

**Artificial Intelligence Systems**

Lab Report # 04

**Submitted by:** Ali Hamza

**Registration No:** B23F0063AI106

**Submitted to:** Dr. Mohsin

Date: 25th February , 2025

# Lab Task 1:

# Implement the A\* algorithm to find the shortest path in a grid where certain cells are blocked (obstacles). You are given a grid with obstacles. You need to implement the A\* search algorithm to find the shortest path from a starting point (start \_x, start \_y) to a goal point (goal\_x, goal\_y).

# The grid is represented as a 2D list where:

# 0 represents an open space.

# 1 represents an obstacle.

# The A\* algorithm should use the Manhattan distance as the heuristic function.

# Is this complete? Is this optimal?

**How the A Algorithm Works\***

The **A\*** search algorithm finds the shortest path from a **start node** to a **goal node** by considering both:

1. **Cost so far (g-score)** – The cost to reach a node from the start.
2. **Estimated cost to goal (h-score / heuristic)** – The estimated cost from the current node to the goal (Manhattan distance).

It selects the node with the lowest total cost:

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

Where:

* **g(n)** = Cost to reach node n
* **h(n)** = Heuristic estimate to goal (Manhattan distance)

**Python Implementation**

*import* heapq  
   
 *def* manhattan\_distance(node, goal):  
 *"""Compute the Manhattan distance heuristic."""  
 return* abs(node[0] - goal[0]) + abs(node[1] - goal[1])  
   
 *def* a\_star(grid, start, goal):  
 *"""  
 Implements the A\* algorithm to find the shortest path in a 2D grid with obstacles.  
   
 Parameters:  
 grid : List[List[int]] → 2D grid where 0 = open space, 1 = obstacle  
 start : Tuple[int, int] → (x, y) starting coordinates  
 goal : Tuple[int, int] → (x, y) goal coordinates  
   
 Returns:  
 path : List[Tuple[int, int]] → The shortest path as a list of coordinates, or empty if no path.  
 """  
   
 # Grid dimensions* rows, cols = len(grid), len(grid[0])  
   
 *# Possible movements (Up, Down, Left, Right)* directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
   
 *# Priority queue (Min-Heap) for A\* search* open\_list = []  
 heapq.heappush(open\_list, (0, start)) *# (f-score, (x, y))  
   
 # Dictionaries to store cost and parent path* g\_score = {start: 0} *# Cost from start to each node* came\_from = {} *# Store parent nodes to reconstruct path  
   
 while* open\_list:  
 \_, current = heapq.heappop(open\_list) *# Get node with lowest f-score  
   
 # Goal check  
 if* current == goal:  
 *# Reconstruct path* path = []  
 *while* current *in* came\_from:  
 path.append(current)  
 current = came\_from[current]  
 path.append(start)  
 *return* path[::-1] *# Reverse to get path from start to goal* x, y = current  
 *for* dx, dy *in* directions:  
 neighbor = (x + dx, y + dy)  
   
 *# Ensure within grid bounds  
 if not* (0 <= neighbor[0] < rows *and* 0 <= neighbor[1] < cols):  
 *continue  
   
 # Skip obstacles  
 if* grid[neighbor[0]][neighbor[1]] == 1:  
 *continue  
   
 # Compute new g-score* tentative\_g\_score = g\_score[current] + 1 *# Cost is always 1 per move  
   
 # If new path is better, update  
 if* neighbor *not in* g\_score *or* tentative\_g\_score < g\_score[neighbor]:  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score = tentative\_g\_score + manhattan\_distance(neighbor, goal)  
 heapq.heappush(open\_list, (f\_score, neighbor))  
 came\_from[neighbor] = current *# Track path  
   
 return* [] *# Return empty if no path exists  
   
 # Example Usage  
 if* \_\_name\_\_ == "\_\_main\_\_":  
 grid = [  
 [0, 0, 0, 0, 0],  
 [1, 1, 0, 1, 0],  
 [0, 0, 0, 1, 0],  
 [0, 1, 1, 1, 0],  
 [0, 0, 0, 0, 0]  
 ]  
   
 start = (0, 0) *# Top-left corner* goal = (4, 4) *# Bottom-right corner* path = a\_star(grid, start, goal)  
 print("Shortest Path:", path)

Yes, the A\* algorithm with the **Manhattan distance heuristic** is both **complete** and **optimal** under the given problem constraints.

**1. Completeness**

A search algorithm is **complete** if it is guaranteed to find a solution whenever one exists.

* **A\*** is **complete** because:
* It explores all possible paths in an optimal manner.
* The grid is **finite**, and movement is constrained.
* If a path exists, A\* will find it.

If no path exists (i.e., the goal is completely blocked), A\* correctly reports failure.

**2. Optimality**

An algorithm is **optimal** if it always finds the **shortest path**.

* **A\*** is **optimal** when the heuristic function is **admissible** and **consistent**:
* The **Manhattan distance heuristic** is **admissible** (it never overestimates the true cost).
* It is **consistent** (each move increases cost by at most 1, maintaining a non-decreasing heuristic).

Since A\* expands nodes in order of increasing cost, it **always finds the shortest path** when using Manhattan distance in a grid with uniform movement costs.

# Task 02:

# Implement the A\* search algorithm to find the shortest path in two types of graphs:

# Unweighted Graph: A grid where each move between adjacent nodes has the same cost and assume the heuristic values. In this case, how is A\* algorithm will be different from the BFS algorithm. Write your observations and recommendations for using/not using A\* for weighted graph.

# Weighted Graph: A grid where each move between adjacent nodes can have different costs (weights) and heuristics values.

# Write code for both types of graphs using the A\* algorithm.

* **Unweighted Graph:** Each move has the same cost.
* **Weighted Graph:** Each move has different costs.

