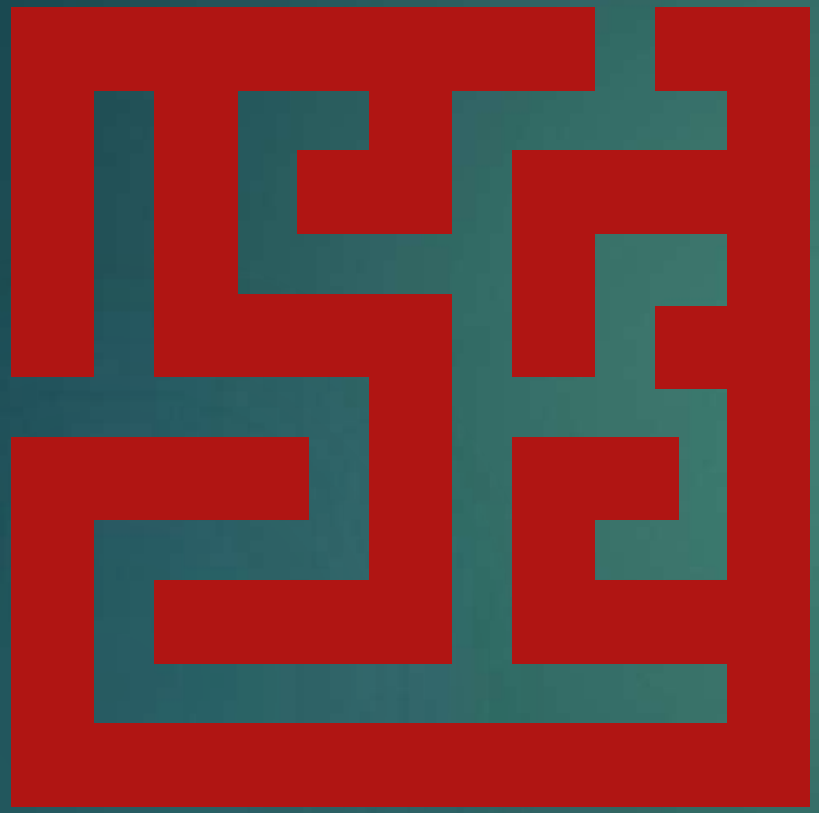
GALALA UNIVERSITY
Powered by
Arizona State University



