TANTA UNVIRISITY

FIRE ESCAPE ROUTE AGENT

Under Supervision:

Dr/Ahmed Selim

Eng/Omar Khaled

Eng/Ahmed Sobhy

TEAM MEMBERS

Maram Hazem Fouad Ismail Ahmed (CS, C4)
Menna Ahmed Ibrahim Agamy (CS, C5)
Wessam Mohammed El-Sayed El-Hanafy (CS, C5)
Heba Ahmed Ibrahim Agamy (CS, C5)
Mohamed Elsayed Mohamed Ahmed Aboelsoud (CS, C4)
Mohamed Khaled El-Daheesh Ahmed (CS, C4)
Mohamed Refat Mostafa Abd-Elmajid Naser (CS, C4)

PROJECT OVERVIEW

Project Definition

"Fire Escape Route Agent" is a pathfinding problem study, which studies and aims to the find the fastest and most safe path to take in a mall in case of a fire.

Scenario Description

- The environment is represented as a **3D grid-based maze**
- Walls represent blocked areas
- The agent starts from a predefined position
- The goal is a **safe exit gate**
- Only **one algorithm runs per experiment**
- The simulation visualizes:
  - Exploration phase
  - Final escape path
- This setup allows observing algorithm behavior under **hard and constrained conditions**

Project Objective

- Implement and compare classical AI search algorithms
- Simulate a realistic evacuation scenario
- Identify the most efficient algorithm for guiding people safely to exits
- Evaluate algorithms based on efficiency and optimality

PROJECT IMPLEMENTATION

The code used was divided into 3 modules:

The Maze Module

This module defines the maze environment and provides 3D visualization capabilities, and the agent moment behavior

The Main Module

This module acts as the program's entry point and orchestration layer

The Search Algorithms Module

This module implements the 6 AI pathfinding algorithms for the study.

THE MAZE MODULE

Check if position is valid Get valid neighbors for a position

```
def is_valid_position(x, z):
    """Check if position is valid (within bounds and not a wall)"""
    return (0 <= x < COLS and 0 <= z < ROWS and maze[z][x] == 0)

def get_neighbors(x, z):
    """Get valid neighbors for a position"""
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    neighbors = []
    for dx, dz in directions:
        nx, nz = x + dx, z + dz
        if is_valid_position(nx, nz):
            neighbors.append((nx, nz))
    return neighbors
```

THE MAZE MODULE

Update agent position based on time

```
def update_agent(self):
    """Update agent position based on time"""
    current_time = time.time()

    if current_time - self.last_update_time >= self.move_delay:
        if self.is_exploring and self.current_move_index < len(self.movement_sequence):
            pos = self.movement_sequence[self.current_move_index]
            self.agent.move_to(pos[0], pos[1])
            self.current_move_index += 1
            self.last_update_time = current_time

            if self.current_move_index >= len(self.movement_sequence):
                self.is_exploring = False
                self.is_following_path = True
                self.current_move_index = 0
                #the agent now is following the optimal path


        elif self.is_following_path and self.current_move_index < len(self.final_path):
            pos = self.final_path[self.current_move_index]
            self.agent.move_to(pos[0], pos[1])
            self.current_move_index += 1
            self.last_update_time = current_time

            if self.current_move_index >= len(self.final_path):
                self.is_following_path = False
                #the agent reched the goal
```

THE SEARCH ALGORITHMS MODULE

Implement the backtracking technique

```
def get_path_between(start_pos, end_pos, parent_map):

    if start_pos == end_pos:
        return []

    # Build path from start_pos to root
    path_from_start = []
    current = start_pos
    while current is not None:
        path_from_start.append(current)
        current = parent_map.get(current)

    # Build path from end_pos to root
    path_from_end = []
    current = end_pos
    while current is not None:
        path_from_end.append(current)
        current = parent_map.get(current)
```

```
    # Find common ancestor (lowest common ancestor)
    path_from_start_set = set(path_from_start)
    common_ancestor = None
    for node in path_from_end:
        if node in path_from_start_set:
            common_ancestor = node
            break
```

```
    # Build backtrack path: start -> common ancestor
    backtrack_path = []
    current = start_pos
    while current != common_ancestor:
        current = parent_map.get(current)
        if current is not None:
            backtrack_path.append(current)

    # Build forward path: common ancestor -> end
    forward_path = []
    current = end_pos
    while current != common_ancestor:
        forward_path.append(current)
        current = parent_map.get(current)
    forward_path.reverse()

    # Combine paths
    return backtrack_path + forward_path
```

