CSC3206 Artificial Intelligence Assignment 1

Group: Old Driver

Team Members:

1. Teoh Zhan Tao (15022999)
2. Ang Yu Hern (17107202)
3. Chong Chein Yeap (17119876)
4. Eng Janson (17102047)

Introduction

In our project, Breadth-first search is selected as our uninformed search algorithm to play the snake game, whereas Greedy best-first search will be our informed search algorithm for the snake game.

Problem Statement

Breadth-first Search (BFS)

The reason we chose Breadth-first Search (BFS) for our uninformed search algorithm for the snake game is because of its completeness and optimality. The algorithm is guaranteed to find the path to the shallowest goal node, due to the depth restrictions imposed by the size of the maze. The algorithm is optimal because the step costs are all identical.

The algorithm will search for the solution with the least amount of steps required to traverse from the root node to the goal node. (In this project, the root node is the coordinates of the snake's head, and the goal node is the coordinates of one of the foods) For this specific project, the BFS algorithm also needs to be aware of the coordinates of the maze boundaries and snake's body at all times. This is because the algorithm needs to find the solution with the least steps for the snake to move, which avoids the snake bumping into itself or the boundaries of the maze. The BFS algorithm should expand the shallowest node in the search tree, until it finds the goal node during a goal test. (The goal test is performed during the generation of the node)

Once the goal node has been found, the algorithm will backtrack the steps/solutions required to traverse from the root node to the goal node. The steps/solution is then sent back to the snake game, which will allow the snake to move from its current location to the location of the food(s). Therefore, we have used breadth-first search in the snake game as the algorithm is trying to find the shortest path for the player to reach the snake food inside the maze.

Greedy Best-First Search

The algorithm will search for the solution with the least estimated cost to traverse from the root node to the goal node. (In this project, the root node is the coordinates of the snake's head, and the goal node is the coordinates of one of the foods) Like BFS, it also needs to be aware of the coordinates of the maze boundaries and the snake's body.

This is because it needs to find the solution with the least steps for the snake to move, which avoids the snake bumping into itself or the boundaries of the maze.

The estimated cost to travel from one node to another can be calculated with the help of a heuristic function. The heuristic function used here calculates the Manhattan distance between two coordinates. The Manhattan Distance can be calculated using the formula **|x1 - x2| + |y1 - y2|**, where x1,y1 are the xy coordinates of the current node, and x2,y2 are the xy coordinates of the goal node. The algorithm expands the node with the lowest estimated cost, and calculates the estimated cost to the goal node for each node that it generates. Unlike BFS, the goal test for this algorithm is done during the expansion of nodes. Manhattan distance is suitable for this algorithm as it finds the shortest path each element could take to reach the food.

Parameters used in both algorithms

Both of our algorithms take the same amount of input parameters, namely maze_size, static_snake_length, snake_locations, current_direction, food_locations.

maze_size

maze_size is a list of two integers that dictate the number of rows and columns of the maze. This input parameter is essential for the program as the algorithm will not know what the maze boundaries are if the maze size is undefined.

static_snake_length

static_snake_length is a boolean. If this parameter has a value of *true*, the snake length remains unchanged. Else, the snake length will increase by 1 every time the snake consumes a food. If static_snake_length has a value of *false*, the snake will have less available steps to the coordinates of the food, due to the movement restrictions set by the body of the snake, which causes the algorithm to process slower, whereas a snake with static length will have more available steps for the algorithm to process when compared to the snake without static length.

snake_locations

snake_locations is a list of list of integers that define the coordinates of the snake's body in the maze. This parameter is required for the algorithm to understand the starting node for the search algorithm.

current_direction

current_direction is a string that defines the direction the snake's head (first element in **snake_locations**) is currently facing. This parameter is required to allow the algorithm to understand where the snake head is facing at all times, so that the algorithm can know what available directions the snake can move in. This parameter also prevents the snake from making a u-turn by removing the opposite direction the snake head is currently facing from the available directions for the snake to move in.

food_locations

food_locations is a list of list of integers that define the locations of the locations of the food available in the maze. It is also required by the BFS algorithm during the goal test, and by the Greedy Best-first Search algorithm during the calculation of estimated cost to goal node.

