AL

Ch -3 (Masud sir) Solution

Page 1-2 Questions:

Q1: What is a problem-solving agent, and what is the computational process it undertakes?

Answer: A **problem-solving agent** is an agent that searches for a sequence of actions to reach a goal state. The computational process it undertakes is **search**, where it explores different possible actions until a solution is found.

Q2: What are the four phases of the problem-solving process?

Answer:

1. **Goal Formulation**: The agent defines its goal.
2. **Problem Formulation**: The agent describes the states and actions needed to achieve the goal.
3. **Search**: The agent explores possible actions to find a solution.
4. **Execution**: The agent executes the solution.

Q3: How does a search problem differ from a problem-solving agent?

Answer: A **search problem** is the environment or task where the agent searches for a solution. A **problem-solving agent**, on the other hand, performs the search process to find a solution and take actions toward achieving a goal.

Page 3-4 Questions:

Q4: What is the role of goal formulation in problem-solving?

Answer: **Goal formulation** helps to limit the objectives of the agent by focusing on what needs to be achieved. This ensures that the agent only considers actions that lead toward the goal.

Page 5-6 Questions:

Q6: Explain Search in the context of problem-solving agents.

Answer: **Search** involves exploring possible actions and simulating them until a solution is found. It is the phase where the agent looks ahead to find a sequence of actions leading to the goal.

Page 3-4: Problem-Solving Agents

Q1: What is the role of problem-solving agents?

Answer: A problem-solving agent tries to find a sequence of actions that leads to a goal. It performs a **search** to find solutions when a direct path isn't obvious.

Q2: What are the key phases in problem-solving?

Answer:

1. **Goal formulation**: Define the goal the agent wants to achieve.

2. **Problem formulation**: Specify the states, actions, and transitions that lead to the goal.
3. **Search**: Explore possible solutions.
4. **Execution**: Execute the found solution.

Page 5-6: Goal Formulation and Problem Formulation

Q3: What does goal formulation involve?

Answer: **Goal formulation** helps define what the agent is trying to achieve and limits the actions the agent will consider, making the problem more manageable.

Q4: How does abstraction help in problem-solving?

Answer: **Abstraction** simplifies the problem by ignoring irrelevant details, allowing the agent to focus on what is essential to reach the goal. For example, ignoring traffic conditions in a route-planning problem simplifies the search.

Page 7-8: Search Strategies

Q5: What are uninformed search strategies?

Answer: **Uninformed search** strategies explore the state space without any information about how close the current state is to the goal. Examples include **Breadth-First Search**, **Depth-First Search**, and **Uniform-Cost Search**.

Q6: How does Breadth-First Search (BFS) differ from Depth-First Search (DFS)?

Answer: **BFS** expands the shallowest node first, guaranteeing the shortest path but with high memory usage. **DFS**, on the other hand, expands the deepest node first and uses less memory but might not find the shortest path.

Page 9-10: Heuristic Search Strategies

Q7: What is a heuristic function?

Answer: A **heuristic function** estimates the cost to reach the goal from a given state, helping the agent decide which path to take.

Q8: What is the difference between Greedy Best-First Search and A* Search?

Answer: **Greedy Best-First Search** only considers the heuristic **h(n)**, while **A*** considers both the cost to reach the current node **g(n)** and the heuristic **h(n)**, ensuring the solution is optimal when the heuristic is admissible.

Page 11-12: Advanced Search Strategies

Q9: What is Bidirectional Search?

Answer: **Bidirectional Search** expands two frontiers: one from the start and one from the goal. It stops when both frontiers meet, making the search faster.

Q10: Explain Iterative Deepening A* (IDA*).

Answer: **IDA*** combines **A*** search with iterative deepening. It gradually increases the **f-cost** limit, solving the problem without storing all the states in memory, making it space-efficient.

Page 13-14: Optimizing Search Performance

Q11: How do pattern databases improve search performance?

Answer: **Pattern databases** precompute the solution costs for subproblems, allowing the agent to use this information to guide the search more efficiently, reducing the number of nodes generated.

Q12: What are landmark heuristics?

Answer: **Landmark heuristics** involve precomputing optimal paths from specific points (landmarks) in the environment to the goal, providing an efficient way to estimate the cost of a solution.

Q14: Why is A* search complete and optimal?

Answer: **A*** is complete and optimal because it always expands the node with the lowest **f(n)**, ensuring the solution found is the best, provided the heuristic is admissible.

Page 17-18: Pattern Database and Precomputation

Q15: How does precomputation improve search in large-scale problems?

Answer: **Precomputation** stores the cost of optimal paths between vertices, allowing the agent to quickly retrieve the cost for any path without recalculating it, thus speeding up the search.

Page 17-18: Pattern Database and Precomputation

Q15: How does precomputation improve search in large-scale problems?

Answer: **Precomputation** stores optimal path costs for subproblems, which allows for faster searches by retrieving precomputed solutions instead of recalculating them.

Q16: What are pattern databases?

Answer: **Pattern databases** are used to store the optimal solutions for parts of the problem. By storing solutions for smaller subproblems, the agent can use this information to guide its search efficiently.