THE SEARCH ALGORITHMS MODULE

Build the heuristic function and assign a consistent random cost for every node

```python
def heuristic(a, b):
    # a = (x1, y1), b = (x2, y2)
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

```python
random.seed(42)
node_costs = {
    (c, r): random.randint(1, 20)
    for c in range(COLS)
    for r in range(ROWS)
}
```

THE SEARCH ALGORITHMS MODULE

Handle reached goal function

```
def handle_goal_reached(
    current,
    goal,
    parent_map,
    discovered_nodes,
    movement_sequence,
    algo_name,
    path_label
):


    # Reconstruct path
    final_path = []
    temp = current
    while temp is not None:
        final_path.append(temp)
        temp = parent_map[temp]
    final_path.reverse()

    return movement_sequence, discovered_nodes, final_path
```

THE MAIN MODULE

plan how to run the search algorithms

```python
def run_algorithm(algorithm_choice, move_delay=0.15):
    global visualizer, algorithm_name
    if algorithm_choice.lower() == 'bfs':
        algorithm_name = "BFS (Breadth-First Search)"
        movement_seq, discovered, path = bfs(START_POS, GOAL_POS)
    elif algorithm_choice.lower() == 'dfs':
        algorithm_name = "DFS (Depth-First Search)"
        movement_seq, discovered, path = dfs(START_POS, GOAL_POS)
    elif algorithm_choice.lower() == 'ucs':
        algorithm_name = "UCS (Uniform Cost Search)"
        movement_seq, discovered, path = ucs(START_POS, GOAL_POS)
    elif algorithm_choice.lower() == 'astar':
        algorithm_name = "A* (A-Star Search)"
        movement_seq, discovered, path = astar(START_POS, GOAL_POS)
    elif algorithm_choice == 'ids':
        algorithm_name = "IDS (Iterative Deepening Search)"
        movement_seq, discovered, path = ids(START_POS, GOAL_POS)
    elif algorithm_choice == 'hill':
        algorithm_name = "Hill Climbing Search"
        movement_seq, discovered, path = hill_climbing(START_POS, GOAL_POS)

    else:
        print(f"Error: Unknown algorithm '{algorithm_choice}'")
        print("Available: 'bfs', 'dfs', 'astar','ids',''ucs,'hill climbing'")
        return False

    if not path:
        print("No path found! But here's what was explored:")
        path_found = False
```

```python
    stats = get_algorithm_stats(discovered, movement_seq, path, viz_time)
    print("\n" + "="*50)
    print("ALGORITHM STATISTICS")
    print("="*50)
    print(f"Algorithm: {algorithm_name}")
    print(f"Explored nodes: {stats['nodes_explored']}")
    print(f"Total movements: {stats['total_movements']}")
    print(f"Final path length: {stats['path_length']}")
    print(f"Visualization time: {viz_time:.2f} seconds")
    print("="*50)
```

THE MAIN MODULE

Program execution prints

```python
def main():

    print("\n" + "="*60)
    print("FIRE ESCAPE ROUTE AGENT - AI PATHFINDING PROJECT")
    print("="*60)
    print(f"Maze Size: {ROWS}x{COLS}")
    print(f"Start Position: {START_POS}")
    print(f"Goal Position: {GOAL_POS}")
    print("="*60)
    print("\nAlgorithms:")
    print("  bfs     - Breadth-First Search")
    print("  dfs     - Depth-First Search")
    print("  astar   - A-Star Search")
    print("  ids     - Itrative Deeping Search")
    print("  ucs     - uniform Cost Search")
    print("  hill    - Hill Climbing Search")
    print("  compare - Compare all algorithms (no visualization)")

    algorithm = input("Enter the algorithm to run (bfs, dfs, ucs, astar, ids, hill, compare): ").lower()
    valid_algorithms = ['bfs', 'dfs', 'ucs', 'astar', 'ids', 'hill', 'compare']
```

```python
    if algorithm not in valid_algorithms:
        print("Invalid algorithm. Exiting.")
        exit()

    if algorithm == 'compare':
        compare_algorithms()
        return

    # Run algorithm
    if not run_algorithm(algorithm,0.15):
        return
```

THE MAIN MODULE

```
PS C:\Users\MG Magic\Documents> python -u "c:\Users\MG Magic\OneDrive\Documents\AI project

