A Comparative Study of Solving Maze

The path-finding algorithms are:

Breadth First Search, A* Search and Greedy_BSF

Krishna Gopal Sharma

Student Number: S3454618

CIS4049

Foundation of Artificial Intelligence

Total Words: 3793

Module Leader: Di Stefano, Alessandro

Email: A.DiStefano@tees.ac.uk

Lab Tutor: Ghareeb, Shatha

Email: s.ghareeb@tees.ac.uk

Table of Contents

1. Abstract

This project investigates the performance of 3 search algorithms, **Breadth-First Search (BFS)**, **A* Algorithm, and Greedy Best-First Search (Greedy BFS)** in solving a **maze of 50 x 120 size.** This maze is created by python script. with **loopPercent = 48%.** [1]

On same maze, the algorithms were tested under 3 scenarios, with **goal positions at (1, 1), (49, 2), and (1, 119).** I am evaluating the effectiveness of these search algorithms based on path length, exploration order, and computational efficiency. The findings provide insight into the advantages and disadvantages of each algorithm under different goal positions. Results of this project shows that BFS consistently finds the shortest path but at the cost of high exploration value, while Greedy_BFS gives faster exploration but less optimal paths. A* balances both, providing optimal solutions with efficient exploration. The study of these algorithms Let's get started.[2]

---

2. Introduction

Pathfinding and search algorithms are fundamental to Artificial Intelligence (AI), with applications in robotics, gaming, Geographical Information System (GIS), optimization problems and many more. This project focuses on implementing and comparing these below search algorithms:

1. **Breadth-First Search (BFS):** An uninformed search algorithm exploring all neighbours at the current depth level before moving deeper.
2. ***A* Algorithm:*** An informed search combining cost from start to node (g(n)) and a heuristic estimate to the goal (h(n)).
3. **Greedy Best-First Search (Greedy BFS):** An informed search prioritizing exploration based solely on heuristic estimates (h(n)).

By implementing and analysing these algorithms in the given maze to solve it by finding the optimal path from starting position to the goal, and by analysis of these algorithm's performance across three different goal positions, this report briefs their methodology, implementation and results.

---

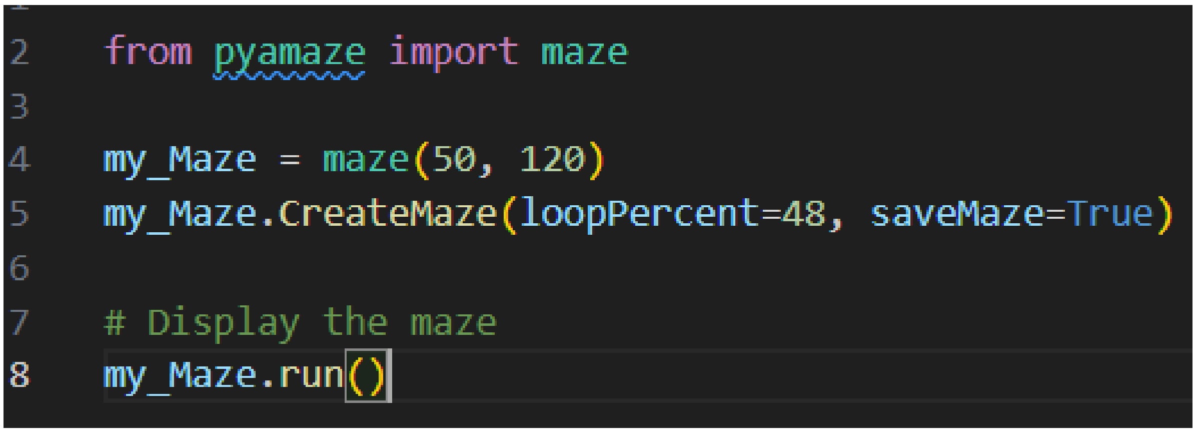
3. Methodology

3.1 Problem Setup

- The maze dimensions, structure and obstacles were predefined and loaded from a CSV file.
  - Dimensions: 50 x 120.
  - Structure: Cell (x, y)
    Direction: E, W, N, S
  - Design: Cell (1, 1) represent cell number or cell position and (0 and 1) represent the presence of wall or not in given direction i.e. E, W, N and S.

cell	E	W	N	S
(1, 1)	1	0	0	1
(2, 1)	1	0	1	0
(3, 1)	1	0	0	1
(4, 1)	0	0	1	1
(5, 1)	1	0	1	0
(6, 1)	1	0	0	1
(7, 1)	1	0	1	1
(8, 1)	1	0	1	0
(9, 1)	0	0	0	1
(10, 1)	0	0	1	1

Table 1: Maze Structure



```python
from pyamaze import maze

my_Maze = maze(50, 120)
my_Maze.CreateMaze(loopPercent=48, saveMaze=True)

# Display the maze
my_Maze.run()
```

Code 1: Create maze using pyamaze package

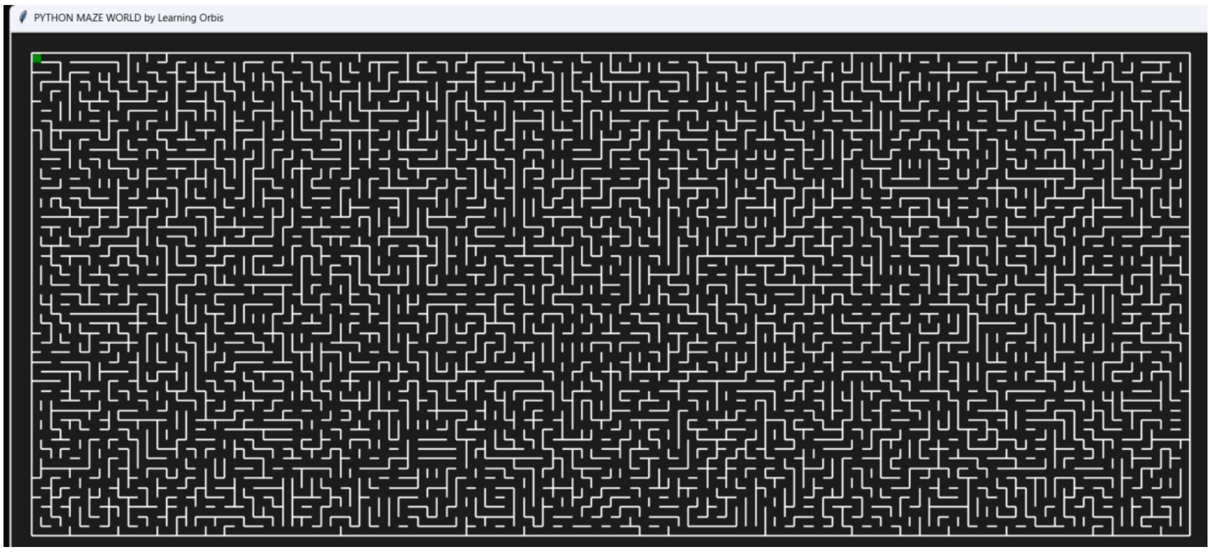


Fig 1: Created maze look like this

- The start position for each algorithm is the **bottom-right corner (50, 120)** of the maze.
- Three scenarios were defined with varying goal positions: **(1, 1), (49, 2), and (1, 119).**

