

BSC (HONS) COMPUTER SCIENCE

COURSEWORK

Subject : Artificial Intelligence

Subject code : CSC 3206

Lecturer name: Dr. Richard Wong Teck Ken

Due Date: 21st May 2021

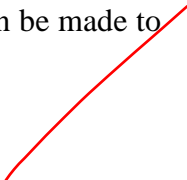
NAME	STUDENT ID
Nicholas Lee Choon Sin	18073957
Cheong Keng Son	18074229
Adrian Chin Liansheng	18063685
Ng Wei Jinn	18064154

1. Introduction

Snake is a timeless video game genre which originated from the 1976 arcade game “Blockade”, developed by Gremlin (hmdglobal, n.d.). After its initial success, many variants of Snake were created. However, a resurgence in popularity began when a variant was preloaded onto Nokia phones in 1998. Subsequently, Snake is traditionally played by a single human player. The objective of the game is to control the snake to eat as many apples as possible, while the snake’s body length increases the more apples it consumes. The player can move the snake in 4 directions - up, down, left and right, relative to the direction of the snake’s head. Consequently, the game ends when the snake collides with a wall or its own body.

Although Snake is traditionally played by humans, it is also a popular game to test and learn about Artificial Intelligence (AI) algorithms, specifically search algorithms. Therefore, this report will implement uninformed and informed search algorithms. The search algorithms used respectively are breadth-first search and greedy best-first search. In summary, breadth-first search explores all neighbour nodes of nodes in the current depth before moving onto the nodes at the next depth level. Contrarily, greedy best-first search selects the best path to expand in the moment based on a calculated heuristic value.

In this report, Section 2 will define the problem description and the problem formulation. Next, Section 3 will focus on elaborating the implementation of the search algorithm with regards to the discussed problems. Section 4 will then display the results of our AI, proceeding with an analysis segment. Following will be Section 5, a discussion pertaining to problems encountered and future improvements. Finally, a conclusion segment in Section 6 will wrap up our findings and suggest future improvements that can be made to our algorithm.



2. Problem

In this section, a detailed description of the snake implementation and the problem formulation will be provided.

2.1. Description

The snake game will be played on a square grid-like maze, with a dimension of 10 rows and 10 columns. The snake body will be represented with a pink hue, starting from magenta at the head to a light pink at the tail, whereas the apples are green in colour. The snake will move around the board, attempting to eat as many apples as possible without hitting a wall or biting its own body. However, if the snake does any from the latter, it will die and the final score will be taken to measure algorithm performance.

Consequently, the following challenges were imposed to ensure that the search algorithm meets a certain standard.

- Challenge 1: One food at any time; Non-increasing snake length; Snake length of one; Reaching at least 15 points
- Challenge 2: One food at any time; Increasing snake length with food; Starting snake length of one; Reaching at least 10 points
- Challenge 3: Two food generated when no food is available on the maze; Increasing snake length with food; Starting snake length of one; Reaching at least 10 points

The degree at which these challenges were completed will be addressed in Section 4 of this paper.

As a result, depending on which challenge the algorithm attempts, it would impose additional rules while testing the algorithm. Notably, the snake's length can either be static or dynamic. In this report, if the snake's length is referred to as static, it means it does not increase as the snake consumes the apples. Vice versa, if referred to as dynamic, the snake's length increases by 1 for each apple consumed. Furthermore, the number of apples that are generated can vary, giving the potential for there to be more than one goal state at a given time.

2.2. Formulation

2.2.1. Initial State

Position of snake	Maze(0, 5)
Direction of snake	Facing east
Length of snake	1 box
Food location	Randomly spawns within the grid, except for position Maze(0, 5)

Table 1. Initial state of a new snake game

2.2.2. Actions

{North, South, East, West}

2.2.3. Transition Model

In this instance, a function called Move(state, action) can be used to obtain the new position of the snake in the maze. For example, Move(Maze(0,5), South) will return the state Maze(0, 6).

Specifically, each action will add the following values to the given state to produce the resultant state.

North	[-1, 0]
South	[+1, 0]
East	[0, -1]
West	[0, +1]

Table 2. Actions and its corresponding values

However, the transition model will not return a resultant state if the following conditions are met.

- Condition 1: The resultant state is a part of the snake's body
- Condition 2: The resultant state is out of bounds

This is because the above conditions will return an unreachable state. Therefore, the model should not return the state.

2.2.4. Goal Test

Check whether the current state is the location of the food.