============================================================
FIRE ESCAPE ROUTE AGENT - AI PATHFINDING PROJECT
============================================================
Maze Size: 25x30
Start Position: (28, 23)
Goal Position: (1, 1)
============================================================

Algorithms:
  bfs     - Breadth-First Search
  dfs     - Depth-First Search
  astar   - A-Star Search
  ids     - Itrative Deeping Search
  ucs     - uniform Cost Search
  hill    - Hill Climbing Search
  compare - Compare all algorithms (no visualization)
Enter the algorithm to run (bfs, dfs, ucs, astar, ids, hill, compare): ids
```

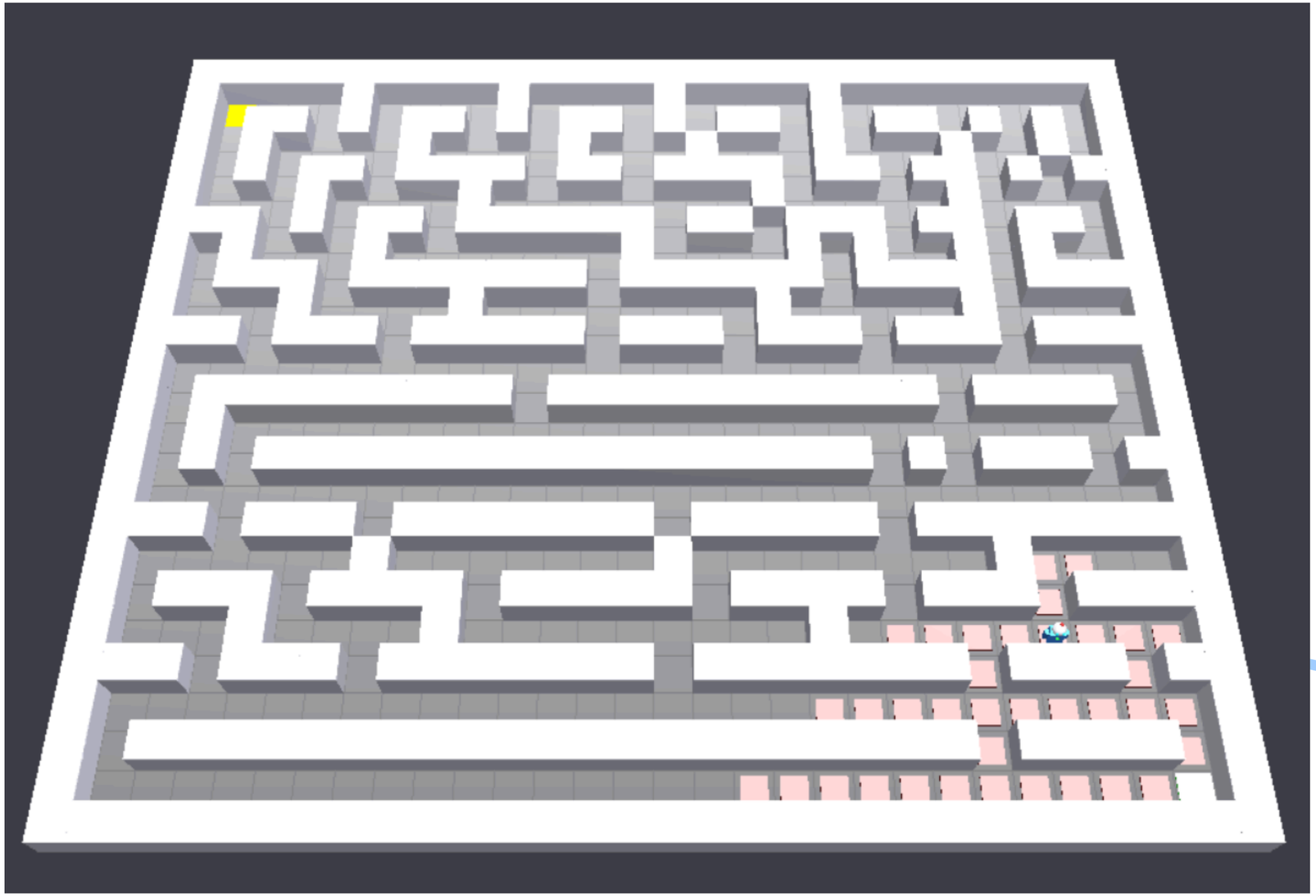
Program output

```
==================================================
ALGORITHM STATISTICS
==================================================
Algorithm: IDS (Iterative Deepening Search)
Explored nodes: 324
Total movements: 18103
Final path length: 74
Visualization time: 2726.55 seconds
==================================================

  Phase 1: Watch agent explore the maze
  Phase 2: Watch agent Follow the optimal path to goal
```

THE MAIN MODULE

3D visualization



Breadth-First Search (BFS)

- Reliable and optimal
- inefficient for emergency evacuation.

