SCHOOL OF SCIENCE AND TECHNOLOGY

COURSEWORK FOR THE BSC (HONS) COMPUTER SCIENCE; YEAR 3
BSC (HONS) INFORMATION TECHNOLOGY; YEAR 3

ACADEMIC SESSION MARCH 2021;

CSC3206: ARTIFICIAL INTELLIGENCE

DEADLINE: 21ST MAY 2021 (FRIDAY), 5:00 PM

NO.	STUDENT ID	STUDENT NAME
1	18098517	Ong Yu Mhing
2	18125799	Tristan Elisha Chong Ze Han
3	18123588	Chuah Kim Hooi
4	15025034	Tan Ju Bhin

Table of Contents

1.0 Introduction

This report is presented as a requirement in Assignment 1 for the course CSC3026 Artificial Intelligence. In the assignment, we are required to create agents to play a snake game using AI uninformed and informed search algorithms in Python. The performance of the two different algorithms are then assessed throughout the snake game. Also, the results and differences between the different approaches will be shown at the end of this report.

2.0 Problem Description

In this assignment, we are required to create two agents, where one agent will be using uninformed search algorithm, while informed search algorithm on the other agent. Both the agents are expected to carry out the snake game on themselves, looking and approaching for food in the maze automatically. As the length of the snake is getting longer after eating the food, the risk of the snake bumping on itself and eating its own body will be higher. We have to hence keep the snake safe from itself as long as possible to avoid the game ending too early.

In addition, there are three challenges to be solved in the snake game, where challenge 1 is to reach at least 15 points in the game, with only one food at any time and non-increasing snake length. Secondly, with one food at any time and snake length starting with one, the snake in challenge 2 is required to reach at least 10 points with increasing snake length with food. Lastly, the snake in challenge 3 starting with the length of one is needed to score at least 10 points with generating of two food when no food is available on the maze and increasing snake length with food.

3.0 Explanation of Problem Formulation

3.1 For Uninformed Search Algorithm

For an uninformed search algorithm, it does not know where the goal of its search is until it reaches the goal state. Through brute force, it checks every possible route available from its starting location until it discovers the goal and then uses the connecting nodes made along that path to provide the route to the goal. As such, for a snake game, the goal would be the food for the snake, while the starting path would be the head of the snake.

As for the type of uninformed search algorithm used to search for the snake's food, the functionality should be that the search does not loop on itself and keeps going while the performance of the algorithm is finding the shortest path to the next food. To determine the shortest path to the next food, the least depth level it needs to expand on before reaching its goal. As such, the memory used by the algorithm is not limited since there is no time limit for the algorithm to calculate its solution. It also would not require a cost saving solution as the cost for moving in each step is the same, meaning the lowest number of movements would be the lowest cost.

3.2 For Informed Search Algorithm

An informed search algorithm is a technique using the Heuristic function for a search operation. To look for an optimal solution reaching the goal in a certain amount of time, an informed search algorithm will be used most of the time to evaluate the information available. It is also often seen that with the Heuristic function, an informed search algorithm tends to work faster than an uninformed search algorithm.

In the snake game, as the snake uses an informed search algorithm to search for the goals, which are the randomly-placed food across the maze, the heuristic evaluating function will iteratively calculate the distances for each snake direction cost until the food is reached. The snake will then take the shortest paths to reach each food in the maze. This saves more time in the finding process as it constantly uses more efficient paths with lowest costs.

3.3 Challenges

For challenge 1, the problem found is the routes that are available, since each movement of the snake is in either a direction of north, south, east or west respectively, we can say that each movement takes a cost of 1 movement to make. For a snake game, the direction it can move in is in a fixed size depending on the size of the maze when the snake game initiates. For example, a maze of 10 by 10 size, would have 100 tiles available for movement for the snake. For brute forcing through all the available movements, each available movement that could be made in each tile must be linked to each other, so that the snake will not go out of the maze size and crash into the wall.

For challenge 2, if the snake length is set to dynamic, meaning that whenever a snake eats a food, its length of body increases by 1. This creates a new obstacle that the snake would have to avoid to not lose in the game. As such, a way to prevent the snake head from colliding onto its own body is needed. This also makes it so the snake cannot go in the reverse direction as it would crash into its own body instantly.

Lastly, for challenge 3, if there are multiple foods on the maze, the snake will have to find each food until all food on the maze is eaten before a new set of food appears. So, the snake will have to constantly be checking for more food and proceed to find available food during each search.

4.0 Search Algorithm Implementation

4.1 For Uninformed Search Algorithm

From the various types of uninformed search algorithms, the picked algorithm is based on its performance to gather food. Since the objective of the algorithm is to collect as much food as possible, the cost of steps taken and memory usage of the algorithm is not limited. As such, Breadth First Search (BFS) algorithm was chosen as it provides any solution by searching through every type of path to the food. It also provides the minimal solution with the least number of steps. In contrast, if depth first search was chosen, the snake will go a longer path towards the food instead, it also may infinitely loop resulting in the food not being collected at all. It cannot be a Depth-Limited Search algorithm too since if the food is further from the snake than the depth limit implemented, the solution would be incomplete, resulting in the food not being collected as well. Lastly, Uniform-cost search is not needed as the value between each node is the same cost of 1, resulting in the same result as BFS.

Implementation of the code:

explaining code (what is saved in what variable) does not give additional value.

The code can be found in the zip file provided under 'Uninformed Search - Player 1 BFS'.