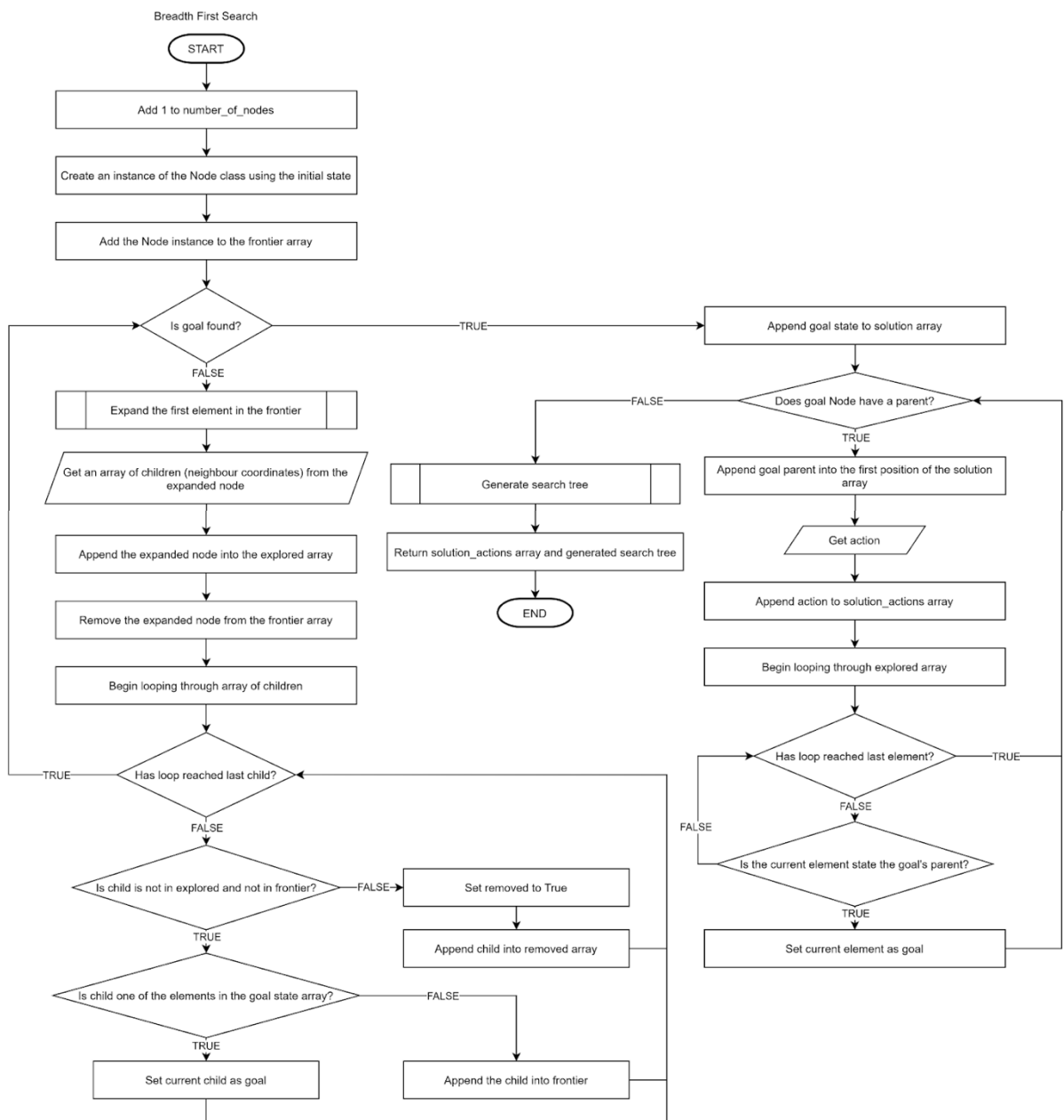
2.2.5. Path Cost

Each step costs 1 box. Therefore, the path cost is the total number of steps taken to complete the operation.

3. Implementation

This section will discuss the implementation of our search algorithms by visualizing flowcharts and explaining them.

3.1. Breadth-First Search



Flowchart 1. Breadth-First Search

The breadth-first search algorithm aims to traverse a graph and obtain the shortest path possible. Its traversal strategy expands nodes in a breadthwise manner, meaning that the expansion starts from the root node, continuing with the root node's successors, followed by their successors, and so on. Subsequently, the goal test is completed after node generation. Hence, this search algorithm can be broken down into 3 main parts - the Node class, the expansion function, and the searching algorithm.

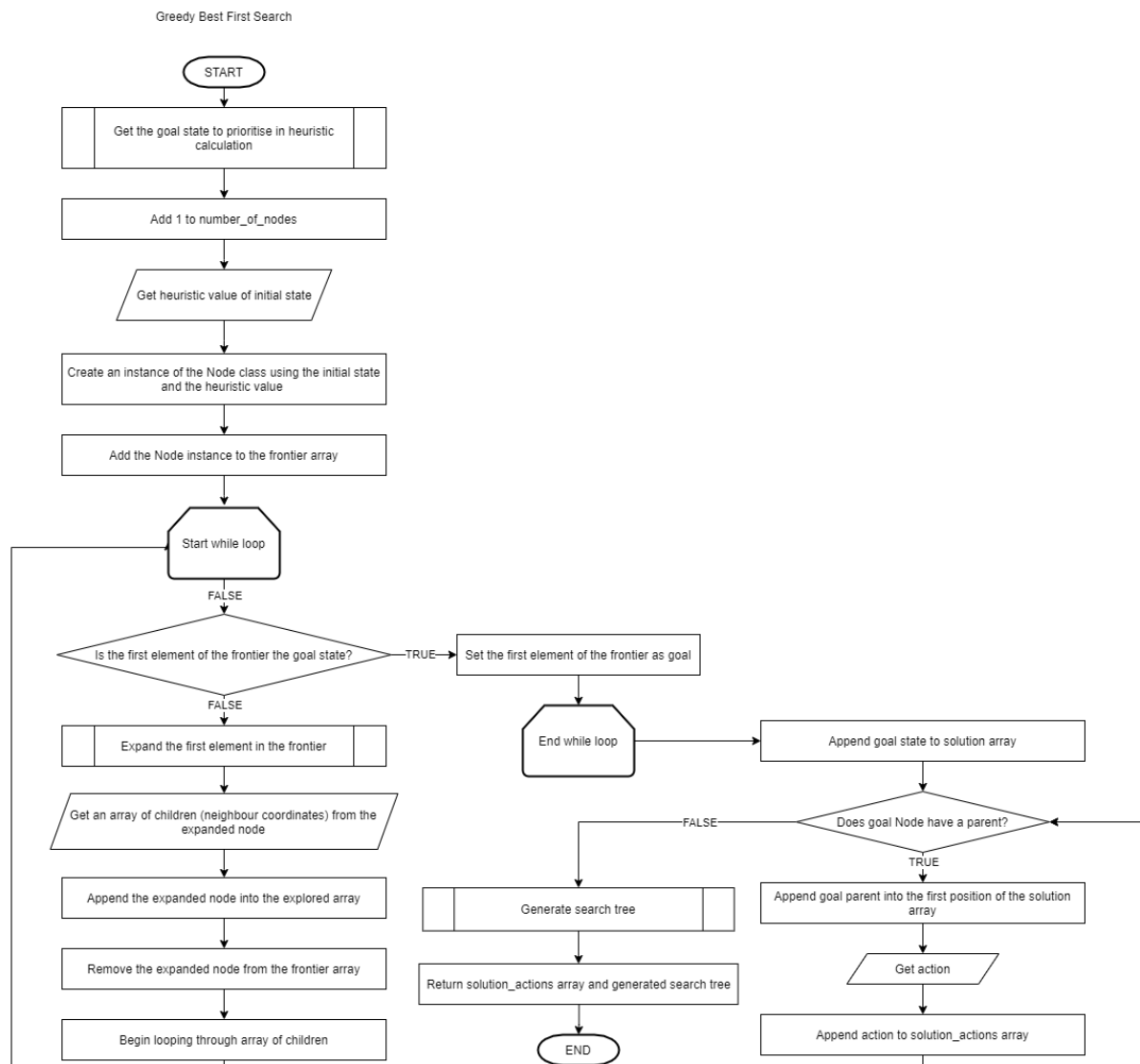
Firstly, the Node class is used to store information about the Nodes. Each Node class will store the following information: the state, the parent state and the children states. These 3 pieces of information are extremely crucial to the implementation of the search algorithm.

