SUNWAY UNIVERSITY

SCHOOL OF SCIENCE AND TECHNOLOGY
ASSIGNMENT FOR
BSC (HONS) COMPUTER SCIENCE; YEAR 3

ACADEMIC SESSION APRIL 2021;
CSC3206: ARTIFICIAL INTELLIGENCE
DEADLINE: 21 MAY 2021 (FRIDAY); 5:00PM

STUDENT NAME AND ID	:	1. Lim Han Shen (18124693)
		2. Brandon Wong Ching Yong (17097692)
		3. Bryan Tan Guan Eu (17083445)
		4. Yeoh Qing Tuan (18002220)

INSTRUCTIONS TO CANDIDATES

- This assessment will contribute 15% to your final result.
- There will be deduction of marks for submissions later than 21 May 2021, 6pm.

Lecturer's Remark (Use additional sheet if required)
We, *Lim Han Shen, Brandon Wong Ching Yong, Bryan Tan Guan Eu, Yeoh Qing Tuan,* with student IDs, *18124693, 17097692, 17083445 18002220*, received the assignment and read the comments.

.... (Signature/date)

Academic Honesty Acknowledgement

"We, *Lim Han Shen, Brandon Wong Ching Yong, Bryan Tan Guan Eu, Yeoh Qing Tuan*, verify that this paper contains entirely our own work. We 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, we have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. We realize the penalties (refer to the student handbook and undergraduate programme) for any kind of copying or collaboration on any assignment."

.... (Student's signature / Date)

1.0 PROBLEM DEFINITION

Snake is a classic computer game where players control the snake to eat food that randomly appears within the enclosure or grid on the screen. Food eaten by the snake represents points and the main goal of the game is to eat as much as possible without the snake crashing into the walls or biting its body.

With the use of AI techniques, it is possible to create autonomous agents that controls the snake and plays the game automatically, while considering the constraints of the game. This project aims to create two autonomous agents for the snake game provided in the form of two search algorithms: One using informed search and one using uninformed search.

These algorithms should return valid solutions to control the snake towards the food each round. The solutions and movements of the snake should also demonstrate the differences between informed search and uninformed search, as well as generate accurate search trees for each round by providing the game with appropriate arguments.

2.0 PROBLEM FORMULATION

Search algorithms are common algorithms used to locate specified goals such as data and objects within a data structure. As such, they are suitable for the snake game because many of the game’s elements can be represented using data structures and objects. The grid of the game is represented by a data structure of nodes to be searched with corresponding coordinates, the snake head represents the starting point of the search (the initial state) and the food represents the goal states of the search.

3.0 IMPLEMENTATION

3.1 Parameters

Since both the algorithms have to fit in the same snake game, the parameters used and values returned to the game have to be similar. The game provides the algorithms with several arguments that need to be used by the algorithms to return proper solutions and values.

One of the parameters is the `maze_size` parameter, which can be adjusted in the game settings. Our algorithms take this parameter into consideration to adapt to player settings and to create the state space accordingly.

Next, `snake-locations` and `food-locations` are essential parameters for the both algorithms. They are both represented by nested lists in which, `snake-locations` contains the states of the snake head and its body, while `food-locations` stores one or more states of the food depending on player settings. The first index of `snake-locations` parameter identifies the snake's head, which is used as the initial state of the search. Besides that, `snake-locations` also records the node states to avoid in both search algorithms so that the snake does not bite itself. As for the `food-locations` parameter, the list identifies the goal states of the search. If there are more than one food, our algorithm will target either the closest food or the most accessible food before proceeding to the next.

3.2 Return Values

As with the parameters, the return values of both search algorithms are the same, namely `solution` and `search_tree`.

The `solution` return value is a list of strings which represent the moves that the snake takes to eat the food for each round. There can be four types of strings representing north, south, west and east in the list, shown as "n", "s", "w" and "e". Since our algorithms deal with nodes and states, we created the `convertToDirection` function to obtain the directions from the initial node to the goal node. This function is covered in the following section.

The `search_tree` return value in our algorithm contains a list of nodes with their respective details, such as node ID, expansion sequence, children, parent and actions. These nodes are represented as dictionary objects. The construction of the search tree is done in the shared `buildSearchTree` function, which will also be covered in the next section.

3.3 Shared Functions

This section covers the common functions used by both of our informed and uninformed search algorithms. These functions have the same usage in both algorithms.