1. Setting up BFS algorithm:

   The function of the BFS algorithm is located at def bfs (line 144). It uses the class Node to store its state, parent, and child value for each node / step. It is called in the run function at line 126. It is used to return the solution of the maze, the cost of the solution and the search tree. If no solution is found, it will try to go to the next explored state; if there is no explored state, it will move to the next frontier state; if both also are empty, the snake will just move North. [Line 210 - 221]

   First, it takes the state it can move in as state_space, the starting location of the snake's head as the initial_state, the food location as the goal_state and the snake's body as the snake parameter. [Line 147]

Then, there are the variables initialized, such as the frontier, explored and solution variables. Once it is initialized, the function will go into a loop until the food is found. hile food is not found, the variable found_goal is set to False in the while loop. During the loop, the children of the last frontier will be expanded and added into the frontier list. The frontier that was expanded upon will then be deleted from frontier and sent to the explored list. Next, it will check each child's state, whereas if the state is the same as the food state, the loop will break. If not, it continues the loop until found. [Line 170 - 206]

After a goal is found, the goalie variable will have the solution inserted. This goalie.state will then be sent to the solution variable and return as the solution of the BFS. [Line 285 - 299]

2. Setting up maze connection

   The maze size is stored in the 'setup' variable of class Player. Once the value of the maze is taken using the python dictionary call for each key and value using setup.items(), the value of the 'maze_size' key can be found. Using a range of the maze size, each coordinate in the maze is linked to the node beside it, the coordinate is then converted into string values using mapping and join. Lastly, the state is appended into the state_space to be used in the BFS later. [Line 52 - 62]

3. Setting up snake body avoidance

   The snake's body coordinates are stored in the snake_locations of the 'problem' variable. Using the problem.items() for the key 'snake_locations', the value of the snake_locations is then stored in the 'snake' variable and used in the BFS. [Line 35-42]

The snake's body coordinates are used to prevent the snake from eating itself. This is done by removing the coordinates of the body of the snake from the state space. By doing this, the BFS function will not expand to any positions of the snake's body. [Line 147- 167]

4. Setting up multiple food collection and path to food

   The food coordinates are stored in the food_locations of the 'problem' variable. Using the problem.items() for the key 'food_locations', the value of the snake_locations is then stored in the 'food' variable and used in the BFS. [Line 35-42]

   Once the solution for the path to the food is found by the BFS function, the BFS function will return the coordinates to the food. These coordinates will be compared between each other to determine the directions the snake has to go in order to reach the food. [Line 71- 94]

5. Setting up search-tree

   In the BFS, each time it does a while loop, it goes through the next state, the states ID are saved and used for each 'loop' variable increment going through the while loop. [Line 178,183,264]

   Each state that the loop went through is just the frontier that is currently being explored, which would be the last 'explored.state' value which is stored in the search_visit variable. [Line 230, 265]

   The children of the current node which are saved in the children.state would be the next ID/loop, as such, a child_id is initiated before the while loop and each child will be given a unique ID for it to be used in the future. The direction of the kid is also saved by comparing the kid's coordinate to the parent's coordinate. [Line 233 - 252]

   For the parent, if it is the first node which has no parent, it will be set to None. Else, the parent will be incremented by 1 ID when all the current children of the parent are completed. This is done by taking the number of children the parent has and decrementing it by one until it reaches 0, when it reaches 0 it means it's the next node’s parent. [Line 256 - 281]

For the expansion sequence of the search tree, for each state found in the solution, it sets the expansion sequence of the node to the ‘expseq’ value and then increments it by 1 until all solution’s state is found. [Line 100 - 114]

4.2 For Informed Search Algorithm

To test the snake game with an informed search algorithm, A* Search algorithm is chosen as it comes with both the best features of Uniform Cost Search and Heuristic Search, where it looks for optimality while fulfilling completeness of the path and optimal efficiency in the state space search tree. A* Search algorithm implements cost estimation to extend the snake path for the food in every iteration. The function f(n) = g(n) + h(n) is the formulae of the algorithm, where g(n) is the path cost from the start node to node n, h(n) is the estimated cheapest cost of the node n to the goal node, and f(n) is the estimated cheapest path cost of the solution through node n. The A* Search algorithm will always then pick the least cost path for the snake to reach the food if the heuristic function is acceptable, leading to lower time complexity.

what's your calculation of heuristic values?

1. Setting up A* Search algorithm:

   The function of the A* Search algorithm is located in the def astar, it uses the class Node to store its state, parent, and child value for each node / step. [line 10]

   The snake body is defined as body, while start is the starting location of the snake’s head and end is the food location. [line 100]

   A while loop is used to find the solution from the start to the end. The function then creates two lists which are open and closed that are empty. Open list is used to compare between the F cost of the node and the lowest is switched to closed list. The start node is added into the open list where the F cost is 0. For each next move going north, south, east and west from the start node, if it is not in the open list then add it into the open list and make the current node, which is the start node as the parent. It then records the F, G and H cost of the current node. The function then checks to see if the path to the next node is better by using F cost as measurement. A lower F cost means it is a better path. It then changes the node of the better path to the parent and recalculates the G and F scores. The while loop stops when the goal is reached.

The paths are saved and going backwards from the current node which is the goal node to the start node, the solution is found.

2. Setting up children:

   The next move of the snake is labeled as children and the current position of the snake is the parent. Children are generated to compare between the G cost of the moves taken. The children are looped through the open list and compared between the G cost and if the G cost is lower, it is added into the open list. If G cost is higher then ignore and return to the beginning of the loop. If the child is in the closed list, the function ignores it as well.

3. Setting up snake body avoidance:

   The snake's body coordinates are stored in the snake_locations of the 'problem' variable. Using the problem.items() for the key 'snake_locations', the value of the snake_locations is then stored in the 'snake' variable and used in the A* Search algorithm. [line 29-36]

   An algorithm is created to loop through the available paths with the snake's body. If the snake's body is in the path, it is removed from the path in order to prevent the snake from eating itself.

4. Setting up multiple food collection and path to food:

   The food coordinates are stored in the food_locations of the 'problem' variable. Using the problem.items() for the key 'food_locations', the value of the snake_locations is then stored in the 'food' variable and used in the A*. [Line 32-36]

Once the solution for the path to the food is found by the A* function, the A* function will return the coordinates to the food. These coordinates will be compared between each other to determine the directions the snake has to go to reach the food. [Line 65-82]

5. Setting up search-tree

   A search tree is created by returning all possible routes that can be taken by the snake alongside with their F cost to show that the cost taken by the snake to reach its goal contains the lowest F cost. The lower the F cost of the path taken, the better the path as it is shorter compared to others.