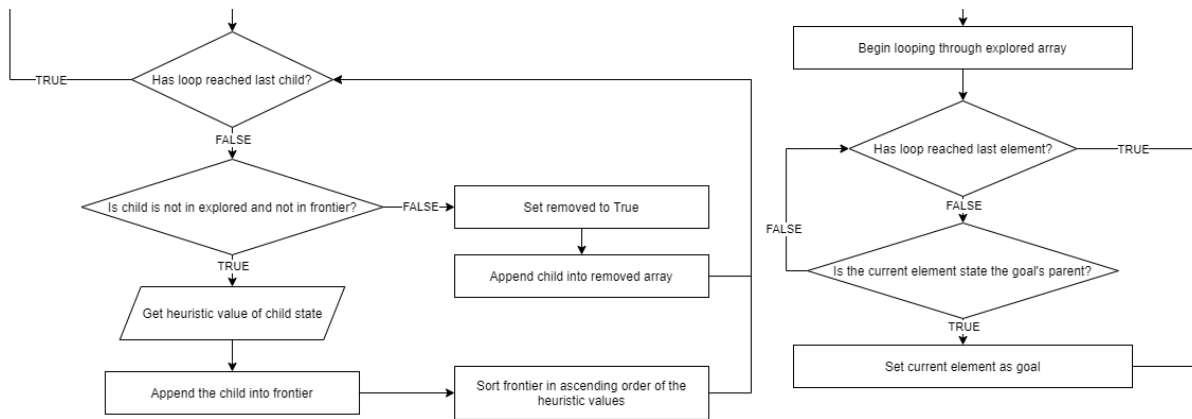
Next, the node expansion function will return the children of a given state. In this instance, the children are the neighbouring nodes of the state. Therefore, when a state is passed into the expansion function, it will use the transition model mentioned in Section 2 Part 2.2 to return the state's north, south, east and west nodes. Furthermore, this function implements the condition checking of the transition model. The function would not consider the nodes that are in the snake's body or out of maze bounds as a legitimate child node. Consequently, after the processing, an array of possible children Node classes will be returned.

Lastly, the search algorithm primarily operates with lists called frontier and explored. When the algorithm is first executed, it will take the initial state and make it a Node class with a parent value of None, signifying that it is the root node. This Node is then added to the frontier. Next, the algorithm will take the first element of the frontier and call the previously explained expansion function to obtain the state's children. Once the expansion is complete, the Node is removed from the frontier and subsequently added to the explored list. Then, each generated child will be checked to see whether it is already in the frontier or explored lists. If it is not, a goal test will be performed and the node will be added to the frontier. Otherwise, it would be ignored as it is a loopy path. The process will be repeated for each element in the frontier list until the goal state has been found or the frontier list becomes empty.

Once the goal state is found, it will take the goal state and search through the explored list, tracing back its steps using the parent attribute, until it meets a node with the parent value None. Simultaneously, the action will be retrieved by subtracting the coordinates of the node state with the coordinates of its parent state. Finally, the action can be inferred from the results and added to an actions array.

3.2. Greedy Best-First Search





Flowchart 2. Greedy Best-First Search

The greedy best-first search algorithm works with a heuristic function - $h(n)$. The algorithm aims to expand the node with the lowest $h(n)$ value. This expansion process is repeated, with the goal test being performed on node expansion. As a result, this search algorithm can be split into 4 parts - the Node class, calculating the heuristic function, the expansion function and the searching algorithm. However, the expansion function is identical to the function used in the Breadth-First Search algorithm.