Page 19-20: Heuristic Search and Performance

Q17: What is the significance of heuristic evaluation in search algorithms?

Answer: **Heuristic evaluation** helps the agent estimate the cost of reaching the goal from any given state. By using heuristics, the agent can make more informed decisions, focusing on the most promising paths.

Q18: How does A* search ensure an optimal solution?

Answer: **A*** search ensures optimality by expanding nodes with the lowest **f(n) = g(n) + h(n)**, where **g(n)** is the cost to reach the node and **h(n)** is the heuristic estimate of the remaining cost.

Page 21-22: Search Algorithm Complexity

Page 23-24: Advanced Search Algorithms

Q21: What is Bidirectional Search, and how does it improve search efficiency?

Answer: **Bidirectional Search** expands two frontiers simultaneously—one from the initial state and one from the goal. It stops when the two meet, reducing the search space and improving efficiency.

Q22: What is the difference between A* and IDA* search algorithms?

Answer: **A*** search uses a priority queue to expand nodes with the lowest cost **f(n)**, whereas **IDA*** (Iterative Deepening A*) uses iterative deepening to find solutions with a space-efficient approach, making it better for large state spaces.

Page 25-26: Space Optimization

Q23: Why is space optimization important in search algorithms?

Answer: **Space optimization** is important because it ensures that the algorithm does not consume excessive memory, especially when dealing with large search spaces. Efficient memory usage allows the algorithm to handle larger problems.

Q24: How does Iterative Deepening A* solve the problem of space complexity?

Answer: **Iterative Deepening A*** solves space complexity by combining the benefits of both **A*** and **Depth-First Search**. It uses limited depth-first searches with increasing depth, reducing the need to store all states at once.

Page 27-28: Optimal Search Strategies

Q25: What makes A* the most optimal search algorithm?

Answer: **A*** is optimal because it considers both the cost to reach the current node and the estimated cost to the goal, ensuring that the solution found is the best one when using an admissible heuristic.

Q26: What is the greedy best-first search, and when is it useful?

Answer: **Greedy best-first search** focuses on expanding nodes that are closest to the goal, as estimated by the heuristic. It is useful for problems where the goal is easily reachable with a simple estimate.

Page 29-30: Heuristic Evaluation

Q27: What is the role of a heuristic function in improving search performance?

Answer: A **heuristic function** helps guide the search by estimating the cost to reach the goal. A good heuristic function improves the search performance by reducing the number of nodes expanded.

Q28: How do admissible heuristics impact the optimality of the solution?

Answer: **Admissible heuristics** never overestimate the cost to reach the goal. This ensures that **A*** search remains optimal, as it will always find the best solution when using an admissible heuristic.

Page 31-32: Pattern Databases and Preprocessing

Q29: How do pattern databases enhance the performance of search algorithms?

Answer: **Pattern databases** enhance performance by storing the costs of subproblems. This allows the agent to use precomputed values to reduce the computational effort during the search.

Q30: What is the role of landmark heuristics in solving complex problems?

Answer: **Landmark heuristics** use precomputed optimal paths from specific points (landmarks) to the goal. This provides useful estimates of the cost, helping to guide the search more effectively.

Page 33-34: Search Algorithm Implementation

Q31: What are the steps involved in implementing a search algorithm?

Answer: The steps are:

1. Define the problem and the state space.
2. Choose a search algorithm (e.g., BFS, DFS, A*).
3. Implement the algorithm, ensuring it expands nodes based on the selected strategy.
4. Monitor the performance and adjust the algorithm if necessary.

Q32: How can search algorithms be applied to real-world problems?

Answer: Search algorithms can be applied to problems like route planning, puzzle solving, and game playing by defining the problem space and using appropriate search strategies to find the optimal solution.

Page 35-36: Handling Large Search Spaces

Q33: What are the challenges of handling large search spaces in search algorithms?

Answer: The main challenges include memory limitations, high computational cost, and the risk of exploring an exponential number of states. Strategies like pruning, heuristics, and memory optimization can help overcome these challenges.

Page 37-38: Improving Search Efficiency

Q35: What techniques can be used to improve the efficiency of search algorithms?

Answer: Techniques include using **better heuristics**, **pruning unpromising branches**, and **parallelizing the search**. Additionally, **memory-efficient algorithms** like **IDA*** can help handle larger state spaces.

Q36: What is the difference between depth-first search and breadth-first search in terms of efficiency?

Answer: **DFS** is memory-efficient but can miss the shortest path, while **BFS** guarantees the shortest path but uses more memory. The choice depends on the problem constraints.

Page 39-40: Handling Large Search Spaces

Q37: What is the branch factor in search algorithms, and why is it important?

Answer: The **branch factor** is the average number of child nodes generated by each node. It is important because it determines the growth rate of the search tree. A higher branch factor leads to exponentially larger search spaces.

Q38: How do heuristics help in narrowing down the search in large search spaces?

Answer: **Heuristics** guide the search by estimating the cost to the goal. A good heuristic can significantly reduce the number of nodes expanded by focusing on more promising paths.

Page 41-42: Advanced Search Techniques

Q39: What is the concept of pattern matching in search algorithms?