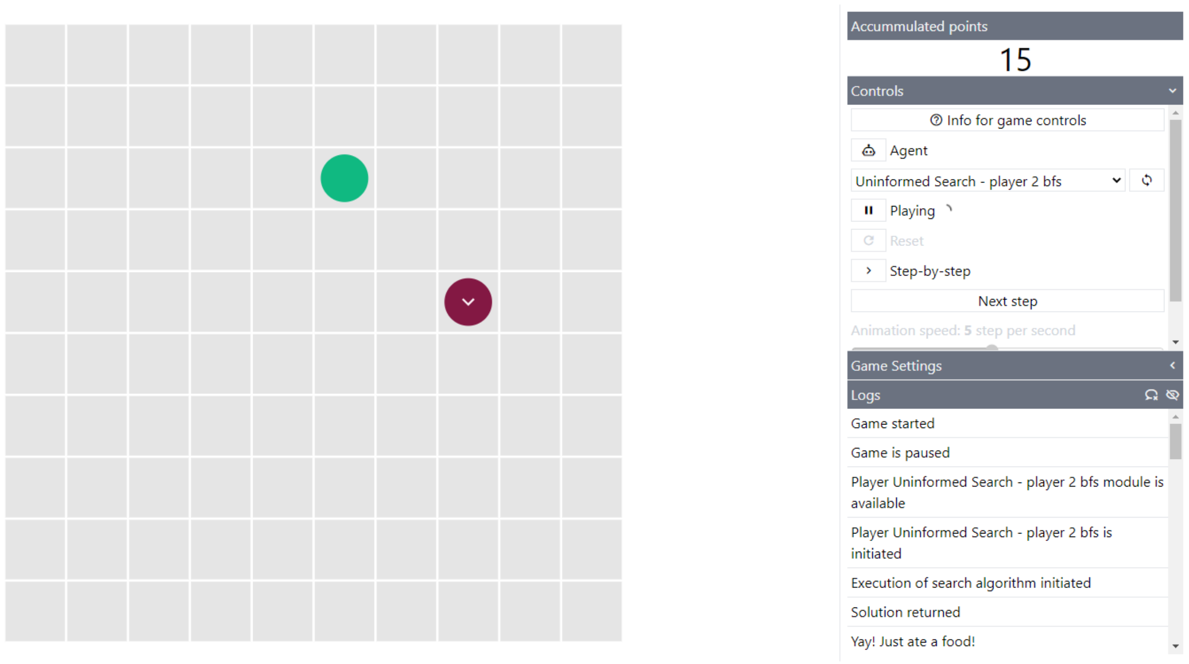
6. Setting up loop catching

   When the A* Search algorithm is caught in a search, the number of iteration is counting down until it reaches the maximum number of available node. Once it reaches it, it will only move to the next node in line and a search will be reconducted with the new positions. [Line 236 - 247]

5.0 Search Algorithm Results

5.1 For Uninformed Search Algorithm

5.1.1: Challenge 1 Figures:



Accummulated points

15

Controls

Info for game controls

Agent

Uninformed Search - player 2 bfs

Playing

Reset

Step-by-step

Next step

Animation speed: 5 step per second

Game Settings

Logs

```
Game started
Game is paused
Player Uninformed Search - player 2 bfs module is available
Player Uninformed Search - player 2 bfs is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
```

Figure 5.1.1.1: Challenge 1. 15 Food, Static snake length, 1 food spawn achieved

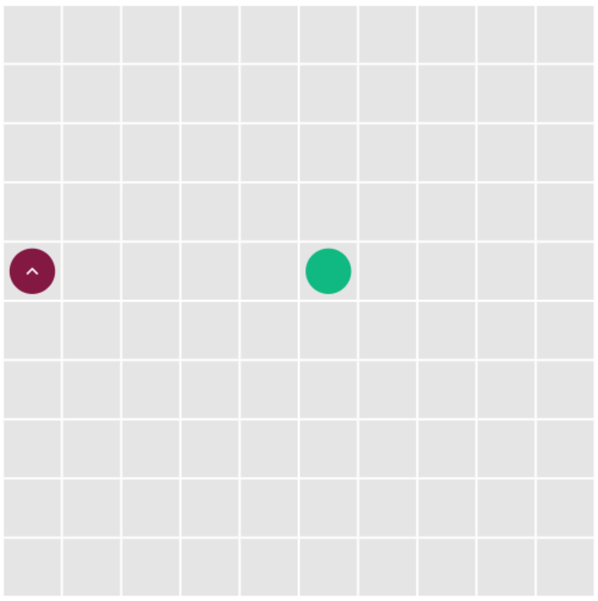


Figure 5.1.1.2 Example of current state

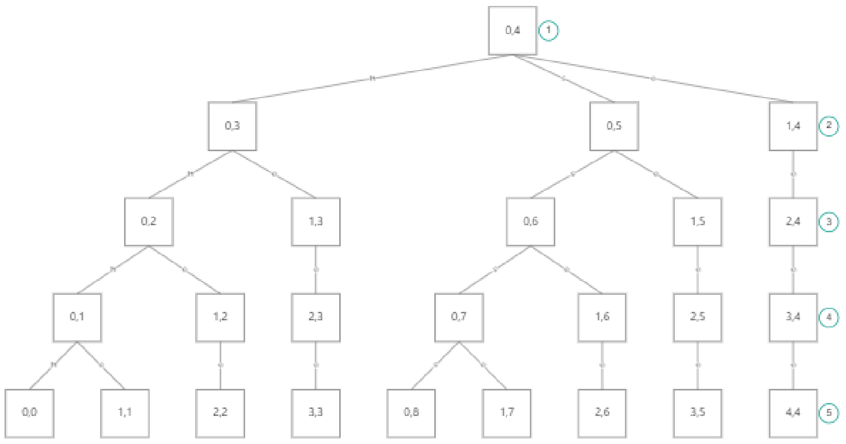
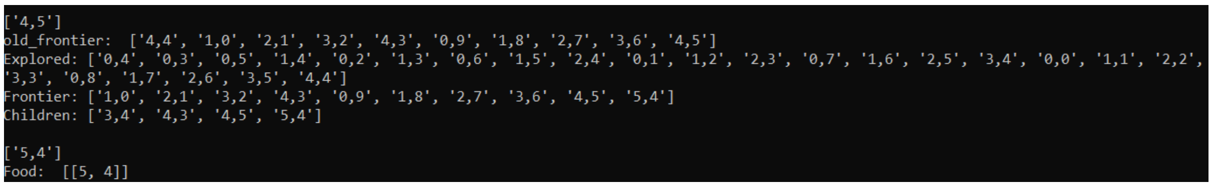