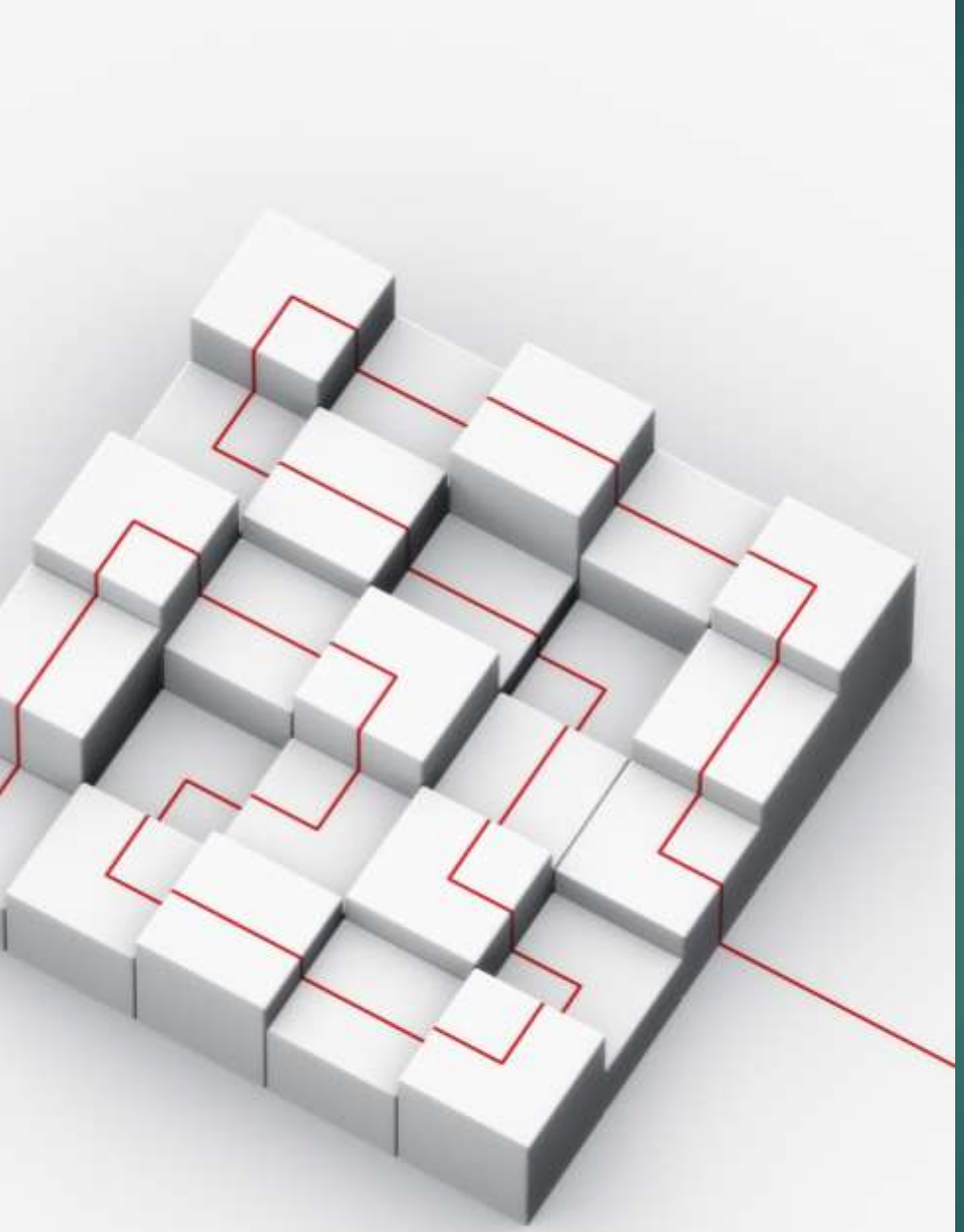
Maze Generator & Solver with Tkinter

AN ALGORITHMIC EXPLORATION USING DFS, BFS, A*, DIJKSTRA, AND GREEDY BFS

Team members

- Khadija Osama 223102197
- Youssef ElDasoky 223101109
- Aser Mohamed 222102487
- Mahmoud Metwally 223103852
- Ziad Mohamed A20000395

Under Supervision: Dr. Mohamed Ghetas



Project Objective

- To design and implement a graphical maze generator and solver using different pathfinding algorithms.
- Provides a visual comparison of efficiency, path optimality, and performance.

Tools and Technologies

LIBRARIES: TKINTER (GUI), HEAPQ (PRIORITY QUEUE), RANDOM (MAZE GENERATION)

ALGORITHMS IMPLEMENTED:

DEPTH-FIRST SEARCH (DFS)

BREADTH-FIRST SEARCH (BFS)

DIJKSTRA'S ALGORITHM

A* SEARCH

GREEDY BEST-FIRST SEARCH

Maze Generation (DFS-Based Backtracking)

- Maze is generated using randomized depth-first search (recursive backtracker).
- Each cell has walls; we remove walls to create paths.

Pathfinding Algorithms Overview

DFS: Deep but not optimal.