- **`fetchChildren`**

The `fetchChildren` function receives the state space and current node as parameters and uses the `isValidNode` function to check all adjacent nodes in four directions (n, s, w, e). If the states are valid, the extracted nodes will be added into the current node's children list, while the current node will be assigned as the children's parents.

- **`isValidNode`**

This function is only called by the `fetchChildren` node in both algorithms. As mentioned, the `isValidNode` function takes in node states to check if they are valid or not. Validity refers to states within the boundaries of the grid and states that avoid the snake's body. This function ensures that the snake does not crash into walls or bite its own body.

- **`convertToDirection`**

The `convertToDirection` function compares the states (coordinates) of the current node to the parent node and returns the `move` value in String format. The `move` value represents the direction for the snake to move from the parent to the child node. This function is mainly used at the end of the search algorithm to create the solution list of strings and within the `moveStep` function. It is also used to append to the list of actions attribute for each node object.

- **`buildSearchTree`**

As mentioned, this function extracts the details from each node from an explored list or valid nodes list passed into it and generates a list of dictionary objects. The game uses this list to generate the search tree to display on the UI for each round.

- **`moveStep`**

This function tries to extend the gameplay in both algorithms. As the snake grows, its body might end up forming a wall between the snake's head and the food. This causes the algorithm to terminate since the goal is inaccessible and no solution can be found. Hence, the **`moveStep`** function extracts and randomly selects a valid node adjacent to the snake head and returns it as a solution. This results in the snake moving one step each round until a solution can be found.

3.4 Uninformed Search

A breadth-first search (BFS) algorithm was developed as the uninformed search algorithm performed by the **`bfs`** function. BFS is a common search algorithm where nodes are expanded horizontally before expanding into deeper nodes. It is considered to be a blind search as the algorithm does not consider any values while expanding the nodes.

When nodes are expanded through the `fetchChildren` function, the children are appended directly into the frontier list. The frontier acts as a queue where the nodes are expanded on a first-in-first-out (FIFO) basis. Then, the expanded node is appended into the explored list and removed from the frontier list.

The goal test is done each time a child node is generated at the end of the loop and the algorithm terminates once the goal node is generated. If a solution is found, the algorithm returns the solution in String format by backtracking from the goal node to the initial node with the `convertToDirection` function. If a solution is not found, the `moveStep` function is invoked instead, returning a random solution and a list of valid nodes. At the end of the algorithm, the `buildSearchTree` function is then invoked with either the explored list or the valid nodes list passed in as the argument, generating the search tree.

3.5 Informed Search

For the informed search, we developed a Greedy breadth-first-search or Greedy Search algorithm. This algorithm is a modified version of the BFS, performed by the `greedy_search` function. In the Greedy Search, the algorithm has a different sequence of node expansion and considers an extra factor called the Heuristic function. There are many measures that can be used as the Heuristic function to estimate the cost for node expansion. In our Greedy Search algorithm, we used the Euclidean Distance from each node to the goal node as the Heuristic function. The Euclidean Distance is a straight-line measure between two points on a grid and in our algorithm, it is calculated by the `getEuclideanDist` function.