Figure 5.1.1.3 The Search Tree for the example

Figure 5.1.1.4 The Solution to the example

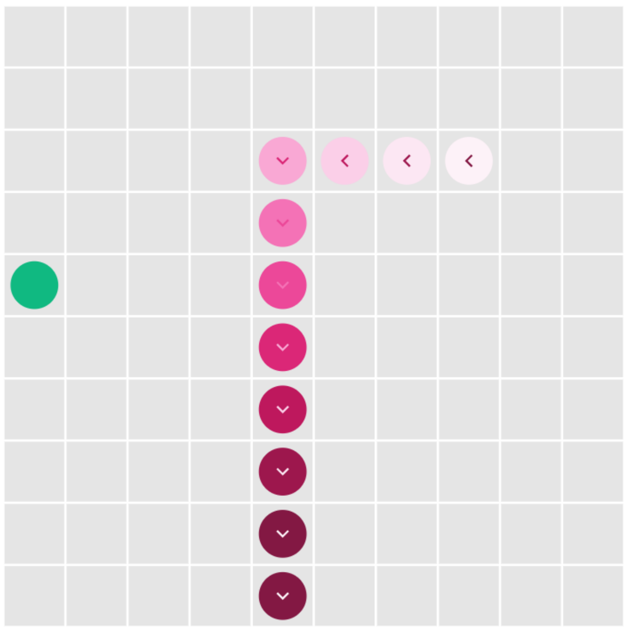
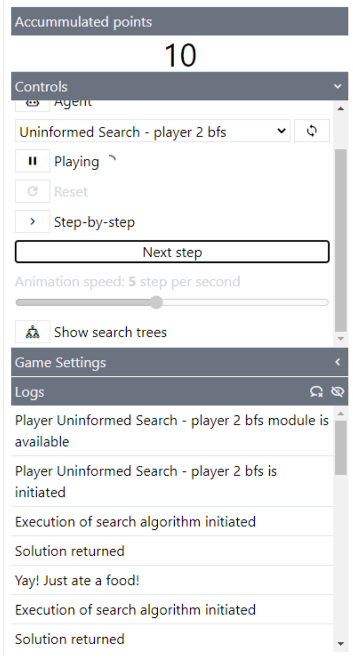


```
['4,5']
old_frontier:  ['4,4', '1,0', '2,1', '3,2', '4,3', '0,9', '1,8', '2,7', '3,6', '4,5']
Explored: ['0,4', '0,3', '0,5', '1,4', '0,2', '1,3', '0,6', '1,5', '2,4', '0,1', '1,2', '2,3', '0,7', '1,6', '2,5', '3,4', '0,0', '1,1', '2,2',
'3,3', '0,8', '1,7', '2,6', '3,5', '4,4']
Frontier: ['1,0', '2,1', '3,2', '4,3', '0,9', '1,8', '2,7', '3,6', '4,5', '5,4']
Children: ['3,4', '4,3', '4,5', '5,4']

['5,4']
Food:  [[5, 4]]
```

Figure 5.1.1.5 The solution shown in the terminal

5.1.2: Challenge 2 Figures:



Accummulated points

10

Controls

Agent

Uninformed Search - player 2 bfs

Playing

Reset

Step-by-step

Next step

Animation speed: 5 step per second

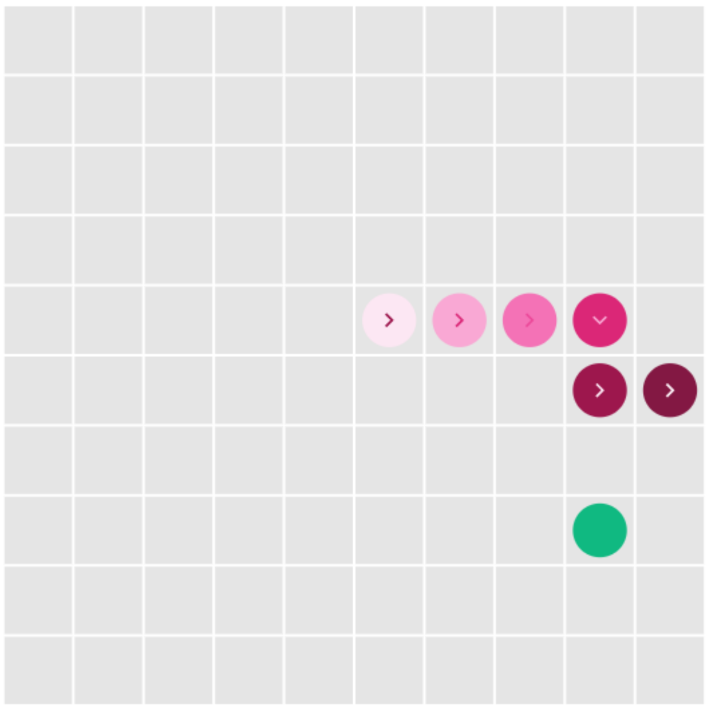
Show search trees

Game Settings

Logs

```
Player Uninformed Search - player 2 bfs module is available
Player Uninformed Search - player 2 bfs is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
```

Figure 5.1.2.1: Challenge 2. 10 Food, Dynamic snake length, 1 food spawn achieved



