SCHOOL OF SCIENCE AND TECHNOLOGY

ACADEMIC SESSION AUGUST 2020; SEMESTER 4

CSC3206: Artificial Intelligence 202104 **DEADLINE: 21st May 2021**

Member Name 1: TAN HANLEY **Student ID: 18033936**

Member Name 2: LAU ZHEN YU **Student ID: 18024182**

Member Name 3: LIM SHAO YONG **Student ID: 19052059**

Member Name 4: KEANU TAN SOON JIE **Student ID: 18020552**

IMPORTANT

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

Courseworks must be submitted on their due dates. If a coursework is submitted after its due date, the following penalty will be imposed:

- ONE day late : 5 % deducted from the total marks awarded.
- TWO days late : 10 % deducted from the total marks awarded.
- THREE : 15% deducted from the total marks awarded.
- 1 week more days late : Assignment will not be marked and 0% will be awarded.

Lecturer's Remark (Use additional sheet if required)
I..... (Name) .....std. ID received the assignment and read the comments..... (Signature/date)

Academic Honesty Acknowledgement

"I **TAN HANLEY, LAU ZHEN YU, LIM SHAO YONG, KEANU TAN SOON JIE** (student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment."

/ (19/5/2021) (Student's signature / Date)

1.0 Introduction

This report is presented as a requirement for the Assignment 1 of the CSC3206 Artificial Intelligence course. In this assignment, we have been tasked to set up two agents to play the Snake game using different search algorithms, which are uninformed search algorithm - Breadth-first Search (BFS), and informed search algorithm - Greedy best-first search (GBFS).

The game Snake's design can be traced back to Gremlin's arcade game Blockade, which was released in 1976 (Goggin, 2010). Snake is traditionally played by a human in which the snake will switch relative to the way it is headed if the human player uses only the left and right arrow keys to play the game. When the snake hits a target, which in this report is referred to as food, it will automatically step forward and increase its length. Therefore, it's difficult to get a high score with such game logic. The game's ultimate objective is to get the snake to eat as many food as possible **without running out of available board space** or **hitting the wall** or **biting itself**. When this occurs, the game is over and a final score will be shown to the player.

Both of the algorithms share a solution in common, which is to firstly identify the position of the goal node (the food) and generate all possible paths or solutions towards the goal node. Next, the algorithm will explore all possible paths that the snake may take to reach the goal node, and will select the first path which leads to the goal node, rather than selecting the most optimal / efficient one. Despite the similarities, there are key differences on how optimal each algorithm could be. Hence, we will be discussing both of these algorithms in detail below.

2.0 Understanding the challenges with Snake game

In this part, we'll go through some of the key characteristics of our Snake game implementation. In a broad sense, our implementation consists of a snake running along a 10x10 square board, attempting to eat as much food as possible while avoiding biting itself and hitting the wall. When the snake consumes a food, a new food is placed on a free spot on the board, and the snake's length increases by one unit (this setting can be changed). The game ends when the snake has no choice except to bite itself and a final score will be given. In the implementation, the score is just the number of foods eaten by the snake or equivalently, the snake's length.

The basic rule of the game followed by our implementation are shown below:

1. The board size is fixed to 10x10 square.
2. The snake can move in four distinct directions, which are *north, south, east and west*. However, under the scenario where the snake's length is longer than 1, the opposite direction of the snake may not be available due to its tail placement. For example, the

snake cannot swap from facing north to south or else it will bite itself and the game will be ended.

3. The objective of the game is to consume as much food as possible in a limited board space. The game's main objective is to avoid the snake from biting itself, while increasing the score as much as possible.
4. When the snake consumes a food, it will grow by one unit. The length of the tail will develop instantly as a result of the food consumption. This feature can be turned off depending on the challenges faced.
5. Following the snake's consumption of the food, another food is put randomly on one of the board's available squares with uniform chance. An available square means the square is not occupied by the snake including the expanded tail.

3.0 Algorithms applied to solve the problem

In this section, we will be discussing the two AI algorithms applied as the agent of the Snake. For clarification, these algorithms are designed to run under the following conditions and will not generate accurate results otherwise:

1. The board size is a 10x10 square.
2. The snake can move in four distinct directions.
3. The snake always prioritizes moving *north* first, then *south*, then *west*, and *east*.
4. The snake aims to eat as much food as possible and will continue to do so unless stopped by the player.
5. When the snake eats a piece of food, its length will not increase.
6. Every time the snake eats a piece of food, a new piece of food is randomly generated on an empty space on the board.