Design of Uninformed search and Informed search algorithm

For breadth-first search, each node in the search tree list contains a few information, namely id, state (the coordinates of the snake's head), expansion sequence, children, parent, actions, removed (a flag to determine whether the node can be used), snake_locations. The algorithm starts off by creating the list **search_tree** for the initial node where its expansion sequence is "-1". The algorithm will quit when the list is empty, as the algorithm is unable to provide a solution and will stop the execution. The **search_tree** list is always sorted ascendingly based on the id of each node, because breadth-first search will always expand from the shallowest node. The algorithm then expands on the first element in the list and all the information of the current expanding element is being stored in variables for later use. The parent of the current element is also being identified in order to retrieve the direction of the current element based on the parent node. The algorithm will then use the information to remove the available directions of the current element so that the snake does not u-turn or bump into itself. Once the direction of the current element to step is decided, a new snake location is generated where it is a copy of the original snake location, but the snake head location (new state) is inserted into the new snake location based on the current element direction, whereas the snake tail is removed using the pop() function. The expansion

you are talking about the design of your code, not the design of algo.

sequence of the current element will also be added by 1 so that the algorithm will not expand on the same element in the next loop.

After that, the program will generate a new element with the latest information retrieved by the algorithm. The new element will include a new id, a new state which contains x, y coordinates based on the direction of the current element, an expansion sequence of -1, parent element and the new list of snake locations. The algorithm will then check if the new element has the snake collided into itself or has bumped into the maze boundaries by comparing the input parameter maze_size with the snake locations of the current element. If the new element will collide, the removed flag of the current element will be true and not be used for the solution. In our breadth-first search algorithm, there is a possibility where new elements with redundant snake coordinates may be generated and added into the search tree. Therefore, the algorithm will also check if the coordinates of the element already existed in the search tree. If yes, the removed flag of the element will become true. Then, the current element the algorithm is processing will update its available directions and its children elements.

Finally, the algorithm will perform the goal test and check if the element will reach the snake food and its removed flag is false. The algorithm will compare the snake locations of the current element with the input parameter food_locations. If yes, the program will trace back the directions used from the parents of the elements, which will be stored into the solution list. The search tree will also be displayed to show all the nodes identified. As breadth-first search only performs goal tests during the creation of nodes, the algorithm will stop executing once the solution or path is found. It will not expand or search for another shorter path in breadth-first search. Therefore, we have a found_goal as a flag in our algorithm to determine whether the solution is found where the snake's head is located at the food. Once the solution is formed, the while loop stops and the snake moves along the path according to the solution formed towards the snake food.If the goal test fails, the algorithm will loop again until a solution is found or no possible solution can be made.

