**Maze Solver: A Comparative Study of Pathfinding Algorithms for Maze Solving**

Sudarshan N Sankaran  
 *22BCE1045*  
sudarshan.nsankaran2022@vitstudent.ac.in

**Abstract:**

**A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. The user starts from source and has to reach the destination and can move only in two directions: forward and down. In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. For example, a more complex version can be that the user can move in 4 directions and a more complex version can be with a limited number of moves. The project generates random mazes, and compares algorithm performance in terms of execution time, path length, and memory usage.**

**This project addresses the need for a versatile maze-solving tool that enables users, including students, educators, and developers, to efficiently navigate mazes while comparing the performance of different pathfinding algorithms. It involves efficiently solving mazes with defined Start and Exit points while comparing the performance of multiple pathfinding algorithms.**

Keywords: Pathfinding, Mazes, Algorithms, Comparison, Navigation

**Introduction:**

Maze-solving algorithms form the backbone of navigation systems, robotics, gaming, and various other fields where efficient pathfinding is crucial. These algorithms aim to find the shortest or most optimal path from a starting point to a destination within a maze, which can be represented as a grid or a network of interconnected nodes. The complexity of mazes varies from simple, grid-like structures to intricate, multi-dimensional configurations, posing challenges for traversal. For a digital mind that calculates its path in real time, that calculation also needs to finish within a reasonable amount of time, otherwise the risk of collision increases drastically. Pathfinding algorithms play a pivotal role in efficiently exploring and navigating these mazes, enabling applications in autonomous vehicles, video games, and even logistical optimization.

Various pathfinding algorithms have been developed, each with its unique approach and trade-offs. For instance, classical algorithms like Depth-First Search (DFS) and Breadth-First Search (BFS) provide fundamental methodologies for traversing mazes but may not always guarantee the shortest path. On the other hand, more sophisticated algorithms such as Dijkstra's Algorithm and the A\* Algorithm incorporate heuristics and weighted approaches, offering optimality and efficiency in finding the shortest path but requiring more computational resources. Understanding the strengths and limitations of these algorithms is crucial in selecting the most suitable method for maze-solving in different contexts, considering factors such as maze complexity, computational efficiency, and real-time constraints in practical applications.

**Literature Survey:**

Here are a few research papers that delve into the topic and provide valuable insights into the state of the art and recent advancements in the field:

* "A\* Search Algorithm" by Peter Hart, Nils Nilsson, and Bertram Raphael.

This seminal paper introduces the A\* algorithm, a widely used pathfinding algorithm applicable to maze solving.

Paper Link: https://www.cs.auckland.ac.nz/courses/compsci709s2c/lectures/H120\_20.pdf

* "A Survey of Path Planning and Solving Techniques for Robotic Mazes" by Marios K. Papachristou

This survey paper provides an overview of various path planning and maze-solving techniques used in robotics.

Paper Link: <https://www.researchgate.net/publication/225750225_A_survey_of_path_planning_and_solving_techniques_for_robotic_mazes>

* "Solving mazes with parallel processes" by A. C. Yao

This paper explores the use of parallel processing for solving mazes efficiently.

Paper Link: https://dl.acm.org/doi/10.1145/800119.803890

* "Robot Navigation in Unknown Terrains by Virtual Sensing" by Avinash Kak

This paper discusses the use of virtual sensing and path planning for maze solving.

Paper Link: https://www.researchgate.net/publication/2426983\_Robot\_Navigation\_in\_Unknown\_Terrains\_by\_Virtual\_Sensing

* "A New Approach to Path Planning for Redundant Robots" by Mark A. Minor and Robert L. Wainwright

