Maze Pathfinding Algorithms

1. Problem Description and Formulation

The objective of this project is to find the shortest path from a given start position to a goal position within a maze environment. The maze is represented as a two-dimensional grid, where each cell can either be free or blocked by a wall.

Maze Representation

- Free cell is represented by 0
- Wall (blocked cell) is represented by 1

State Representation

A state is defined as the agent's position in the maze and is represented by a coordinate pair:

- **State = (x, y)**

Initial State

- The starting position of the agent:
    - **start = (sx, sy)**

Goal State

- The target position that the agent aims to reach:
    - **goal = (gx, gy)**

Actions (Successor Function)

From any valid cell, the agent can perform one of the following actions:

- Up
- Down
- Left
- Right

An action is valid only if:

- The resulting cell lies within the maze boundaries
- The resulting cell is not a wall

Cost Function

Each valid movement has a uniform cost of **1**.

---

2. Algorithm Implementations and Pseudocode

This project implements and compares three search algorithms:

- Breadth-First Search (BFS)
- A* Search
- Hill Climbing

Breadth-First Search (BFS):

BFS is an Uninformed Search algorithm that explores a search space level by level. Starting from the root node (the start position), it visits all direct neighbors, then all neighbors of those neighbors, and so on.

Application to the Maze Problem:

State Representation:

Every cell in the maze (represented as (row, col)) is a Node.

A valid move between adjacent cells is an Edge.

The Queue (FIFO):

BFS uses a Queue (First-In, First-Out) data structure. The first node added to the queue is the first one to be processed. This ensures that we explore all nodes at distance d before moving to distance d+1.

Systematic Exploration:

The algorithm pops a node from the queue and checks its neighbors (Up, Down, Left, Right).

If a neighbor is valid (within the grid and not a wall) and has not been visited yet, it is added to the queue and marked as visited. This preventing the algorithm from getting stuck in infinite loops.

Optimality (Shortest Path):

Because BFS expands uniformly in all directions (like a ripple in a pond), the first time it hits the goal node, it is guaranteed to have found the path with the minimum number of steps.

Path Reconstruction:

While searching, we maintain a parent_map that links each node to the node that discovered it. Once the goal is found, we backtrack through this map to build the final path from start to finish

Characteristics:

- Complete and optimal
- High memory usage

Pseudocode:

```
BFS(start, goal, maze):
  create an empty queue Q
  create an empty set VISITED
  create an empty map PARENT
  enqueue start into Q
  add start to VISITED
  PARENT[start] ← NULL
  while Q is not empty:
    current ← dequeue(Q)
    if current == goal:
      return RECONSTRUCT_PATH(PARENT, goal)
    for each neighbor in GET_NEIGHBORS(current, maze):
      if neighbor is not in VISITED and neighbor is not a wall:
          add neighbor to VISITED
          PARENT[neighbor] ← current
          enqueue neighbor into Q
  return FAILURE
RECONSTRUCT_PATH(PARENT, goal):
  path ← empty list
  current ← goal
  while current is not NULL:
    add current to the beginning of path
    current ← PARENT[current]
  return path
```

A* Search

A* (A-Star) is an informed search algorithm used to find the shortest path between a start state and a goal state.

Unlike uninformed algorithms (such as BFS or DFS), A* uses heuristic information to guide the search toward the goal more efficiently.

The main idea of A* is to evaluate each node using the following function:

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

Where:

$g(n)$: the actual cost from the start node to the current node

$h(n)$: a heuristic estimate of the cost from the current node to the goal

$f(n)$: the estimated total cost of the solution path through node n

How A* Works in the Maze:

Initialize the search with the start node

Add the start node to the Open List

Repeatedly:

Select the node with the lowest $f(n)$ value

If it is the goal, terminate the search

Otherwise, expand the node and generate its neighbors

For each neighbor:

Compute g(n) (path cost so far)