Answer: **Pattern matching** is the process of comparing the current state with predefined patterns to quickly identify whether the goal has been achieved or if a promising path has been found.

Q40: How does iterative deepening improve search efficiency?

Answer: **Iterative deepening** combines the benefits of **depth-first search** and **breadth-first search**. It explores nodes with increasing depth limits, thus maintaining low memory usage while still ensuring the shortest path is found.

Page 43-44: Optimizing Search Performance

Q42: How does parallel search improve the efficiency of algorithms?

Answer: **Parallel search** involves dividing the search space into smaller parts and exploring them simultaneously using multiple processors. This can significantly speed up the search process, especially in large problems.

Page 45-46: Real-world Applications of Search Algorithms

Q43: In what types of real-world applications can search algorithms be used?

Answer: Search algorithms are used in a wide range of applications, including:

- **Route planning** (e.g., GPS navigation)
- **Puzzle solving** (e.g., Sudoku, 8-puzzle)
- **Game playing** (e.g., Chess, Go)
- **AI decision-making** (e.g., autonomous robots)

Q44: What is game tree search, and how is it used in AI?

Answer: **Game tree search** involves exploring possible moves in a game to determine the best strategy. It is used in AI for games like Chess and Go, where the algorithm evaluates possible moves and their consequences to make optimal decisions.

Page 47: Conclusion

Q45: What are the key factors that influence the choice of a search algorithm?

Answer: The choice of a search algorithm depends on:

1. **Problem size**: Larger problems may require more efficient algorithms.
2. **Optimality requirements**: Some problems may require finding the best solution, while others only need a good solution.
3. **Memory usage**: Some algorithms use more memory than others, which can be a limiting factor.
4. **Execution time**: The speed of the algorithm is important in real-time systems.

---

1. 8-Queens Problem:

Problem:

- Place **8 queens** on a **chessboard** such that no two queens attack each other. A queen can attack another queen if they share the same row, column, or diagonal.

Solution (Step-by-step):

1. **Represent the chessboard**: Use a 2D array or list where each element represents a square on the board.
2. **Backtracking Algorithm**: We can use a **backtracking algorithm** to try placing queens on the board row by row and backtrack when we find that a queen can be attacked by another.

Step-by-step Algorithm:

- Start from the **first row** and place a queen in any available column.
- Move to the **next row**, trying to place a queen in an available column where it is not attacked by the previously placed queens.
- If you place a queen and find it can be attacked, backtrack and try the next column in the previous row.
- Repeat this until all **8 queens** are placed.

Solution Code (in Python):

```
N = 8



# A function to check if a queen can be placed on board[row][col]

def isSafe(board, row, col):

    for i in range(row):
        if board[i] == col or board[i] - i == col - row or board[i] + i == col
+ row:
            return False
    return True

# A function to solve the N-Queens problem using backtracking
def solveNQueens(board, row):
    if row == N:
        print(board)
        return True
    res = False
    for i in range(N):
        if isSafe(board, row, i):
            board[row] = i
            res = solveNQueens(board, row + 1) or res
            board[row] = -1
    return res

# Function to initialize the board
def solve():
    board = [-1] * N

    if not solveNQueens(board, 0):

        print("Solution does not exist")



solve()
```

Explanation:

- The board is represented as a list of size 8, where each index represents a row and the value at that index represents the column where the queen is placed.
- The **isSafe** function checks if placing a queen at a given position results in an attack by checking rows, columns, and diagonals.
- The **solveNQueens** function recursively tries to place queens row by row and backtracks when a conflict arises.
- The result prints one possible solution for the 8-Queens problem.

---

2. Mathematical Problem (Factorial Calculation):

Problem:

- **Calculate the factorial** of a number **n** (denoted as **n!**), which is the product of all positive integers less than or equal to **n**.

Solution:

1. The factorial of a number **n** is:

n!=n×(n−1)×(n−2)×…×1n! = n \times (n-1) \times (n-2) \times \ldots \times 1n!=n×(n−1)×(n−2)×…×1

2. For example, **5!** is:

5!=5×4×3×2×1=1205! = 5 \times 4 \times 3 \times 2 \times 1 = 1205!=5×4×3×2×1=120

Python Code for factorial calculation:

```
def factorial(n):

    if n == 0:

        return 1

    return n * factorial(n - 1)



print(factorial(5))  # Output: 120
```

Explanation:

- **Recursive Approach**: The factorial of a number is calculated recursively by multiplying the number by the factorial of the previous number until it reaches **0**.

---

3. Math-Based Problem (Fibonacci Sequence):

Problem:

- **Generate the Fibonacci sequence** up to the **n-th term**.