3.2 Implementation

Each algorithm was implemented in Python using the pyamaze library for maze creation, visualization, and pathfinding. The library provides a flexible framework for defining maze dimensions, walls, and goal positions, making it suitable for comparative studies of search algorithms. The details of each implementation are outlined below: The uploaded files include the following implementations:

3.2.1 the_BSF.py: BFS implementation.

```python
# Importing required modules for maze creation and visualization
from pyamaze import maze, agent, COLOR, textLabel
from collections import deque
```

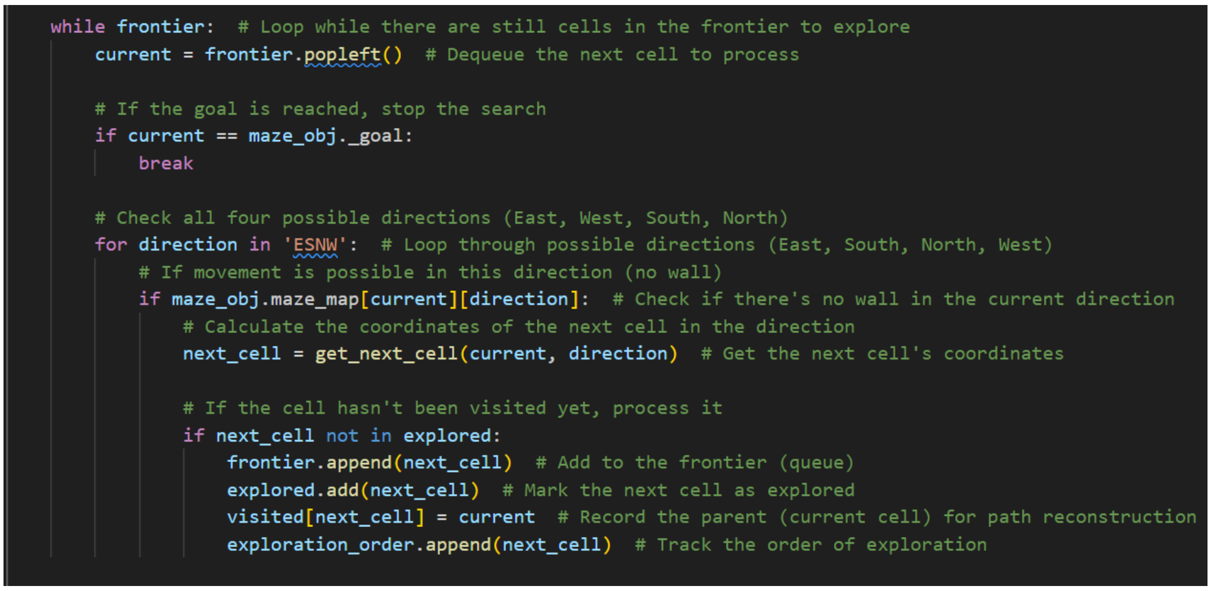
Code 2: Import necessary modules

This code imports the necessary modules for creating and visualizing a maze, as well as implementing an agent that can navigate through it. Specifically, it imports the maze, agent, COLOR, and textLabel classes from the pyamaze library. The maze class is used to generate and manage the maze structure, while the agent class represents the entity that will traverse the maze. The main agents I will use in my project is, **Algorithm's Agent:** This agent will visualize the movement of algorithm while searching the path from start point to goal position over maze. **Path Agent:** This agent will visualize the path from start to goal position, each algorithm has its own path, and these paths will show in different color, therefore The COLOR module is used to provides color options for customizing the maze's visual elements, and textLabel can be used to display text annotations within the maze. Additionally, the code imports deque from the collection's module, which is a double-ended queue commonly used for efficiently managing data structures like queues and stacks, especially for pathfinding or traversal algorithms within the maze. This is very important to understand that use of deque as data structure here with search algorithms is to increase the flow of adding and removing of nodes from both ends, deque also help to manage this flow as well.[3]

```python
def BFS_search(maze_obj, start=None):
    # If no start position is provided, use the bottom-right corner of the maze as the start
    if start is None:
        start = (maze_obj.rows, maze_obj.cols)

    # Initialize BFS frontier with the start point
    frontier = deque([start])  # BFS queue initialized with the start position

    # Dictionary to store the path taken to reach each cell
    visited = {}  # This will map each cell to its parent (previous cell)

    # List to track cells visited in the search process
    exploration_order = []  # This will store the order in which cells are explored

    # Set of explored cells to avoid revisiting
    explored = set([start])  # Start with the start cell marked as explored
```

Code 3: The BFS part I

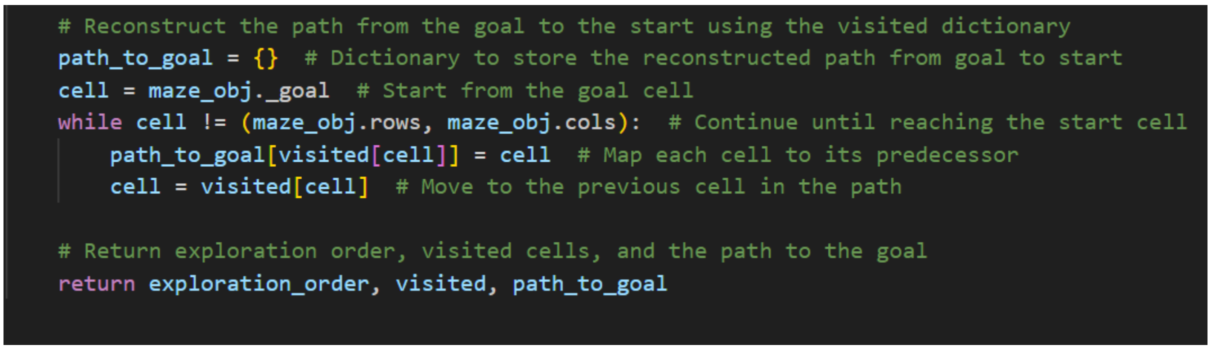
This is the beginning of the BFS algorithm code, ***start*** is the starting point for **BFS_agent** (explained here in detailed), the starting default point is at (50, 120) the bottom right corner of the maze. The frontier is deque which initiates from starting point, it is not just a simple choice of variable name. In BFS, the frontier refers to the set of nodes, here cells (x, y). In the case of a maze that are currently being explored. These are the nodes that have been discovered but not yet fully explored, meaning their neighbours are about to be visited. As BFS progresses, nodes are added to the frontier, and the algorithm explores them by removing nodes from the frontier one by one.



```python
while frontier:  # Loop while there are still cells in the frontier to explore
    current = frontier.popleft()  # Dequeue the next cell to process

    # If the goal is reached, stop the search
    if current == maze_obj._goal:
        break

    # Check all four possible directions (East, West, South, North)
    for direction in 'ESNW':  # Loop through possible directions (East, South, North, West)
        # If movement is possible in this direction (no wall)
        if maze_obj.maze_map[current][direction]:  # Check if there's no wall in the current direction
            # Calculate the coordinates of the next cell in the direction
            next_cell = get_next_cell(current, direction)  # Get the next cell's coordinates

            # If the cell hasn't been visited yet, process it
            if next_cell not in explored:
                frontier.append(next_cell)  # Add to the frontier (queue)
                explored.add(next_cell)  # Mark the next cell as explored
                visited[next_cell] = current  # Record the parent (current cell) for path reconstruction
                exploration_order.append(next_cell)  # Track the order of exploration
```