5.1.2.2 Example of current state

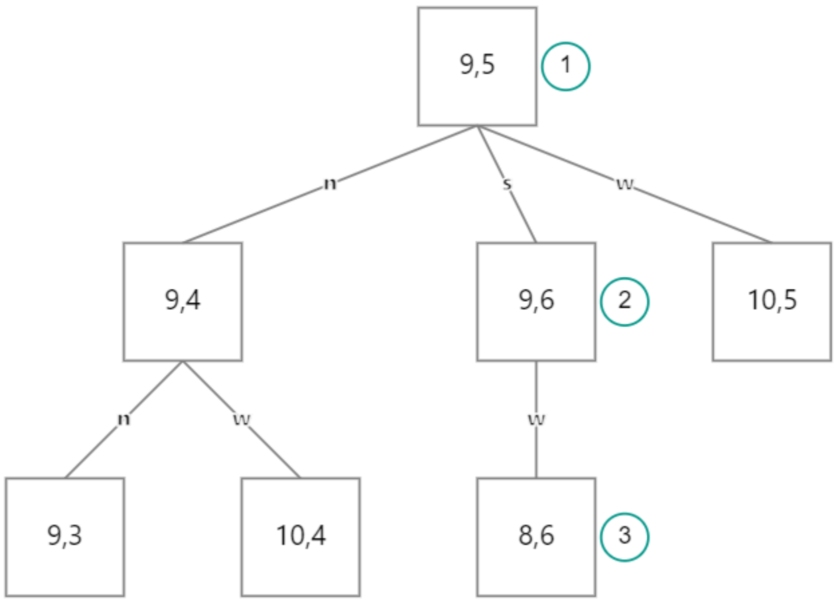


Figure 5.1.2.3 The Search Tree for the Example

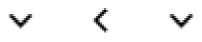
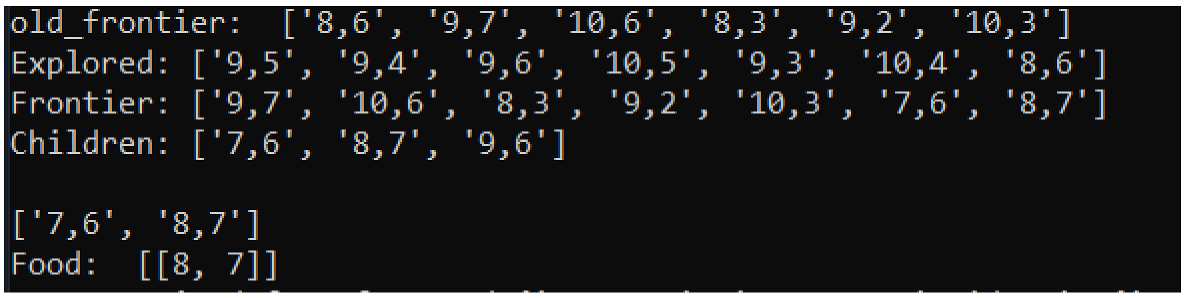


Figure 5.1.2.4 The Solution to the Example

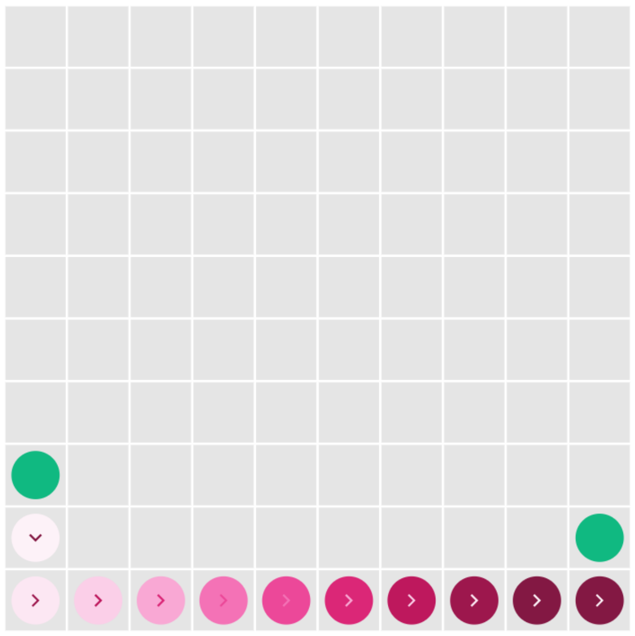


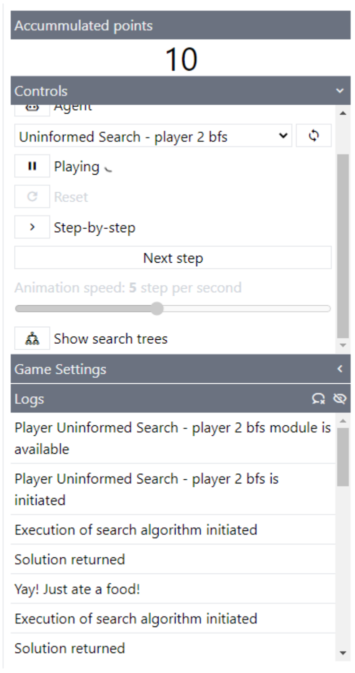
```
old_frontier:  ['8,6', '9,7', '10,6', '8,3', '9,2', '10,3']
Explored: ['9,5', '9,4', '9,6', '10,5', '9,3', '10,4', '8,6']
Frontier: ['9,7', '10,6', '8,3', '9,2', '10,3', '7,6', '8,7']
Children: ['7,6', '8,7', '9,6']

['7,6', '8,7']
Food:  [[8, 7]]
```

Figure 5.1.2.5 The Solution shown in the terminal

5.1.3: Challenge 3 Figures:




Accumulated points

10

Controls

Agent

Uninformed Search - player 2 bfs

Playing

Reset

Step-by-step

Next step

Animation speed: 5 step per second

Show search trees

Game Settings

Logs

```
Player Uninformed Search - player 2 bfs module is available
Player Uninformed Search - player 2 bfs is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
```

Figure 5.1.3.1: Challenge 3. 10 Food, Dynamic snake length, 1 food spawn achieved