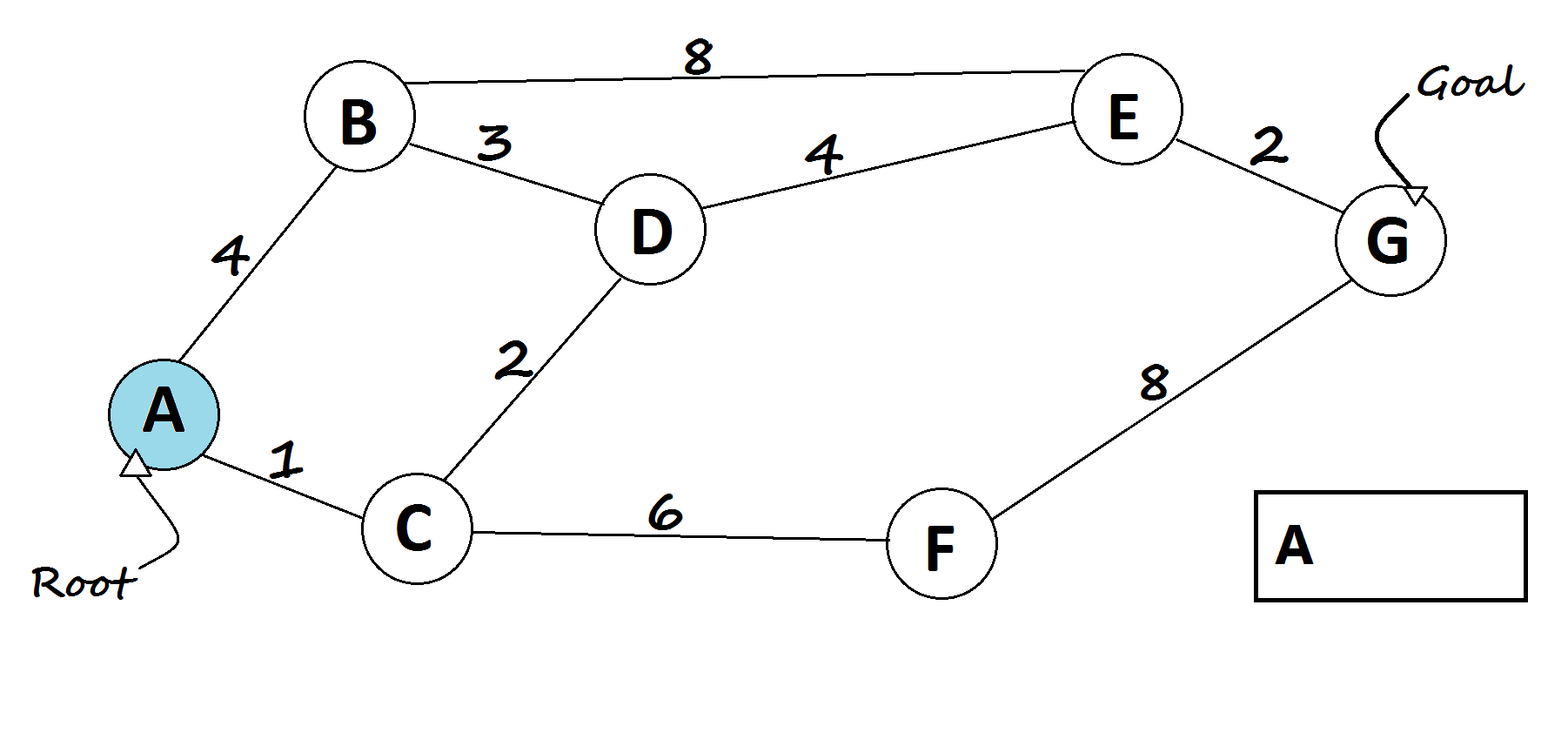
This paper discusses path planning for redundant robots, which can be relevant for maze-solving scenarios.

