**ASSIGNMNET ANSWERS 2**

**1. Maze-Solving with Greedy Best-First Search (GBFS)**

**Working Principle of Greedy Best-First Search (5 marks)**

Greedy Best-First Search (GBFS) is an informed search algorithm that expands the node closest to the goal based on a heuristic function **h(n)**. It follows these steps:

1. Start at the initial position (0,0).
2. Select the neighboring cell with the lowest heuristic value **h(n)** (i.e., estimated distance to the goal).
3. Move to that cell and repeat until reaching the goal (4,4).
4. If a dead end is reached, backtrack to the last valid position and choose another path.

**Application in Maze-Solving:**

* GBFS prioritizes paths that seem promising based on heuristic estimates.
* It works well in open mazes but may fail in cases with dead ends, as it does not consider the actual cost of reaching a node.

**Heuristic Function for Maze-Solving (5 marks)**

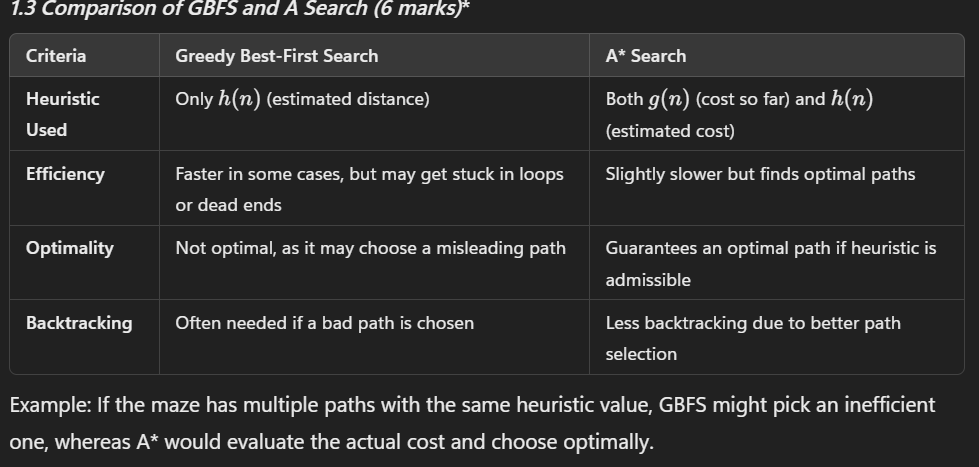
A common heuristic function for this problem is the **Manhattan Distance**, which calculates the shortest path assuming no obstacles:

h(n)=∣xgoal−xn∣+∣ygoal−yn∣h(n) = |x\_{\text{goal}} - x\_n| + |y\_{\text{goal}} - y\_n|h(n)=∣xgoal​−xn​∣+∣ygoal​−yn​∣

For example, at position (2,3), the heuristic is:

h(2,3)=∣4−2∣+∣4−3∣=2+1=3h(2,3) = |4 - 2| + |4 - 3| = 2 + 1 = 3h(2,3)=∣4−2∣+∣4−3∣=2+1=3

This function is **admissible** (never overestimates) and **consistent** (always satisfies the triangle inequality).



**Limitations of GBFS (4 marks)**

1. **Gets Trapped in Dead Ends:** GBFS follows the heuristic blindly, leading it into dead ends, requiring costly backtracking.
2. **Ignores Path Cost:** It may choose a longer path just because it looks closer to the goal.
3. **Multiple Good Paths Confusion:** If multiple paths have the same heuristic, GBFS may not consistently pick the best one.
4. **Performs Poorly in Mazes with Obstacles:** It does not consider obstacles effectively, which can lead to unnecessary detours.

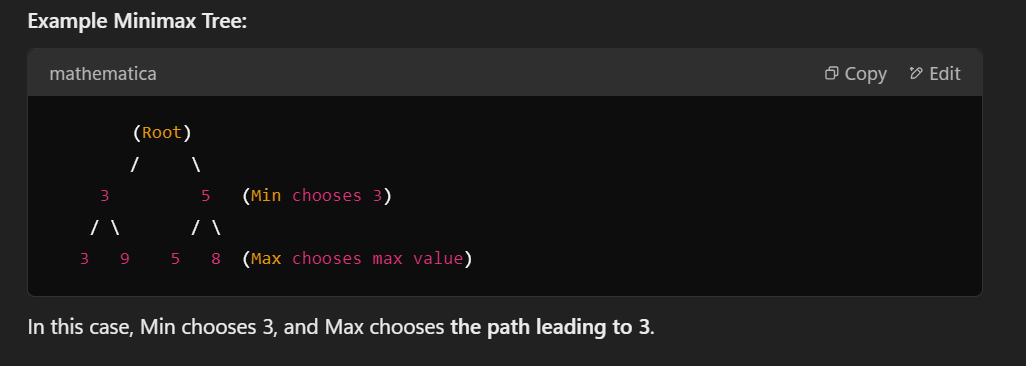
**2. Chess-like Board Game AI**

**2.1 Minimax Algorithm in Adversarial Search (5 marks)**

Minimax is a **decision-making algorithm** for two-player games where one player maximizes the score while the other minimizes it.

**Working Principle:**

1. The game is represented as a **tree**, where each node is a game state.
2. Player 1 (Max) tries to **maximize** the score, while Player 2 (Min) tries to **minimize** it.
3. The algorithm evaluates possible moves and chooses the best one for the maximizing player.
4. It recursively explores all possible moves up to a depth **d** and picks the best move using: Best Move=max⁡(min⁡(next state values))\text{Best Move} = \max(\min(\text{next state values}))Best Move=max(min(next state values))



**Alpha-Beta Pruning for Efficiency (5 marks)**

Alpha-Beta pruning **optimizes Minimax** by eliminating branches that cannot influence the final decision.