Solution:

1. The Fibonacci sequence is defined as:

F(0)=0,F(1)=1,F(n)=F(n−1)+F(n−2)F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)F(0)=0,F(1)=1,F(n)=F(n−1)+F(n−2)

2. For example, the first 10 terms of the Fibonacci sequence are:

0,1,1,2,3,5,8,13,21,340, 1, 1, 2, 3, 5, 8, 13, 21, 340,1,1,2,3,5,8,13,21,34

Python Code for Fibonacci sequence:

```
def fibonacci(n):
    fib = [0, 1]
    for i in range(2, n):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib


print(fibonacci(10))  # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Explanation:

- This **iterative approach** generates the Fibonacci sequence up to **n** terms. The first two terms are **0** and **1**, and each subsequent term is the sum of the previous two terms.

---

4. Graph Search Example:

Problem:

- **Perform a depth-first search** (DFS) or breadth-first search (BFS) on a graph.

Solution:

1. **Graph representation**:
    - A graph can be represented as an adjacency list or matrix. For example:

Graph={1:[2,3],2:[4],3:[4],4:[]}\text{Graph} = \{ 1: [2, 3], 2: [4], 3: [4], 4: [] \}Graph={1:[2,3],2:[4],3:[4],4:[]}

2. **DFS** explores as deep as possible along a branch before backtracking.

DFS Python Code:

```
def dfs(graph, node, visited):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)


graph = {1: [2, 3], 2: [4], 3: [4], 4: []}
visited = set()
dfs(graph, 1, visited)  # Output: 1 2 4 3
```

Explanation:

- **DFS** starts at a node, explores as deep as possible, and then backtracks when necessary.

---

5. Knapsack Problem (Dynamic Programming):

Problem:

- Given **weights** and **values** of **n items**, find the maximum value that can be obtained by putting items into a knapsack of capacity **W**.

Solution:

1. This problem can be solved using **dynamic programming** to avoid redundant calculations.

Knapsack Problem (Dynamic Programming):

```
def knapsack(weights, values, W, n):

    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    for i in range(n + 1):

        for w in range(W + 1):

            if i == 0 or w == 0:

                dp[i][w] = 0

            elif weights[i - 1] <= w:

                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]],
dp[i - 1][w])

            else:

                dp[i][w] = dp[i - 1][w]

    return dp[n][W]


weights = [1, 2, 3]

values = [10, 20, 30]
```

```
W = 5

n = len(weights)