Paper Link: <https://journals.sagepub.com/doi/abs/10.1177/027836498800700205>

**Proposed Architecture:**

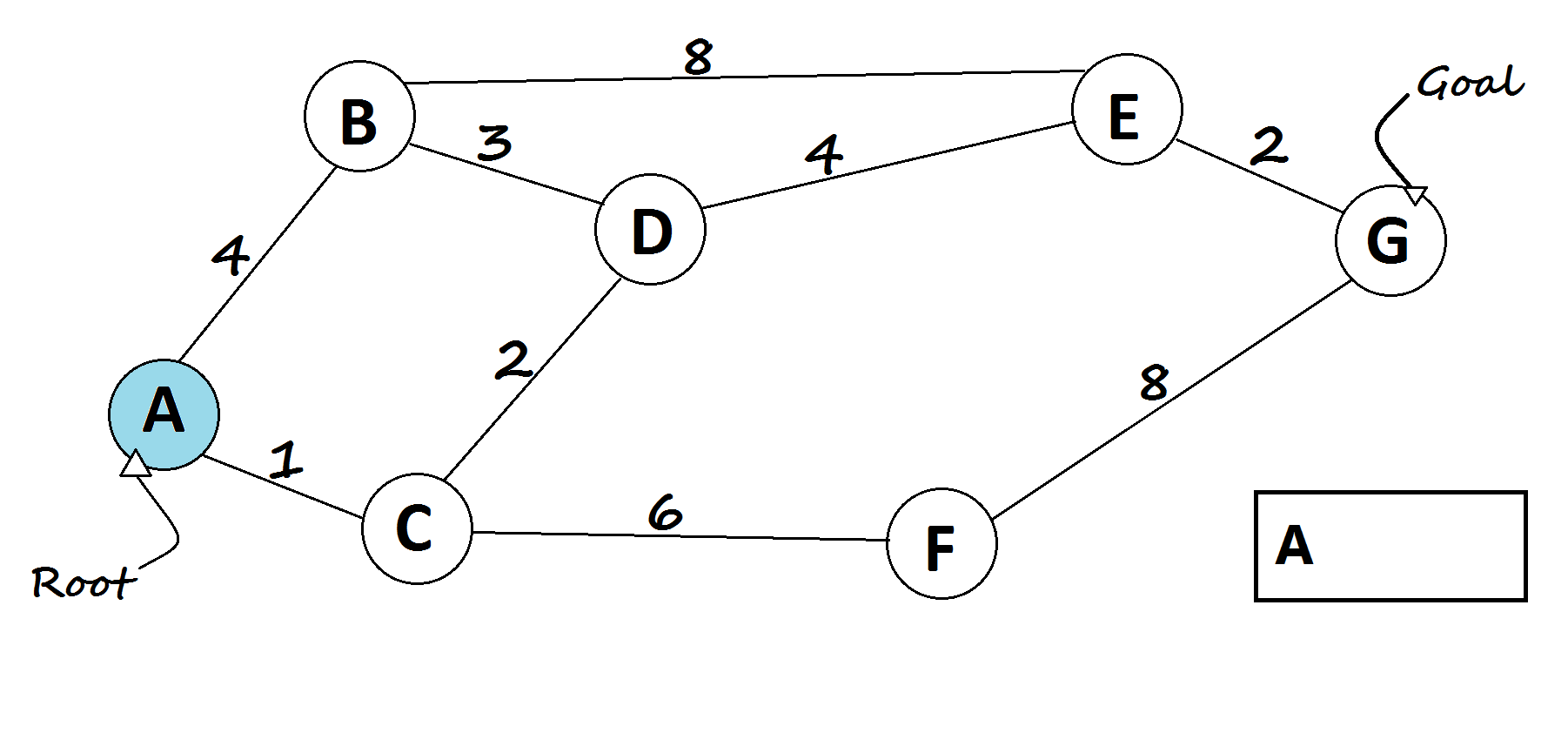
* **Breadth First Search (BFS)**

Breadth-First Search is an essential graph traversal algorithm used to explore and search for the shortest path in unweighted graphs or trees. It starts at the root node and explores all neighbor nodes at the current depth level before moving on to deeper levels. BFS is known for its **simplicity** and guarantee of finding the shortest path in unweighted graphs. It is commonly used in applications where finding the shortest path is essential, such as network routing and maze solving.