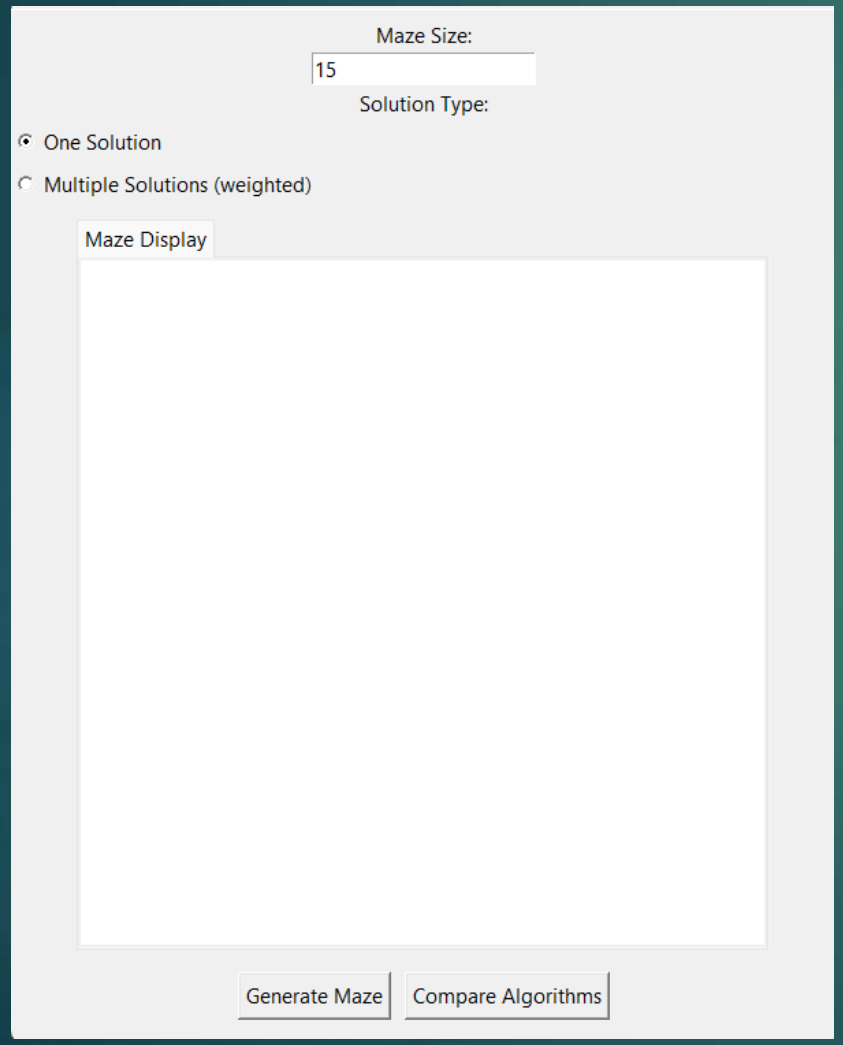
BFS: Shallow, finds shortest path in unweighted graphs.

Dijkstra: Optimal for weighted graphs.

A*: Optimal with heuristic (uses cost + estimate).

Greedy BFS: Fast, not always optimal (uses heuristic only).

Maze Display



```
def bfs(start, goal):
    queue = deque([start])
    visited = set()
    parent = {start: None}

    while queue:
        current = queue.popleft()
        if current == goal:
            break
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)
    return reconstruct_path(parent, start, goal)
```

Breadth-First Search (BFS)

Breadth-First Search (BFS)