3.1 Breadth-first Search (BFS)

As for the uninformed search algorithm that is appointed to our agent, we have chosen to use Breadth-first Search (BFS) Algorithm.

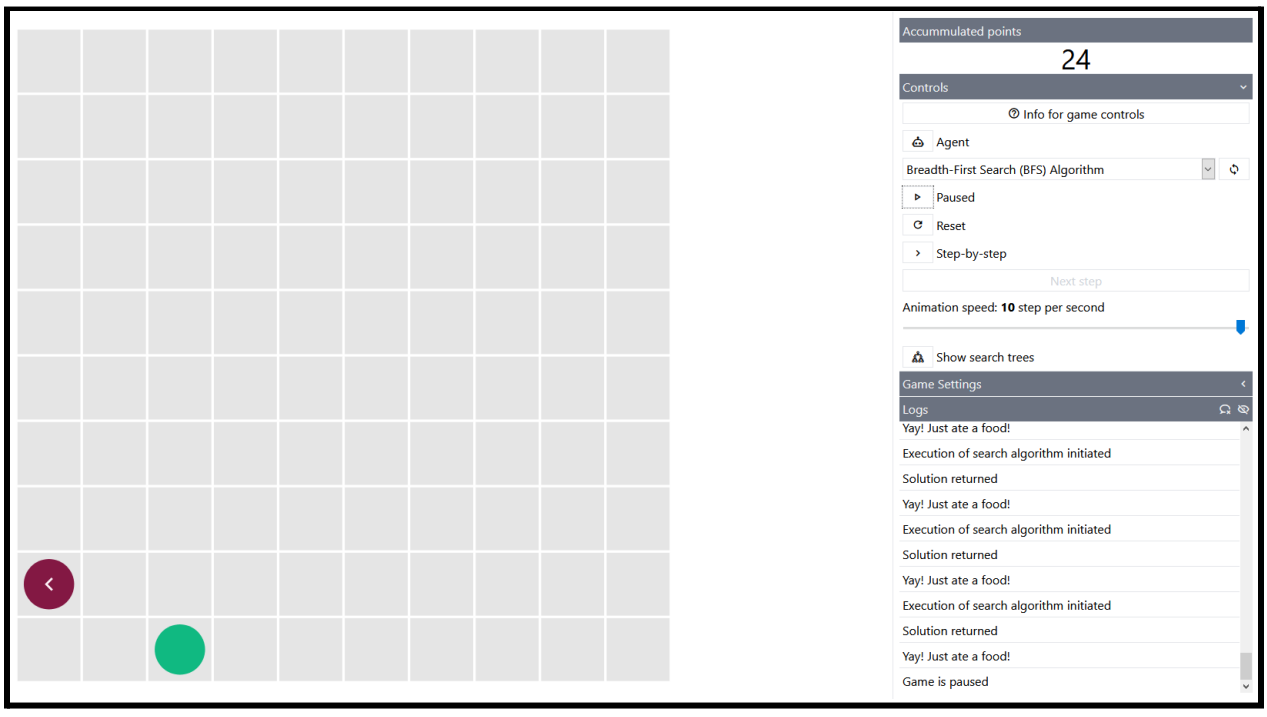
In BFS, nodes are expanded from left to right, as opposed to other algorithms (i.e. Depth-first search, where nodes are expanded from top to bottom). In the context of the Snake game, this means that many child nodes will be generated from expanding all possible nodes from a single cycle of movement. Because the nature of the BFS algorithm focuses on breadth rather than depth or any other metric, it is known to have a reduced computing speed due to redundancies in searching.

Initially, the BFS algorithm begins traversal from the root node (snake's head) at the initial state and aims to reach the goal state (the food). In doing so, the frontier is appended with the children generated from the expansion of the initial state. Next, the algorithm traverses through the first node at the top of the frontier from the list of children returned from expanding the previous node. By doing this, the resulting child nodes generated from expansion are added to the frontier. This cycle repeats and nodes from the top of the frontier are procedurally expanded until the goal node is reached, forming the solution. As an additional note, any potential loopy paths that are generated during expansion are not added to the frontier.