Firstly, the calculation of the heuristic function is the most crucial part of this search algorithm. In theory, a heuristic function is used to estimate the distance between the current node and the goal node. Hence, the distance that we have chosen to estimate in our algorithm is the Manhattan distance. The Manhattan distance is calculated as the sum of the absolute differences between two vectors. Figure 1 below visualizes the Manhattan distance on a grid.

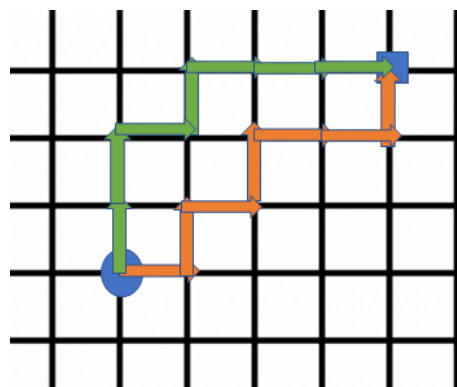


Figure 1. Manhattan Distance

The first vector will be the current node and the second will be the goal node. However, Challenge 3 poses a problem as there are multiple goal states. To overcome this problem, the algorithm implemented a goal state prioritisation function, whereby it ensures that the closer goal state is used in the heuristic calculation. The closer goal state is the goal state with the lower Manhattan distance from the initial state.

As a result, the Node class has been altered to fit the Greedy Best-First Search algorithm. In addition to the 3 attributes used by the Breadth-First Search, an additional h variable is required for Greedy Best-First Search. The h variable will store the $h(n)$ value of the node returned from the heuristic function.

Lastly, the search algorithm's implementation differs in these areas compared to Breadth-First Search: the processing for each generated child; the order of elements in the frontier; and the location of the goal test. Firstly, after expanding the children, the program will still check whether the node exists in the frontier or explored lists. If it is not, the algorithm no longer performs the goal test here. Contrarily, it will calculate the heuristic value using the function, assigning its return value to the h attribute of the child node. Next, the order of elements in the frontier will be sorted in ascending order of the node's h cost after each generated child is processed. This is to ensure that the first node expanded has the lowest h cost. Lastly, the goal test is moved prior to the expansion of the first node in the frontier. Similarly, the process will be repeated for each element in the frontier list until the goal state has been found or the frontier list becomes empty. The action retrieval process after the goal state has been found is identical to the process used in Breadth-First Search.

3.3. Exception Handling

So far, there has been one outcome that has not been explained - the frontier list becoming empty. When this occurs, it means that the algorithm has failed to find a solution to the problem presented. There are many ways of implementing a workaround, such as forward-checking. However, that would unnecessarily increase the complexity of the algorithm. As a result, the workaround our algorithm employs is shown through the pseudocode below.

```

try:
    bfs = BFS(snake_body, food_locations, maze_dimensions)
    solution, search_tree = bfs.bfs()
catch IndexError:
    food_locations.insert(snake_body[-1])
    k_value = int(snake_body.length() / 4)
    res = snake_body. splice (snake_body. length() – k_value)
    bfs = BFS(res, food_locations, maze_dimensions)
    solution, search_tree = bfs.bfs ()

```

Pseudocode 1. Implementation of Exception Handling for Breadth-First Search

The diagram above shows that the exception is handled through a try and except block. This is because when all the nodes in the frontier are expanded and there is no solution found, it will throw an “Index out of range” error. Hence, the try and except block aims to catch this error. Next, in the except portion, the search algorithm is run once again. However, before executing the search algorithm, two key components of the search algorithm are altered. Firstly, the last node of the snake body is added as a goal state. Additionally, the final quarter of the snake’s body will be removed. This is to encourage the snake to chase after its own tail as the snake often gets trapped when its body encircles the head of the snake, even though there is ample space to avoid hitting the tail. By doing so, it increases the probability of the snake exiting a dead end and finding a new path to the food location. The effectiveness of this exception handling implementation is discussed in Section 4.

4. Results

To fairly measure the performance of the search algorithms, each method was run continuously for 50 times on both challenge 2 and challenge 3 requirements respectively. All 50 runs were completed on a fixed 10x10 maze size, on the algorithm that utilizes exception handling. Additionally, to ensure that the data was not heavily affected by variants such as the apple’s location, the average of all data was obtained. Moreover, the runs were conducted across multiple devices to ensure that the results were replicable and consistent. The various outcomes are tabulated in tables and analysed below.

	Breadth-First Search	Greedy Best-First Search
Average score	> 100	> 100

Table 3. Average scores of Challenge 1

Table 3 displays the average score of both methods executing under the rules of Challenge 1. Based on Table 3, it can be concluded that when the snake's length is static, both Breadth-First Search and Greedy Best-First Search can consistently find a path towards the food location. This is proven as the average scores across both algorithms are more than a 100. The runs were halted when the score surpassed a 100 points as the score would not stop increasing otherwise.

		Breadth-First Search	Greedy Best-First Search
Challenge 2	Average score	28.32	26.70
	Average number of steps taken	226.98	217.16
	Average number of expansions	1034.16	266.08
Challenge 3	Average score	29.12	28.04
	Average number of steps taken	202.64	216.12
	Average number of expansions	850.64	272.06

Table 4. Data recorded from Challenge 2 and Challenge 3

In Table 4, the average score, average steps, and the average number of node expansions were obtained from the 50 respective runs for both challenges. In this instance, steps are defined as the number of actions calculated to reach the goal state. Therefore, it can be concluded that both search algorithms have outperformed the requirements of Challenge 1 and Challenge 2, with average scores of more than 2.5x the original requirement. However, it can be seen that through both challenges, Breadth-First Search manages to achieve a higher average score compared to Greedy Best-First Search. However, the most notable difference is seen in the average number of expansions. To achieve a higher average score, Breadth-First Search expands more nodes. Therefore, the node expansions from the Greedy Best-First Search algorithm is 3.89 times lower in Challenge 2 and 3.13 times lower in Challenge 3.