1. **Algorithm for an Unweighted Graph\***

* The movement cost between adjacent nodes is uniform (cost = 1).
* The heuristic function used is **Manhattan Distance**.

*import* heapq  
  
  
*def* manhattan\_distance(a, b):  
 *"""Computes the Manhattan distance heuristic."""  
 return* abs(a[0] - b[0]) + abs(a[1] - b[1])  
  
  
*def* a\_star\_unweighted(grid, start, goal):  
 *"""  
 A\* algorithm for an unweighted graph where movement cost is uniform.  
 """* rows, cols = len(grid), len(grid[0])  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] *# Up, Down, Left, Right* open\_set = []  
 heapq.heappush(open\_set, (0, start)) *# (f-score, node)* g\_score = {start: 0}  
 came\_from = {}  
  
 *while* open\_set:  
 \_, current = heapq.heappop(open\_set)  
  
 *if* current == goal:  
 path = reconstruct\_path(came\_from, current)  
 *return* path  
  
 x, y = current  
 *for* dx, dy *in* directions:  
 neighbor = (x + dx, y + dy)  
  
 *if* 0 <= neighbor[0] < rows *and* 0 <= neighbor[1] < cols *and* grid[neighbor[0]][neighbor[1]] == 0:  
 tentative\_g\_score = g\_score[current] + 1 *# Uniform cost  
  
 if* neighbor *not in* g\_score *or* tentative\_g\_score < g\_score[neighbor]:  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score = tentative\_g\_score + manhattan\_distance(neighbor, goal)  
 heapq.heappush(open\_set, (f\_score, neighbor))  
 came\_from[neighbor] = current  
  
 *return None # No path found  
  
  
def* reconstruct\_path(came\_from, current):  
 *"""Reconstructs the path from the goal to the start."""* path = []  
 *while* current *in* came\_from:  
 path.append(current)  
 current = came\_from[current]  
 path.reverse()  
 *return* path  
  
  
*# Example grid (0 = open, 1 = obstacle)*grid\_unweighted = [  
 [0, 0, 0, 0],  
 [1, 1, 0, 1],  
 [0, 0, 0, 1],  
 [0, 1, 0, 0]  
]  
  
start = (0, 0)  
goal = (3, 3)  
  
path\_unweighted = a\_star\_unweighted(grid\_unweighted, start, goal)  
print("Shortest Path (Unweighted Graph):", path\_unweighted)

1. **Algorithm for a Weighted Graph\***

* Movement cost varies for different positions.
* Uses **Manhattan Distance** as a heuristic.

*import* heapq  
  
  
*def* manhattan\_distance(a, b):  
 *"""Computes the Manhattan distance heuristic."""  
 return* abs(a[0] - b[0]) + abs(a[1] - b[1])  
  
  
*def* reconstruct\_path(came\_from, current):  
 *"""Reconstructs the shortest path from the goal to the start."""* path = []  
 *while* current *in* came\_from:  
 path.append(current)  
 current = came\_from[current]  
 path.append(current) *# Add the start node* path.reverse()  
 *return* path  
  
  
*def* a\_star\_weighted(grid, weights, start, goal):  
 *"""  
 A\* algorithm for a weighted graph where movement cost varies.  
 """* rows, cols = len(grid), len(grid[0])  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] *# Up, Down, Left, Right* open\_set = []  
 heapq.heappush(open\_set, (0, start)) *# (f-score, node)* g\_score = {start: 0}  
 came\_from = {}  
  
 *while* open\_set:  
 \_, current = heapq.heappop(open\_set)  
  
 *if* current == goal:  
 *return* reconstruct\_path(came\_from, current)  
  
 x, y = current  
 *for* dx, dy *in* directions:  
 neighbor = (x + dx, y + dy)  
  
 *if* 0 <= neighbor[0] < rows *and* 0 <= neighbor[1] < cols *and* grid[neighbor[0]][neighbor[1]] == 0:  
 move\_cost = weights[neighbor[0]][neighbor[1]]  
 tentative\_g\_score = g\_score[current] + move\_cost  
  
 *if* neighbor *not in* g\_score *or* tentative\_g\_score < g\_score[neighbor]:  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score = tentative\_g\_score + manhattan\_distance(neighbor, goal)  
 heapq.heappush(open\_set, (f\_score, neighbor))  
 came\_from[neighbor] = current  
  
 *return None # No path found  
  
  
# Example grid (0 = open, 1 = obstacle)*grid\_weighted = [  
 [0, 0, 0, 0],  
 [1, 1, 0, 1],  
 [0, 0, 0, 1],  
 [0, 1, 0, 0]  
]  
  
*# Corresponding weights for each cell*weights = [  
 [1, 2, 1, 1],  
 [0, 0, 2, 0],  
 [1, 1, 3, 0],  
 [1, 0, 2, 1]  
]  
  
start = (0, 0)  
goal = (3, 3)  
  
path\_weighted = a\_star\_weighted(grid\_weighted, weights, start, goal)  
print("Shortest Path (Weighted Graph):", path\_weighted)

**Comparison of A and BFS\***

1. **Unweighted Graphs**

* **Breadth-First Search (BFS)** is optimal for unweighted graphs since it explores all nodes at the same level before moving deeper.
* **A**\* still works but **adds an unnecessary heuristic**, making it slightly less efficient.

1. **Weighted Graphs**

* **BFS fails** in weighted graphs because it does not consider cost differences.
* **A**\* is optimal as it prioritizes paths with the lowest cost.

**Conclusion**

* The *A algorithm*\* is optimal and complete, guaranteeing the shortest path when a heuristic is admissible.
* It is **best suited for weighted graphs** but can still be used for unweighted graphs.
* For unweighted graphs, **BFS is often a better choice** due to its simplicity and efficiency.