Comparison of BFS, DFS, and A* in the Snake Game
May 8, 2023

Matin Horri
horri031@umn.edu
University of Minnesota
Minneapolis, Minnesota, USA

Evelyn Junaid
junai015@umn.edu
University of Minnesota
Minneapolis, Minnesota, USA

Abstract

This research paper evaluates the performance of DFS, BFS, and A* algorithms with four different heuristics (Manhattan, Diagonal, Euclidean, and Euclidean squared) on the Snake game. The study analyzes the algorithms based on time to completion, the number of actions needed to reach the goal, and two calculated scores: food per actions score and food per time score. The study was conducted by executing the program ten times and gathering data to compute the average performance metrics. The results show that each algorithm has its strengths and weaknesses, and the best algorithm depends on the specific requirements of the problem. Overall, BFS performed the best on all metrics, followed by A* with Manhattan and diagonal heuristics. A* with Euclidean and Euclidean squared heuristics performed worse in comparison, while DFS performed the worst. The study concludes that if the goal is to minimize the number of actions required or the time it takes to reach the goal, then BFS is the best algorithm. A* with Manhattan and diagonal heuristics would also be suitable algorithms to choose for this problem. A* with Euclidean heuristic would be an appropriate choice if time to completion and number of actions required do not need to be optimal, but A* with Euclidean squared distance heuristic is not recommended as it often overestimates the true distance. DFS does not appear to be a good choice for a pathfinding algorithm in the Snake game.

1 Introduction

Pathfinding algorithms remain a highly relevant and sought-after topic in the field of Artificial Intelligence today. With a diverse range of applications, including but not limited to multi-player interactive video games and GPS systems, there is an ongoing demand for the development and improvement of these algorithms. One such game that can be implemented using pathfinding algorithms is the Snake Game. In this game, the player must control the movement of a snake and navigate it through a map to collect food. While traditionally played by humans, the Snake Game can also be played by utilizing a controller that utilizes a pathfinding algorithm to guide the snake towards the food.

Two popular uninformed search algorithms that have a long history in solving single agent pathfinding games are Breadth-First Search (BFS) and Depth-First Search (DFS). In addition to these algorithms, the more advanced informed search algorithm A Star Search (A*) has been introduced. This algorithm utilizes a heuristic evaluation function that can enhance pathfinding abilities across various metrics, such as time, space, and path length.

BFS is a search algorithm that begins at the starting position of the snake and explores all the neighboring positions in each direction. It then continues to expand the search to the next level of nodes until it either finds the food or exhausts all possible paths. While BFS is guaranteed to find the shortest path to the food if one exists, it can be slow and memory-intensive for larger search spaces.

DFS, on the other hand, is a search algorithm that also begins at the starting position of the snake but explores as far as possible in each direction before backtracking. While DFS can be faster than BFS for small search spaces, it does not guarantee the shortest path to the food.

A* Search is a more advanced search algorithm that combines the best of both BFS and DFS. It uses a heuristic function to estimate the distance between the current node and the goal node (the food). A* search then expands the search in the direction of the lowest estimated cost. A* search is faster than BFS for larger search spaces and guarantees the shortest path to the food. The time and space complexities of A* searches are highly dependent on the heuristic function used. Common heuristic functions used in A* search include the Manhattan distance and the Euclidean distance. The Manhattan distance heuristic calculates the distance between two points by summing the absolute differences between their x and y coordinates. The Euclidean distance heuristic calculates the straight line distance between two points.

To solve the Snake Game using these search algorithms, one would first represent the game board as a graph, with nodes representing the snake's position and edges representing the possible movements. Then, the search algorithm would be applied to the graph to find the path to the food. The performance of each algorithm can be measured using different parameters such as the time it takes for the snake to reach the food, the number of actions it takes for the snake to reach the food, and the amount of memory used.

The Snake Game is a simple yet challenging problem that can be solved using various search algorithms, making it an excellent platform for learning and practicing search techniques. Attempting to solve the Snake Game using DFS, BFS,

and A* with different heuristic functions can highlight the importance of not only using heuristic functions but choosing the right one. Measuring the performance of these algorithms can help to gain insight into which algorithms perform well under certain conditions. By using the right algorithm for a specific situation, we can save time and memory. This will improve the accuracy of our solutions and can lead to better outcomes in a wide range of applications.