		Without exception handling	With exception handling
Breadth-First Search	Average score	22.22	28.32
	Average number of steps taken	169.38	226.98
	Average number of expansions	819.04	1034.16
Greedy Best-First Search	Average score	21.18	26.70
	Average number of steps taken	163.88	217.16
	Average number of expansions	205.58	266.08

Table 5. Data recorded from Challenge 2 with and without exception handling

Additionally, the exception handling was vital in consistently achieving the goal as depicted in Table 5. This is because it would help the snake avoid situations where the program perceives the state space to be unsolvable, such as in Figure 2 below.

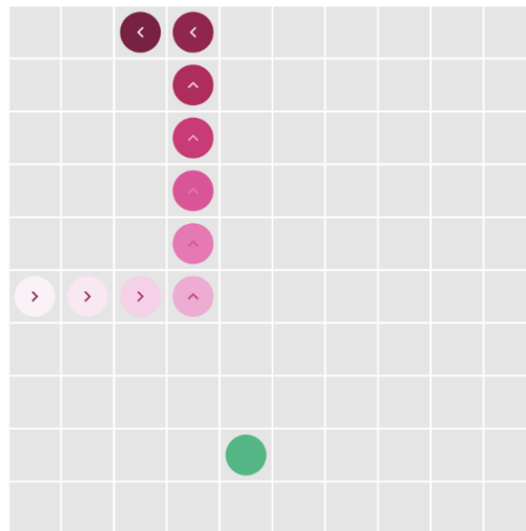


Figure 2.

4.1. Completeness

Both Breadth-First Search and Greedy Best-First Search are complete algorithms. This is because it is able to consistently find a solution when it exists. This is proven in

Table 1, as without an increasing snake length, it can always find the solution. However, for challenges 2 and 3, due to the increasing snake length, the algorithm may not find a solution even though it exists, as seen in Figure 2 above.

4.2. Optimality

With regards to optimality, there are instances where both algorithms would not find the most optimal solution. There are three notable instances where the optimal solution would not be found.

The first problem is when the snake takes a longer, less optimal route to the food. This problem persists across both algorithms, and is caused by the practical implementation which factors in the coordinates of the initial snake body. Combined with the snake being a moving obstacle as it moves 1 box every time an action is executed, the updated locations of the snake body is not taken into consideration. This leads to the solution path avoiding the previous locations of the snake body entirely, even though it is the most optimal route as there would be no snake there when the snake’s head reaches the coordinate.

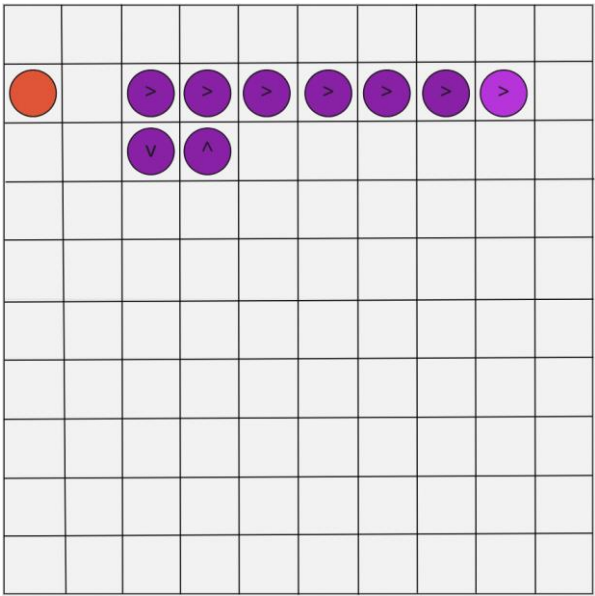


Figure 3.1.

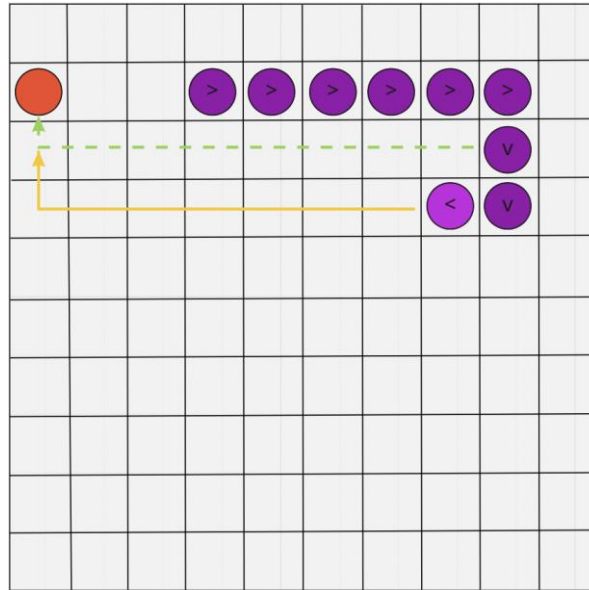


Figure 3.2.