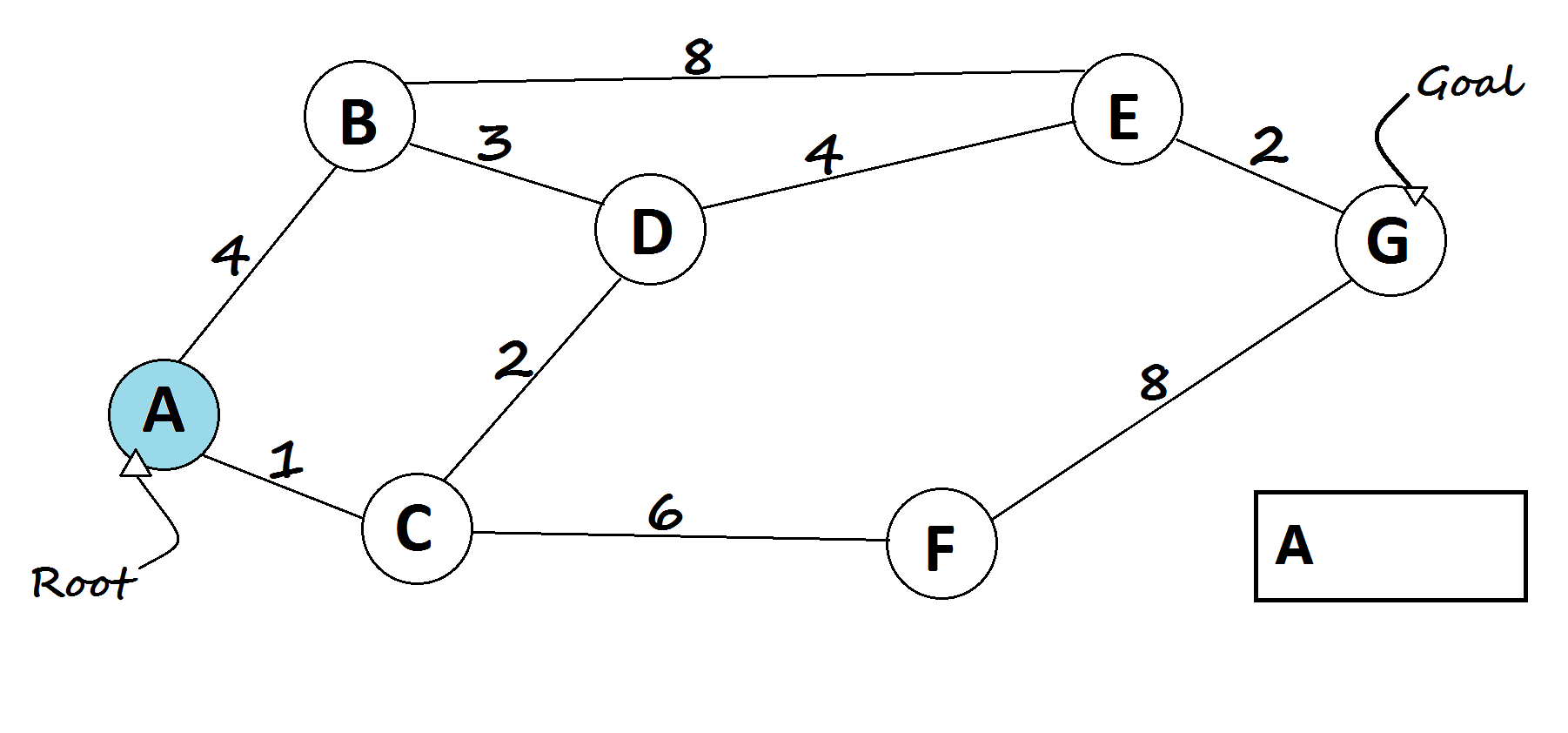
* **Depth First Search (DFS)**

Depth-First Search is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It is used to traverse and search **tree** and **graph** structures. DFS is **memory-efficient** as it uses a stack to keep track of visited nodes. DFS is more efficient than BFS for large graphs, but it can be less complete. It is well-suited for tasks like topological sorting, cycle detection, and solving puzzles, but it does not guarantee finding the shortest path, making it less suitable for certain applications like maze solving.



