**Q.1. State space representation of a problem helps us search for the solution(s) using known search algorithms. Represent the following problems as state-space.**

**i) Missionaries and cannibals**

**ii) Tower of Hanoi**

**iii) Bridges of Konigsberg**

**iv) Travelling Salesperson**

**v) Knapsack**

**i) Missionaries and Cannibals**

**Problem Statement:**

Three missionaries and three cannibals must cross a river using a boat that can hold up to two people. The condition is that at no point should the cannibals outnumber the missionaries on either side of the river; otherwise, the cannibals will eat the missionaries.

**State-Space Representation:**

* **State**: A state can be represented as a tuple (M,C,B) where:
  + M​: Number of missionaries on the current side of the river.
  + C​: Number of cannibals on the current side of the river.
  + B: Boat's position (either on the current side or on the opposite side).
* **Initial State**: (3, 3, current side).
* **Goal State**: (0, 0, opposite side) — all missionaries and cannibals are on the opposite side of the river.
* **Actions**: Possible moves are to transport 1 or 2 people across the river (missionaries, cannibals, or both), ensuring the boat's capacity is not exceeded.
* **Transition Model**: Moves change the number of missionaries and cannibals on each side of the river and update the boat’s position.

**ii) Tower of Hanoi**

**Problem Statement:**

There are three rods and in n disks of different sizes. The goal is to move all the disks from one rod to another, obeying the rules that only one disk can be moved at a time, and no disk may be placed on top of a smaller disk.

**State-Space Representation:**

* **State**: A state can be represented as a tuple of three lists (A,B,C)(A, B, C)(A,B,C), where each list represents the disks on a rod, ordered from bottom (largest) to top (smallest).
* **Initial State**: All disks on the first rod, e.g., A=[n,n−1,…,1]A = [n, n-1, \dots, 1]A=[n,n−1,…,1], B=[]B = []B=[], C=[]C = []C=[].
* **Goal State**: All disks moved to the third rod, e.g., A=[]A = []A=[], B=[]B = []B=[], C=[n,n−1,…,1]C = [n, n-1, \dots, 1]C=[n,n−1,…,1].
* **Actions**: Move the top disk from one rod to another, subject to the rule that no larger disk can be placed on a smaller one.
* **Transition Model**: Transitions occur by removing the top disk from one rod and placing it on another rod.

**iii) Bridges of Königsberg**

**Problem Statement:**

This classic graph theory problem involves finding a walk through the city of Königsberg that crosses each of its seven bridges exactly once.

**State-Space Representation:**

* **State**: The state can be represented as a node in a graph that corresponds to the position on the map (at a specific landmass or bridge).
* **Initial State**: Starting at any node representing one of the landmasses.
* **Goal State**: Visiting all bridges exactly once.
* **Actions**: Crossing a bridge to move from one landmass to another.
* **Transition Model**: A valid transition happens when a person moves from one landmass to another by crossing a bridge. Each bridge can be crossed only once.

**iv) Travelling Salesperson**

**Problem Statement:**

The traveling salesperson problem (TSP) involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city.

**State-Space Representation:**

* **State**: A state is represented as a tuple of cities visited in a sequence.
* **Initial State**: Starting city.
* **Goal State**: A complete tour where all cities are visited once, and the salesperson returns to the starting city.
* **Actions**: Moving from one city to another.
* **Transition Model**: A move from one city to another based on the cost (distance) between them, updating the sequence of visited cities.

**v) Knapsack Problem**

**Problem Statement:**

The knapsack problem involves selecting a subset of items, each with a weight and a value, to maximize the total value without exceeding the weight capacity of the knapsack.

**State-Space Representation:**

* **State**: A state is represented as a tuple (W,V,S)(W, V, S)(W,V,S), where:
  + WWW: Current total weight of selected items.
  + VVV: Current total value of selected items.
  + SSS: A list of selected items.
* **Initial State**: (0, 0, []) — no items selected, total weight and value are zero.
* **Goal State**: Any state where the total weight is less than or equal to the capacity and the total value is maximized.
* **Actions**: Adding an item to the knapsack (if it doesn't exceed the weight limit).
* **Transition Model**: Each action of adding an item updates the total weight and value.

**Q.2. A variety of search techniques can be used to search for a problem solution. Explain the following algorithms giving their pseudocode representation.**

**i) Beam search**

**ii) Iterative deepening A\* search**

**iii) Bidirectional search**

**iv) Simulated Annealing search**

**i) Beam Search**

**Explanation:**

Beam search is a heuristic search algorithm that is an optimization of the **best-first search**. Instead of exploring all child nodes, it explores only a limited number of the most promising nodes (based on a heuristic) at each level of the search tree. The size of the "beam" defines how many nodes are kept at each level. It sacrifices completeness for efficiency, as it does not guarantee finding the optimal solution but can be useful in large search spaces.