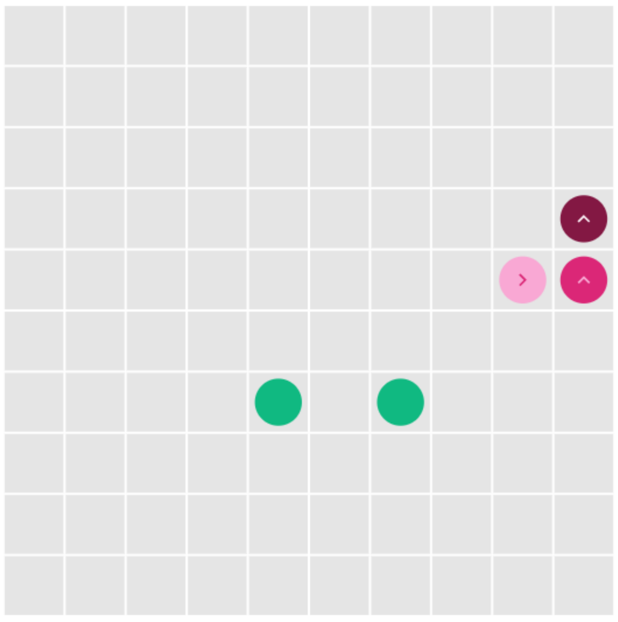


Figure 5.1.3.2 Example of current state

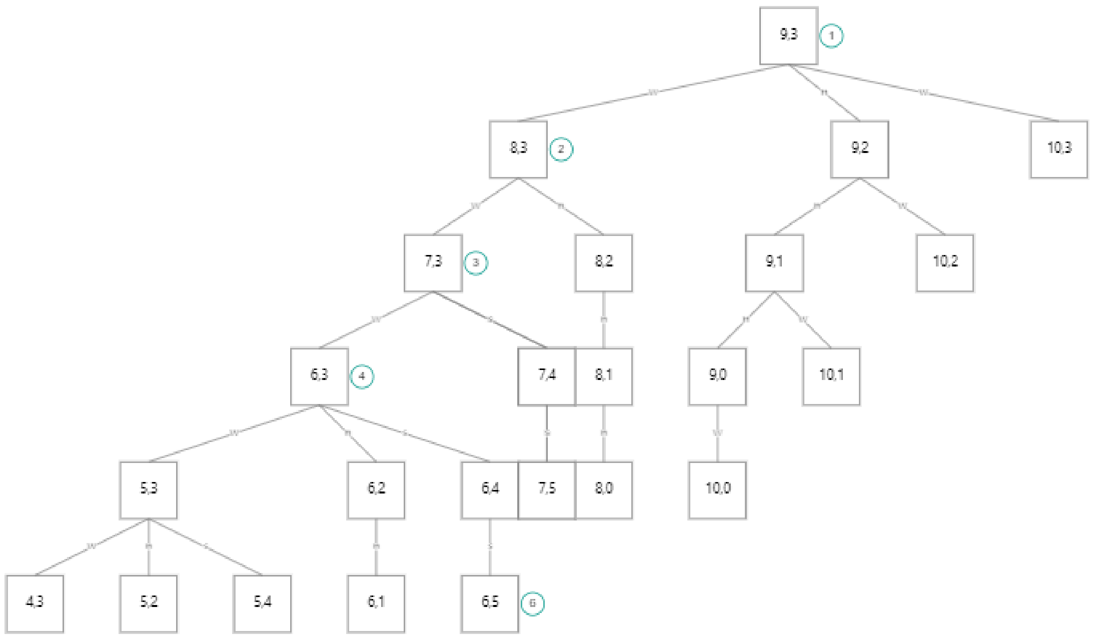


Figure 5.1.3.3 The Search Tree for the Example

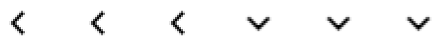
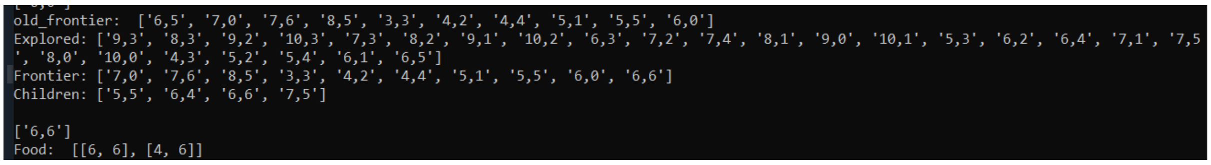


Figure 5.1.3.4 The Solution to the Example



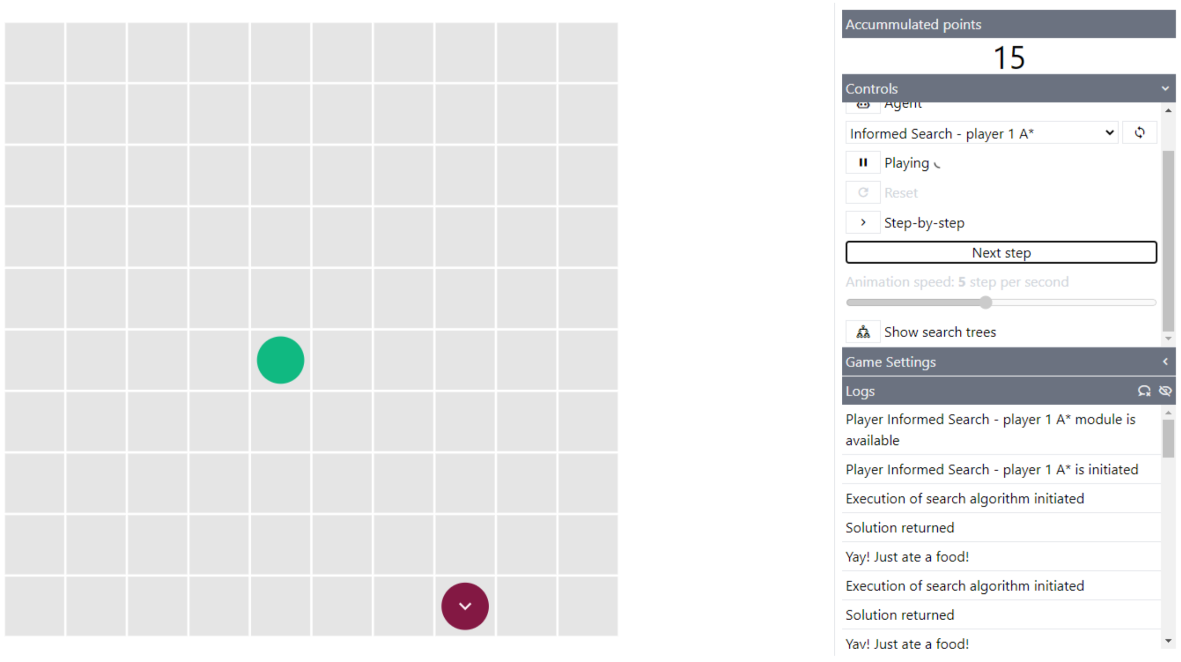
```
old_frontier:  ['6,5', '7,0', '7,6', '8,5', '3,3', '4,2', '4,4', '5,1', '5,5', '6,0']
Explored: ['9,3', '8,3', '9,2', '10,3', '7,3', '8,2', '9,1', '10,2', '6,3', '7,2', '7,4', '8,1', '9,0', '10,1', '5,3', '6,2', '6,4', '7,1', '7,5
', '8,0', '10,0', '4,3', '5,2', '5,4', '6,1', '6,5']
Frontier: ['7,0', '7,6', '8,5', '3,3', '4,2', '4,4', '5,1', '5,5', '6,0', '6,6']
Children: ['5,5', '6,4', '6,6', '7,5']

['6,6']
Food:  [[6, 6], [4, 6]]
```

Figure 5.1.3.5 The Solution shown in the terminal

5.2 For Informed Search Algorithm

5.2.1 Challenge 1 Figures:



```
Player Informed Search - player 1 A* module is available
Player Informed Search - player 1 A* is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
```