Code 4: The BFS part II

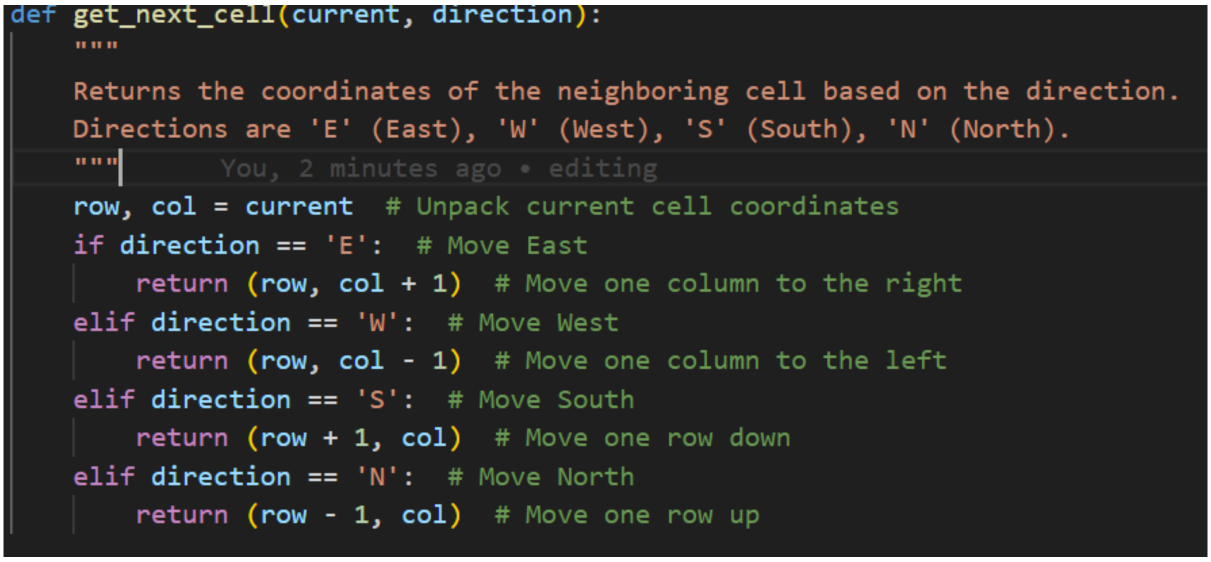
This code implements the core of the BFS algorithm for exploring a maze. Every cell will go to frontier then each cell is taken in current variable and if ***current !=goal*** then the neighbour cell is checked again and again by following the *for loop* with possible directions E,W, S, N. This moment is constrained by walls with Boolean values (0 and 1, where 0 means no wall and 1 means wall). these walls contain the moment in that direction, if moment (direction) has wall (1) then it can't move to that direction and check another direction if the moment is allowed (0) then go to the next cell and check if goal is there or not. Now to go to the next cell I have ***next_cell*** variable which use ***get_next_cell*** function. This next cell is added to frontier for future reference and if the goal is not there then these cells are put under visited variable that have list of all visited cells yet. If ***current = goal*** then the function break. Now to track this exploration I use ***exploration_order***, this will use later when I want to see the path from start to goal.



```python
# Reconstruct the path from the goal to the start using the visited dictionary
path_to_goal = {}  # Dictionary to store the reconstructed path from goal to start
cell = maze_obj._goal  # Start from the goal cell
while cell != (maze_obj.rows, maze_obj.cols):  # Continue until reaching the start cell
    path_to_goal[visited[cell]] = cell  # Map each cell to its predecessor
    cell = visited[cell]  # Move to the previous cell in the path

# Return exploration order, visited cells, and the path to the goal
return exploration_order, visited, path_to_goal
```

Code 5: Construct the path to goal.