```
==================================================
ALGORITHM STATISTICS
==================================================
Algorithm: IDS (Iterative Deepening Search)
Explored nodes: 324
Total movements: 18103
Final path length: 74
Visualization time: 2726.55 seconds
==================================================
```

```python
def bfs(start, goal,):

    queue = deque([start])
    visited = {start}
    parent = {start: None}

    movement_sequence = [start]
    discovered_nodes = {start}
    current_position = start

    while queue:
        current = queue.popleft()

        if current_position != current:
            path_to_current = get_path_between(current_position, current, parent)
            movement_sequence.extend(path_to_current)
            current_position = current


        if current == goal:
            return handle_goal_reached(
                current=current,
                goal=goal,
                parent_map=parent,
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="BFS",
                path_label="Shortest path length"
            )

        for neighbor in get_neighbors(current[0], current[1]):
            if neighbor not in visited:
                visited.add(neighbor)
                discovered_nodes.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)

    return movement_sequence, discovered_nodes, []
```

Depth-First Search(DFS)

Fast and memory efficient, but unsafe due to lack of optimality.

```
ALGORITHM STATISTICS
==========================================
Algorithm: DFS (Depth-First Search)
Explored nodes: 241
Total movements: 328
Final path length: 116
Visualization time: 66.60 seconds
```

```python
def dfs(start, goal):

    stack = [start]
    visited = {start}
    parent = {start: None}

    movement_sequence = [start]
    discovered_nodes = {start}
    current_position = start

    while stack:
        current = stack.pop()

        if current_position != current:
            path = get_path_between(current_position, current, parent)
            movement_sequence.extend(path)
            current_position = current

    if current == goal:
        return handle_goal_reached(
            current=current,
            goal=goal,
            parent_map=parent,
            discovered_nodes=discovered_nodes,
            movement_sequence=movement_sequence,
            algo_name="DFS",
            path_label="Shortest path length"
        )

    for neighbor in reversed(get_neighbors(current[0], current[1])):
        if neighbor not in visited:
            visited.add(neighbor)
            discovered_nodes.add(neighbor)
            parent[neighbor] = current
            stack.append(neighbor)

return movement_sequence, discovered_nodes, []
```

```python
def ids(start, goal):

    movement_sequence = [start]
    discovered_nodes = set()
    current_position = start


    def dls(node, depth_limit, parent, visited):
        nonlocal movement_sequence, discovered_nodes, current_position

        discovered_nodes.add(node)

        if current_position != node:
            path = get_path_between(current_position, node, parent)
            movement_sequence.extend(path)
            current_position = node

        if node == goal:
            return True

        if depth_limit == 0:
            return False

        for neighbor in get_neighbors(node[0], node[1]):
            if neighbor not in visited:
                visited.add(neighbor)
        for neighbor in get_neighbors(node[0], node[1]):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = node
                if dls(neighbor, depth_limit - 1, parent, visited):
                    return True

        return False

    max_depth = ROWS * COLS

    for depth in range(max_depth):
        parent = {start: None}
        visited = {start}
        current_position = start

        if dls(start, depth, parent, visited):
            return handle_goal_reached(
                current=goal,
                goal=goal,
                parent_map=parent,
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="IDS",
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="IDS",
                path_label="Path length"
            )

    return movement_sequence, discovered_nodes, []
```

Iterative Deepening Search (IDS)

- ✅ Reliable and optimal
- ❌ inefficient for emergency evacuation.

