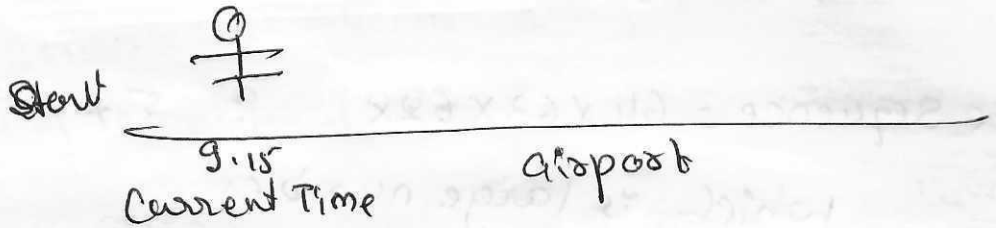
Route-finding Problem

→ Defined in terms of specified location & transitions along links between them

→ Routing in CN, Military operation planning airline travel planning systems

Airline Travelling problem

→ State : Each state represents a location (airport) & current state



- Initial state : This is specified by the problem [where you are & the time you are there]

- Successor function : This function generates possible next states based on scheduled flights. A flight can only be taken if.

- The departure time is later than the current time
- you are at the correct airport to board

Goal Test: The Goal test is to check if you have reached the destination airport within specific time

- Path cost → This depends upon the monetary cost, waiting time, flight time, customs & immigration seat quality, time of date & so on

TOURING PROBLEMS

→ closely related to route-finding problems
visit every city atleast once

- Initial State: The agent starts in Bucharest & has not yet visited
- Intermediate state: The agent might be in other city visiting A, B & C as well.
- Goal: The Goal is to check if all cities have been visited, meaning the agent has traveled through each one atleast once.

THE TRAVELLING SALESPERSON PROBLEM (TSP)

→ Is a touring problem in which each city must be visited exactly once.
→ The aim is to find shortest tour.
→ The problem is called NP-hard

VLSI layout

→ Layout problem requires positioning millions of components & connections on a chip to minimze area
↳ 2 problems — Cell layout
— Channel routing

Robot Navigation

→ Robot navigation is a generalization of the route-find problem.
→ Rather than a discrete set of routes a robot can move in continuous space with an infinite set of possible actions & states.

Automatic Assembly Sequencing

→ assembly electric motors
→ aim in " pblm is to find the order in which to assemble the parts of some objects.

UNINFORMED SEARCH STRATGES

Strategies that know whether one non goal state is "more promising" than another are

Informed Search / heuristic Search

5 - Uniformed

1) Breadth - first
2) Uniform - cost Search
3) Depth - first Search
3) Depth - limited search
4) Iterative deepening Search

Searching for Solutions

TREE SEARCH & GRAPH - SEARCH

(1) TREE SEARCH : fundamental search algorithm in AI

1. Search Tree : Search tree is created starting with initial state (root)
   & Expands all possible actions (Branches)
   → Each action leads to a new state.

Ex Arad to Bucharest, each stat (city) is connected by actions (roads & flights)

(2) Frontier (or open List)

The frontier is a set of all leaf nodes in the Search tree, set of nodes that have been generated but not yet expanded