Understanding the differences and strengths of different pathfinding algorithms can help developers choose the best algorithm for their specific use case. For instance, in video game development, the choice of algorithm can impact the performance and user experience of the game. In GPS navigation, the choice of algorithm can affect the accuracy and speed of the directions provided to the user. Therefore, by studying and comparing different algorithms in a practical setting, we can gain a better understanding of their strengths and limitations, and make more informed decisions in the development of AI applications. The performance of different search algorithms is highly dependent on the problem instance, so predicting the optimal algorithm is impossible without testing. In order to compare the effectiveness of different algorithms under specific conditions, it is crucial to conduct experiments. A simple game like the Snake Game, with its adjustable variables such as grid size, fruit placement, and snake speed, provides a great platform for testing and benchmarking pathfinding algorithms. Snake Game is not only a classic single-player game, but also a crucial first step towards solving more complex games and tasks, including real-time strategy games and robotic path planning. Pathfinding algorithms can be used in the Snake Game to study trade-offs between metrics such as time, space, and optimality, and to compare informed and uninformed search strategies. It can be a fun and engaging way to learn about AI, programming, and game development. The Snake Game has been used to evaluate several pathfinding algorithms in several research studies, and it is interesting to learn more about it for researchers, programmers, and students.

2 Relevant Work

2.1 Introduction

The implementation of pathfinding algorithms in games has and continues to be an area of great interest within artifical intelligence. A plethora of experiments to evaluate the performance of these algorithms in various environments have been conducted throughout A.I. history. This review will explore such experiments and their findings.

2.2 DFS

DFS explores a state space that begins at a root and explores as deep as possible before returning to the root to explore other nodes[2]. DFS has been known to perform suboptimally in large and complex state spaces. The time complexity of DFS is

$$O(b^d)$$

and the space complexity is

$$O(b * d)$$

where b represents the maximum branching factor and d represents the maximum depth[12]. The linear space complexity of DFS is a major advantage of the algorithm that makes it worth exploring.

2.3 BFS

BFS explores a search tree level by level, expanding all nodes in a level before moving on to the subsequent level. BFS is always guaranteed to find an optimal solution, has a space and time complexity of

$$O(b^d)$$

, where b is the maximum branching factor and d is the maximum depth[12]. Since BFS is guaranteed to find an optimal solution, it will be interesting to see how it compares to A* in the Snake Game.

2.4 A*

A* has been found to be an effective pathfinding algorithm in video games. The A* algorithms can effectively guide characters through game worlds and handle dynamic environments[8]. Considering that the Snake Game has a dynamic environment and consists of the snake navigating the map, A* may be a good choice for the game. When the game world is small, either BFS or DFS are sufficient for pathfinding as they do not use large amounts of time or memory in small environments[8]. The same is not true as the environment increases in size and complexity. The slowest component of A* is searching for which move the algorithm will take. This is done by examining each possible move and executing whichever one returns the lowest value of the heuristic function. Using a binary heap instead of a queue is an effective way of speeding up the search process[8]. A* tends to be the preferred pathfinding algorithm because it outperforms uninformed search algorithms in efficiency and adaptability due to its heuristic evaluation[4].

2.5 Comparing Search Algorithms in the Snake Game

In the article written by Appaji, BFS, DFS, and A* were used as search algorithms in the Snake Game. In the first experiment each algorithm was run for 120 seconds. The results showed that when the controller was using A* the snake ate the highest number of fruits (92), followed by BFS (73), and DFS (22)[2]. For the second experiment, the fruits were placed at fixed coordinates and the algorithms were all run for a set amount of time. Under these conditions, the snake ate the same amount of fruit for all three algorithms but BFS took 104s, DFS took 203, and A* took 82. The researchers

concluded that BFS was more successful than DFS and a human but not as successful as A*[2]. In a similar experiment conducted by Kong, five different search algorithms were compared in solving the Snake Game. Best-First Search, A*, A* with forward checking, Almighty Move, and Random Move. A* with forward checking is similar to A* except that it considers what state the snake will be in after the fruit is reached for each potential move as part of its path evaluation[7]. The researcher found that A* with forward checking outperformed A* in that it produced the highest score while taking fewer steps. A sequential implementation of BFS followed by A* with forward search then by Almighty Move was suggested by Sharma et al. Through trial and error the authors concluded that for an AI bot to achieve the highest number of scores with the least amount of moves, the bot should move using BFS for first 4 moves, A* with forward search for next 34 moves, and Almighty with 62 moves[13]. The authors discovered that the AI Bot can be utilized to train players in the field of "Electronic Sports"[13]. Although this discovery will not be realized in this project, the findings are impressive.