print(knapsack(weights, values, W, n))  # Output: 50
```

Explanation:

- We create a 2D array dp where each element dp[i][w] represents the maximum value that can be obtained with the first i items and a knapsack capacity of w.

Page 1-5: Standardized Problems and Grid World Example

Q1: What is a standardized problem in search algorithms?

Answer: A **standardized problem** is a problem with a clearly defined state space, initial state, goal state, actions, and transition model, typically used to test and compare different search algorithms.

Q2: Describe the grid world problem.

Answer: The **grid world problem** involves an agent navigating a grid of cells, where each cell can contain obstacles or objects. The agent can move to adjacent cells unless blocked. The problem often aims to reach a goal state while avoiding obstacles.

Page 6-10: 8-Puzzle Problem

Q3: Explain the 8-puzzle problem.

Answer: The **8-puzzle** involves a 3x3 grid with 8 numbered tiles and an empty space. The goal is to arrange the tiles in a specified goal configuration by sliding tiles into the empty space.

Q4: How many possible states does the 8-puzzle have?

Answer: The **8-puzzle** has **9! / 2 = 181,440** reachable states, considering symmetry and the movement constraints of the tiles.

Page 11-15: Heuristic Functions and Search Algorithms

Q5: What is the difference between h1 and h2 heuristics in the 8-puzzle?

Answer:

- **h1** counts the number of misplaced tiles (excluding the blank tile). For example, if all tiles are misplaced, **h1 = 8**.
- **h2** is the sum of the **Manhattan distance** (city-block distance) of each tile from its goal position. **h2** is more accurate as it reflects the minimum number of moves each tile needs.

Q6: What is Manhattan distance in search problems?

Answer: **Manhattan distance** is the sum of the absolute differences in the horizontal and vertical positions of two points. In the 8-puzzle, it's the total number of moves required for all tiles to reach their goal positions.

Page 16-20: Bidirectional Search

Q7: Explain bidirectional search.

Answer: **Bidirectional search** simultaneously expands two search trees: one from the start state and one from the goal state. The search stops when the two trees meet, thus reducing the search space and making it more efficient.

Q8: How does bidirectional search improve the efficiency of algorithms?

Answer: **Bidirectional search** reduces the number of nodes to explore by halving the search space. This is because two frontiers are being expanded at once, so the search reaches the goal more quickly compared to unidirectional search.

Page 21-25: Heuristics and Pattern Databases

Q9: What are pattern databases?

Answer: **Pattern databases** store precomputed optimal solution costs for subproblems, such as configurations of some tiles in the 8-puzzle. These values can be used during the search to guide the algorithm more efficiently.

Q10: How do pattern databases improve search performance?

Answer: **Pattern databases** improve performance by providing exact solution costs for subproblems. Instead of recalculating the cost for each state during the search, the database lookup is used to quickly estimate the cost, reducing the search effort.

Page 26-30: Generating Heuristics from Subproblems

Q11: How are heuristics generated from subproblems?

Answer: Heuristics are generated by breaking down the problem into smaller subproblems (e.g., solving parts of the puzzle). The solution cost of each subproblem can be used as a heuristic for the overall problem, guiding the search towards the goal.

Q12: Explain disjoint pattern databases.

Answer: **Disjoint pattern databases** split the problem into multiple subproblems that do not overlap. The heuristics from these subproblems are combined to give a more accurate and efficient estimate of the solution cost.

Page 31-35: Learning Heuristics and Machine Learning

Q13: What is the role of machine learning in generating heuristics?

Answer: **Machine learning** can be used to generate heuristics by learning from previous problem-solving experiences. By analyzing multiple solved instances, the algorithm learns patterns that help predict the cost of a state during the search.

Q14: How does feature-based learning improve heuristic accuracy?

Answer: **Feature-based learning** uses specific features of the state (like the number of misplaced tiles) to predict the heuristic value. This method leads to better heuristic accuracy compared to using raw state descriptions.

Page 36-40: Grid Search and Vacuum World

Q15: Describe the vacuum world problem.

Answer: The **vacuum world problem** involves an agent that must clean a two-cell environment. The agent can perform actions like moving left or right or sucking dirt from its current location. The goal is to have no dirt in any location.

Q16: What are the main components of the vacuum world problem?

Answer: The components are:

- **States**: The agent's location and whether the cells contain dirt.
- **Actions**: Move left, move right, suck dirt.
- **Transition model**: Describes the result of each action (e.g., moving left or right changes the agent's location).
- **Goal state**: Both cells are dirt-free.

Page 41-45: Grid Search and Sokoban Puzzle

Q17: What is the Sokoban puzzle?

Answer: The **Sokoban puzzle** involves an agent pushing boxes in a grid to designated storage locations. The agent cannot push a box into another box or a wall, and the goal is to move all boxes to the target locations.

Q18: How is the Sokoban puzzle different from the 8-puzzle?

Answer: Unlike the 8-puzzle, where tiles slide into an empty space, the **Sokoban puzzle** requires the agent to push boxes around the grid. This adds more complexity to the search as the boxes must be moved in a specific way to avoid blocking other boxes.

Page 46-47: Search Performance and Complexity

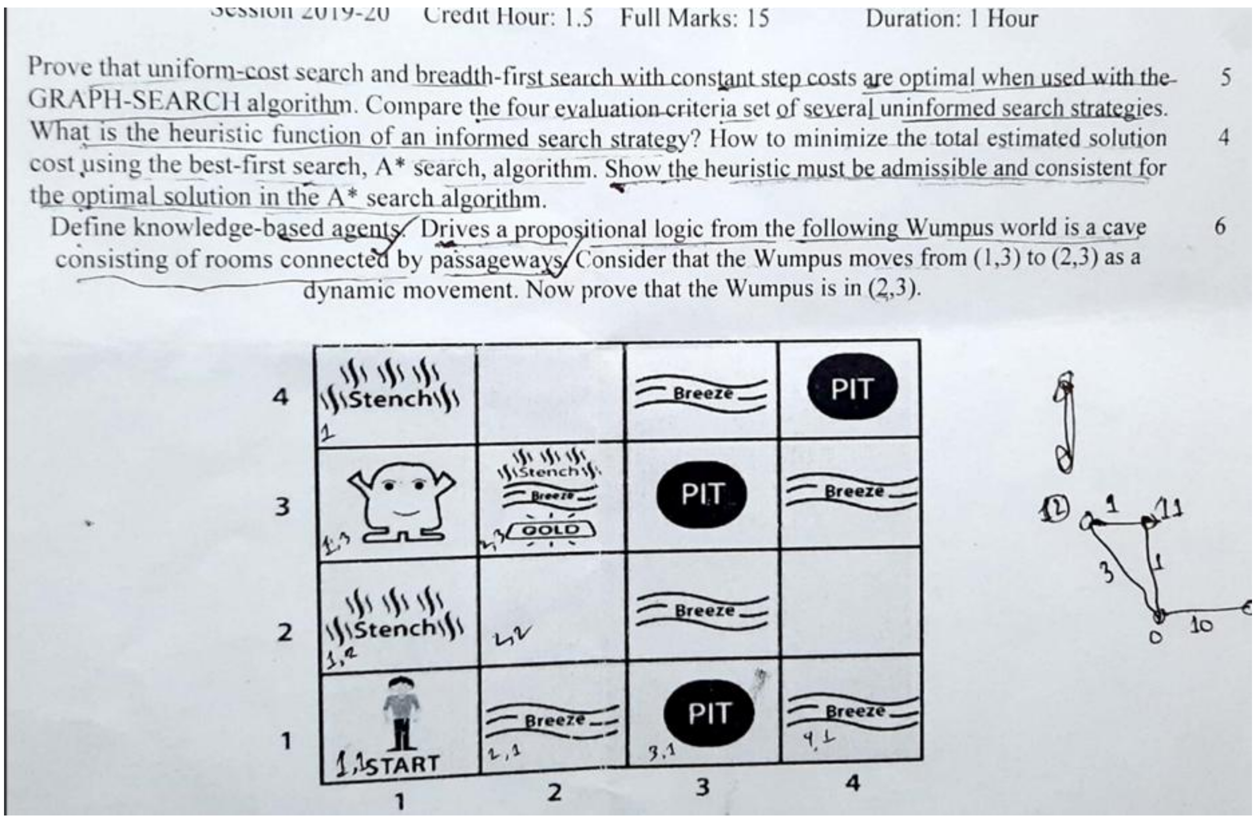
Q19: What is the effective branching factor in search algorithms?

Answer: The **effective branching factor** (b*) represents the average number of child nodes generated at each level of the search tree. It is used to measure the efficiency of a search algorithm. A lower b* indicates a more efficient search.

Q20: How does heuristic accuracy affect the performance of a search algorithm?