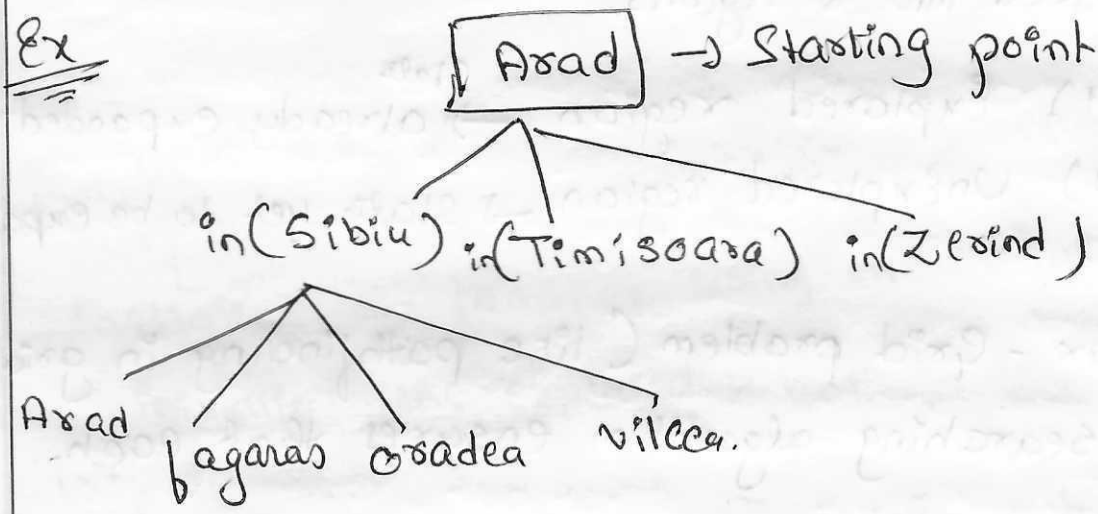
At each step the algorithm selects a node from the frontier, expands it & add its child node to the frontiers.

Key Steps of TREE - SEARCH

- Initialize the frontiers [Start with the initial State

- Loop until the frontiers is empty
  - If Empty → No more nodes to explore
  - If node contains the goal state → The algorithm terminates Succesfully.

Ex



Function TREE-SEARCH (problem) return a solution or failure
initialize the frontier using the initial state of problem
loop do
if the frontier is empty then return failure
Choose a leaf node & remove it from the frontiers
if the node contains a goal state then return the corresponding Solution
Expand the chosen node, adding the resulting nodes to the frontiers

GRAPH - SEARCH ALGORITHM

In the Graph - Search algorithm, the State - space graph is divided into 2 regions.

- (1) Explored region → state already expanded
- (2) Unexplored region → states yet to be expand

Rectangular - Grid problem (like pathfinding in grid) the Graph-searching algorithm ensures that each State is only expanded once.

Illustration

Consider a rectangular grid where the Goal is to find a path from the top - left corner to the bottom - right Corner

1. The start state is placed in the frontier
2. The first state from frontier is expanded & moved into explored region
3. After expanding a state, its neighbors are added to the frontier, which grows as more States are generated

The process continues until the algorithm finds the goal state.

Figure 3.9

Function GRAPH-SEARCH (problem) returns a solution or failure

initialize the frontier using the initial state of problem

initialize the explored set to be empty

loop do

if the frontier is empty then return failure

Choose a leaf node & remove it from frontier

If the node contains a goal state then return the corresponding solution

add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

UNINFORMED SEARCH STRATEGIES

(1) Breadth - first Search

Def²

Breadth - first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next then their successors.

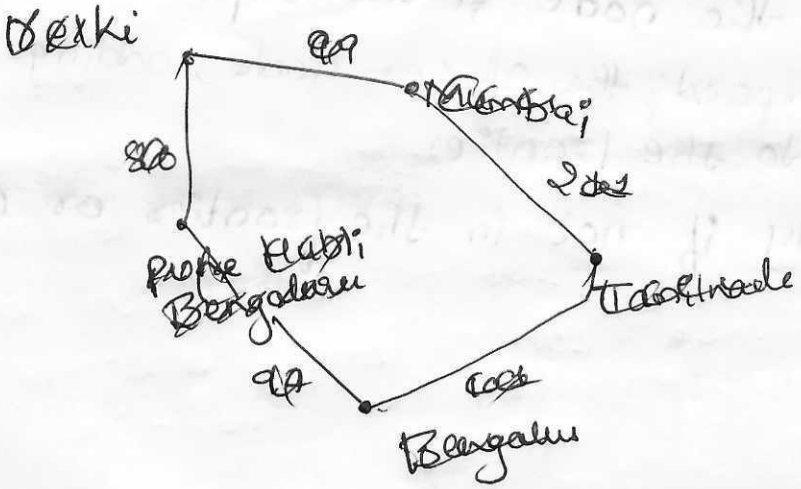
→ Explore the graph/tree level-by-level.