2.6 Comparing BFS, DFS, and A* in Various Environments

The comparison of BFS, DFS, and A* can be analyzed in many different environments. For example, in the article written by Banerjee, BFS and DFS were compared based on optimality, completeness, and time and space complexity in a simulated restaurant environment. The environment contains custom start and goal nodes, tables, and a mobile serving robot[3]. Both BFS and DFS were able to find a path to the goal node but BFS had lower time and space complexity compared to DFS[3]. In an experiment conducted by Mahmud et al., the ability of three variants of DFS were examined to determine their effectiveness at discovering a map on an unknown maze. Their implementation of DFS with a Greedy Approach performed better in terms of number of movements than classic DFS[9]. The Greedy Approach worked better than the Simple DFS Algorithm because it prioritized finding the shortest path in the maze. By first going to unchecked very short dead-ends, followed by unchecked front adjacent cells, and then unchecked cells on the right or left, the Greedy Approach helped to minimize the movements and rotations required to navigate through the maze, resulting in a more efficient path.This suggests that although classic DFS is often suboptimal, it can improve with modifications to the algorithm.

In the article written by Permana et al., BFS, A*, and Dijkstra's pathfinding algorithms were used to solve the Maze Runner game. The Maze Runner game consists of an NPC who is trying to reach a goal state and has to navigate through a labyrinth of blocks created by players.The performance of each algorithm was measured by time, path length, and number of blocks played. The authors found that all three algorithms find the optimal path length, with A* being the overall best due to short searching times and smallest number of blocks[11]. Similarly to Permana et al., Iloh used a maze game to analyze the performance of DFS, BFS, and A*. The author was able to conclude that each algorithm can locate the shortest path length. It was found that DFS takes a long time to reach the goal. The author also concluded that A* is the most effective pathfinding algorithm, taking the shortest amount of time to reach the goal[6].

Alhassan et al. compared the performance of BFS, DFS, and A* for the game of Bloxorz. Bloxorz is a single agent path-finding problem in which a block must be moved in one of four directions (up, down, left, right) to reach a goal square. It is a 3-D block sliding game with a map of 1x1 boxes arranged in a special shape. The navigation of the block from its initial state to its goal state will be similar to the snake game, such that there are also four directions the snake can move in to reach the goal state. The authors consistently found that BFS and A* are able to solve the game in the number of optimal moves, while DFS was not always able to do so. Depending on the starting box and ending box, DFS can perform better than BFS and A* but, this is generally not the case. In conclusion, the authors noted that BFS always finds the optimal path and that A* performance is strongly dependent on the heuristic function used. Additionally, because A* uses tree search, it expands more nodes than BFS and DFS, using more memory[1].

Shi assessed the differences between BFS, DFS, and A* in solving the infamous eight puzzle. Consistent with the findings of other articles previously discussed, A* was found to be the most effective algorithm in solving this puzzle[14]. A* outperformed BFS and DFS in elapsed time and number of search steps. In this specific puzzle, DFS performed close to ten times worse than BFS, and BFS performed four times worse than A*. It appears that in puzzles similar to the eight puzzle, DFS does not find the optimal solution, although it does find a solution.

2.7 Conclusion

Pathfinding algorithms play an important role in the field of Artificial Intelligence, and the Snake Game is an excellent platform for evaluating the performance of different search algorithms, including DFS, BFS, and A*. Evidently the consensus is that with a thoughtful heuristic evaluation, A* performs optimally compared to BFS and DFS. While DFS has linear space complexity, it may not be suitable for larger and complex environments. It can be modified, implementing a greedy variation to improve the speed. BFS is guaranteed to find an optimal solution, but its time and space complexity may become impractical in certain scenarios. A* has emerged as the most efficient algorithm due to its heuristic evaluation, and overall efficiency. Based on the knowledge gained from

this literature review, it is probably that BFS will perform optimally compared to DFS, and that A* will perform similarly to BFS.