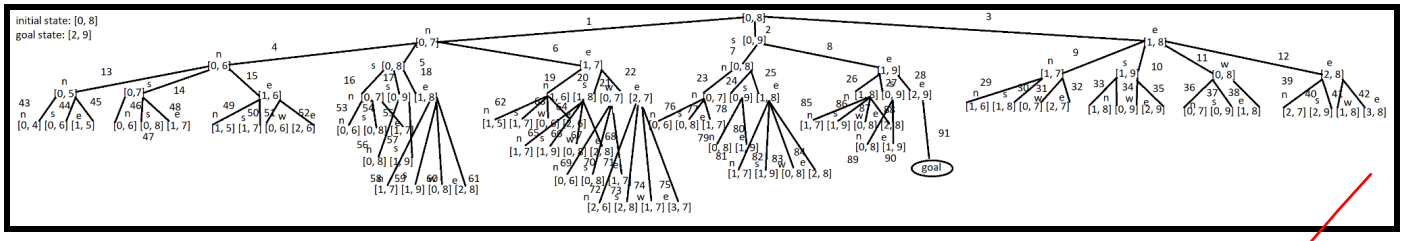
The performance of the algorithm greatly varies depending on the distance of the snake from the goal node as it is a form of brute-force searching. For example, the best case scenario and worst case scenario of the algorithm are O(N) where it depends on N, the number of nodes in the tree representing the search space. Hence, it results in low average moves as it must process through many nodes (low number of moves at high CPU cost)

3.1.1 Challenges with BFS

Below is the solution garnered from the implemented informed search algorithm:



Result of challenge (a) using BFS algorithm:
One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points



Generated search tree for the solution of the above problem (very long)

As shown above, the BFS algorithm expands every node in a row before moving on to the next, and stops expansion once the goal node is expanded. Due to the uninformed nature of this search, it can be slightly inefficient as it has no real way of identifying whether one path is more efficient than the other. Nevertheless, the solution is easily reached, and with a static snake length, we are able to reach a score of positive infinity. One thing to note is that the above example is a result of an iteration cycle where the snake's initial location is located somewhat close to the food. In different iterations where the snake can be located much farther away from the food, the computing speed of this algorithm drops significantly.

3.2 Greedy Best-first Search (GBFS)

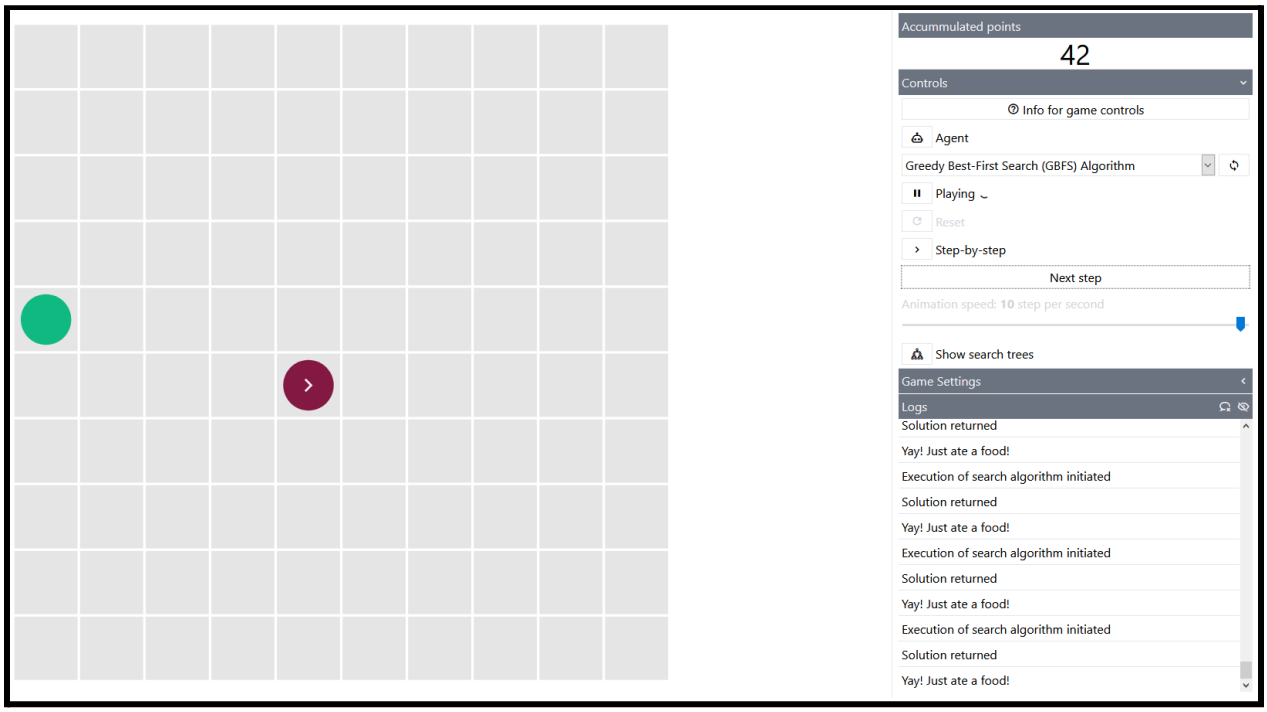
As for the informed search algorithm that is appointed to our agent, we have chosen to use Greedy Best-first Search (GBFS) Algorithm. Greedy Best-first Search algorithm is a combination of both Depth-first search and Breadth-first search algorithms. It is concentrated on the next best move by prioritizing the queue based on the lowest path cost or distance from the goal node.