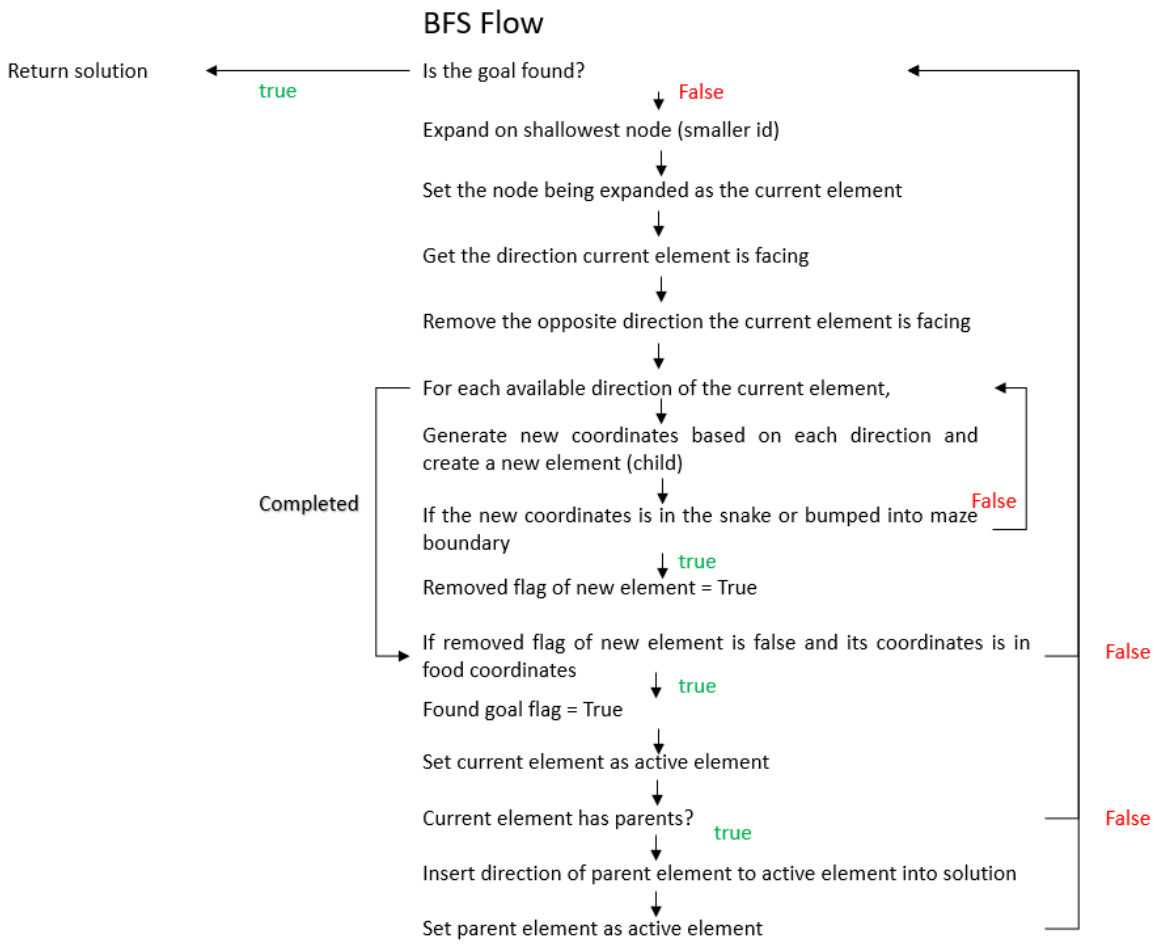

Figure 1: Breadth-first search design in our algorithm

In greedy-best first search, the design of the algorithm is similar to the one of breadth-first search. The algorithm will take Each element in the search tree includes the id, state, expansion sequence, children, action, removed flag, parent, snake locations and the estimated cost between the element and the food. Similarly, the algorithm will execute a while loop where the algorithm will keep looping until it has found the goal. It first creates a list of elements with expansion sequence as -1, and sort them based on the estimated cost ascendingly. Therefore, the algorithm will always expand on the element with the least estimated cost.

After that, the algorithm will store the information of the current expanding element in local variables for later use. Available directions of the current element will also be determined based on the direction the current element is facing. For example, the north direction will be removed from the available directions if the current element is facing

south. The algorithm will then increase the expansion sequence of the expanding element by 1.

One of the differences between breadth-first search and greedy best-first search is that breadth-first search will perform the goal test during the generation of a node, whereas greedy best-first search only performs the goal test during the expansion of a node. Therefore in greedy best-first search, the goal test is performed before the algorithm has actually generated the new node. Greedy best-first search also only expands on the element with the minimal estimated cost to the food, hence there is no need to check if there are redundant elements in the search tree. In the goal test of our greedy best-first search, the algorithm will check if the estimated cost from the snake head to the coordinates of the food is 0. If yes, the algorithm will perform a while loop and insert the current element into the solution list, then loop again by inserting the parent element until it reaches the root element. After the goal test, the algorithm finds the available directions of the current node and generates new elements with new coordinates based on the available directions. The algorithm also checks if the elements with newly generated coordinates have either collided into its own body or the maze boundaries. Finally, the parent of the current element will update its children and available directions by using the information retrieved from the current element.