3 Methods

3.1 Overview

An implementation of the Snake game was found on github[5]. The code includes a full implementation of a 20x20 game environment as well as Breadth First Search, Depth First Search, and A* search with the Manhattan distance heuristic. To further evaluate the performance of A* search, three additional heuristic functions were implemented; Diagonal distance, Euclidean distance, and Euclidean Squared distance.

BFS uses a queue data structure to keep track of spaces within the search space. The algorithm operates in a while loop until the queue is empty. If the first state in the queue has not been visited before, it is added to a visited data structure. If the visited state is the goal state, the function updates the score and adds a cube to the snake, and the length of the move sequence is appended to a list data structure. If the popped state is not the goal state, the function generates its child nodes and adds these nodes to the queue if they have not been visited or added to the queue before.

Similarly, DFS works in a while loop until the stack is empty. If the popped state has not been visited, it is added to the visited set and explored. If the popped state is the goal state, the function updates the score and adds a cube to the state. The length of the move sequence is then appended to a list data structure. If the popped state is not the goal state, its child nodes are generated, and these nodes are added to the stack if they have not been visited or added to the stack before.

In the code provided, the A* algorithm is implemented with a Manhattan distance heuristic. The Manhattan distance heuristic calculates the distance between two points by summing the absolute differences between their x and y coordinates. The A* algorithm uses a priority queue initialized with the starting state and an empty list of directions. The while loop runs until the priority queue is empty. In each iteration of the loop, the node with the lowest cost (sum of the cost of the path so far and the estimated cost to the goal) is removed from the priority queue. If the node has not been visited before, it is added to the visited set. If the node is the goal state, the snake is moved according to the directions in the list, and the length of the list is appended to a list data structure. For each successor of the current node, if the child node has not already been added to the priority queue, its estimated cost is computed and added to the queue, along with the current path cost and the direction list. The formula for Manhattan distance calculation is as follows:

```
manhattanHeuristic(position):
  xy1 = position
  xy2 = tempFood.pos
  return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
```

3.2 Additional A* Heuristics

3.2.1 Diagonal Distance. The diagonal distance heuristic is used when diagonal movement is allowed. It is calculated by finding the Manhattan distance (the sum of absolute differences in x and y coordinates) plus a factor that takes into account the cost of diagonal movement. The diagonal distance heuristic is often used in grid-based environments such as video games and robotics. When the cost of diagonal movement is the same as the cost of vertical or horizontal movement, the diagonal distance heuristic can be simplified to use the Chebyshev distance, which is simply the maximum of the absolute differences in x and y coordinates [10].The formula that we used for this heuristic was [10]:

```
function heuristic(node) =
  dx = abs(node.x - goal.x)
  dy = abs(node.y - goal.y)
  return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

where D represents the cost of moving between adjacent squares on the game board. The value of D should be chosen based on the cost and the game board and should be set to the lowest cost between adjacent squares.

3.2.2 Euclidean Distance. The Euclidean distance heuristic calculates the straight-line distance between two points. This heuristic is appropriate when the movement of the agent is unconstrained and can move in any direction. However, the Euclidean distance can sometimes overestimate the true cost of the path, especially when obstacles are present. This can lead to slower convergence times for the A* algorithm. The formula that we used for this heuristic was [10]:

```
function heuristic(node) =
  dx = abs(node.x - goal.x)
  dy = abs(node.y - goal.y)
  return D * sqrt(dx * dx + dy * dy)
```

3.2.3 Squared Euclidean Distance. The squared Euclidean distance heuristic is a variant of the Euclidean distance heuristic that uses the square of the straight-line distance instead of the actual distance. This is done to avoid the computational cost of taking the square root. However, using the squared Euclidean distance heuristic can lead to an overestimation of the true cost of the path, especially for longer distances. This can cause A* to degrade into Greedy Best-First-Search, which is a less optimal algorithm. The formula that we used for this heuristic was [10]:

```
function heuristic(node) =
  dx = abs(node.x - goal.x)
  dy = abs(node.y - goal.y)
```

return D * (dx * dx + dy * dy)

3.3 Differences Between Diagonal, Manhattan, Euclidean, and squared Euclidean distance Heuristics

The main difference between the Diagonal, Manhattan, Euclidean, and squared Euclidean distance heuristics lie in how they estimate the distance from the current cell to the goal cell in a grid-based environment.

The Manhattan distance heuristic considers only the horizontal and vertical distance between the two cells and is appropriate when the agent can only move in four directions (up, down, left, right). For example, to calculate the Manhattan distance between two points (6,1) and (9,5), you would add up the absolute differences of their x and y coordinates: |6-9| + |1-5| = 3 + 4 = 7.

On the other hand, the diagonal heuristic calculates the distance between two points as the maximum of the absolute differences of their x and y coordinates. For example, the diagonal distance between points (6,1) and (9,5) would be the maximum of |6-9| and |1-5|, which is 4.

The Euclidean distance for these points is calculates to be sqrt((6 - 9)*(6 - 9) + (5 - 1)*(5 - 1), which works out to be 5. The Squared Euclidean distance is therefore 25. In the above example, it is easy to see how the Squared Euclidean distance may overestimate the true cost of the path. The choice of heuristic is highly dependent on the nature of the problem and the agent's movement capabilities.

4 Experiment

The aim of the experiment is to use the previously discussed search algorithms to solve the snake game and analyze the performance of the search algorithm based on the time to completion, number of actions needed to reach the goal, and two calculated scores. The first score will be calculated from the following parameters, ((number of food items eaten)/(number of actions taken)) * 100. The second score will be calculated similarly, except the time taken to eat the food will be accounted for, ((number of food items eaten)/(time)).

The initial state of the game consists of the snake having a body size of one cube appearing at a position on the grid, in this case (0,0), with a body size of one cube, as well as a fruit at another position on the grid.

It is important to ensure that the state space is consistent for each search algorithm to reduce potential bias in the results. To guarantee a consistent state space, the positions of the goal states are hard coded such that the food will appear in the same spot for each level during the run of each algorithm. A total of 400 random positions for the food are stored in a constant array. Each algorithm will iterate over the 400 positions and conduct their search.

Another important factor to consider when thinking about reducing potential bias in the state space between algorithms is that the snake should always start its search from the same state across algorithms. This functionality is already built into the code such that after each algorithm is finished all 400 searches, a reset function is called that restores the game to its original state.

The algorithms are called sequentially, operating for as many iterations as there are food positions. The time for each search is recorded from the beginning of the first iteration until the end of the last iteration. Within the implementation of each search algorithm, there is a counter variable that is incremented whenever an action is taken. The total actions taken over the course of 400 searches will be stored at an index in a global array corresponding to the order the algorithms are run in. This will allow the actions required for each algorithm to be easily accessible for score calculation. This process will be repeated 10 times with the final results being presented as averages across all 10 trials.

5 Results

The results generated by the experiment are present in the figures below. The average number of actions taken by each algorithm and the average time taken by each algorithm is presented as a table in figure 1. The Food per Action score is presented in a bar graph in figure 2 and the Food per Time score is presented as a bar graph in figure 3.

Average Actions and Time

Algorithm	Actions	Time (s)
BFS	937	18.01
DFS	4461	46.12
A* Manhattan	1191	21.05
A* Diagonal	1203	21.59
A* Euclidean	1401	22.23
A* Euclidean Squared	1581	23.33

