**Analysis of Graph Search Algorithms**

**Depth-First Search, Breadth-First Search, Greedy\_BSF and A\* Search Algorithms in Maze Solving**

**Krishna Gopal Sharma**

**Student Number: B1055988**

**Artificial Intelligence ICA S3454618**

**Total Words:**

**1. Abstract**

This project investigates the performance of four search algorithms, the uninformed search algorithms i.e. Breadth-First Search (BFS) and informed searched algorithms i.e. {A\* Algorithm, and Greedy Best-First Search (Greedy BFS)} in solving a **maze of 50 x 120 size.** This maze is created by python code as shown below. with **loopPercent = 48%.**

**A computer screen shot of a program

Description automatically generated**

**Code 1: Create maze using pyamaze package.**

On same maze, the algorithms were tested under three 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.

**2. Introduction**

Pathfinding and search algorithms are fundamental to Artificial Intelligence (AI), with applications spanning robotics, gaming, and optimization problems. This project focuses on implementing and comparing these 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)).

This report documents the methodology, implementation, results, and analysis of these algorithm’s performance across three different goal positions.

**2. Methodology**

**2.1 Problem Setup**

* The maze dimensions, structure and obstacles were predefined and loaded from a CSV file.
  + Dimensions: 50 x 120.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

* + Structure: Cell (x, y)

Direction: E, W, N, S

* + Design: Cell (1,1 ) represent cell number or position and E,W,N,S (0 and 1 ) represent the presence of wall or not )

Table 1: Maze Structure

A black and white maze

Description automatically generated

**Fig 2: Created maze look like this.**

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

**2.2 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:

* **1.1 the\_BSF.py:** BFS implementation​.

A black background with green and blue text

Description automatically generated

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, **Algorithms 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 as in BFS to increase the flow of adding and removing of nodes from both end, deque also help to manage this flow as well.

A screen shot of a computer program

Description automatically generated

Code 3: The BFS part I

This is the beginning of the BFS algorithm code, start is the starting point for BFS\_agent (this will I explain later in agent section), 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.

A computer screen shot of a program code

Description automatically generated

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 from 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 direction*  with possible direction E,W, S, N, this moment is constrain by walls with Boolean values (0 and 1, where 0 means no wall and 1 means wall and 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 allow (0) then go to the next cell and check if goal is there 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. Now to track this exploration I use ***exploration\_order***, this will use later when I want to see the path from start to goal. If ***current = goal*** then the function break.

A screen shot of a computer code

Description automatically generated

Code 5: Construct the path to goal.

Now, from above code we will reach the goal position, so I need to see what path 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.

A screen shot of a computer code

Description automatically generated

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.

A screen shot of a computer program

Description automatically generated

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 show 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.

* **1.2 the\_Astar.py:** 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.

A screen shot of a computer

Description automatically generated

Code 8: The heuristic function.

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. [\*]](#The_real_reason_of_differene_in_formula)

A computer screen shot of text

Description automatically generated

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.

**1.3 the\_Greedy\_BFS.py:** *Same code as A\* just greedy used only heuristic value to find path towards goal.*

A computer screen shot of text

Description automatically generated

Code 10: The Greedy\_ BFS Algorithm

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 calculated in both the algorithms, but formula od both algorithm is different.

[\*] The real reason of formula difference while calculating Manhattan distance is the way/values used to calculate the Manhattan distance between goal and starting position.

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

A\* 🡪 f(n) = (h(n) + g(n)

Greedy\_BFS 🡪 f(n) = h(n)

**2.3 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)):**

In Scenario 2, the goal is at **(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.

A graph of different colored bars

Description automatically generated

Fig: Path Length Comparison with all 3 Scenarios

A graph of bar graph

Description automatically generated with medium confidence

**3. Results and Discussion**

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.

**3.4 Algorithm Comparison**

* **Breadth-First Search (BFS):**
  + Performed optimally in finding the shortest path due to its exhaustive exploration.
  + High computational cost as all nodes at a given depth are explored.
* **Greedy Best-First Search (Greedy BFS):**
  + Faster than A\* but prone to getting stuck in local minima.
  + Explored fewer nodes but sometimes resulted in longer paths.
* *A*\*:
  + Consistently achieved optimal paths with efficient exploration.
  + Performance depended on the heuristic accuracy (Manhattan distance used).

**4. Achievements and Insights**

**I**n 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:

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.

**5. Conclusion and Future Work**

1. 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.
2. Future work could focus on testing these algorithms on more complex, dynamic, or real-world problems, such as navigating in larger maps or environments with changing conditions. Additionally, experimenting with different heuristics in A\* could further optimize its 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.
3. The project also opens opportunities to explore how the performance of these algorithms might change in environments with obstacles or non-uniform cost grids, which would add complexity and realism to the search problems**.**

**6. Appendix**

* **Code Files:** The complete implementation is included in the files attached to this report.
* **Execution Environment:** *(Put the details of your software, libraries, and hardware here.)*
* **Data Sources:** *(Put information about the CSV files or maze generation details here.)*