Compute h(n) using Manhattan distance

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

Update the node information if a better path is found

Continue until the goal is reached or no nodes remain

Pseudocode:

```
A_STAR(start, goal):
   open_list ← priority queue ordered by f(n)
   closed_list ← empty set
   g(start) ← 0
   f(start) ← h(start)
   add start to open_list
   while open_list is not empty:
      current ← node in open_list with lowest f value
      if current == goal:
         return RECONSTRUCT_PATH(current)
      remove current from open_list
      add current to closed_list
      for each neighbor of current:
         if neighbor is a wall or in closed_list:
            continue
         tentative_g ← g(current) + cost(current, neighbor)
         if neighbor not in open_list
           OR tentative_g < g(neighbor):
            parent(neighbor) ← current
            g(neighbor) ← tentative_g
```

```
        f(neighbor) ← g(neighbor) + h(neighbor)
        if neighbor not in open_list:
            add neighbor to open_list
return FAILURE
```

Hill Climbing :

Hill Climbing Search is a local search algorithm used in artificial intelligence to solve optimization problems.

It focuses on gradually improving a solution by always moving toward a neighboring state that appears closer to the goal according to a heuristic function.

How the Algorithm Works in the Maze:

Start at the initial cell and add it to the path.

Generate neighbors that are within the maze and not walls.

Calculate H(n) (Manhattan distance) for each neighbor.

Select the neighbor with the smallest H(n) and move to it.

Add the new position to the path.

Repeat until:

You reach the goal → return the path

No neighbor improves H(n) → return FAILURE

◆ **Key Points**

Only one state is stored at a time (current).

Previous states are forgotten (no backtracking).

May get stuck in:

Local maxima

Plateaus

Ridges

Does not guarantee the shortest path, only moves toward improvement.

Characteristics:

- Fast and memory-efficient
- Not complete
- May get stuck in local optima

Pseudocode:

```
function HillClimbingMaze(start, goal, maze):
    current = start
    path = [current]              # Initialize path
    while current != goal:
        neighbors = get_neighbors(current, maze)
        if neighbors is empty:
            return FAILURE          # No valid moves available
        # Evaluate heuristic H(n) for each neighbor
        best_neighbor = None
        best_h = infinity
        for neighbor in neighbors:
            h = heuristic(neighbor, goal)
            if h < best_h:
                best_h = h
                best_neighbor = neighbor
        # Check for improvement
        if best_h >= heuristic(current, goal):
            return FAILURE or path    # Stuck at local maximum / plateau
        # Move to best neighbor
        current = best_neighbor
        path.append(current)
    return path                   # Goal reached
# Helper function to generate valid neighbors
function get_neighbors(cell, maze):
    neighbors = []
    for direction in [up, down, left, right]:
        new_cell = move(cell, direction)
        if new_cell is inside maze and not a wall:
            neighbors.append(new_cell)
    return neighbors
# Heuristic function (Manhattan distance)
function heuristic(cell, goal):
    return abs(cell.x - goal.x) + abs(cell.y - goal.y)
```

3. Heuristic Design and Justification (for A*)

The heuristic used for the A* algorithm is the **Manhattan Distance**, defined as:

$h(x, y) = |x - gx| + |y - gy|$

Justification

- Suitable for grid-based environments
- Admissible (never overestimates the true cost)
- Consistent and efficient

4. Experimental Methodology

To evaluate the performance of each algorithm, several maze configurations of different sizes and obstacle densities were used.

Evaluation Metrics

- Execution Time
- Memory Usage
- Number of Nodes Explored
- Path Cost
- Solution Optimality

Each algorithm was executed multiple times, and the average results were recorded to ensure fairness.

---

5. Results and Analysis

Performance Comparison

- **BFS** consistently found the optimal path but explored a large number of nodes.
- **A*** achieved optimal solutions while significantly reducing the number of explored nodes due to heuristic guidance.
- **Hill Climbing** was the fastest but often failed to find a solution or produced suboptimal paths.

Summary Table

Algorithm	Optimal	Nodes Explored	Time	Reliability
BFS	Yes	High	Medium	High
A*	Yes	Medium	Low	High
Hill Climbing	No	Low	Very Low	Low

---

6. Conclusions and Recommendations

This project demonstrates the trade-offs between completeness, optimality, and efficiency in search algorithms.

- **A*** is recommended for most maze pathfinding problems due to its balance between optimality and performance.
- **BFS** is suitable for small problem spaces where memory is not a concern.
- **Hill Climbing** can be used when speed is critical and approximate solutions are acceptable.

---

7. References

1. Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach*. Pearson.
2. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths.
3. Course lecture notes and materials.