Figure 1. Figure 1. Table of average actions taken and time for algorithms BFS, DFS, A* with Manhattan Heuristic, A* with Diagonal heuristic, A* with Euclidean heuristic, and A* with Euclidean squared heuristic

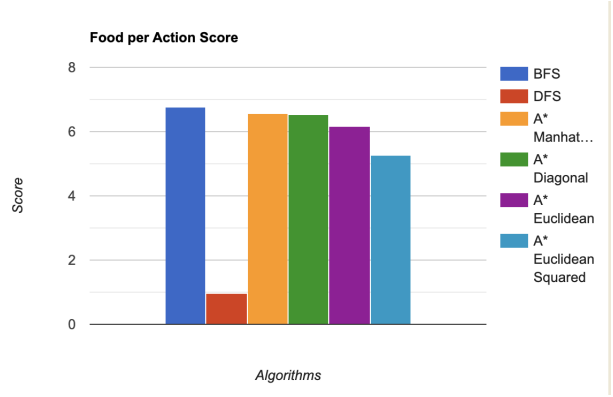

Figure 2. Figure 2. Bar graph of the average food items consumed per average actions taken score for algorithms BFS, DFS, A* with Manhattan Heuristic, A* with Diagonal heuristic, A* with Euclidean heuristic, and A* with Euclidean squared heuristic

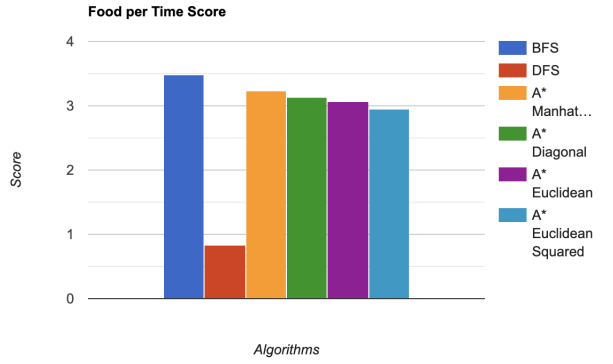

Figure 3. Figure 3. Bar graph of the average food items consumed per average time taken score for algorithms BFS, DFS, A* with Manhattan Heuristic, A* with Diagonal heuristic, A* with Euclidean heuristic, and A* with Euclidean squared heuristic

6 Analysis

The program was executed a total of 10 times and gathered data to compute the average performance metrics for the algorithms being studied. We evaluated the performance of DFS, BFS, and A* with the Manhattan heuristic, Diagonal heuristic, Euclidean heuristic, and Euclidean squared heuristic. We evaluated the algorithms based on four metrics: time to completion, the number of actions needed to reach the goal, and two calculated scores. The first score, the food per actions score is based on the number of food items eaten divided by the number of actions taken to reach the goal and the second score, the food per time score, is based on the number of food items eaten divided by the time it took to reach the goal.

The first metric to examine is the number of actions taken by each algorithm to reach the goal. The results show that on average, DFS required the highest number of actions (4461), while BFS required the lowest number of actions (937). A* with the Manhattan heuristic and A* with the diagonal heuristic required 1191 and 1203 actions, respectively. It appears that when A* was implemented with the Manhattan and diagonal heuristics a higher number of actions were required that BFS but a lower number of actions were required than by DFS. A* with Euclidean heuristic and A* with Euclidean squared heuristic required 1401 and 1581 actions, respectively, which are higher than the number of actions required by A* with the Manhattan and diagonal heuristics but still lower than DFS.

The next metric to examine is the food per action score, which is the number of food items eaten (goal states reached) divided by the number of actions required to reach the goal. BFS received the highest score (6.78), indicating that it required fewer actions than the other algorithms to reach the goal. A* with the Manhattan heuristic and A* with the diagonal heuristic had similar scores (6.57 and 6.55, respectively). This is not surprising considering both A* with the Manhattan heuristic and diagonal heuristic required more actions to reach the goal state than BFS. A* with the Euclidean and Euclidean squared heuristics had scores of 6.16 and 5.26, respectively, which is also not surprising based on the number of actions both algorithms required to reach the goal state. Overall, all the above algorithms performed better than DFS, with DFS receiving a score of 0.965.

The results of the algorithmic analysis show that the BFS algorithm achieved the best performance in terms of time taken to reach the goal. The average food per time score for the BFS algorithm is 3.491, with the average time being 18.02 seconds. The A* algorithms took slightly longer, with the Manhattan distance heuristic achieving a score of 3.241 and an average time of 21.92 seconds, the diagonal distance heuristic scoring 3.141 and an average time of 22.62 seconds, the Euclidean distance heuristic averaging 3.071 and an average time of 23.57 seconds, and the Euclidean distance squared heuristic achieving an average score of 2.956 26.78 seconds. The DFS algorithm took the longest time to reach the goal, with an average score of 0.837 and an average time of 41.87 seconds.

In conclusion, the results show that each algorithm has its strengths and weaknesses. Overall on average, BFS required the fewest actions and the least amount of time to reach the goal state, allowing BFS to receive the highest scores for both food per time and food per actions. A* performed the best on all metrics when implemented with the Manhattan distance heuristic function. When A* was implemented using the diagonal distance heuristic function performed comparably to A* with the Manhattan heuristic. While A* with the Manhattan and Diagonal heuristics performed similarly, A* with Euclidean heuristic and A* with Euclidean squared heuristic