How the GBFS algorithm differentiates itself from the similar BFS algorithm is that it is informed; it utilizes a heuristic function to assign weightages / values to each node. By doing so, the GBFS selects and prioritizes nodes which have a lower heuristic value, as this usually results in an overall shorter path distance / cheaper total path cost when searching.

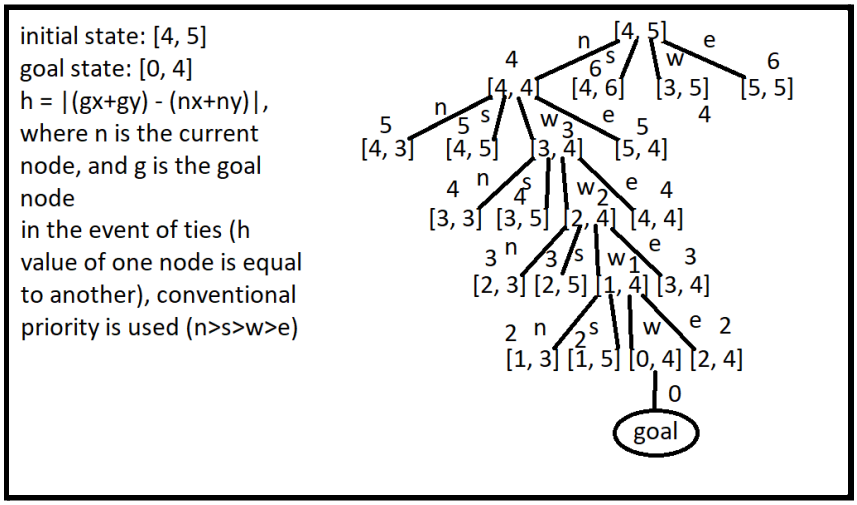
Initially, the GBFS algorithm traverses from the root node (snake's head) at the initial state and aims to reach the goal state (the food). In doing so, the frontier is populated with child nodes generated from expanding the initial state. Next, the algorithm traverses through the top node of the frontier from the returned solution based on which node has the shortest path cost from the goal node. The algorithm will evaluate each frontier from the returning solution by sorting it in an ascending order, which helps identify nodes which are the best candidates or have the shortest path cost towards the goal node (food).

3.2.1 Challenges with GBFS

Below is the solution garnered from the implemented informed search algorithm:



Result of challenge (a) using GBFS algorithm:
One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points



Generated search tree for the solution of the above problem.

4.0 Code implementation / explanation

Below is a list of functions, as well as their respective role in the overall system:

1. expandAndReturnChildren(self, state)

This function takes one set of coordinates (a list of integers) and returns a list of lists of lists of integers (a list of paired coordinates). This list of lists represents the potential coordinates that the Snake object can move towards. For example, if passing a set of coordinates [1, 1] as an argument into the expandAndReturnChildren() function, we obtain a list of [[[1, 1], [1, 0]], [[1, 1], [1, 2]], [[1, 1], [2, 1]], [[1, 1], [0, 1]]]. In this case, [1, 1] is the initial location of the snake, whereas the four locations [1, 0], [1, 2], [2, 1], and [0, 1] are the new locations that the snake can move to.

When this function is called, it takes the argument (a set of coordinates), and initializes an empty list called “children”. It will then check if its child nodes exist within the boundary of the board space. What this means is that it will not add any nodes with coordinates below 0 or greater than 9, as those refer to locations outside the boundary of the board space. For each child node that has coordinates which are not lesser than 0 and do not exceed 9, they are appended to the children list. At the end of the loop, the list of coordinates are returned to the method call.

2. bfs(self, initial_state, goal_state)

This function takes a pair of coordinates, one an initial state, and one a goal state to be reached. It will return a solution as well as a search tree detailing the procedures of the search. This solution is returned in the form of a list of String objects (i.e. [“n”, “n”, “n”, “e”, “e”, “e”]).

When calling the function, the variables state, frontier, explored, solution, found_goal are initialized. State is a copy of the initial_state argument, whereas frontier, explored, and solution are empty lists which will be filled later. The variable found_goal is a Boolean statement which acts as a loop control variable.