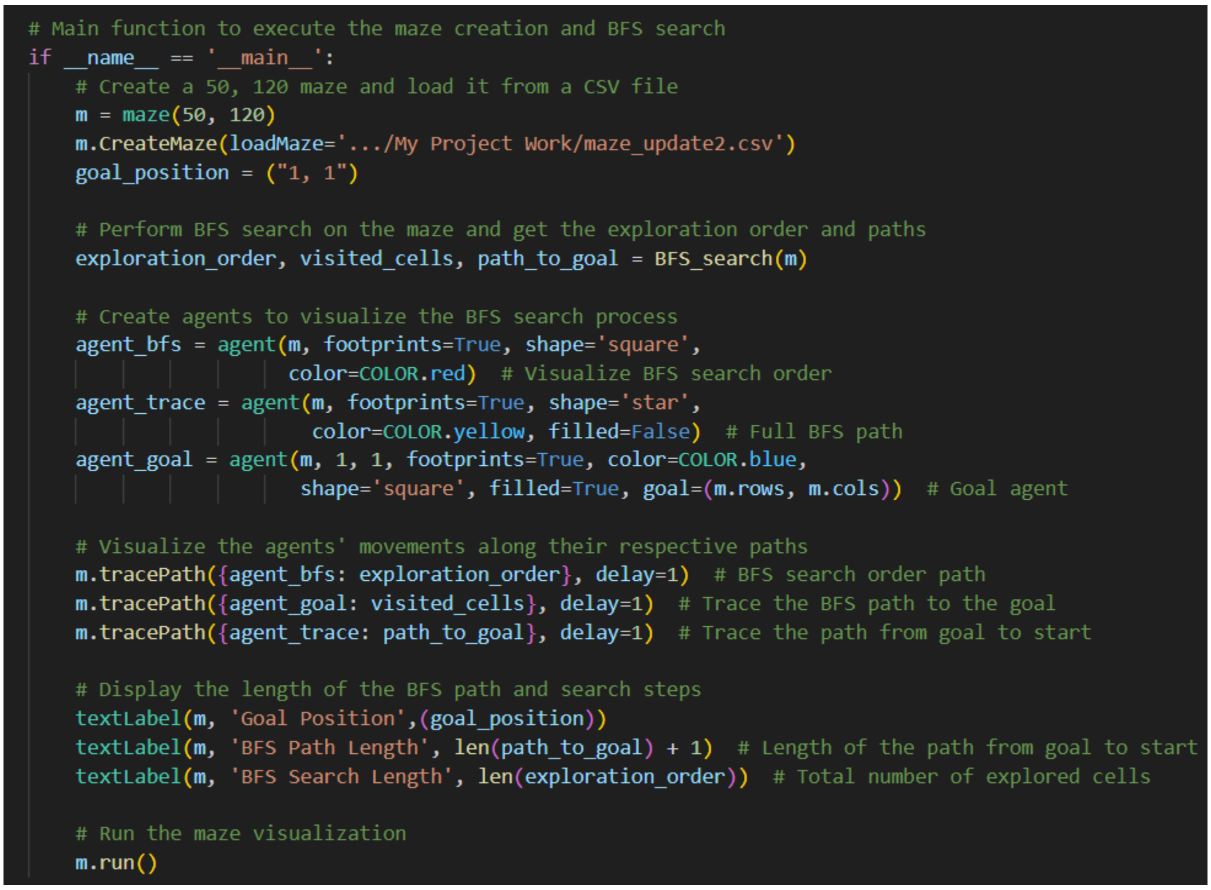
Now, the algorithms reach the goal position, so I need to see what path optimal BFS does has taken to reach the goal from starting point, so each cell needs to be in on systematic order after each visit, therefore this code comes in light. The dictionary is needed to save the path, so that traversing to predecessor cell become possible after adding each new visited cell and after reaching the goal the path can be constructed with the help of visited cells.



```python
def get_next_cell(current, direction):
    """
    Returns the coordinates of the neighboring cell based on the direction.
    Directions are 'E' (East), 'W' (West), 'S' (South), 'N' (North).
    """
    row, col = current  # Unpack current cell coordinates
    if direction == 'E':  # Move East
        return (row, col + 1)  # Move one column to the right
    elif direction == 'W':  # Move West
        return (row, col - 1)  # Move one column to the left
    elif direction == 'S':  # Move South
        return (row + 1, col)  # Move one row down
    elif direction == 'N':  # Move North
        return (row - 1, col)  # Move one row up
```

Code 6: Get next cell by reading direction ad walls

So, basically if an algorithm wants to move in any direction and if there is no wall, the return (row, Col+1) will add one column to that direction and our algorithm moves one step or cell. And if the algorithm choose direction that has wall, then no moment occurs, and next direction is chosen.

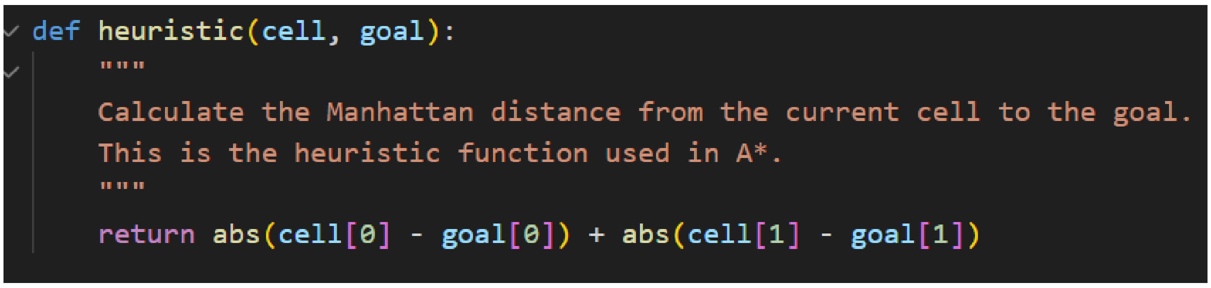


```python
# Main function to execute the maze creation and BFS search
if __name__ == '__main__':
    # Create a 50, 120 maze and load it from a CSV file
    m = maze(50, 120)
    m.CreateMaze(loadMaze='.../My Project Work/maze_update2.csv')
    goal_position = ("1, 1")

    # Perform BFS search on the maze and get the exploration order and paths
    exploration_order, visited_cells, path_to_goal = BFS_search(m)

    # Create agents to visualize the BFS search process
    agent_bfs = agent(m, footprints=True, shape='square',
                      color=COLOR.red)  # Visualize BFS search order
    agent_trace = agent(m, footprints=True, shape='star',
                        color=COLOR.yellow, filled=False)  # Full BFS path
    agent_goal = agent(m, 1, 1, footprints=True, color=COLOR.blue,
                       shape='square', filled=True, goal=(m.rows, m.cols))  # Goal agent

    # Visualize the agents' movements along their respective paths
    m.tracePath({agent_bfs: exploration_order}, delay=1)  # BFS search order path
    m.tracePath({agent_goal: visited_cells}, delay=1)  # Trace the BFS path to the goal
    m.tracePath({agent_trace: path_to_goal}, delay=1)  # Trace the path from goal to start

    # Display the length of the BFS path and search steps
    textLabel(m, 'Goal Position',(goal_position))
    textLabel(m, 'BFS Path Length', len(path_to_goal) + 1)  # Length of the path from goal to start
    textLabel(m, 'BFS Search Length', len(exploration_order))  # Total number of explored cells

    # Run the maze visualization
    m.run()
```

Code 7: The main function to run the algorithm

50 x 120 maze is loaded ***.csv***, here the goal position is (1, 1). 3 agents are created and tracepath Is created for them. The 3 agents the ***Agent_bfs:*** which show the actual movement of bfs algorithm***. Agent_path:*** which activate when goal is reached and shows the path from starting till goal. ***Agent_goal:*** which is shows the position of the goal. It stays at goal position and help to find the location of goal on the maze. The agents are designed like an entity to make decisions and take actions as according to the maze situation. These decisions are made according to the logic of algorithms.

3.2.2 the_Astar.py:



```python
def heuristic(cell, goal):
    """
    Calculate the Manhattan distance from the current cell to the goal.
    This is the heuristic function used in A*.
    """
    return abs(cell[0] - goal[0]) + abs(cell[1] - goal[1])
```

Code 8: The heuristic function.