performed worse in comparison. A* implemented with the Euclidean heuristic performed better on all metrics than A* with Euclidean squared heuristic. This is not surprising as the Euclidean squared heuristic produces a larger estimation, which can easily lead to an unnecessarily large overestimation of the true distance. DFS performed the worst overall on all metrics. In a game such as the Snake game where the state space is relatively large, the nature of DFS does not typically allow the most efficient path to the goal to be found.

Overall, the best algorithm depends on the specific requirements of the problem. In the instance of the Snake game, if the goal is to minimize the number of actions required to reach the goal or the time it takes to reach the goal, then BFS is the best algorithm. That being said, A* when implemented with the Manhattan heuristic and the diagonal heuristic performed very similarly to BFS in terms of time taken and actions required, so they too would also be suitable algorithms to choose for this problem. In instances where time to completion and number of actions required do not need to be optimal, A* with the Euclidean heuristic would be an appropriate choice of algorithm. However, the A* algorithms when implemented with Euclidean squared distance heuristic is evidently not the best choice, as it often overestimates the true distance. While DFS has real world applications outside of the Snake game, it does not appear to be a good choice for a pathfinding algorithm in this game.

7 Conclusion

The snake game is a classic game that has been used as a benchmark for evaluating search algorithms. In this analysis, we examined the performance of six search algorithms - DFS, BFS, A* with Manhattan heuristic, A* with diagonal heuristic, A* with Euclidean heuristic, and A* with Euclidean squared heuristic - based on the number of actions required to reach the goal, and the time to reach the goal, and the score of those parameters.

In conclusion, we analyzed the performance of DFS, BFS, and A* with four different heuristics, namely Manhattan, Diagonal, Euclidean, and Euclidean squared, on solving the snake game. We have evaluated these algorithms based on three performance measures, namely time to completion, number of actions needed to reach the goal, and the score. We have found that the BFS algorithm outperforms all other algorithms based on these measures, as it required the least number of actions and the shortest time to reach the goal, and the score of the BFS algorithm was also the highest in the case of the time and the number of actions. The A* algorithms performed almost as well as the BFS algorithm, while the DFS algorithm performed the worst in all metrics.

To make further progress, we would like to explore different heuristics and optimization techniques for the A* algorithm. We can also evaluate the algorithms' performance on larger game boards or more complex games to test their scalability. Moreover, we can explore the performance of different algorithms based on their memory usage and evaluate their efficiency in terms of memory usage. This investigation can provide valuable insights into finding the best algorithm for memory usage. Based on the research papers we have read, we believe DFS can be a good candidate for the memory-efficient algorithm. In addition, adding walls to the game can add an interesting element to the evaluation and possibly yield unexpected results. Additionally, Future work could focus on improving the heuristics used by A* algorithms to further optimize their performance. Another direction for future work is to explore the use of other search algorithms, such as iterative deepening search or greedy best-first search, to see if they can achieve better performance than the algorithms studied here. Additionally, incorporating machine learning techniques to learn and optimize the heuristics used by the A* algorithm could be explored. The problem of using search algorithms to solve the snake game highlights the importance of pathfinding in game development. Therefore, this problem can be used as a platform for learning and practicing search techniques. Finally, studying the performance of the search algorithms on more complex snake games with multiple snakes or obstacles would be an interesting direction for future work.