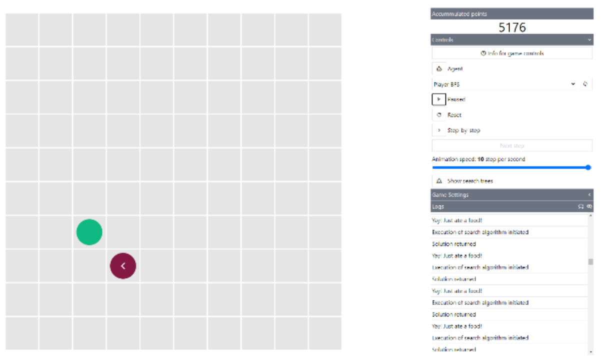
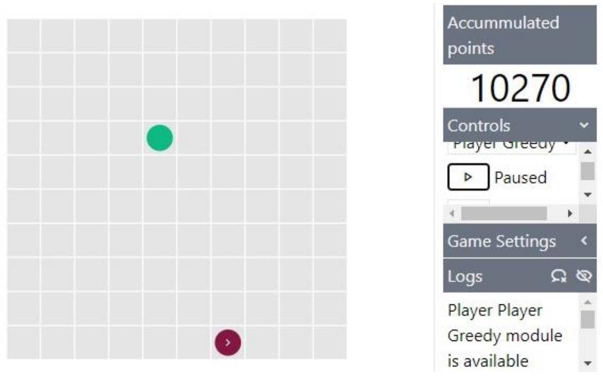
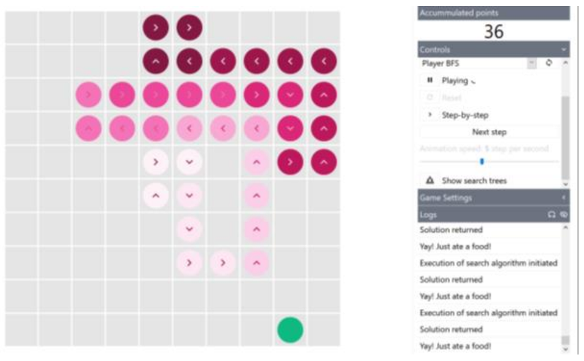
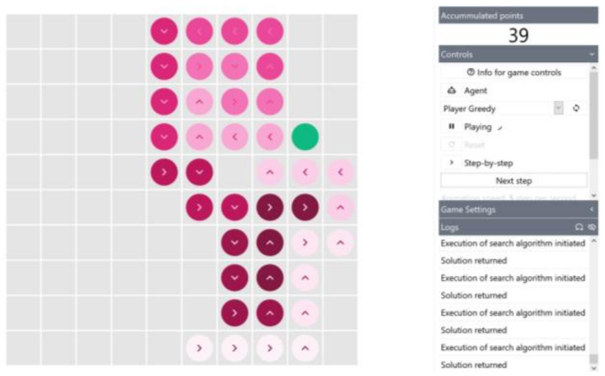
Besides having the Heuristic function, two key differences with the Greedy Search algorithm compared to the BFS are the sequence of actions and the frontier arrangement. The Greedy Search algorithm performs the goal test when nodes are expanded at the start of each loop. After a node is expanded, the children are appended into the frontier list which goes through the `sortFrontier` function. This function arranges the frontier based on the Euclidean Distance of each node to the goal in ascending order before the next expansion takes place. The sorted frontier affects the expansion sequence, where the node with the closest estimated distance to the goal will be prioritised and expanded first.

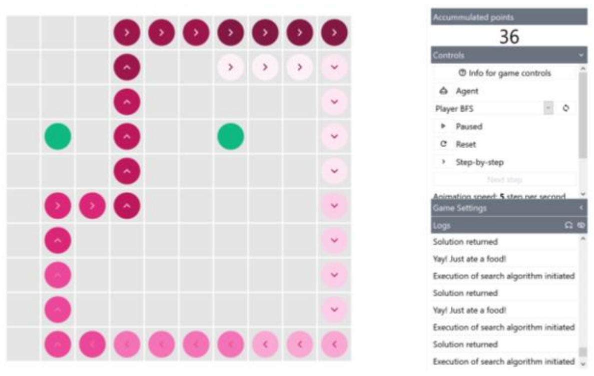
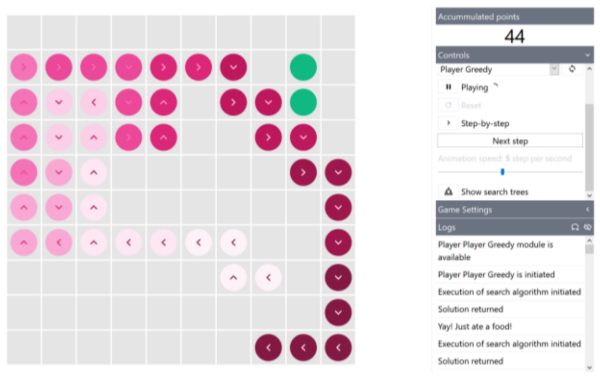
When the goal node is identified during its expansion, the loop breaks and the process of obtaining the solution and search tree is performed. This process is similar to that of the BFS algorithm. If no solution is found, the `moveStep` function is similarly invoked to return a random solution and a list of valid nodes. The only difference in this part is that if a solution is returned, the explored list will be combined with the frontier list before being passed into the `buildSearchTree` function. This is because in the Greedy Search algorithm, many generated nodes are not expanded and will not make it into the explored list.