Figure 5.2.1.1 Challenge 1. 15 Food, Static snake length, 1 food spawn achieved

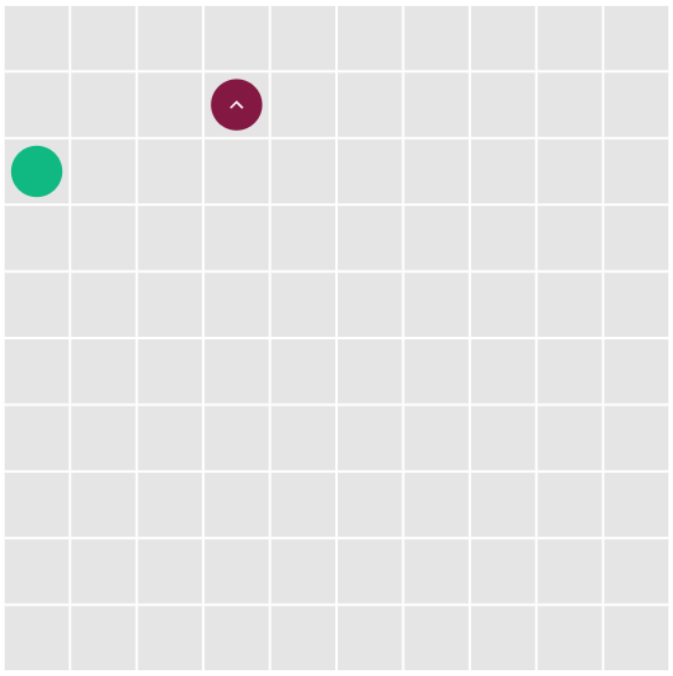
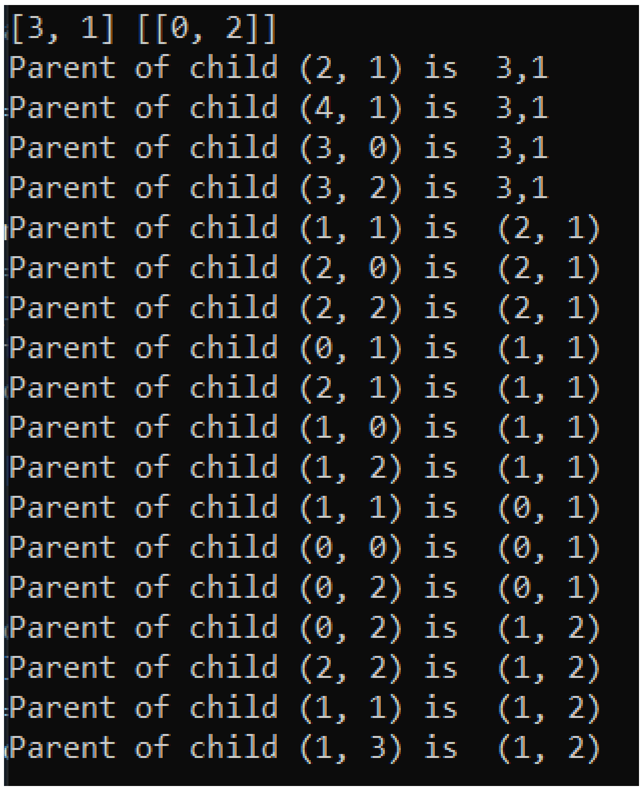


Figure 5.2.1.2 Example of current state

‹ ‹ ‹ ⌄

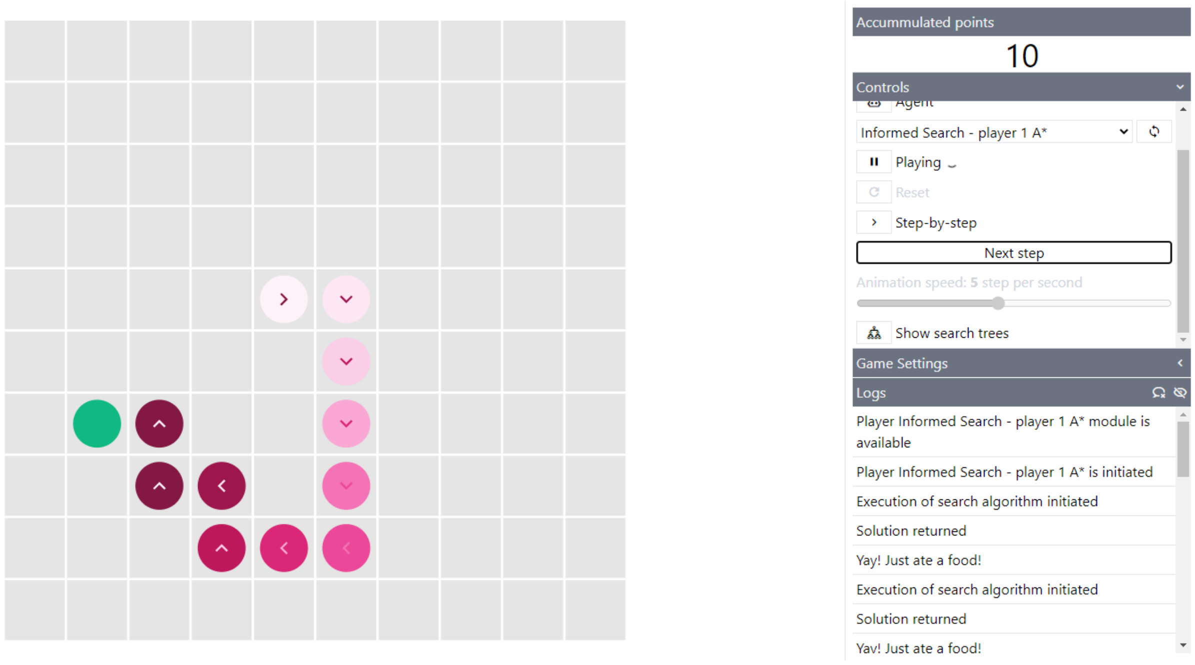
Figure 5.2.1.3 The Solution to the Example



```
[3, 1] [[0, 2]]
Parent of child (2, 1) is  3,1
Parent of child (4, 1) is  3,1
Parent of child (3, 0) is  3,1
Parent of child (3, 2) is  3,1
Parent of child (1, 1) is  (2, 1)
Parent of child (2, 0) is  (2, 1)
Parent of child (2, 2) is  (2, 1)
Parent of child (0, 1) is  (1, 1)
Parent of child (2, 1) is  (1, 1)
Parent of child (1, 0) is  (1, 1)
Parent of child (1, 2) is  (1, 1)
Parent of child (1, 1) is  (0, 1)
Parent of child (0, 0) is  (0, 1)
Parent of child (0, 2) is  (0, 1)
Parent of child (0, 2) is  (1, 2)
Parent of child (2, 2) is  (1, 2)
Parent of child (1, 1) is  (1, 2)
Parent of child (1, 3) is  (1, 2)
```