The second issue applies to Greedy Best-First Search when facing Challenge 2. Based on Figure 4, it is illustrated that the search algorithm will opt for the longer path instead of the closer path. This is because the search algorithm expands its nodes based on the lowest h cost. This means that the algorithm does not consider the obstacle in the way. Consequently, it will wrap around the tail to avoid the obstacle. On the other hand, the green dotted line is the path that Breadth-First Search will opt for, which is the most optimal path. This is because Breadth-First Search will expand the nodes in a breadth-like manner, thus being able to find the shortest path in this situation.

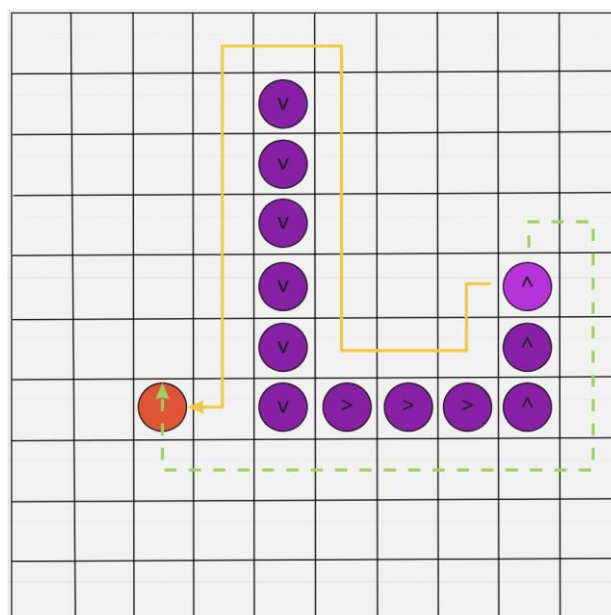


Figure 4.

Finally, another problem is encountered when Greedy Best-First Search is applied to Challenge 3. The challenge requires the program to execute with 2 pieces of food in the maze. As a result, a goal prioritization method is implemented, as justified in Section 3 Part 3.2. However, the prioritization method does not factor the snake's body when determining which goal state is closer to the snake's head. Hence, in situations depicted in Figure 5.1, the snake will end up following the route in Figure 5.2. This problem can be seen in the results shown in Table 4, under Challenge 3. Greedy Best-First Search uses more steps on average compared to Breadth-First Search. Hence, it can be concluded based on the two problems that the Greedy Best-First Search algorithm has encountered so far, that the algorithm performs extremely well when the snake body is short.

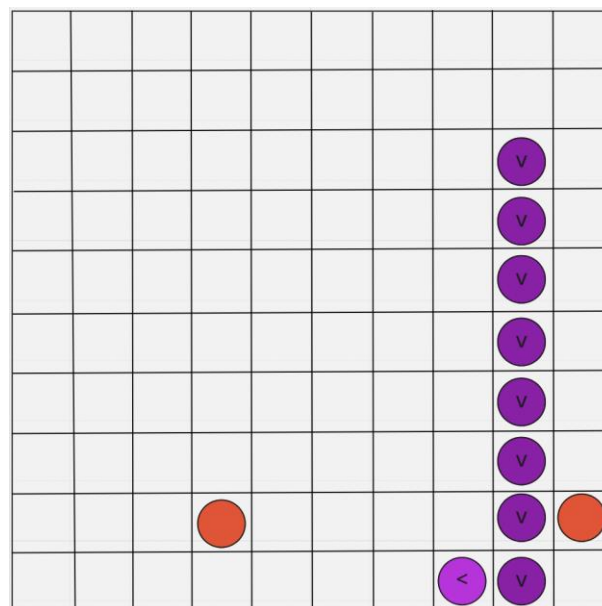
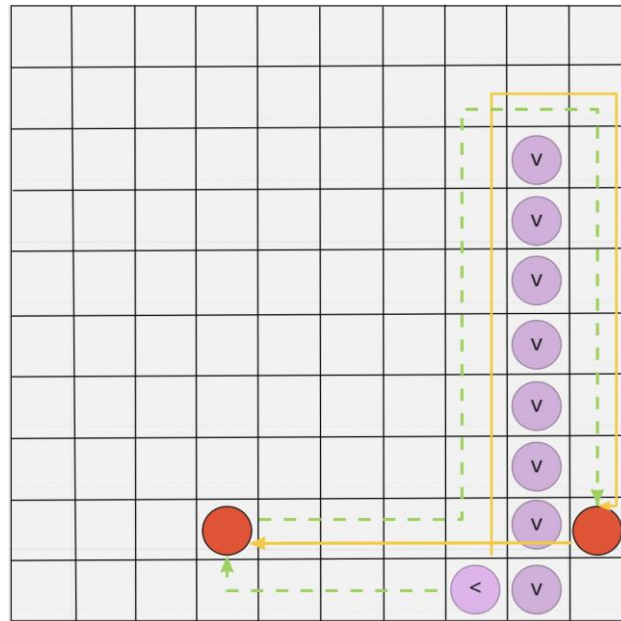


Figure 5.1.



4.3. Time and Space Complexity

Time complexity is defined as the time taken to return a solution path, given a maze state. The time was not recorded as part of the dataset, however it can still be compared as there are minor changes between the implementations of both search algorithms. The time taken to find a solution is faster using Greedy Best-First Search as opposed to Breadth-First Search. This is because Greedy Best-First Search will not explore as many nodes to find the solution, as evident by the data illustrated in Table 4. Breadth-First Search expanded almost 3-4 times the number of nodes expanded by Greedy-Best First Search.