4.0 RESULTS

4.1 Automated Gameplay

The automated gameplay of both the BFS algorithm and the Greedy Search algorithms produced expected results and achieved the minimum objectives proposed. Shown in the table below are screenshots of snake game UI running our algorithms and achieving the minimum objectives. Note that the record score listed are not for comparison, but just a record of the highest score observed.

Uninformed Search (BFS)	**Informed Search (Greedy)**
Objective 1: One food, Static snake length, achieve minimum 15 points	
 Accummulated points 5176 Controls Info for game controls Agent Player BFS Paused Reset Step by step Next step Animation speed: **10** step per second Show search trees Game Settings Logs Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned *Record score observed: 5176*	 Accummulated points 10270 Controls Player Greedy Paused Game Settings Logs Player Player Greedy module is available *Record score observed: 10270*
Objective 2: One food, Dynamic snake length, achieve minimum 10 points	
 Accummulated points 36 Controls Player BFS Playing Reset Step-by-step Next step Animation speed: 5 step per second Show search trees Game Settings Logs Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! *Record score observed: 36*	 Accummulated points 39 Controls Info for game controls Agent Player Greedy Playing Reset Step-by-step Next step Game Settings Logs Execution of search algorithm initiated Solution returned Execution of search algorithm initiated Solution returned Execution of search algorithm initiated Solution returned Execution of search algorithm initiated Solution returned *Record score observed: 39*

Objective 3: Two food, Dynamic snake length, achieve minimum 10 points	
 Accummulated points 36 Controls Info for game controls Agent Player BFS Paused Reset Step-by-step Next step Animation speed: 5 step per second Game Settings Logs Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned Execution of search algorithm initiated *Record score observed: 36*	 Accummulated points 44 Controls Player Greedy Playing Reset Step-by-step Next step Animation speed: 5 step per second Show search trees Game Settings Logs Player Player Greedy module is available Player Player Greedy is initiated Execution of search algorithm initiated Solution returned Yay! Just ate a food! Execution of search algorithm initiated Solution returned *Record score observed: 44*

4.2 Search Tree Generation

As for the search tree generation, both algorithms accurately provided the details of the nodes such as the node expansion sequence and the nodes' respective parents and children.

Examples of a search tree for the BFS algorithm (Figure 1) and the Greedy Search algorithm (Figure 2) are shown below. To illustrate the differences in the algorithms, the initial state and goal states are the same: (0, 5) and (3, 6) respectively.

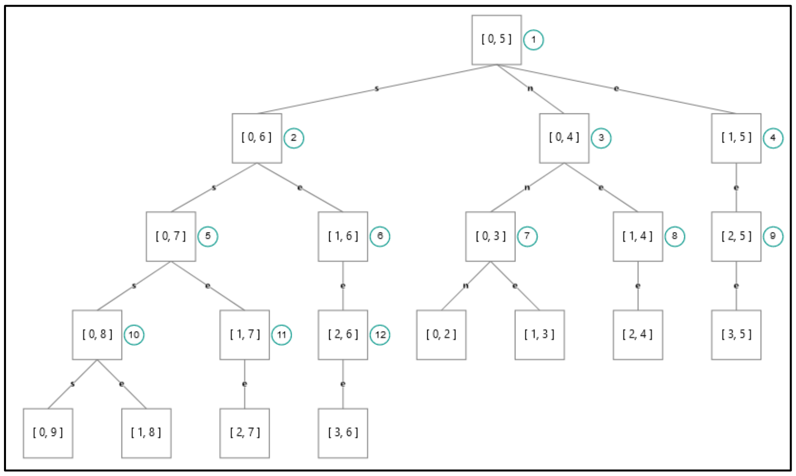


Figure 1: Search Tree for Breadth-First Search (BFS)