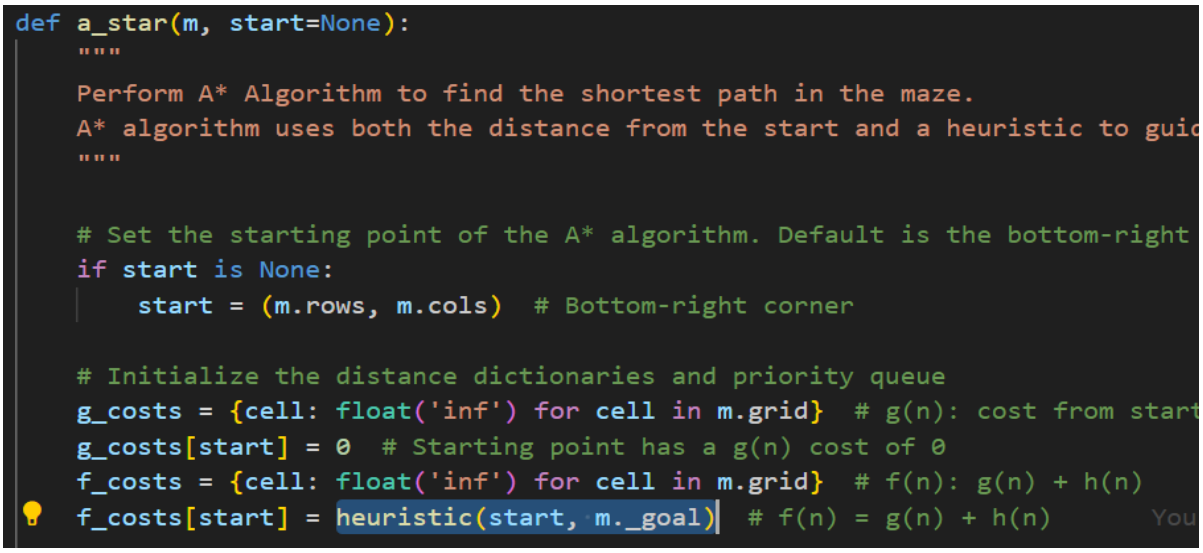
In A* there is heuristic values which inform them about the direction of the goal from starting position, each cell has same heuristic value i.e. 1 and the below code is representing to implement the heuristic (Manhattan distance) from current cell to goal position.[4]

The heuristic function is most obvious function used in A* algorithm, here the term,

return abs (cell [0] – goal [0]) + abs (cell [1] - goal [1])

is showing the calculation of Manhattan Distance. In mathematically it is done by calculating the distance between two points (A and B) by applying Manhattan distance formula. **A—B = {A(y) – A(x)} +{B(y) – B(x)}**

Where, x and y are coordinates for A and B point respectively. The calculation of heuristic function is same in A* and Greedy_BFS. I will explain it a little later. [*]



```python
def a_star(m, start=None):
    """
    Perform A* Algorithm to find the shortest path in the maze.
    A* algorithm uses both the distance from the start and a heuristic to gui
    """

    # Set the starting point of the A* algorithm. Default is the bottom-right
    if start is None:
        start = (m.rows, m.cols)  # Bottom-right corner

    # Initialize the distance dictionaries and priority queue
    g_costs = {cell: float('inf') for cell in m.grid}  # g(n): cost from star
    g_costs[start] = 0  # Starting point has a g(n) cost of 0
    f_costs = {cell: float('inf') for cell in m.grid}  # f(n): g(n) + h(n)
    f_costs[start] = heuristic(start, m._goal)  # f(n) = g(n) + h(n)        You
```

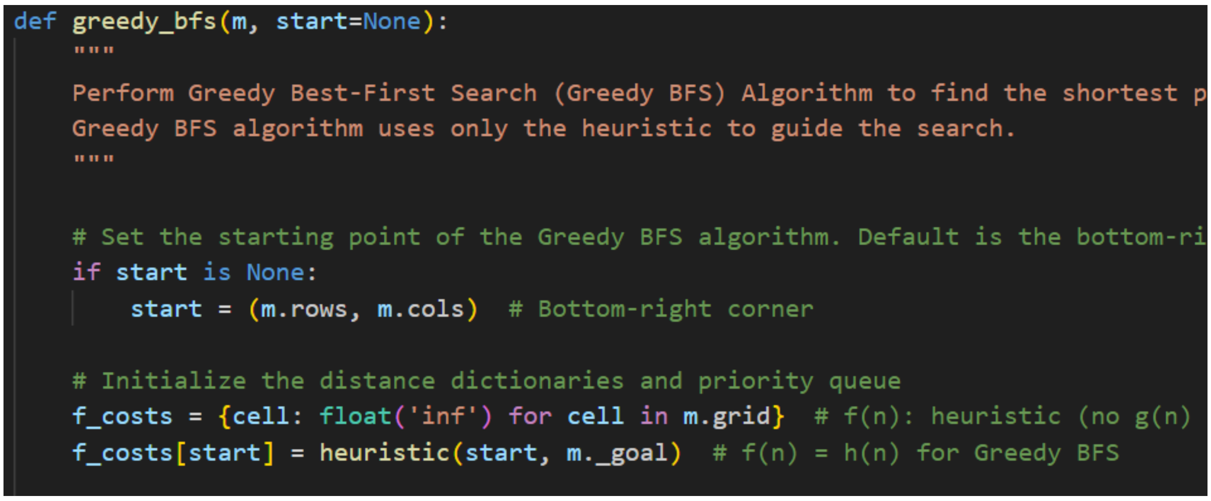
Code 9: The A* Algorithm

In this code for each neighbour, I need to calculate total cost f(n) which is equal to the sum of current cost g(n) and heuristic cost h(n).

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

The use of heuristic value is to help in calculating the search efficiency when we move towards the goal. For eg: if the goal is at left direction w.r.t the current position then heuristic value give point to move at left side towards goal from current position. Same with right, up or down direction. The heuristic value will tell the A* where it should move to get the goal ASAP. A* does not have exact direction of a goal but have estimated it and this estimation help to move the algorithm towards goal path.

3.2.3 the_Greedy_BFS.py:



```python
def greedy_bfs(m, start=None):
    """
    Perform Greedy Best-First Search (Greedy BFS) Algorithm to find the shortest p
    Greedy BFS algorithm uses only the heuristic to guide the search.
    """

    # Set the starting point of the Greedy BFS algorithm. Default is the bottom-ri
    if start is None:
        start = (m.rows, m.cols)  # Bottom-right corner

    # Initialize the distance dictionaries and priority queue
    f_costs = {cell: float('inf') for cell in m.grid}  # f(n): heuristic (no g(n)
    f_costs[start] = heuristic(start, m._goal)  # f(n) = h(n) for Greedy BFS
```

Code 10: The Greedy_ BFS Algorithm

Same code as A* just greedy used only heuristic value to find path towards goal.

Here, **f(n) = h(n)**

This is a basic reason why the greedy_BFS is not an optimal algorithm but faster. As compared to A* the greedy_BFS is only focus on moving forward. Greedy_BFS only use heuristic values to calculate the total cost. It ignores the current cost and just focus on the value which let it to move faster towards goal.

Now, as I discuss about both the algorithms the **A* and Grredy_BFS,** let's understand the reason why heuristic value is same in both the algorithms, but the formula is different for calculating the total cost.

[*] The real reason behind this is, in Greedy_BFS the total cost is calculated on the bases of heuristic value, it means that no current cost is being consider when the agent move towards goal.

A—B = (hn) = {A(y) – A(x)} +{B(y) – B(x)}.