```
==================================================
ALGORITHM STATISTICS
==================================================
Algorithm: BFS (Breadth-First Search)
Explored nodes: 333
Total movements: 6582
Final path length: 50
Visualization time: 994.80 seconds
==================================================
```

```python
def ucs(start, goal):

    priority_queue = [(0, start)]
    visited = set()
    parent = {start: None}
    g_cost = {start: 0}

    movement_sequence = [start]
    discovered_nodes = {start}
    current_position = start

    while priority_queue:
        cost, current = heapq.heappop(priority_queue)

        if current in visited:
            continue

        visited.add(current)

        if current == goal:
            return handle_goal_reached(
                current=current,
                goal=goal,
                parent_map=parent,
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="UCS",
                path_label="Path to goal"
            )

        for neighbor in get_neighbors(current[0], current[1]):
            step_cost = node_costs[neighbor]
            new_cost = g_cost[current] + step_cost
            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                parent[neighbor] = current
                heapq.heappush(priority_queue, (new_cost, neighbor))
                discovered_nodes.add(neighbor)

    return movement_sequence, discovered_nodes, []
```

Uniform-Cost Search(UCS)

Optimal but impractical for real-time fire escape scenarios.

```
============================================
ALGORITHM STATISTICS
============================================
Algorithm: UCS (Uniform Cost Search)
Explored nodes: 326
Total movements: 8814
Final path length: 50
Visualization time: 1329.60 seconds
============================================
```

A* Search

Best balance between optimality and guided search, most suitable algorithm for this problem.

```
============================================================
ALGORITHM STATISTICS
============================================================
Algorithm: A* (A-Star Search)
Explored nodes: 322
Total movements: 9014
Final path length: 50
Visualization time: 1359.60 seconds
============================================================
```

```python
def astar(start, goal):

    priority_queue = []
    priority_queue = [(0, start)]

    parent = {start: None}
    g_cost = {start: 0}

    visited = set()
    discovered_nodes = {start}
    movement_sequence = [start]
    current_position = start

    while priority_queue:
        f_cost, current = heapq.heappop(priority_queue)

        if current in visited:
            continue
        visited.add(current)

        if current_position != current:
            path = get_path_between(current_position, current, parent)
            movement_sequence.extend(path)
            current_position = current

        if current == goal:
            return handle_goal_reached(
                current=current,
                goal=goal,
                parent_map=parent,
                discovered_nodes=discovered_nodes,
                movement_sequence=movement_sequence,
                algo_name="A*",
                path_label="Optimal path length"
            )

        for neighbor in get_neighbors(current[0], current[1]):
            step_cost = node_costs[neighbor]
            new_cost = g_cost[current] + step_cost

            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                parent[neighbor] = current

                h = heuristic(neighbor, goal)
                f = new_cost + h

                heapq.heappush(priority_queue, (f, neighbor))
                discovered_nodes.add(neighbor)
    return movement_sequence, discovered_nodes, []
```

```python
ef hill_climbing(start, goal):

    current = start
    movement_sequence = [current]
    discovered_nodes = {current}
    parent = {current: None}

    while current != goal:
        neighbors = get_neighbors(current[0], current[1])
        neighbors = [n for n in neighbors if n not in discovered_nodes]
        if not neighbors:
            break
        next_node = min(neighbors, key=lambda n: heuristic(n, goal))
        parent[next_node] = current
        discovered_nodes.add(next_node)
        movement_sequence.append(next_node)
        current = next_node

    if current == goal:
        return handle_goal_reached(
            current=current,
            goal=goal,
            parent_map=parent,
            discovered_nodes=discovered_nodes,
            movement_sequence=movement_sequence,
            algo_name="Hill Climbing",
            path_label="Path length"
        )
    return movement_sequence, discovered_nodes, []
```

Hill Climbing

Fast but unreliable, unsafe for evacuation planning.

```
ALGORITHM STATISTICS
==========
Algorithm: Hill Climbing Search
Explored nodes: 56
Total movements: 56
Final path length: 0
Visualization time: 8.40 seconds
```

ALGORITHMS COMPARISON

Metric	BFS	DFS	UCS	A*	IDS	Hill
Discovered nodes	333	241	326	322	324	56
Moves	6582	328	8814	9014	18103	56
Finel path	50	116	50	50	74	0
Viz Time (s)	994.80	66.60	1329.60	1359.60	2726.55	8.40

Final Comparison Conclusion

Overall Comparison and Best Algorithm Selection

Based on the experimental results:

· Optimal and successful algorithms: BFS, UCS, A*, IDS

· Fast but non-optimal: DFS

· Fast but failed: Hill Climbing

Among all algorithms, A* Search provides the best balance between optimality and intelligent guidance, making it the most suitable choice for a fire escape route

THANK YOU