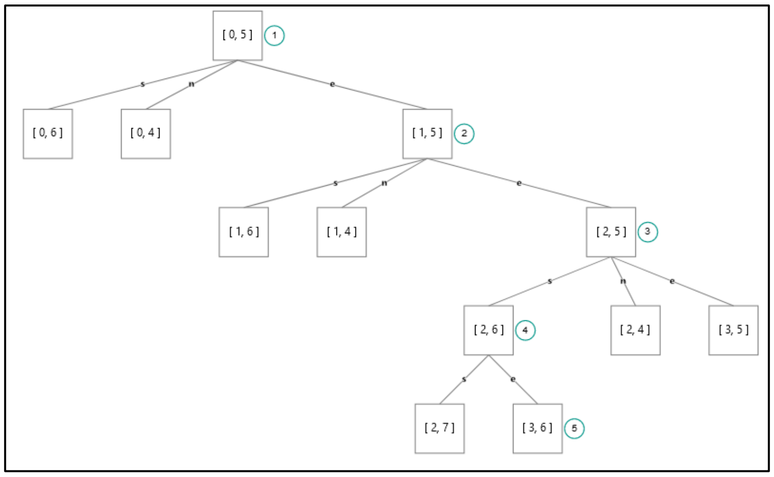


Figure 2: Search Tree for Greedy Search Algorithm

As shown, the BFS tree appears more complete and more expansive compared to the Greedy search tree; there are 20 nodes generated with 12 of them expanded. Note that the nodes expansion sequence is numbered from the left to right and that the leaf nodes were not expanded. This shows that the BFS terminates on goal node generation.

As for the Greedy Search tree, nodes are selectively expanded based on the Euclidean Distance calculated for each node. This makes the tree less expansive; there are only 12 nodes generated and only five nodes are expanded. Note that one of the leaf nodes is expanded. This is because, unlike BFS, the Greedy Search checks for the goal during the expansion of a node.

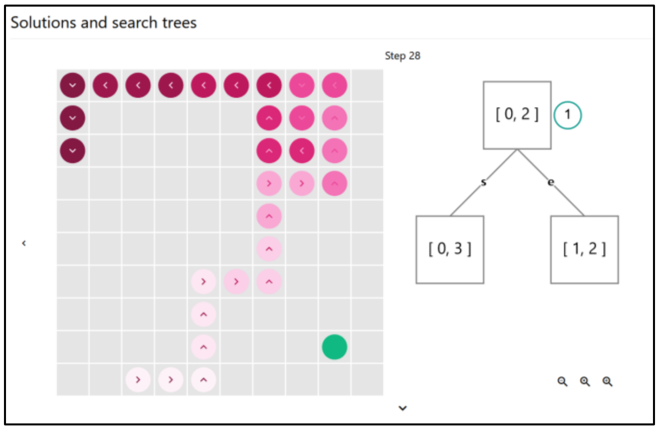


Figure 3: Search Tree with only the Valid Nodes

Figure 3 above shows a special case where a solution cannot be found. As shown in the game field, the body of the snake formed a wall between its head and the food. This event invokes the `moveStep` function, which returns a random solution and a list of valid nodes. When this occurs, the search tree is generated using only the list of valid nodes, hence the small search tree.

5.0 DISCUSSION

As shown in the Results, the BFS algorithm and the Greedy Search algorithm can produce valid solutions to control the snake to target the food continuously in each round. In this section, we discuss the performance and limitations of the algorithms developed and propose ideas that might be able to solve the issues faced.

5.1 Performance

In theory, BFS algorithm has lower efficiency and higher complexity compared to the Greedy Search because of the way nodes are expanded and explored. This could be seen in the search trees discussed earlier, as the BFS blindly expands and explores more nodes than necessary.

Through the use of a Heuristic function in the Greedy Search, the complexity of BFS is reduced, producing solutions more efficiently without expanding nodes unnecessarily. Although the Greedy Search simplifies the BFS, the Heuristic function merely produces an estimated distance between the initial state and goal state. The Greedy Search also does not compute the cost of the complete path and hence, might not produce the most optimal solution.

From our testing, we noticed that, when running BFS for extended periods of time, there would be a lag in the gameplay and a delay in the computation of solutions. This case is not as noticeable when running the Greedy Search. However, this might be because of the lack of optimisation from our implementation of the algorithms, which might not have followed the best coding practices.

5.2 Limitations

While our algorithm can run indefinitely with a static snake length, there is a limitation when running with dynamic snake length. The snake would end up trapping itself against the walls or within its own body. This causes the game to terminate since no solutions can be computed and returned.

Hence, we developed the random step function, `moveStep`. Although this is a valid way to solve the limitation of the snake getting trapped, the function only delays the game from ending for a short while and its performance is entirely dependent on chance.

One of the best ways to deal with this issue is through the use of a Hamiltonian Cycle, which creates a one-way path that passes each node on the grid exactly once. After linking up all the nodes, the path loops back to its starting point. As the snake grows, it will follow closer and closer to the defined path. However, as the game progresses with the Hamiltonian Cycle, there will be less search involved. The snake’s movement becomes more erratic and would appear to be chasing its tail rather than moving towards the food.