A* → f(n) = (h(n) + g(n). as discuss previously in A* the total cost is calculated by adding heuristic value along with current cost.[5]

Greedy_BFS → f(n) = h(n) in greedy_BFS is only needed heuristic value therefore if impact of current cost is null in this case and greedy_BFs will only focus on achieving the goal in fastest way without taking care of optimality of a path.

4. Results and Discussion

4.1 Performance Metrics

The algorithms were evaluated based on:

1. **Path Length:** The number of steps from the start to the goal.
2. **Exploration Length:** The number of nodes explored before reaching the goal.

Scenario	Algorithms	Goal Position (cell No)	Path Length	Exploration Length
1	**BFS**	(1, 1)	189	5991
	Greedy_BFS	(1, 1)	229	401
	A*	(1, 1)	189	4264
2	**BFS**	(49, 2)	174	5707
	Greedy_BFS	(49, 2)	204	331
	A*	(49, 2)	174	2358
3	**BFS**	(1, 119)	83	2253
	Greedy_BFS	(1, 119)	101	152
	A*	(1, 119)	83	657

Table 2: Comparing Algorithm's path and search length

By using this table I'm comparing the performance of the 3 algorithms in 3 different scenarios. The goal is to find the best path from a starting point to goal position which chances.

Scenario 1(Goal Position: (1, 1): When goal position is at (1,1), Both **BFS** and **A*** find the shortest path to the goal at **(1, 1)**, which is **189 steps** long. However, **Greedy_BFS** takes a longer path of **229 steps**. When it comes to exploration, **A*** explores **4264 steps** to find the optimal path, while **BFS** explores **5991 steps**. On the other hand, **Greedy_BFS** explores only **401 steps** to reach the goal, but it doesn't find the shortest path, making its exploration much smaller but less optimal.

Scenario 2 (Goal Position: (49, 2)): BFS and **A*** both find the shortest path of **174 steps**, while **Greedy_BFS** takes a longer path of **204 steps**. In terms of search length, **A*** explores **2358 steps**, and **BFS** explores **5707 steps** to find the optimal path. In contrast, **Greedy_BFS** only explores **331 steps** to reach the goal, making it the fastest in terms of exploration, though it doesn't find the shortest path like the other two algorithms.

Scenario 3 (Goal Position: (1, 119)):

In Scenario 3, the goal is at **(1, 119)**. **BFS** finds the shortest path of **83 steps**, while **Greedy_BFS** takes a longer path of **101 steps**. For search length, **BFS** explores **2253 steps**, while **Greedy_BFS** only explores **152 steps**. As in the previous scenarios, **Greedy_BFS** explores fewer steps but takes a less optimal route, while **BFS** finds the shortest path with more exploration.