→ Uses a FIFO (First-in, First out) queue to keep track of nodes

→ Visit nodes at each level before moving to next

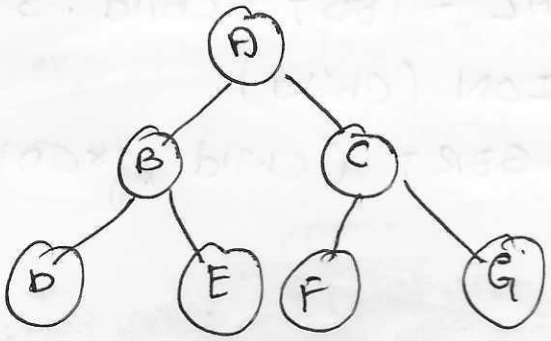
→ Ensure the shortest path is found first

Ex



(I) ~~Delhi~~ – ~~Hubli~~ – ~~Bengaluru~~ – ~~Tamilnadu~~ = ~~278~~

(II) ~~Delhi~~ – ~~Mumbai~~ – Tamilnadu = ~~310~~



Breadth - first Search on a Graph → Function

Function BREADTH - FIRST - SEARCH (problem) return 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)

add node. STATE to explored

for each action in problem. ACTIONS (node. STATE) do

child ← STATE is not in explored / frontier then

* frontier → set of nodes that are waiting waiting to be expanded.

Assignment No. Date :

if child.STATE is not in explored / frontier then
if problem.GOAL-TEST(child.STATE) then
return SOLUTION(child)
frontier ← INSERT(child, frontier)

Expl?

1. Initialize
   - Start from initial state with cost = 0
   - If this node is the goal, return it as the solution
   - Use FIFO queue called frontier to store nodes to be explored
   - Maintain an explored set to track visited nodes & avoid cycle
2. Loop until solution is found / frontier is empty
   - Remove (POP) the shallowest node from frontier
   - Add its state to the explored set
   - Generate child nodes from available action
   - If a child node is not in the explored set or frontier, insert it into the queue
   - If goal is found return the solution

* POP → The shallowest node (first inserted) is removed (POP) & expanded first

Assignment No. Date :

Use of BFS

→ Guaranteed to find the shortest path
→ Works well for unweighted Graphs
→ Complete (always finds a solution if one exists)

Not memory efficient (store all nodes in memory)

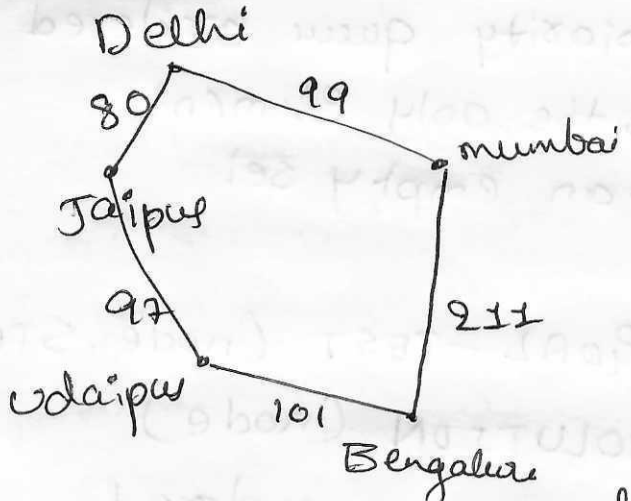
UNIFORM - COST SEARCH

Def?

Uniform - cost search (UCS) is an search algorithm that finds the least - cost path from start node to goal node

Work

- Use a priority queue
- Expand the node with the lowest total path cost ($g(n)$) first
- If a newly discovered path to a node is cheaper than the previously found path, update the cost
- The first goal node selected for expansion is guaranteed to be optional



(I) Delhi - Jaipur - Udaipur - Bengaluru - 278

(II) Delhi - Mumbai - Bengaluru - 310.