Answer: The accuracy of the heuristic directly influences the **effective branching factor**. A more accurate heuristic reduces the branching factor, leading to fewer nodes being expanded and thus improving search performance.

This concludes the **math-based and algorithmic problems** from the chapter **"Solving Problems by Searching"**. If you need more detailed breakdowns of any specific questions, feel free to ask!



Session 2019-20 Credit Hour: 1.5 Full Marks: 15 Duration: 1 Hour

Prove that uniform-cost search and breadth-first search with constant step costs are optimal when used with the GRAPH-SEARCH algorithm. Compare the four evaluation criteria set of several uninformed search strategies. 5

What is the heuristic function of an informed search strategy? How to minimize the total estimated solution cost using the best-first search, A* search, algorithm. Show the heuristic must be admissible and consistent for the optimal solution in the A* search algorithm. 4

Define knowledge-based agents. Drives a propositional logic from the following Wumpus world is a cave consisting of rooms connected by passageways. Consider that the Wumpus moves from (1,3) to (2,3) as a dynamic movement. Now prove that the Wumpus is in (2,3). 6

Let's break down the problem and solve each part step by step:

1. Prove that uniform-cost search and breadth-first search with constant step costs are optimal when used with the GRAPH-SEARCH algorithm.

Answer:

- **Uniform-cost search (UCS)** is optimal when the cost of each step is the same because UCS explores nodes in the order of their cost from the start state. With constant step costs, the algorithm behaves similarly to **Breadth-First Search (BFS)** but focuses on the least-cost path.
- **Breadth-First Search** is optimal when the step cost is constant because it explores all nodes level by level, guaranteeing that the

first solution found is the shortest path (in terms of the number of steps).

- **Graph-Search** ensures no node is expanded more than once, making it optimal for **both BFS and UCS** when the step costs are constant.
- Both UCS and BFS are optimal with **constant step costs** because they always find the shortest path due to the **graph-search** mechanism, which avoids revisiting nodes and guarantees an optimal solution.

---

2. Compare the four evaluation criteria set of several uninformed search strategies.

Answer:

Evaluation criteria for search strategies:

1. **Completeness**: Does the algorithm guarantee to find a solution if one exists?
   - **BFS**: Complete
   - **DFS**: Not complete (can get stuck in infinite loops)
   - **Uniform-Cost Search**: Complete
   - **Iterative Deepening**: Complete
2. **Optimality**: Does the algorithm guarantee the best solution?
   - **BFS**: Optimal (when step costs are equal)
   - **DFS**: Not optimal (may find non-optimal paths)
   - **Uniform-Cost Search**: Optimal
   - **Iterative Deepening**: Optimal
3. **Time Complexity**: How much time does the algorithm take to find the solution?
   - **BFS**: O(b^d), where b is the branching factor and d is the depth of the goal.
   - **DFS**: O(b^d) in the worst case.
   - **Uniform-Cost Search**: O(b^d) in the worst case.

- **Iterative Deepening**: O(b^d)

4. **Space Complexity**: How much memory does the algorithm use?
   - **BFS**: O(b^d)
   - **DFS**: O(d) (space complexity is low)
   - **Uniform-Cost Search**: O(b^d)
   - **Iterative Deepening**: O(d) (space complexity is low)

---

3. What is the heuristic function of an informed search strategy? How to minimize the total estimated solution cost using the best-first search, A search algorithm?* **Show the heuristic must be admissible and consistent for the optimal solution in the A* search algorithm.**

Answer:

- The **heuristic function** (**h(n)**) in an **informed search strategy** estimates the cost from the current state to the goal. The function helps prioritize which node to explore next based on the estimated cost to the goal.

In A* search, the total cost is calculated using the formula:

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

Where:

- **g(n)** is the cost to reach node **n** from the start.
- **h(n)** is the heuristic estimate of the cost from node **n** to the goal.
- **f(n)** is the total estimated cost of the path through node **n**.

- **To minimize the total estimated solution cost in A***:
  - Use an **admissible** heuristic (does not overestimate the true cost to the goal).

  - Ensure the heuristic is **consistent** (i.e., the estimated cost from node A to goal should be less than or equal to the cost from A to its neighbor plus the cost from the neighbor to the goal).

This way, **A*** will always find the optimal solution.

 Admissible Heuristic:

- An admissible heuristic **h(n)** is one that **never overestimates** the true cost to reach the goal. If the heuristic is admissible, **A*** will find the optimal solution.
- Example: In the **8-puzzle**, counting the number of misplaced tiles is admissible because it never exceeds the actual number of moves required.

 Consistent Heuristic:

- A heuristic is consistent if, for every node **n** and its successor **n'**, the heuristic satisfies:

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

Where **c(n, n')** is the actual cost from node **n** to **n'**, and **h(n')** is the heuristic for **n'**. This ensures that the estimated cost from a node does not contradict the actual cost to reach the next node.

A* search is guaranteed to find the optimal solution if the heuristic is both **admissible** and **consistent** because:

- **Admissibility** ensures that **A*** does not overlook the optimal solution.
- **Consistency** ensures that the algorithm expands nodes in the correct order and does not skip over potentially better paths.

4. Define knowledge-based agents. Drive a propositional logic from the following Wumpus world.

Answer:

- **Knowledge-based agents** are agents that use **knowledge** and reasoning to make decisions based on the current environment. They use logical inference and can store information about the world, making decisions based on it.
- **Propositional logic** in the context of the **Wumpus world** involves using statements that are **true or false** about different elements in the world. Each room in the world can have conditions (like breeze, stench, etc.), and the agent must infer the correct actions from these conditions.