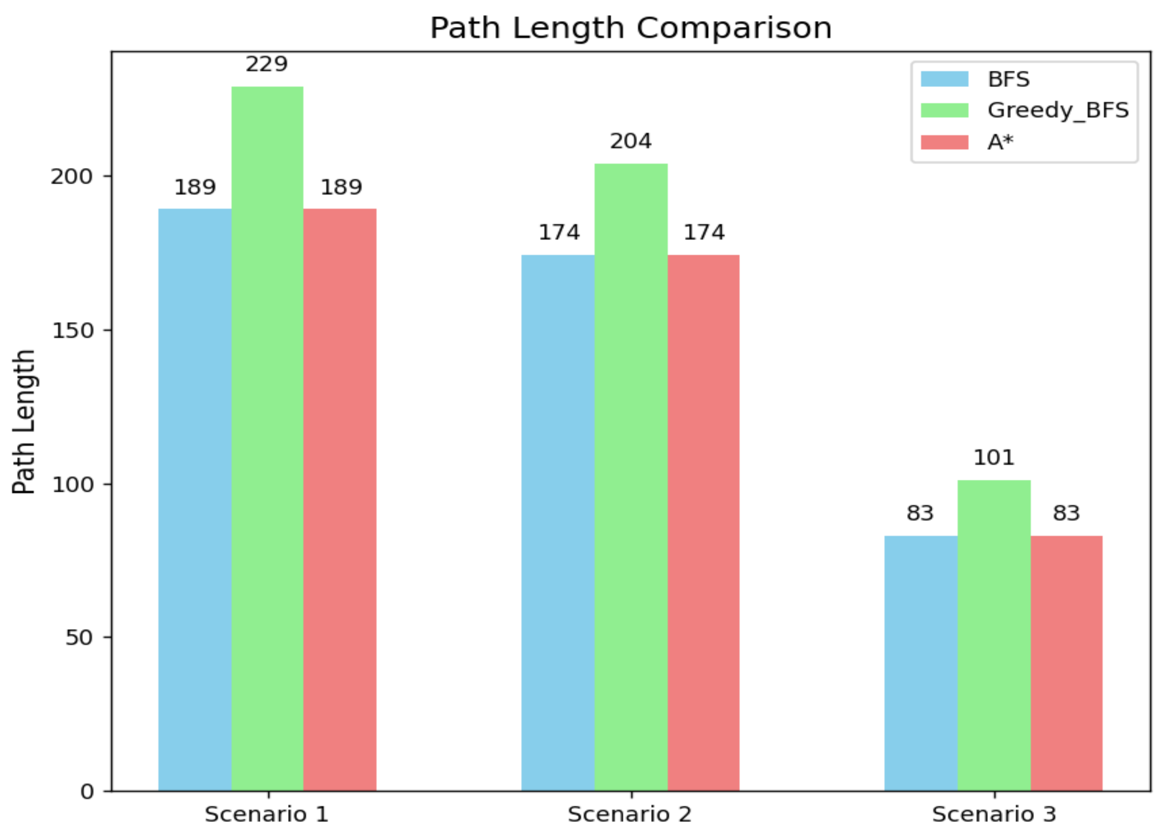


Fig 2: Path Length Comparison with all 3 Scenarios

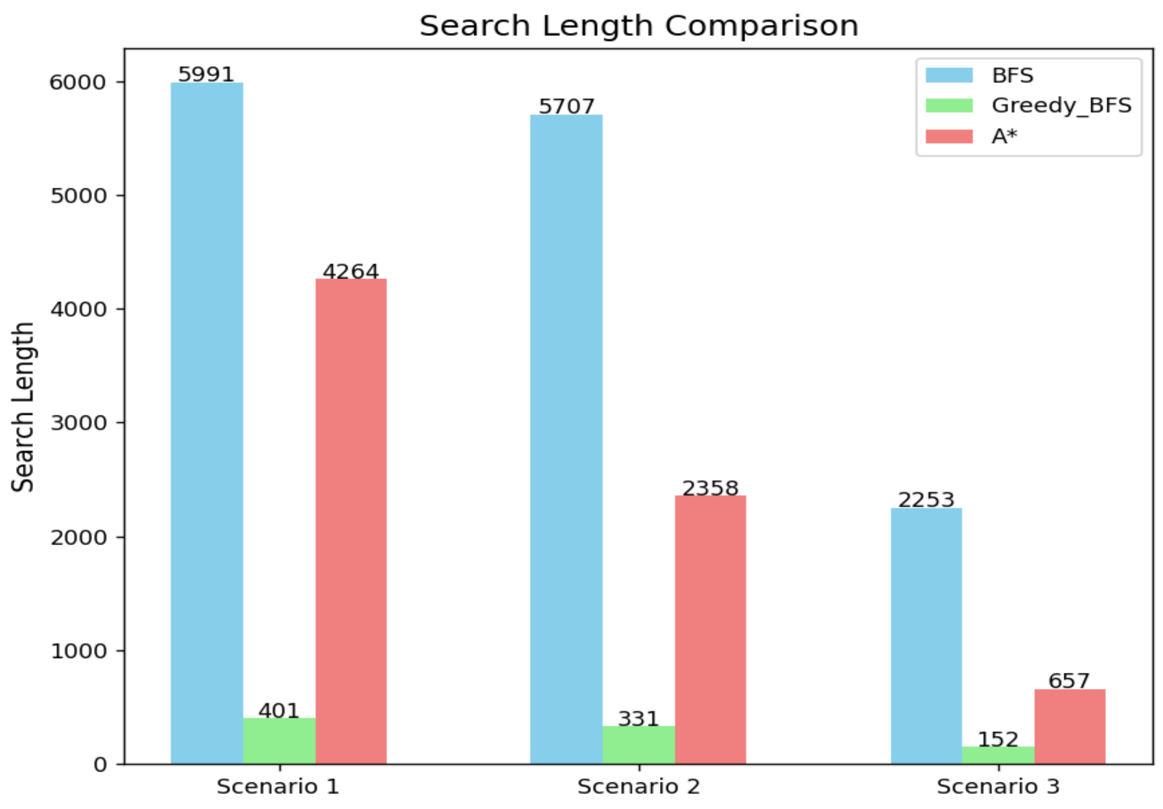


Fig 3: Search Length Comparison with all 3 Scenarios

4.2 Explanation of the Results

The comparison between the search algorithms BFS, Greedy BFS, and A* across different scenarios reveals important insights into their performance.

- BFS performs consistently across all scenarios with the same path length but explores many more nodes, resulting in high search lengths. This is because BFS searches all possible paths level by level, making it exhaustive but less efficient in terms of exploration.
- Greedy BFS, on the other hand, focuses on reaching the goal based on a heuristic, which can make it faster in terms of search length, especially in scenarios like Scenario 1. However, this comes at a cost, it often takes longer paths to reach the goal compared to BFS or A*. Greedy BFS does not guarantee the shortest path, as it is biased towards the most promising nodes based on the heuristic, rather than considering the entire search space.
- A* strikes a balance between path length and search length. It combines the benefits of both BFS and Greedy BFS, using a heuristic to guide its search, but ensuring that it still finds the optimal path. While it sometimes explores more nodes than Greedy BFS, it generally results in shorter paths than Greedy BFS while being more efficient than BFS in terms of search length.

4.3 Algorithm Comparison

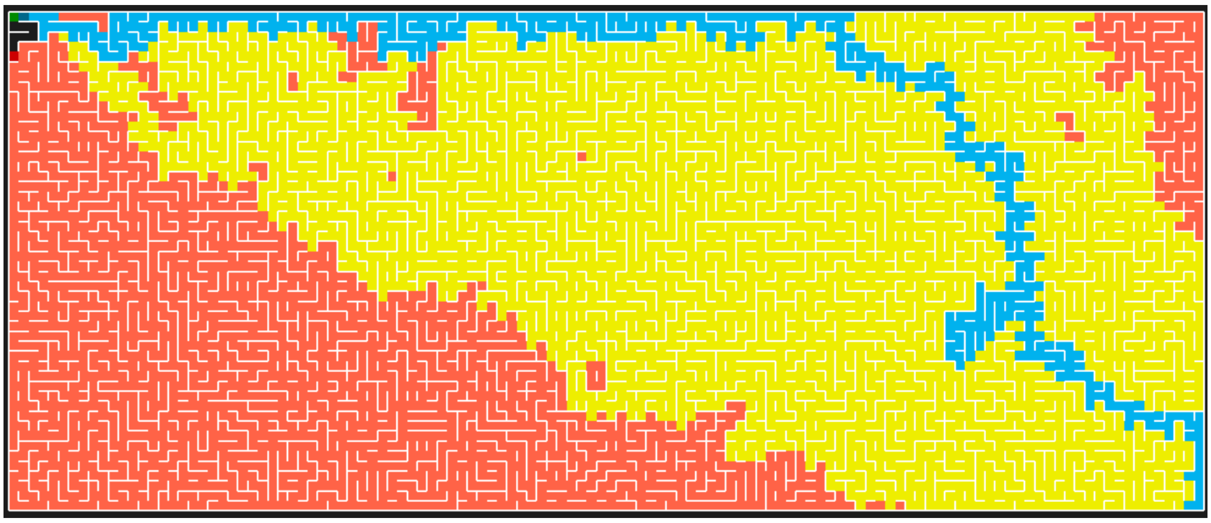


Fig 4: Exploration path comparison of all algorithms with goal position (1, 1)

|| Red path = BFS || **|| Yellow path = A* ||** **|| Blue path = Greedy_BFS ||**

4.3.1 Breadth-First Search (BFS):

- **Strength:** BFS is great for finding the shortest path, especially when all steps or edges have the same cost (like in unweighted graphs).
- **Weakness:** It can be slow and use a lot of memory because it checks all the nodes at each level before moving deeper.

- **Example:** If you're trying to find the shortest path between two people in a social network, BFS is perfect because it will explore all possible connections and guarantee the shortest one, without skipping any possibilities.

4.3.2 A Algorithm:*

- **Strength:** A* is great for finding the best, most efficient path because it combines the current distance with a guess of how far it is to the goal (using a heuristic).
- **Weakness**: It depends on having a good heuristic. If the guess isn't accurate, it can take longer than necessary.
- **Example**: A* is often used in GPS systems for navigation, where it can find the fastest route considering both distance and factors like traffic or road conditions. It's the best choice when you need to be sure you're getting the most efficient route.

4.3.3 Greedy Best-First Search (Greedy BFS):

- **Strength:** Greedy BFS is faster than A* because it focuses more on the goal and explores fewer nodes.
- **Weakness:** It might not always find the best (shortest) path because it can get stuck on a path that seems good at first but isn't the best in the end.
- **Example:** Imagine a robot trying to navigate a grid. If the goal is close and the robot has a good idea of where it is, Greedy BFS can get it there quickly. But if there are obstacles, it might choose a bad path because it only looks at the nearest options.