Eg

→ First path from Delhi — Mumb — Bengaluru
Rs. 25,000/-

→ Late Delhi — Bhopal — Nagpur — Bengaluru
with Rs. 23,000/-

23,000/- Cheaper, we update the cost & choose this path.

→ FUNCTION

```
function UNIFORM-COST-SEARCH (problem)
  returns a solution, or failure
node ← a node with STATE = problem.INITIAL
  -STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST
  with nodes as the only element
    explored ← an empty set
loop do
  if problem.GOAL-TEST (node.STATE) then
      return SOLUTION (node)
  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
frontier ← INSERT (child, frontier)
else if child.STATE is in frontier with higher
PATH-COST then
replace that frontier node with child.

Expln

1. Initialize the search
Start with a initial node with cost zero
- Add this node to the priority queue (frontier) which always prioritize the node with the lowest cost.
- Keep an empty explored set to track visited nodes

2. Expand the lowest-cost node
- Remove the node with the smallest path cost from the queue
- If node is a goal, return the sol?
- otherwise, mark it as explored

3. Explore child nodes.
for each neighbor (child node) of the current
- if it is not in the explored set or the frontier, add it to the frontier
- If it is in the frontier but with a higher cost, replace it with the cheaper path

4. Repeat until goal is found / no nodes left in the queue.

Depth-first Search

Dep

→ DFS always expands the deepest node in the current frontier of the search tree

→ use LIFO queue

1. Time Complexity of DFS

In a graph search, DFS explores a maximu

of $O(|V| + |E|)$ nodes

- $|V|$ → number of vertices
- $|E|$ → Number of edges

→ In tree search DFS can explore $O(b^m)$

b → branching factor (number of children per node

m → maximum depth of the tree

Use of DFS

✓ Space Complexity → DFS only stores $O(bm)$ nodes, whereas BFS requires $O(b^d)$ nodes
↳ dept of shallowest solutio

✓ Memory efficiency

• DFS is the foundation for many AI technique — logic programming
— propositional satisfaction

✓ Backtracking search (help to reduce memory usage

✓ Applications → pathfinding algorithm.
N Queen, Sodoku Solver.

DEPTH - LIMITED SEARCH

Def²

Depth - Limited search (DLS) is a variation of Depth - first search (DFS)

Where the search is restricted to a fixed depth limit l.

Use of Depth - Limited Search

1. Solves the problem of infinite loops in DFS by enforcing a depth limit

2. Uses $O(b^l)$ time complexity

   $O(bl)$ → Space "

   b → branching factor (no. of children / node)

   l → depth limit

Drawbacks : Incomplete: If the solution is deeper than l, it won't be found

- Non - Optimal → If l is too large, it may find a deeper, Sub optimal solution

Function DEPTH-LIMITED-SEARCH (problem, limit)
return Solution | failure | cutoff

if problem.is-goal (node.state):
return node ⟹ Solution found
else if limit = 0 then return cutoff
else
cutoff-occurred ← false
for each in problem.actions (node.state)d
~~Child ← problem.result~~ (node,
Child ← CHILD-NODE (problem, node, action)
result ← RECURSIVE-DLS (Child, problem,
limit − 1)
if result = cutoff then cutoff-occured?
← true
else if result ≠ failure then return result
if cutoff-occured ? then return cutoff else
return failure

ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Def?

IDDFS is a depth-first search
↳ Combination of Depth first search (DFS) & Breadth-first search

→ It repeatedly runs Depth-Limited Search (DLS) with increasing depth limit until a solut is found

function ITERATIVE-DEEPENING-SEARCH (proble
return a solution | failure
for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH (problem, depth)
if result ≠ cutoff then return result

Working

Start with depth limit l = 0
Run depth-limited search with this lim
If no solution found, increase l & rep

4. Continue until a solution is found.