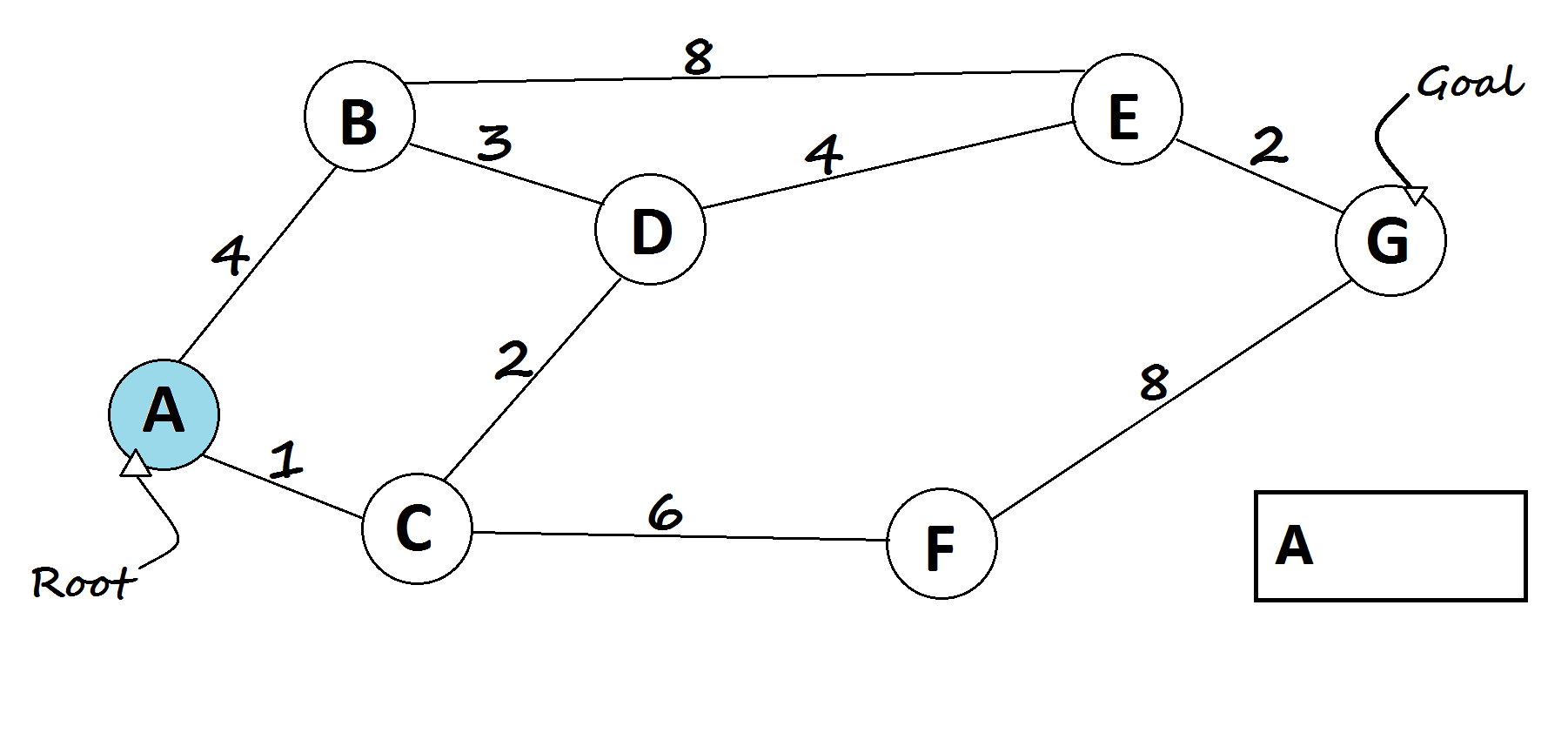
* **Dijkstra’s Algorithm**

Dijkstra's Algorithm is a weighted graph traversal algorithm used for finding the shortest path from a single **source node** to all other nodes in a graph with non-negative edge weights. It maintains a priority queue to continuously select the node with the **smallest tentative distance** from the source and updates the distances as it explores the graph. In each iteration, Dijkstra's algorithm removes the node with the smallest distance from the unvisited set and adds it to the visited set. It then updates the distances of all of the neighbors of the removed node. It is primarily suitable for applications like network routing and GPS navigation.



* **A\* (A-star)**

The A\* Algorithm is a complex heuristic-based search algorithm used for finding the shortest path in graphs. It combines the advantages of both BFS and Dijkstra's Algorithm by considering both the actual cost from the start node and an estimated cost to reach the goal node. This allows A\* search to focus its exploration on the most promising paths, leading to a shorter path to the goal node. A\* intelligently selects nodes to explore based on a priority queue, optimizing efficiency. It is widely used in applications like video games, robotics, and route planning, where efficient pathfinding is crucial.



**Methodology:**

1. **Algorithm Selection**

Identification of Algorithms:

* Chosen algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), A\*, and Dijkstra's algorithm.
* Basis of selection: Diverse exploration techniques and computational characteristics.

2. **Experimental Setup**

Maze Configuration:

* Structure: Utilized a static 10x10 grid maze with predefined walls and open paths.
* Initialization: Defined fixed start and end positions within the maze for consistent testing.

3. **Performance Metrics**

Evaluation Criteria:

- Metrics: Efficiency, Optimality, and Space Complexity.

- Efficiency Measurement: Recorded computational time taken by each algorithm for maze traversal.

- Optimality Assessment: Compared found path lengths against the shortest possible paths.

- Space Complexity Analysis: Documented memory usage and explored node count during execution.

4. **Execution and Data Collection**

Experiment Execution:

- Individual Runs: Executed algorithms separately on the standardized maze, recording relevant data.

- Recorded Metrics: Captured path lengths, execution times, nodes visited, and memory usage.

- Consistency Measures: Maintained uniform experimental settings for consistent measurements.

5. **Comparative Analysis**

Performance Evaluation:

- Metric Comparison: Evaluated BFS, DFS, A\*, and Dijkstra's algorithm performances against predefined metrics.

- Analysis Scope: Examined results to determine superior algorithms based on efficiency, optimality, and space complexity.

- Trade-off Discussion: Explored trade-offs between computational efficiency, path optimality, and algorithmic complexities.

**Code:**

**Breadth First Search:**

def bfs(maze, start, end):

rows = len(maze)

cols = len(maze[0])

visited = [[False] \* cols for \_ in range(rows)]

queue = deque([(start, [])])

visited\_count = 0

while queue:

current, path = queue.popleft()

x, y = current

if current == end:

return path + [current], visited\_count

if visited[x][y]:

continue

visited[x][y] = True

visited\_count += 1

for dx, dy in movements:

new\_x = x + dx

new\_y = y + dy

if 0 <= new\_x < rows and 0 <= new\_y < cols and not visited[new\_x][new\_y] and maze[new\_x][new\_y] == 0:

queue.append(((new\_x, new\_y), path + [current]))

return None, visited\_count

**Depth First Search:**

def dfs(maze, start, end):

rows = len(maze)

cols = len(maze[0])

visited = [[False] \* cols for \_ in range(rows)]

stack = [(start, [])]

visited\_count = 0

while stack:

current, path = stack.pop()

x, y = current

if current == end:

return path + [current], visited\_count

if visited[x][y]:

continue

visited[x][y] = True

visited\_count += 1

for dx, dy in movements:

new\_x = x + dx

new\_y = y + dy

if 0 <= new\_x < rows and 0 <= new\_y < cols and not visited[new\_x][new\_y] and maze[new\_x][new\_y] == 0:

stack.append(((new\_x, new\_y), path + [current]))

return None, visited\_count

**Dijkstra:**

def dijkstra(maze, start, end):

rows = len(maze)

cols = len(maze[0])

visited = [[False] \* cols for \_ in range(rows)]

priority\_queue = []

heapq.heappush(priority\_queue, (0, start))

cost\_so\_far = {start: 0}

came\_from = {}

visited\_count = 0

while priority\_queue:

\_, current = heapq.heappop(priority\_queue)

x, y = current

if current == end:

path = [current]

while current != start:

current = came\_from[current]

path.append(current)

path.reverse()

return path, visited\_count

if visited[x][y]:

continue

visited[x][y] = True

visited\_count += 1

for dx, dy in movements:

new\_x = x + dx

new\_y = y + dy

if 0 <= new\_x < rows and 0 <= new\_y < cols and not visited[new\_x][new\_y] and maze[new\_x][new\_y] == 0:

new\_cost = cost\_so\_far[current] + 1

if (new\_x, new\_y) not in cost\_so\_far or new\_cost < cost\_so\_far[(new\_x, new\_y)]:

cost\_so\_far[(new\_x, new\_y)] = new\_cost

heapq.heappush(priority\_queue, (new\_cost, (new\_x, new\_y)))

came\_from[(new\_x, new\_y)] = current

return None, visited\_count

**A Star(A\*):**

def heuristic(position, end):

x1, y1 = position

x2, y2 = end

return abs(x1 - x2) + abs(y1 - y2)

def a\_star(maze, start, end):

rows = len(maze)

cols = len(maze[0])

visited = [[False] \* cols for \_ in range(rows)]

priority\_queue = []

heapq.heappush(priority\_queue, (0, start))

cost\_so\_far = {start: 0}

came\_from = {}

visited\_count = 0

while priority\_queue:

\_, current = heapq.heappop(priority\_queue)

x, y = current

if current == end:

path = [current]

while current != start:

current = came\_from[current]

path.append(current)

path.reverse()

return path, visited\_count

if visited[x][y]:

continue

visited[x][y] = True

visited\_count += 1

for dx, dy in movements:

new\_x = x + dx

new\_y = y + dy

if 0 <= new\_x < rows and 0 <= new\_y < cols and not visited[new\_x][new\_y] and maze[new\_x][new\_y] == 0:

new\_cost = cost\_so\_far[current] + 1

if (new\_x, new\_y) not in cost\_so\_far or new\_cost < cost\_so\_far[(new\_x, new\_y)]:

cost\_so\_far[(new\_x, new\_y)] = new\_cost

heapq.heappush(priority\_queue, (new\_cost + heuristic((new\_x, new\_y), end), (new\_x, new\_y)))