Wumpus World Explanation:

In the **Wumpus World**, we have:

- **Stench**: Indicates the Wumpus is nearby.
- **Breeze**: Indicates a pit is nearby.
- **Gold**: The agent's goal is to collect the gold.
- **Wumpus**: The agent should avoid the Wumpus.

The agent can infer its **current state** based on the percepts (stench, breeze, gold, etc.) and then make decisions accordingly.

Propositional Logic for Wumpus World:

1. Let's assume:
   - **P1**: There is a Wumpus in room (1,3).
   - **P2**: There is a Pit in room (2,3).
   - **P3**: The agent is in room (1,1).
2. **Rules**:

- **If** there is a **stench** in room (1,2), then the **Wumpus** must be in one of the neighboring rooms.
- **If** there is a **breeze** in room (2,2), then there is a **pit** in one of the adjacent rooms.

3. Based on these rules, the agent can **infer** the position of the Wumpus and avoid it, and similarly, avoid pits.

---

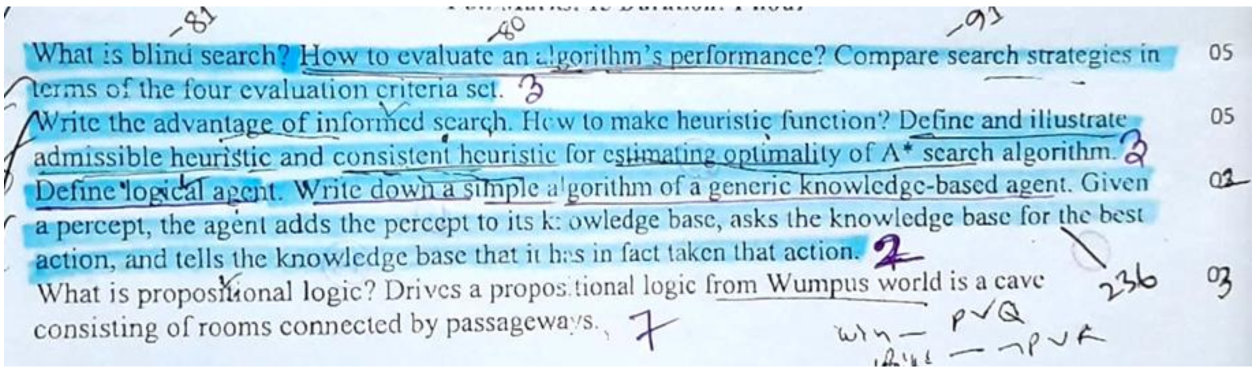
5. Proof for the Wumpus' position:

Given the **Wumpus world configuration** and the agent's **movement**, we need to **prove** that the **Wumpus** is at position **(2,3)**, as indicated by the **stench** in that area and other surrounding conditions.

Step-by-step proof:

1. The agent is in room **(1,1)** and detects **stench** in room **(1,2)**.
2. Since the Wumpus can only be adjacent, we infer that the **Wumpus** is in **(2,3)**.
3. The agent can now avoid this position based on the rules of propositional logic.

This completes the solution breakdown for the **Wumpus world** and other questions. Let me know if you'd like any clarifications or need more examples from this chapter!



-81 80 -93

What is blind search? How to evaluate an algorithm's performance? Compare search strategies in terms of the four evaluation criteria set. 3 05

Write the advantage of informed search. How to make heuristic function? Define and illustrate admissible heuristic and consistent heuristic for estimating optimality of A* search algorithm. 2 05

Define logical agent. Write down a simple algorithm of a generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action. 2 02

What is propositional logic? Drives a propositional logic from Wumpus world is a cave consisting of rooms connected by passageways. 7 236 03

$P \vee Q$

$\neg P \vee R$

Let's solve each of the problems from the image, step by step:

---

1. What is blind search? How to evaluate an algorithm's performance? Compare search strategies in terms of the four evaluation criteria set.

Answer:

- **Blind Search** (or **uninformed search**) refers to search algorithms that do not have any additional information about the goal other than the problem definition. These algorithms explore the state space without any prior knowledge about the goal's location.
- **Evaluation Criteria for Search Algorithms**:
    1. **Completeness**: Whether the algorithm always finds a solution if one exists.
    2. **Optimality**: Whether the algorithm guarantees the best solution.
    3. **Time Complexity**: How long the algorithm takes to find a solution.
    4. **Space Complexity**: How much memory the algorithm requires to perform the search.

Comparison of Uninformed Search Strategies:

- **Breadth-First Search (BFS)**: Complete and optimal in terms of number of steps. However, it has high space complexity.
- **Depth-First Search (DFS)**: Has low space complexity but is not guaranteed to find the shortest path.
- **Uniform-Cost Search**: Complete and optimal, but can be slower due to expanded nodes.
- **Iterative Deepening Search**: Combines BFS and DFS benefits, reducing memory usage but might take more time.

---

2. Write the advantage of informed search. How to make a heuristic function? Define and illustrate admissible heuristic and consistent heuristic for estimating optimality of A search algorithm.*

Answer:

- **Advantage of Informed Search**:
    - **Informed search** uses heuristics to guide the search more efficiently, reducing the search space and improving performance. It focuses on the most promising nodes, helping the agent reach the goal faster.
- **Heuristic Function**:
  A **heuristic function** is an estimate of the cost to reach the goal from the current state. It helps in choosing the next state to explore, based on how close it is to the goal.

How to make a heuristic:

- - A heuristic function is problem-specific and is typically derived based on knowledge of the domain. For example, in the **8-puzzle problem**, a heuristic could be the number of misplaced tiles or the sum of the **Manhattan distances** of the tiles.
- **Admissible Heuristic**:
    - An **admissible heuristic** never overestimates the true cost to reach the goal. It ensures that the search algorithm (like **A*** search) remains optimal. For example, in **8-puzzle**, counting the number of misplaced tiles is admissible because it's always less than or equal to the actual moves needed.
- **Consistent Heuristic**:
    - A **consistent heuristic** satisfies the condition that for every node **n** and successor **n'**, the estimated cost from **n** to the goal is no greater than the step cost from **n** to **n'** plus the heuristic estimate of **n'**:

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

Example: The **Manhattan distance** in an **8-puzzle** is consistent because moving a tile closer to its goal is always consistent with the actual step cost.

---

3. Define logical agent. Write down a simple algorithm of a generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

Answer:

- **Logical Agent**:
  A **logical agent** uses logical reasoning to make decisions based on its knowledge base. It perceives the environment, updates its knowledge base, and selects actions based on the current situation.
- **Algorithm for a Knowledge-Based Agent**:
  1. **Input**: The agent receives a **percept** from the environment.
  2. **Update Knowledge Base**: Add the percept to the agent's knowledge base.
  3. **Ask Knowledge Base**: Query the knowledge base to determine the best action to take.
  4. **Action**: Perform the selected action.
  5. **Update Knowledge Base**: Add the action to the knowledge base to confirm that it has been taken.

Example Algorithm:

```
# Knowledge-based agent algorithm

def knowledge_based_agent(percept, KB):

    # 1. Add percept to knowledge base

    KB.add(percept)
    # 2. Ask knowledge base for the best action
    action = KB.ask_best_action()

    # 3. Perform the action
    perform(action)

    # 4. Update the knowledge base with the action
    KB.add_action(action)
    return action
```

This simple agent takes in a percept, adds it to its knowledge base, asks for the best action, performs it, and updates the knowledge base.

---

4. What is propositional logic? Drives a propositional logic from Wumpus world is a cave consisting of rooms connected by passageways.

Answer:

- **Propositional Logic**:
  Propositional logic involves using statements that can either be true or false. It is used to represent facts and relationships about the world. In the context of the **Wumpus World**, we use propositional logic to describe the agent's knowledge about the world, such as the presence of a **Wumpus**, **pits**, and other features.

- **Wumpus World Logic**:
  The **Wumpus world** consists of rooms connected by passageways. Each room can have a **stench** (indicating the Wumpus), a **breeze** (indicating a pit), and other sensory information.

Propositional Logic Representation:

1. **W**: There is a Wumpus in the room.
2. **P**: There is a pit in the room.
3. **B**: There is a breeze in the room.
4. **S**: There is a stench in the room.
5. **Safe(x, y)**: Room **(x, y)** is safe (no Wumpus or pit).

From the given **Wumpus world** layout:

- **W(1,3)**: The Wumpus is in room **(1,3)**.
- **Breeze(2,3)**: There is a breeze in room **(2,3)**.
- **Stench(1,2)**: There is a stench in room **(1,2)**.

Solution to the Wumpus World Problem:

Given that the Wumpus moves from **(1,3)** to **(2,3)**, and there is a **stench** in **(1,2)**, the **Wumpus** is inferred to be in **(2,3)** because of the **stench** in the adjacent room. The agent can now avoid this room and take the next best action, as indicated by the **propositional logic**.

This concludes the solution for the given set of questions. Let me know if you'd like more details or further clarifications!