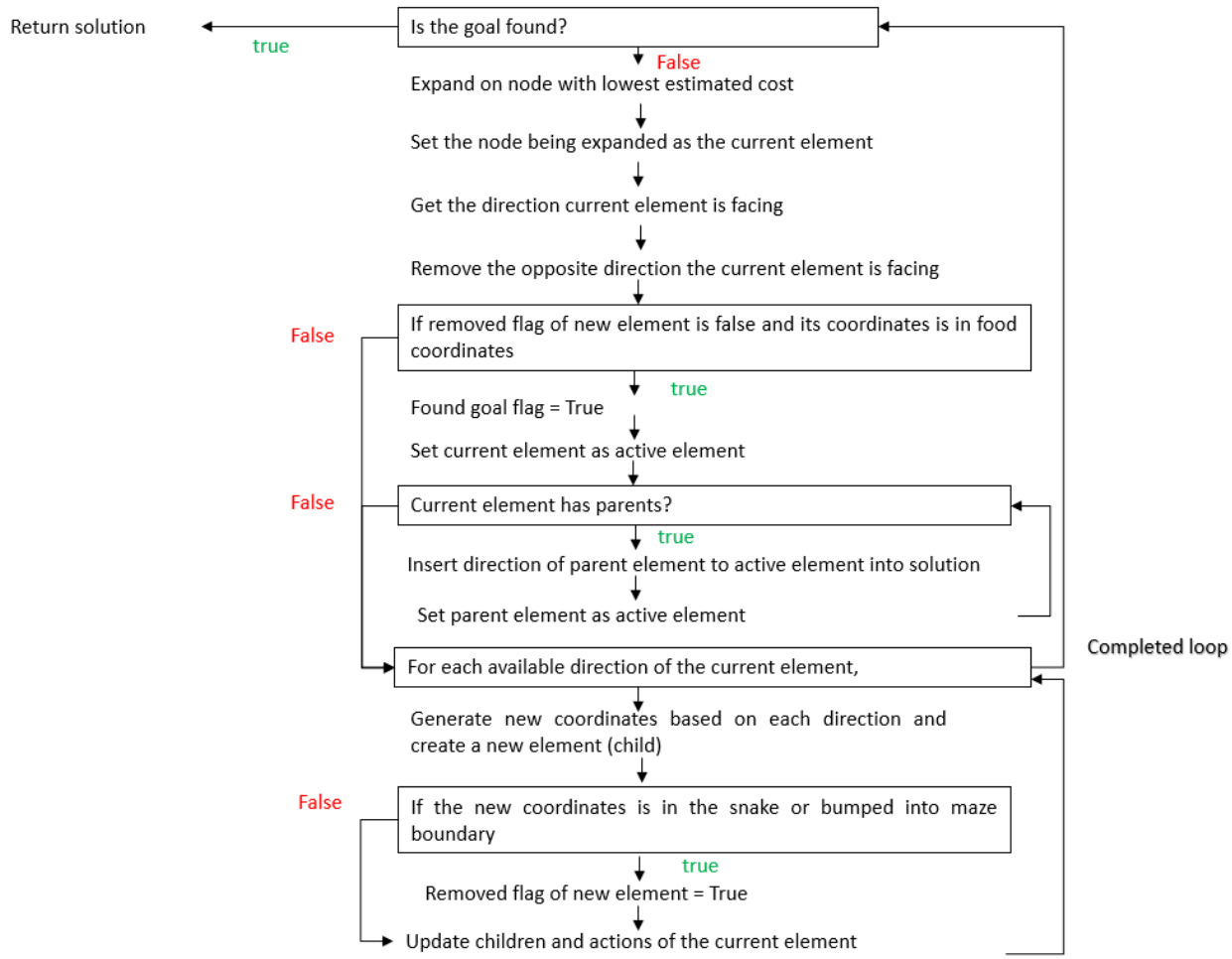

Figure 2: Greedy best-first search design in our algorithm

Outcome of both algorithms

Both the breadth-first search and greedy best-first search algorithms in our program managed to search for the food and obtain the solution path.

Both algorithms are able to reach at least 15 points when the snake has a static snake length and starts with a length of 1. It can easily achieve at least 15 points in this challenge as the algorithm only has to be aware of the coordinates of a singular food, and the boundaries of the maze. The snake body is not considered in this game setting as the snake will never bump into itself due to its static length. For the second challenge where the snake length will increase with food, both algorithms are also able to achieve at least 10 points. Unlike the first challenge, the algorithms will be aware of the coordinates of the snake body due to its increasing length, hence the algorithm will

choose the solution without the snake bumping into itself. For the third challenge where food will appear twice at a time, the food_locations is passed as an array parameter in our algorithm and can hold multiple food locations. Therefore, the algorithms take into account all the coordinates inside the food_locations parameter and perform similarly to the second challenge.

Performance of the algorithms

For this project, we are measuring the performance of each algorithm based on the highest score that the snake is able to achieve.

In the first challenge, both breadth-first search and greedy best-first search algorithms theoretically can achieve infinite marks, since the snake never grows in size and would never bump into itself or the boundaries of the maze. However, the snake response time of breadth-first search would be slightly slower when compared to our greedy best-first search algorithm. This is because the shallowest node that the breadth-first search algorithm expands may or may not be in the direction of the goal node.

In the second challenge, both search algorithms can achieve above thirty points consistently, and it would never bump into itself and the wall due to the logic of both algorithms. The highest score of the Breadth-first search is 37 points, whereas the greedy best-first search has 49 points in the second challenge.

For the third challenge, both algorithms are also able to achieve above 30 points even with two foods appearing at once. Both algorithms need to take into account all the coordinates passed in the food_locations parameter array, instead of just taking the first set of coordinates in the array. From the videos shown for the third challenge, breadth-first search is able to achieve 41 points and greedy best-first search achieves 35 points.

Video Proof of Breadth-first Search Performance:

- 1st challenge (One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points): https://youtu.be/yqh5-vYWPsQ
- 2nd challenge (One food at any time; Increasing snake length with food; Starting snake length of one; reaching at least 10 points): https://youtu.be/9yg0ZjQAAeM
- 3rd challenge (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): https://youtu.be/tZhwvZ3kQzU

Video Proof of Greedy Best-First Search Performance:

- 1st challenge (One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points): https://youtu.be/0Q4yBEizBBk
- 2nd challenge (One food at any time; Increasing snake length with food; Starting snake length of one; reaching at least 10 points): https://youtu.be/HKv6DtOyL00
- 3rd challenge for Greedy Best-First Search (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): https://youtu.be/0bWGFKyrNzk