Next, the function will procedurally generate children via the expandAndReturnChildren() function, based on the current top node of the frontier, and will check if each child node exists either in the list of explored nodes, or in the frontier. If a node is deemed to not be present in either list, it will be appended to the frontier list, and then will be checked to see if it matches the goal state. If it matches the goal state, the value of found_goal is set to True and its value is copied to the solution variable. Otherwise, this process repeats under a while loop as long as the value of found_goal is set to False.

While the function is running, it will print its current explored list, as well as its frontier and children. Once the solution is found, its full path is returned alongside its corresponding search tree to the calling method. (NOTE: The functionality for generating a search tree has not been added to the program, and exists solely to demonstrate its functionality. Do not take the generated search tree as an accurate representation of the algorithm's logic.)

3. gbfs(self, initial_state, goal_state)

This function behaves similarly to bfs(), and takes a pair of coordinates while returning a solution in the form of a full path (an ordered list of coordinates which act as instructions). It uses the same code structure as the bfs() function, but has added functionality to make it a Greedy Best-First Search algorithm.

Unlike the bfs() function, the gbfs() function assigns a heuristic value to each node with respect to its distance from the goal node. This is how the algorithm will sort and prioritize certain nodes to reduce overall computing time.

After appending the children to the frontier in each iteration cycle, it will measure the distance from the goal node and append this value to each node in the frontier. Once all nodes in the frontier have a corresponding distance value, the entire list is sorted based on distance in ascending order. This way, nodes with the shortest distance from the goal node will be expanded first instead of others. In the event of ties, where certain paths have the same heuristic value / distance from the goal node, the following directions take priority as follows: first *north*, then *south*, then *west*, and finally *east*.

Once the solution has been found, it is returned alongside its corresponding search tree to the calling method. (NOTE: The functionality for generating a search tree has not been added to the program, and exists solely to demonstrate its functionality. Do not take the generated search tree as an accurate representation of the algorithm's logic.)

4. run(self, problem)

The run() function is used to initialize the Player() class and to instruct the front-end system to begin computing. When this function is called, the initial and goal states of the program are initialized, which represent the snake's starting location as well as the location of the food respectively. From here, the respective algorithm function (bfs() / gbfs()) is called to generate a set of coordinates to act as a solution.

Once the list of coordinates is returned by the search algorithm function, we will put these coordinates through a for-loop in order to convert them from their coordinate form into

movement instructions (i.e. [0, 0] -> [0, 1] -> [0, 2] -> [1, 2] = [“s”, “s”, “e”]). Finally, the solution is returned to the calling method as a list of String objects alongside a search tree. (NOTE: The functionality for generating a search tree has not been added to the program, and exists solely to demonstrate its functionality. Do not take the generated search tree as an accurate representation of the algorithm’s logic.)

5.0 Performance discussion / evaluation

This section discusses the overall performance differences between the BFS algorithm and the GBFS algorithm, and attempts to evaluate which algorithm is more efficient at performing searches.

When implementing both search algorithms, one thing to note is that the BFS algorithm was much easier to implement during development, as the logic behind the search is to explore every node via brute-force and eventually reach the goal node. As it is an uninformed search, it is naturally more time-consuming compared to that of informed searches, which are able to reduce the burden on the CPU by cutting corners in computing via well-defined metrics / rules.

In this context, it is clear that GBFS is the more advantageous choice of the two as utilizing a heuristic value such as distance from the goal node can help reduce unnecessary node expansion cycles, therefore reducing overall computing time. Unlike in BFS, where nodes are expanded level-by-level and prioritize breadth over depth and the like, GBFS is able to skip unnecessary nodes and efficiently reach the goal node.

However, this does not mean that a Best-First search is the most efficient search algorithm, because while it is able to eliminate moves which deter from achieving the game’s main objectives, it is not generally optimized to handle dynamic snake lengths - game states in which eating a piece of food would increase the snake’s size. In cases like these, our design of the snake’s directional movement priorities would not prove suitable for this use case.

This is because our snake always searches vertical paths before moving on to checking horizontal paths; for this reason, every time a solution is generated, the snake will always move *north* or *south* until it reaches the same y-level as the food, before moving *west* or *east* to consume the piece of food.

With this in mind, we believe that other informed search algorithms would prove to be more beneficial, such as A* Search or even AND-OR Tree. It remains to be seen if the performance difference between these algorithms is significant, but we believe that with larger test samples, there may very well be a noticeable change in performance.

References

Goggin, G. (2010). *Global Mobile Media*. Taylor & Francis.