Contrarily, space complexity is defined as the memory consumed when finding the solution. Similarly, the average number of nodes expanded in Table 4 can be used as a metric. This is because the more nodes expanded, the more memory is required to store the nodes in the frontier list for expansion. Additionally, more nodes will also be added to the explored and removed list. Hence, it can be concluded based on Table 4 that Greedy Best-First Search has a better space complexity compared to the Breadth-First Search algorithm. This is further evident by the fact that even though Greedy Best-First Search encountered an optimality problem in Challenge 3 when prioritizing the food, it still expands substantially lesser nodes, thus having a better space complexity.

A concern when implementing the exception handling was that it would affect the search algorithms performance. However, based on the results gathered in Table 4, it can be said that the exception handling enables Breadth-First Search and Greedy Best-First Search to increase their average scores by 30.5% and 27.3% respectively, with little difference in the number of nodes expanded.

5. Discussion

5.1. Dead ends

This section will further elaborate on how although both implementations consist of exception handling functions, it is still not sufficient enough to avoid unreachable goal states in certain situations. This is because the program implements a temporary solution that removes the last quarter of the snake's body for processing. In Figure 6.1 below, the left state space would be considered a dead end without exception handling. However, with exception handling, the search algorithm perceives the state space as the right state space after removing the last quarter of the body.

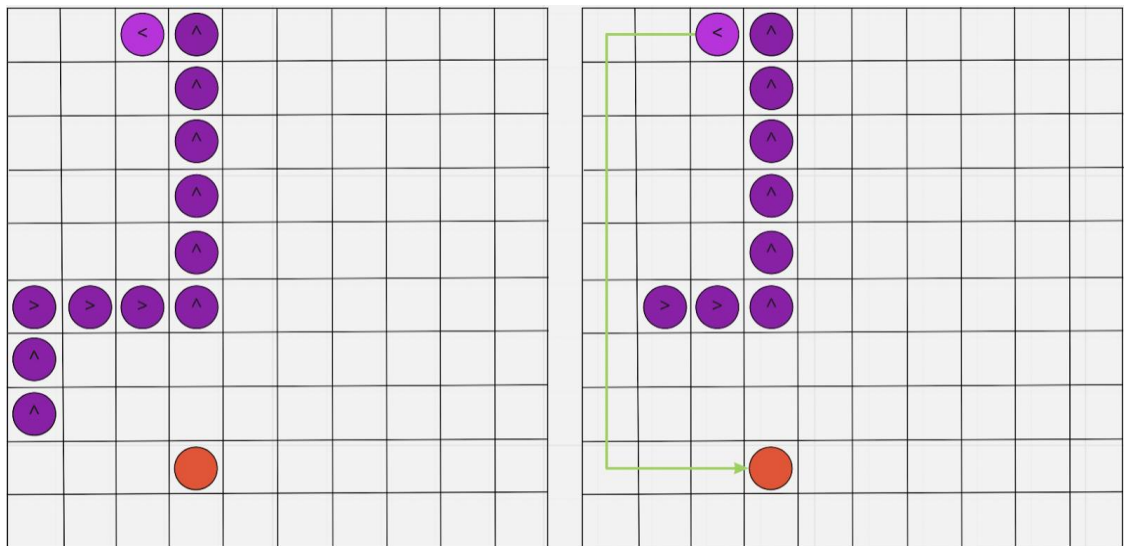


Figure 6.1.

Despite this, it is not the case for all situations. As the snake gets longer, it gets increasingly difficult to remove nodes and find a way out. Otherwise, it could be due to a bad spawn location that will cause a dead end. Take Figure 6.2.1 and Figure 6.2.2 as an example. Although the nodes have been removed, there is still no way out of the

snake. As a result, the program will stop executing due to an “Index out of bounds” error. From human perception, we can tell that Figure 6.2.1 is still a solvable puzzle and Figure 6.2.2 is a dead end. However, with the current exception handling, it is limited, and will deem both situations a dead end.