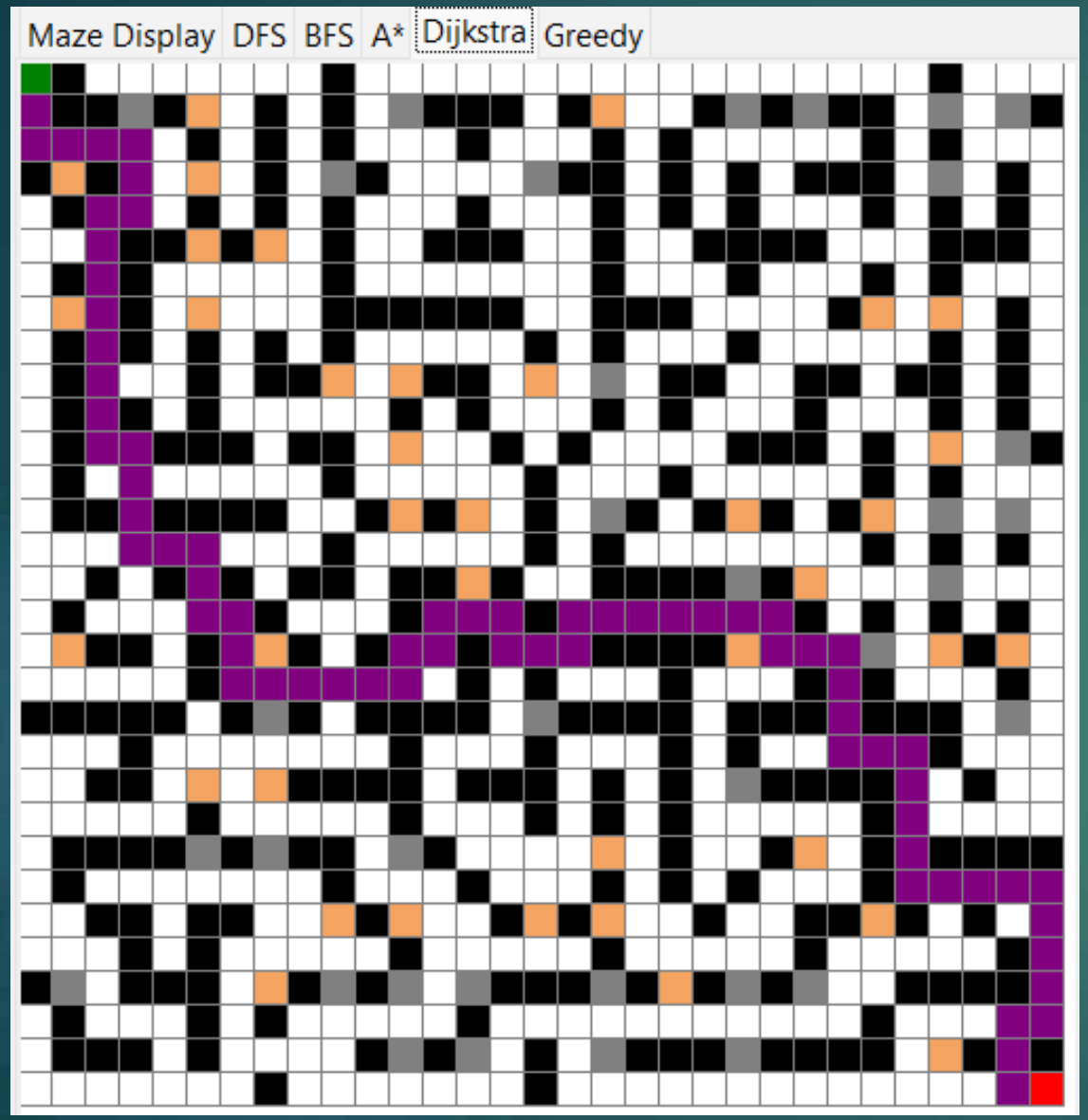
```
def dfs(start, goal):
    stack = [start]
    visited = set()
    parent = {start: None}

    while stack:
        current = stack.pop()
        if current == goal:

            break
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                stack.append(neighbor)
    return reconstruct_path(parent, start, goal)
```

Depth-First Search (DFS)

Depth-First Search (DFS)

Dijkstra's Algorithm

```
def dijkstra(start, goal):
    pq = [(0, start)]
    visited = set()
    dist = {start: 0}
    parent = {start: None}

    while pq:
        cost, current = heapq.heappop(pq)
        if current == goal:
            break
        for neighbor in get_neighbors(current):
            new_cost = cost + 1
            if neighbor not in dist or new_cost < dist[neighbor]:
                dist[neighbor] = new_cost
                heapq.heappush(pq, (new_cost, neighbor))
                parent[neighbor] = current
    return reconstruct_path(parent, start, goal)
```