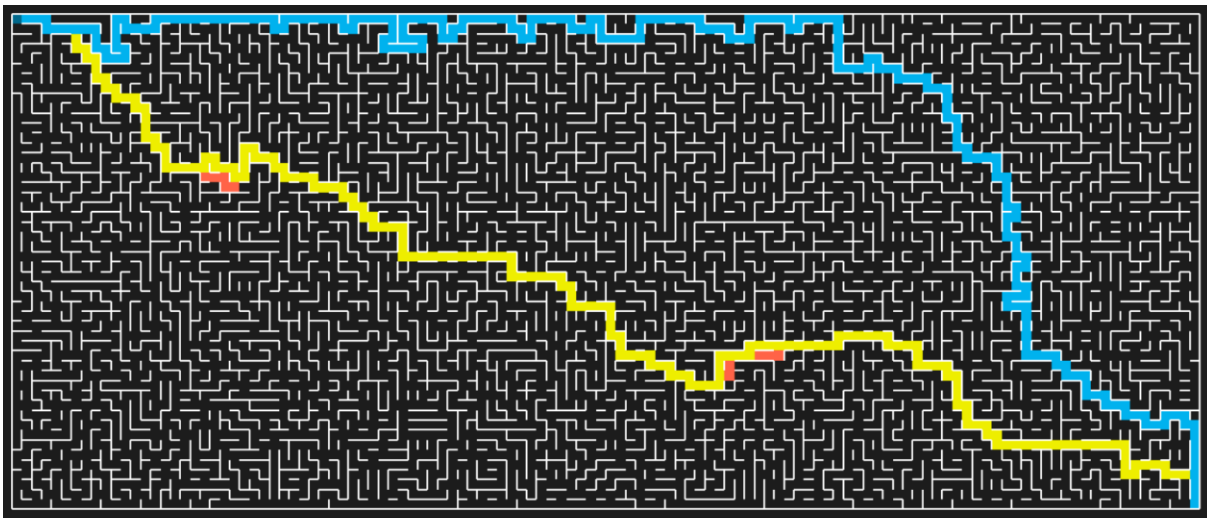


Fig 5: Goal path comparison of all algorithms with goal position (1, 1)

|| Red path = BFS || **|| Yellow path = A* ||** **|| Blue path = Greedy_BFS ||**

Here, we see the A* and BFS have almost same path to reach goal just a little difference in between but that will not affect the lath length that's the reason this difference is acceptable because both are giving the optimal path. Here we can conclude that there might be many optimal paths in a maze but the all optimal path should have same path length like, the BFS and A* has different path as we can see the red color below the yellow color, this is because, BFS(red color) take little different path as compare to A*(Yellow color), but as we can see from the result the path length is same in both algorithms in all 3 cases.

4.4 Discussion

In this project, I had learned that A* had given better result as compared to other algorithms, but it doesn't mean that A* is better in all scenarios [6]. A* is better when we want to focus on optimality and we need result faster than BFS, so A* is good.

Important Question: __ If A* is giving us best optimal path with less node search than BFS why we even need a Greedy_BFS algorithm__??

This project reflects the implementation of above 3 algorithms which is important to give for me to understand the concept which make me able to give answer to this question:

Basically, every algorithm has its own advantages and disadvantages like I already highlighted above, so does have greedy_BFS too. Greedy_BFS is basically used when we want to focus on getting result rather than the cost of getting result.

Eg: Social Media recommendation system: In Facebook, snapchat or any other social media, the suggestion of friends are given by application on the bases of how close the person is to who, like friend's of friend's or friend etc... this system doesn't care about connecting with you the best person, NO it only suggest you nearby connection, here we use Greedy_BFS.

1. **Another eg of using Greedy_BFS is in games like *GTA, farCry or* Assassin's *Creed*,** [7] in these games the NPC(Non-Player Charaters) are used to do some action. Their actions are only based on getting to the goal not to do anything, here we don't need optimal solution, we just need faster solution which can be achieve by greedy_BFS.

---

5. Achievements and Insights

In this project, I successfully compared three classic search algorithms BFS, Greedy BFS, and A*, across three different goal positions, analysing their performance in terms of Path Length and Search Length. The key achievements and insights are:[10]

1. BFS (Breadth-First Search), while consistent in its results, showed that it is a highly exhaustive search algorithm, exploring a significant number of nodes to find the goal. This led to higher search lengths compared to the other algorithms but ensured the shortest path to the goal.

2. Greedy BFS, which uses a heuristic to guide its search, was faster in terms of search length but often resulted in longer path lengths. This highlights that while Greedy BFS reduces the number of nodes explored, it doesn't guarantee the shortest or optimal path.

3. A* emerged as the most balanced algorithm. By combining the best of both BFS and Greedy BFS, it achieved shorter paths than Greedy BFS and explored fewer nodes than BFS, showing its effectiveness in finding optimal solutions efficiently.

6. Conclusion and Future Work

2. In conclusion, this project demonstrated the strengths and weaknesses of BFS, Greedy BFS, and A* algorithms in solving search problems. While BFS is exhaustive and guarantees the shortest path, it is computationally expensive in terms of search length. Greedy BFS offers speed but sacrifices optimality, and A* provides a balanced approach, finding the optimal path with efficient node exploration.**[8]**
3. Future work I'm planning to experiment with different heuristics in A* like Manhattan, Euclidean etc and understand their performance. Another direction for future research could involve comparing these algorithms with other advanced search techniques, such as IDDFS (Iterative Deepening DFS) or Dijkstra's Algorithm, to explore their performance in varied scenarios.
4. Another scenario for future understanding is to use some obstacles or non-uniform cost grids, which would add complexity and realism to the search problems.

7. References

1. *Geeks for Geeks, Searching Algorithms.*
2. *Chang ChiaChi, 2020, (GitHub Project), Maze Shortest Path Finding.*
3. *MAN1986, (2021), (Github Project), Path Finding using pyamaze,*
4. *Stanford theory, Game Programming, Heuristics.*
5. Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley.
6. Korf, R. E. (1985). "Iterative-Deepening A*." *Artificial Intelligence, 27*(1), 97-109.
7. S. R. Lawande, Graceline Jasmine, (2022), A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games, Lila Iznita Izhar.
8. Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach (3rd ed.)*, Pearson.
9. E Badr, O Loubna, E Hiba, H Ayoub, E Chama, B Yassine, EL M Karim, (2024), Semantic_Scholar, Maze Navigation: A Comparative Study
10. Pathfinding Visualizer: Google Source

8. Appendix

- **Code Files: .pynb** files are being attached within zip.
- **Execution Environment:**
  1. ***Programming Languages***: *Python 3.12.8*
  2. ***Libraries***: *numpy, pandas, matplotlib, scikit-learn, pyamaze, deque, tkInter*
  3. ***Development Environment***: *Visual Studio Code, Jupyter notebook*
  4. ***Operating System***: *Windows 11*
  5. ***Hardware***: *Intel Core i7 processor, 16 GB RAM.*
- **Data Sources: maze_update2.csv** file is attached within zip.