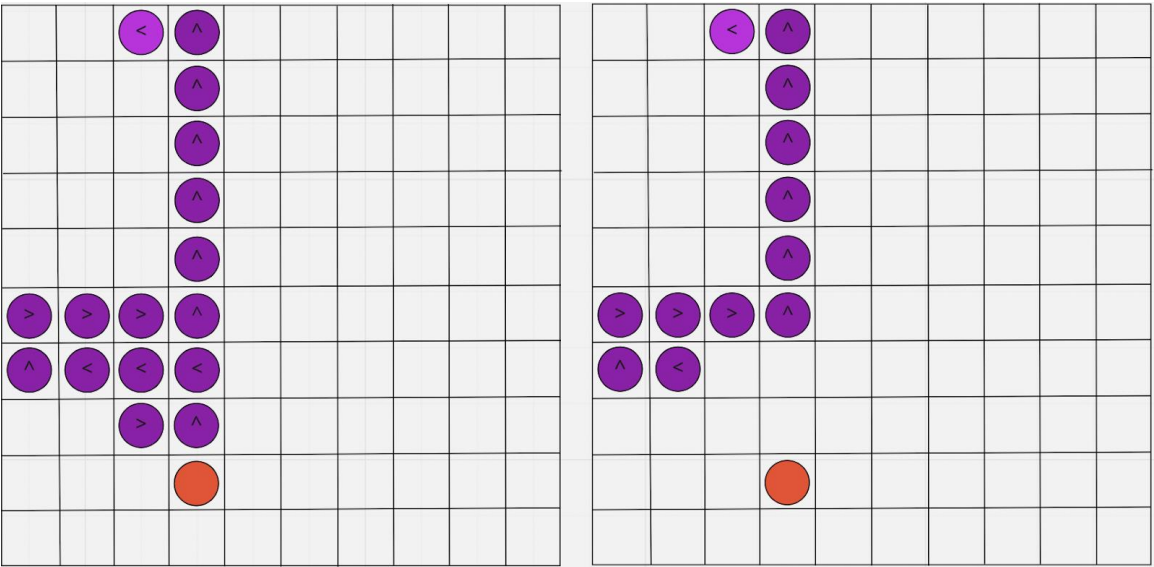


Diagram 6.2.1

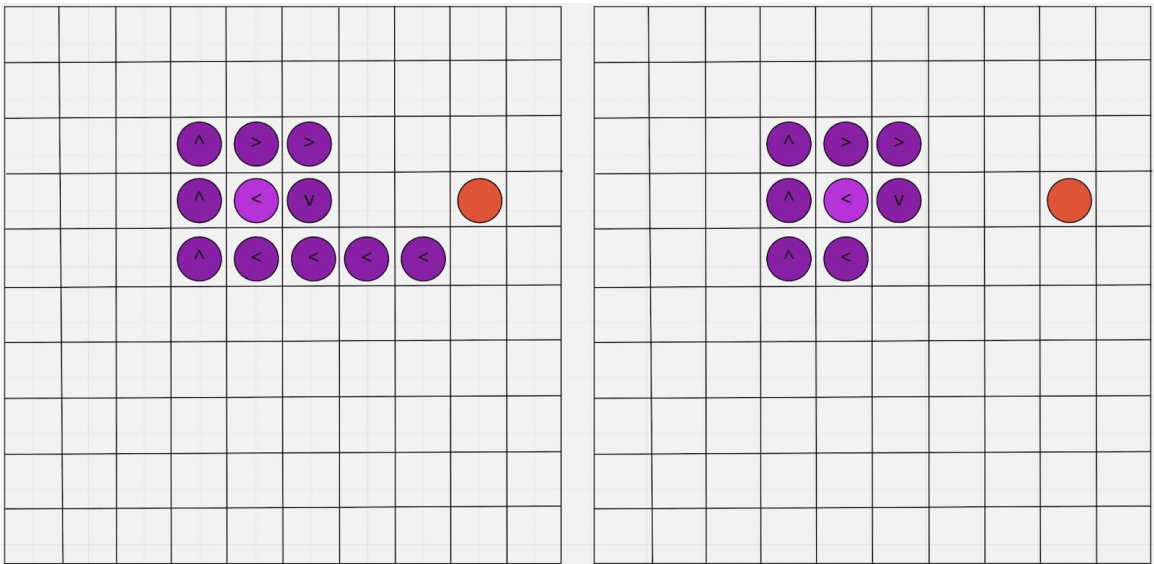


Diagram 6.2.2

5.2. Future Improvements

5.2.1. Handling complex dead ends

In the previous section, a discussion has been made on how the algorithm runs on the exception handling. Instead of using forward checking, the implementation of our exception handling does increase the average score that is able to be achieved by the snake. However, this implementation is not perfect. This is because if there are any conditions that fall outside of the exception handling, the algorithm is no longer able to find an alternative path towards the food, that is to prevent itself getting an unreachable goal state.

Moreover, our algorithm is able to further improve by implementing Forward Checking. Although we do mention that Forward Checking will increase the complexity of the algorithm, it will increase the average score by 5-10%.

5.2.2. Encountering irregular sized mazes

As mentioned in the Problem Description section, the current size of the maze is fixed to 10 rows and 10 columns. This produces a 10x10 square-shaped maze for the snake to move in. As a result, both search algorithms used are able to provide exceptional results and have met all of the requirements in this project. However, an irregular maze size, such as 5x10 or 10x5, causes complications with the algorithms. The algorithm was not able to find the appropriate path towards the goal. The snake ends up chasing its tail instead of finding the correct solution.

Therefore, an improvement to this current system is having the ability for the search algorithms to cater to irregular maze sizes set by the user. Possible enhancements can be made to prepare the algorithms for different maze sizes. Another solution for this problem is by experimenting and implementing different algorithms, such as depth-first search, in response to the irregular maze size set by the user. With this issue resolved, the search algorithms will function even with irregular maze sizes.

6. Conclusion

In this report, 2 search algorithms were implemented to play the Snake game. We also conducted multiple experiments and analysed the results to measure the performance of each algorithm. It can be concluded that the algorithms successfully solved the challenges provided. However, both algorithms completed the challenges with varying levels of performance. Breadth First-Search had an overall better score compared to Greedy Best-First Search. However, Breadth-First Search sacrificed time and space complexity for a better optimality whilst Greedy Best-First Search did the opposite. Furthermore, it was proven that Breadth First-Search would perform better in multiple food scenarios. In conclusion, both algorithms are great for solving the snake game, however the selection of which algorithm to implement depends on the use case. If the player would like to achieve a more consistent higher score with multiple food on the maze without processing and memory constraints, then Breadth-First Search is the optimal algorithm. On the contrary, if the player restricts processing and memory disregarding optimality, then Greedy Best-First Search is the best fit. Despite this, both implementations are limited by the fact that it will not execute if it reaches a dead end.

7. Reference

Iconic Snake game is back and available on Messenger. The home of Nokia phones. (n.d.). <https://www.hmdglobal.com/press-releases/iconic-snake-game-is-back#:~:text=Snake%20first%20appeared%20in%201997,developed%20and%20published%20by%20Gremlin.>
James Aubin. Snake AI. (n.d.). <https://jaubin.net/projects/snake-ai.html>.