**Steps:**

1. Maintain two values:
   * **Alpha (α)**: Best value found for Max.
   * **Beta (β)**: Best value found for Min.
2. If a node's value is worse than an already explored option, **prune** it (ignore it).
3. This reduces the number of nodes to evaluate, speeding up Minimax.

**Example of Pruning:**

1. 5
2. / \
3. 3 (X) Pruned (β ≤ α)
4. / \
5. 3 9

Here, once the left subtree gives **3**, the right subtree is **not evaluated** if it's worse.

**Evaluation Function for Game (6 marks)**

The evaluation function assigns a score to a game state to help Minimax make decisions.

**Components of the Evaluation Function:**

f(state)=W1(Material Value)+W2(Mobility)+W3(King Safety)+W4(Piece Positioning)

Where:

* **Material Value**: Number and type of pieces (e.g., **pawn = 1**, **knight = 3**, **queen = 9**).
* **Mobility**: Number of available moves for a player.
* **King Safety**: Distance of the king from threats.
* **Piece Positioning**: Advantageous positions (e.g., center control in chess).

Example Calculation:

* Player 1 has a **queen (9), rook (5), 2 pawns (2)** → Score = **16**
* Player 2 has a **bishop (3), knight (3), 3 pawns (3)** → Score = **9**
* **f(state) = 16 - 9 = 7** (Player 1 has an advantage)

**Impact of Branching Factor on Performance (4 marks)**

The **branching factor** is the number of possible moves per turn. A high branching factor leads to:

1. **Exponential Growth in Computation**: More branches mean deeper search trees, increasing complexity.
2. **Slower Decision Making**: The AI takes longer to evaluate moves.
3. **Need for Pruning**: Techniques like **alpha-beta pruning** reduce the number of nodes searched.
4. **Use of Heuristic Search**: Instead of searching all moves, **heuristic-based searches like Monte Carlo Tree Search (MCTS)** are used.

**Example:**

* **Chess has ~35 possible moves per turn**, requiring deep search optimization.
* **Tic-Tac-Toe has ~9 moves**, making exhaustive search feasible.

**Conclusion**

* **GBFS is fast but inefficient in complex mazes**; A\* is **better for optimal solutions**.
* **Minimax and Alpha-Beta pruning improve decision-making in board games**.
* **Evaluation functions and pruning strategies are key to optimizing AI performance**.
* **Branching factor impacts AI efficiency**, requiring smart search techniques.

These principles apply to both **maze-solving robots and AI board game players**, showing how search algorithms help intelligent agents make better decisions.

**3. Definition of the 8-Puzzle Problem (2 Marks)**

The **8-puzzle problem** is a classic sliding tile puzzle consisting of a **3×3 grid** with **8 numbered tiles** and **one empty space (0)**. The goal is to rearrange the tiles from a given **initial configuration** to a **goal state** by sliding tiles into the empty space using **valid moves (up, down, left, right)**. The problem is commonly solved using search algorithms like **A**\* and **Breadth-First Search**.

**2. Manhattan Distance Heuristic for the Initial State (4 Marks)**

The **Manhattan Distance heuristic** is the sum of the absolute differences between the current positions of tiles and their correct positions in the goal state.

**Initial State:**

CopyEdit

1 2 3

4 5 6

7 8 0

**Goal State:**

CopyEdit

1 2 3

4 5 6

7 0 8

**Manhattan Distance Calculation**

* Tile **8** is at **(2,2)** but should be at **(2,1)** → Distance = **|2-2| + |1-2| = 1**
* Tile **0** is at **(2,1)** but should be at **(2,2)** → Distance = **|2-2| + |2-1| = 1**
* All other tiles are in the correct positions.

h(n)=1+1=2h(n) = 1 + 1 = 2h(n)=1+1=2

***3. Cost Functions in A Algorithm (3 Marks)*\***

The *A algorithm*\* combines two cost functions:

1. **g(n) – Path Cost:** Number of moves taken from the initial state to reach the current state.
2. **h(n) – Heuristic Cost:** Estimated cost from the current state to the goal state using Manhattan Distance.
3. **f(n) = g(n) + h(n):** Total estimated cost from the start state to the goal via the current state.

This ensures that the algorithm explores the most promising paths efficiently while guaranteeing an optimal solution.

**4. Generating Possible Moves & Calculating f(n) Values (4 Marks)**

The empty space **(0)** is at **(2,1)**. It can move:

1. **Up to (1,1):** Swaps with tile **5**
2. **Left to (2,0):** Swaps with tile **7**
3. **Right to (2,2):** Swaps with tile **8**



**Constructing the Search Tree (4 Marks)**

**Level 0 (Initial State)**

CopyEdit

1 2 3

4 5 6

7 8 0

**f(n) = 2**

**Level 1 (Possible Moves)**

CopyEdit

1 2 3 1 2 3 1 2 3

4 0 6 4 5 6 4 5 6

7 5 8 0 8 7 7 0 8

**f(n) = 3** **f(n) = 3** **f(n) = 1** ✅ *(Best move: Right swap with 8)*

**Level 2 (Next Moves from Best Choice)**

CopyEdit

1 2 3

4 5 6

7 0 8

Since this matches the **goal state**, we **stop** here.

**6. Optimal Solution Path (3 Marks)**

The shortest path found by **A**\*:

1. **Initial State:**

CopyEdit

1 2 3

4 5 6

7 8 0

1. **Move Right (Swap 0 with 8)**

CopyEdit

1 2 3

4 5 6

7 0 8

✅ **Goal State Reached in 1 Move**

**Total Cost:** **g(n) = 1**  
This is the **optimal solution**, as moving right achieves the goal in the fewest steps.