**Pseudocode:**

python

Copy code

function BeamSearch(problem, beam\_width):

# Initialize the beam with the initial state

beam = [problem.initial\_state()]

while beam is not empty:

# Generate all possible successors from the current beam states

successors = []

for state in beam:

successors += problem.get\_successors(state)

# Sort the successors by the heuristic value (ascending order)

successors.sort(key=lambda state: problem.heuristic(state))

# Keep only the top "beam\_width" number of states

beam = successors[:beam\_width]

# Check if any state in the beam is a goal state

for state in beam:

if problem.is\_goal(state):

return state # Goal found

return failure # If no solution is found

* **Beam Width**: Defines how many states to keep at each level.
* **Successors**: All child nodes from the current states.
* **Heuristic**: A function that estimates the cost of reaching the goal from a state.

**ii) Iterative Deepening A\* (IDA\*) Search**

**Explanation:**

Iterative Deepening A\* (IDA\*) is a combination of the depth-first search and A\* search. It performs a series of depth-limited searches where the depth is defined by an increasing cost threshold, which is guided by a heuristic function. IDA\* is optimal like A\*, but it uses less memory.

**Pseudocode:**

python

Copy code

function IDAStar(problem):

threshold = problem.heuristic(problem.initial\_state())

while True:

result = DFS\_Limited(problem, problem.initial\_state(), 0, threshold)

if result is a solution:

return result

if result == infinity:

return failure

threshold = result # Update threshold to the new limit

function DFS\_Limited(problem, node, g, threshold):

f = g + problem.heuristic(node) # f = cost so far + heuristic

if f > threshold:

return f # Return the next threshold value

if problem.is\_goal(node):

return node # Goal found

min\_threshold = infinity

for successor in problem.get\_successors(node):

result = DFS\_Limited(problem, successor, g + problem.cost(node, successor), threshold)

if result is a solution:

return result

if result < min\_threshold:

min\_threshold = result # Update minimum threshold

return min\_threshold

* **Threshold**: Limit based on the estimated cost f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n), where g(n)g(n)g(n) is the cost so far and h(n)h(n)h(n) is the heuristic estimate to the goal.
* **DFS\_Limited**: A depth-first search with a limit based on the current threshold.

**iii) Bidirectional Search**

**Explanation:**

Bidirectional search simultaneously searches forward from the initial state and backward from the goal state, meeting in the middle. This method reduces the search space drastically, making it faster for large problems. The search proceeds until the two frontiers (from the start and the goal) meet.

**Pseudocode:**

python

Copy code

function BidirectionalSearch(problem):

# Initialize frontiers for both forward and backward search

forward\_frontier = Queue()

forward\_frontier.enqueue(problem.initial\_state())

backward\_frontier = Queue()

backward\_frontier.enqueue(problem.goal\_state())

forward\_visited = set() # Set of states visited by forward search

backward\_visited = set() # Set of states visited by backward search

while not forward\_frontier.is\_empty() and not backward\_frontier.is\_empty():

# Expand the forward frontier

forward\_node = forward\_frontier.dequeue()

forward\_visited.add(forward\_node)

if forward\_node in backward\_visited:

return reconstruct\_path(forward\_node) # Path found

for neighbor in problem.get\_successors(forward\_node):

if neighbor not in forward\_visited:

forward\_frontier.enqueue(neighbor)

# Expand the backward frontier

backward\_node = backward\_frontier.dequeue()

backward\_visited.add(backward\_node)

if backward\_node in forward\_visited:

return reconstruct\_path(backward\_node) # Path found

for neighbor in problem.get\_predecessors(backward\_node):

if neighbor not in backward\_visited:

backward\_frontier.enqueue(neighbor)

return failure # No solution found

* **Frontiers**: Two queues for forward and backward exploration.
* **Visited Sets**: To track explored nodes for both directions.
* **Meeting Point**: The algorithm terminates when the two searches meet.

**iv) Simulated Annealing Search**

**Explanation:**

Simulated annealing is a probabilistic search algorithm inspired by the annealing process in metallurgy. The algorithm explores the search space by accepting both better and worse solutions, but over time, it becomes more selective (reducing the "temperature"). The probability of accepting worse solutions decreases as the algorithm "cools down."

**Pseudocode:**

python

Copy code

function SimulatedAnnealing(problem):

current = problem.initial\_state()

T = initial\_temperature()

while T > minimum\_temperature:

next\_state = random\_choice(problem.get\_successors(current))

delta\_E = problem.value(next\_state) - problem.value(current)

if delta\_E > 0:

current = next\_state # Always move to a better state

else:

# Move to a worse state with a probability based on the temperature

if random() < exp(delta\_E / T):

current = next\_state

T = schedule(T) # Reduce the temperature according to the schedule

return current # Return the final state found

* **Initial State**: Start with an initial solution.
* **Temperature**: Controls how likely the algorithm is to accept worse solutions. It starts high and gradually decreases.
* **Acceptance Probability**: A function of the difference in value between the current and next states, scaled by the current temperature.
* **Cooling Schedule**: The temperature TTT decreases over time according to a schedule (e.g., exponentially).