8 Contributions

Evelyn: I wrote half of the project proposal (WA3) and wrote half of the literature review (WA4). For this final report I wrote the abstract, introduction, results and experiment section. I wrote 3.1 of the methods section. For the programming portion of the project, I helped to implement one of the score calculations and the timers. For the writing assignments I formatted the document.

Matin: I focused on analyzing the data (Part 6) that collected and drew conclusions(Part 7) based on the results. Additionally, I researched and wrote about the different variations of the A* algorithm, including the Manhattan, Diagonal, Euclidean, and Squared Euclidean heuristics(part 4.2.1, 4.2.2, 4.2.3, 4.3,). In collaboration with my colleague Evelyn, we split the workload for Assignments 3 and 4. For Assignment 3, I wrote half of the article, and Evelyn wrote the other half. For Assignment 4, we each read and analyzed five articles related to our project, and i focused on the A* algorithm and Evelyn focused into the DFS and BFS algorithms in the Snake Game. In terms of coding, Evelyn and I worked together as a team to implement our findings and create an effective program, and new heuristics to A*, added new functions for calculating the time for each algorithm and captured the output (like the time, number of actions, score based of the action and the time) of each run into a .txt file.

References

[1] Tahani Q. Alhassan, Shefaa S. Omar, and Lamiaa A. Elrefaei. 2019. Game of Bloxorz Solving Agent Using Informed and Uninformed Search Strategies. *Procedia Computer Science* 163 (2019), 391–399. https://doi.org/10.1016/j.procs.2019.12.121 16th Learning and Technology Conference 2019Artificial Intelligence and Machine Learning: Embedding the Intelligence.

[2] Naga Sai Dattu Appaji. 2020. Comparison of Searching Algorithms in AI Against Human Agents in Snake Game. *International Journal of Computer Science and Mobile Computing* (2020).

[3] Ramin Banerjee, Chakraborty and Satti. 2018. Space Efficient Linear Time Algorithms for BFS, DFS, and Applications. *Theory of Computing Systems* 62, 1 (2018).

[4] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. 2021. A Systematic Literature Review of A* Pathfinding. *Procedia Computer Science* 179 (2021), 507–514. https://doi.org/10.1016/j.procs.2021.01.034 5th International Conference on Computer Science and Computational Intelligence 2020.

[5] Ben Greenfield. 2021. SearchSnake. https://github.com/BenGreenfield825/SearchSnake.

[6] Princess Chinemerem Iloh. 2022. A COMPREHENSIVE AND COMPARATIVE STUDY OF DFS, BFS, AND A* SEARCH ALGORITHMS IN A SOLVING THE MAZE TRANSVERSAL PROBLEM. *International Journal of Social Sciences and Scientific Studies* 2, 2 (Apr. 2022), 482–490. https://ijssass.com/index.php/ijssass/article/view/54

[7] Shu Kong. 2013. Automated Snake Game Solvers via AI Search Algorithms. *IEEE* 5, 3 (2013).

[8] Daohong Liu. 2023. Research of the Path Finding Algorithm A* in Video Games. *Highlights in Science, Engineering and Technology* 39 (2023).

[9] Md. Sazzad Mahmud, Ujjal Sarker, Md. Monirul Islam, and Hasan Sarwar. 2012. A Greedy Approach in Path Selection for DFS Based Maze-map Discovery Algorithm for an autonomous robot. In *2012 15th International Conference on Computer and Information Technology (ICCIT)*. 546–550. https://doi.org/10.1109/ICCITechn.2012.6509798

[10] Amit Patel. 2020. A*'s Use of the Heuristic. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[11] Arifitama Syahputra Permana, Bintoro. 2018. Comparative Analysis of Pathfinding Algorithms A*, Dijkstra, and BFS on Maze Runner Game. *International Journal Of Information System Technology* 1, 2 (2018).

[12] S Pooja, S Chethan, and C V Arjun. 2016. Analyzing uninformed search strategy algorithms in state space search. In *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. 97–102. https://doi.org/10.1109/ICGTSPICC.2016.7955277

[13] Shubham Sharma, Saurabh Mishra, Nachiket Deodhar, Akshay Katageri, and Parth Sagar. 2019. Solving The Classic Snake Game Using AI. In *2019 IEEE Pune Section International Conference (PuneCon)*. 1–4. https://doi.org/10.1109/PuneCon46936.2019.9105796

[14] Hua Shi. 2011. Searching algorithms implementation and comparison of Eight-puzzle problem. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 2. 1203–1206. https://doi.org/10.1109/ICCSNT.2011.6182175