Dijkstra's Algorithm

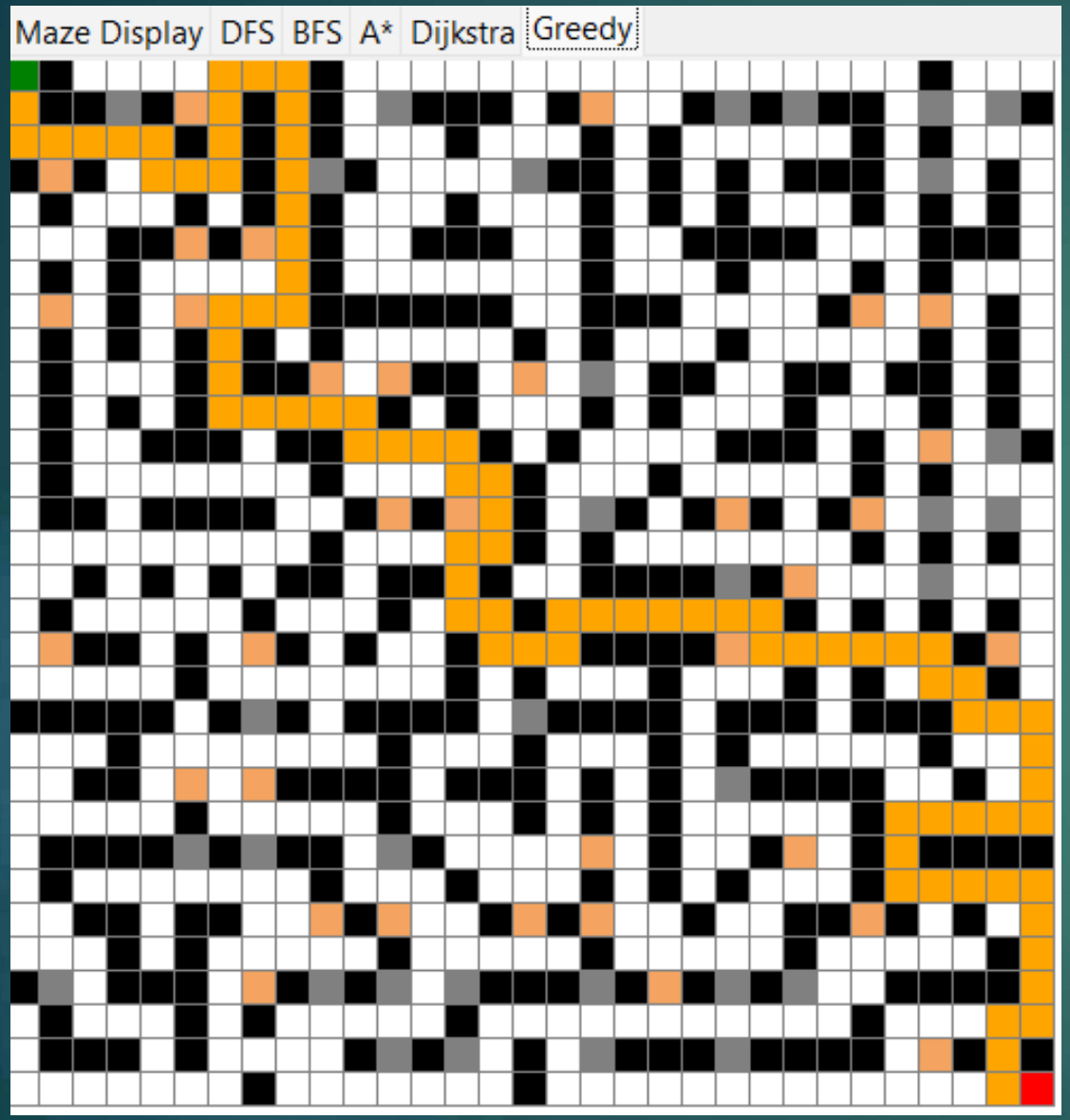
A* Algorithm

```python
def a_star(start, goal):
    pq = [(0, start)]
    g_cost = {start: 0}
    parent = {start: None}

    while pq:
        _, current = heapq.heappop(pq)
        if current == goal:
            break
        for neighbor in get_neighbors(current):
            tentative_g = g_cost[current] + 1
            if neighbor not in g_cost or tentative_g < g_cost[neighbor]:
                g_cost[neighbor] = tentative_g
                f = tentative_g + heuristic(neighbor, goal)
                heapq.heappush(pq, (f, neighbor))
                parent[neighbor] = current
    return reconstruct_path(parent, start, goal)
```

A*
Algorithm

Greedy Best-First Search

```
def greedy_bfs(start, goal):
    pq = [(heuristic(start, goal), start)]
    visited = set()
    parent = {start: None}

    while pq:
        _, current = heapq.heappop(pq)
        if current == goal:
            break
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                heapq.heappush(pq, (heuristic(neighbor, goal), neighbor))
                parent[neighbor] = current
    return reconstruct_path(parent, start, goal)
```



Greedy Best-First Search

Comparison Screenshot from GUI

```
DFS: Path length = 299
BFS: Path length = 67
A*: Path length = 71
Dijkstra: Path length = 71
Greedy: Path length = 85
```

Comparison & Results

Algorithm	Shortest Path?	Speed	Uses Heuristic?
DFS	NO	Fast	NO
BFS	YES	Medium	NO
Dijkstra	YES	Slow	NO
A*	YES	Fast	YES
Greedy Best-First	NO	Fast	YES

Conclusion

- Maze-solving algorithms provide a hands-on way to understand search, graph traversal, and heuristics.
- A* remains the best overall performer for optimal and fast search.

Thank You