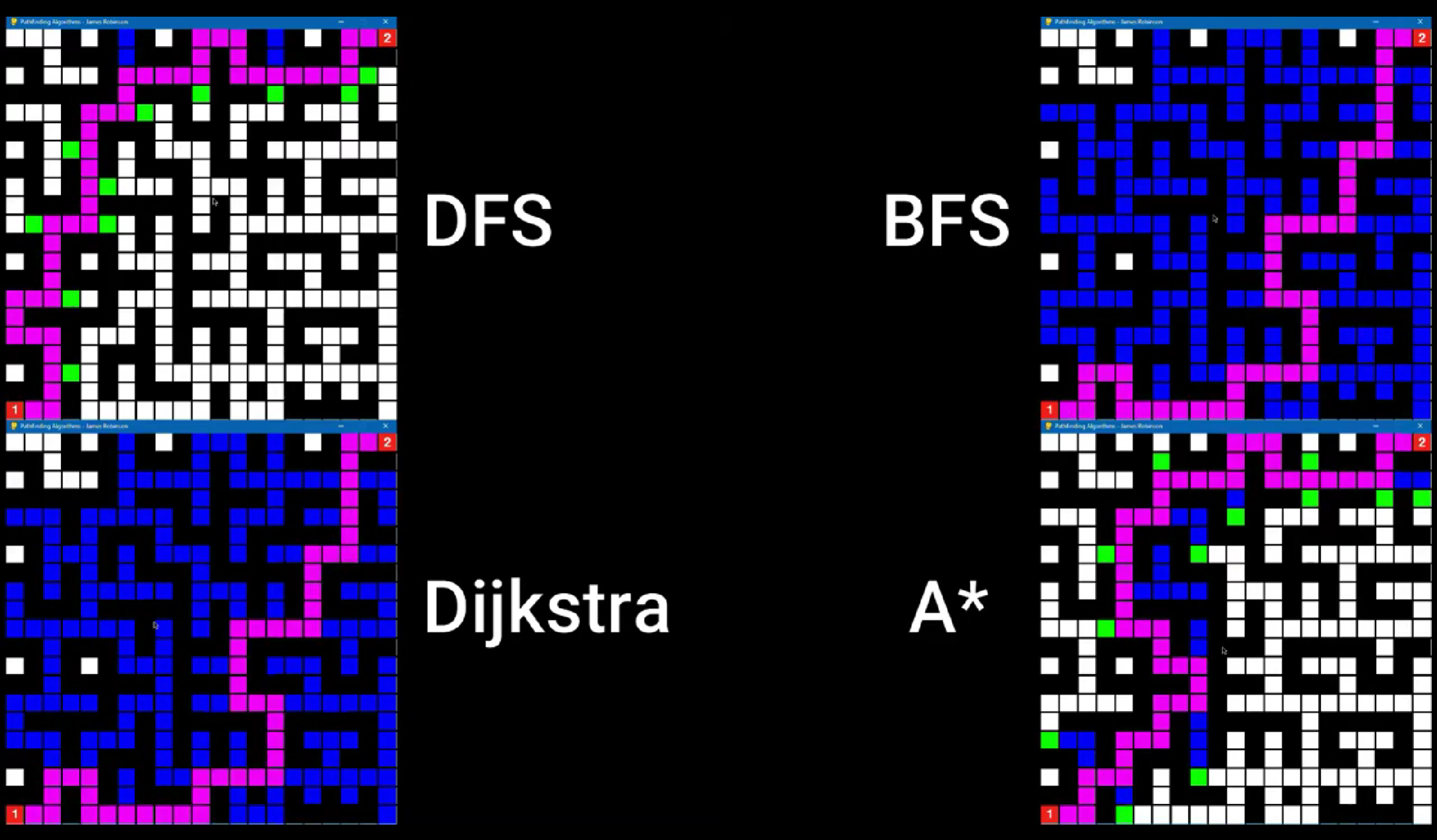
came\_from[(new\_x, new\_y)] = current

return None, visited\_count

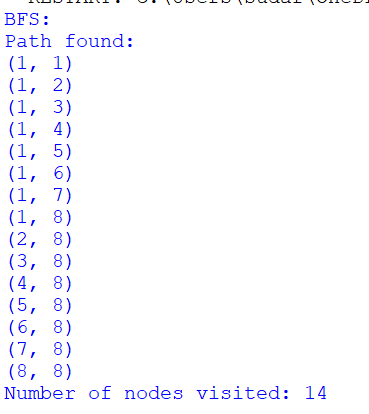
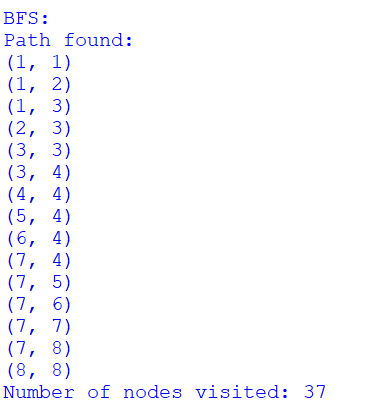
**Analysis:**

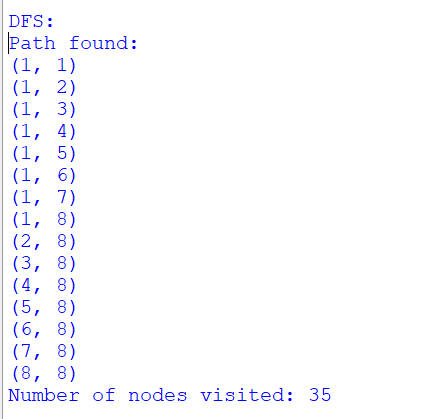
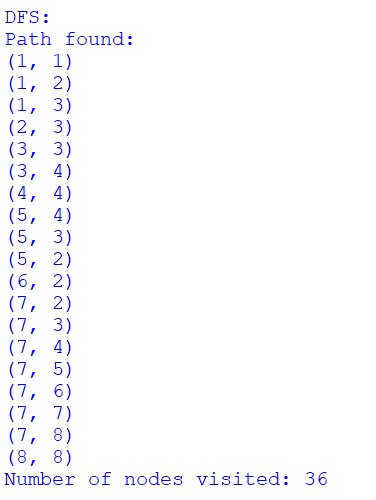
The comparative study of maze-solving algorithms revealed distinct performance characteristics across efficiency, optimality, and space utilization metrics. Algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS) showcased simplicity in their approaches, with BFS displaying consistent performance but longer traversal times due to its breadth-first exploration. DFS, while occasionally faster, exhibited deviations from optimality, resulting in longer paths within the maze. Contrarily, A\* and Dijkstra's algorithms consistently found optimal paths, leveraging heuristic-guided approaches, and demonstrated faster traversal times. However, their reliance on heuristics led to increased memory usage, influencing their space complexity compared to BFS and DFS.

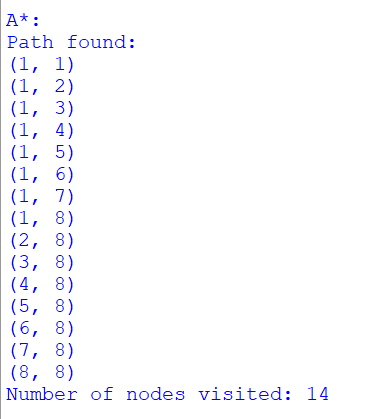
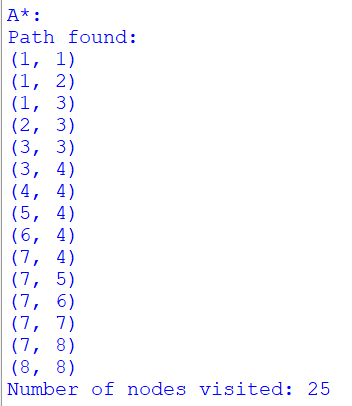
These findings underscore the trade-offs inherent in maze-solving algorithms: a balance between computational efficiency, path optimality, and memory consumption. A\* and Dijkstra's algorithms prioritize optimality but demand higher memory resources, making them well-suited for scenarios requiring precise pathfinding. Meanwhile, BFS and DFS, despite occasional suboptimal paths, offer simpler implementations and lower memory usage, potentially fitting applications prioritizing computational speed and efficiency over absolute optimality. The study elucidates the distinct strengths and trade-offs among these algorithms, providing insights into their applicability across various maze-solving contexts.

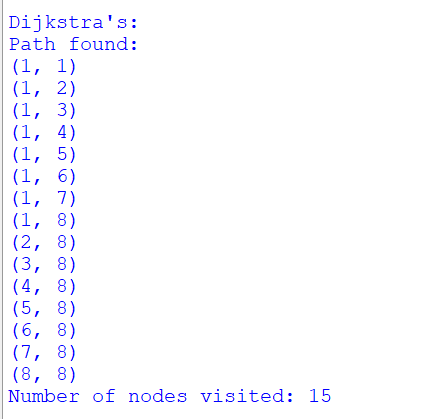
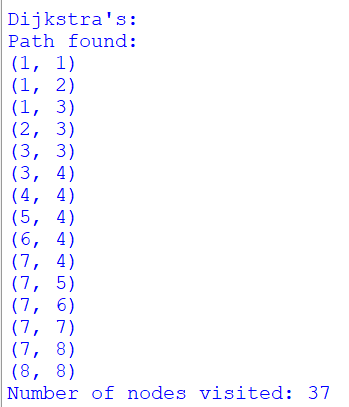


**Experimental Results:**

1. **Efficiency Analysis:**
   * BFS exhibited consistent performance but tended to take longer due to its broad exploration approach.
   * DFS, while faster in certain scenarios, occasionally produced longer paths due to its depth-first exploration.
   * A\* and Dijkstra's consistently showed faster traversal times, leveraging heuristic-guided approaches for efficient maze-solving.
2. **Optimality Evaluation:**
   * BFS and DFS paths occasionally deviated from optimality, resulting in longer paths due to their lack of optimality guarantees.
   * A\* and Dijkstra's consistently found optimal paths, demonstrating reliability in finding the shortest paths within the maze.
3. **Space Complexity Assessment:**
   * BFS and DFS were less memory-intensive but tended to explore a higher number of nodes during traversal.
   * A\* and Dijkstra's, though more memory-demanding, showcased reduced node visits, optimizing traversal efficiency.
4. **Comparative Analysis:**
   * A\* and Dijkstra's exhibited a balance between efficiency and optimality but at the expense of higher memory usage.
   * BFS and DFS, while simpler and less memory-intensive, occasionally sacrificed optimality for faster exploration.

**Conclusion:**

In conclusion, the comparative study of maze-solving algorithms unveiled multifaceted performance dynamics, providing a holistic perspective on their efficacy in navigating complex maze structures. The evaluation, encompassing Breadth-First Search (BFS), Depth-First Search (DFS), A\*, and Dijkstra's algorithms, delineated their distinct operational paradigms and trade-offs. BFS and DFS, despite their simplicity, demonstrated varied strengths and weaknesses, with BFS ensuring consistency in traversal but compromising speed, while DFS occasionally outpaced others but at the expense of path optimality. Conversely, A\* and Dijkstra's algorithms consistently excelled in finding optimal paths, leveraging heuristic-guided approaches, albeit with augmented memory requirements.

These findings underscore the critical interplay between computational efficiency, path optimality, and resource utilization in maze-solving contexts. A\* and Dijkstra's algorithms prioritize path optimality, positioning them as optimal choices for scenarios mandating precise route determination, such as navigation systems and robotics. Their heuristic-guided methodologies prove invaluable in finding shortest paths within mazes, albeit at the cost of higher memory consumption. On the other hand, BFS and DFS, while simpler and more memory-efficient, may occasionally sacrifice path optimality but present viable options for applications emphasizing computational speed over absolute optimality, such as gaming environments or basic pathfinding systems.

In the ever-evolving landscape of algorithmic applications, understanding the nuanced performance nuances and trade-offs among maze-solving algorithms is pivotal. This study sheds light on the diverse strengths and limitations of each algorithm, offering valuable insights into their applicability across a spectrum of maze-solving scenarios. Ultimately, the choice of algorithm hinges on the specific requirements of the application, considering the delicate balance between computational efficiency, path optimality, and resource constraints. The findings of this study serve as a compass for navigating algorithm selection in maze-solving contexts, providing a comprehensive framework for informed decision-making in diverse real-world applications.