Figure 5.2.1.4 The Solution shown in the terminal

5.2.1 Challenge 2 Figures:



Accumulated points

10

Informed Search - player 1 A*

Animation speed: 5 step per second

```
Player Informed Search - player 1 A* module is available
Player Informed Search - player 1 A* is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
```

Figure 5.2.2.1 Challenge 2. 15 Food, Static snake length, 1 food spawn achieved

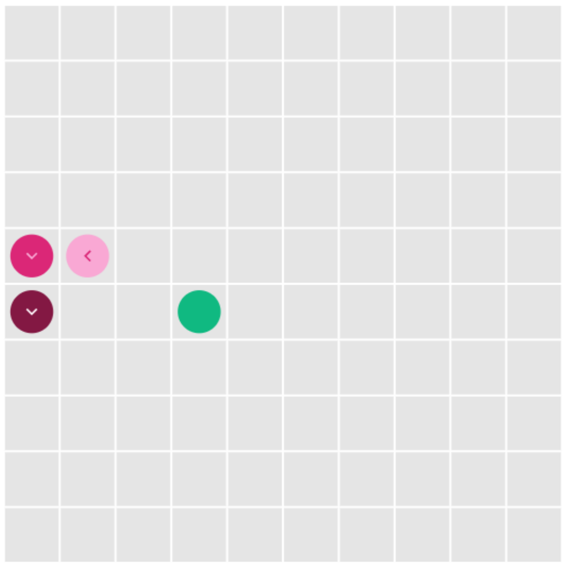
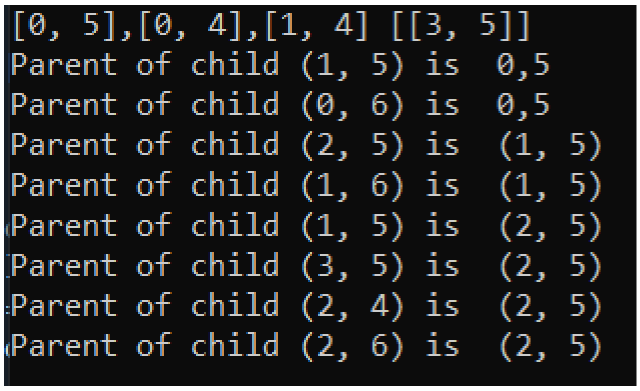


Figure 5.2.2.2 Example of current state

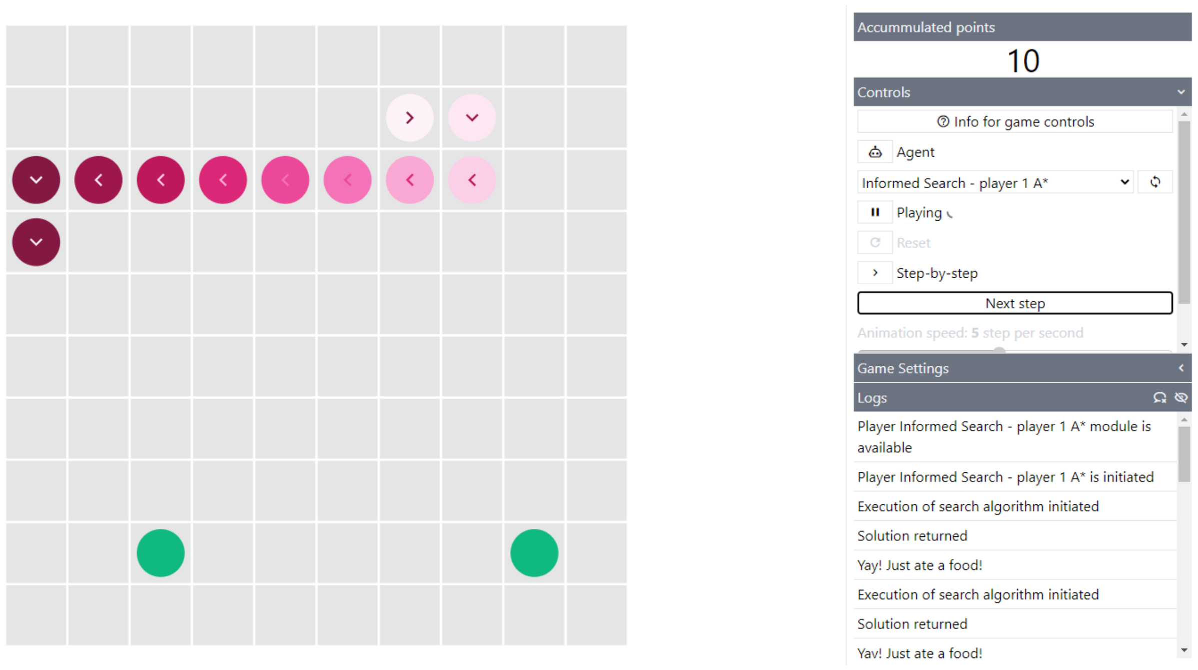
Figure 5.2.2.3 The Solution to the Example



```
[0, 5],[0, 4],[1, 4] [[3, 5]]
Parent of child (1, 5) is  0,5
Parent of child (0, 6) is  0,5
Parent of child (2, 5) is  (1, 5)
Parent of child (1, 6) is  (1, 5)
Parent of child (1, 5) is  (2, 5)
Parent of child (3, 5) is  (2, 5)
Parent of child (2, 4) is  (2, 5)
Parent of child (2, 6) is  (2, 5)
```

Figure 5.2.2.4 The Solution shown in the terminal

5.2.1 Challenge 3 Figures:



```
Player Informed Search - player 1 A* module is available
Player Informed Search - player 1 A* is initiated
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Yay! Just ate a food!
```

Figure 5.2.3.1 Challenge 1. 15 Food, Static snake length, 1 food spawn achieved

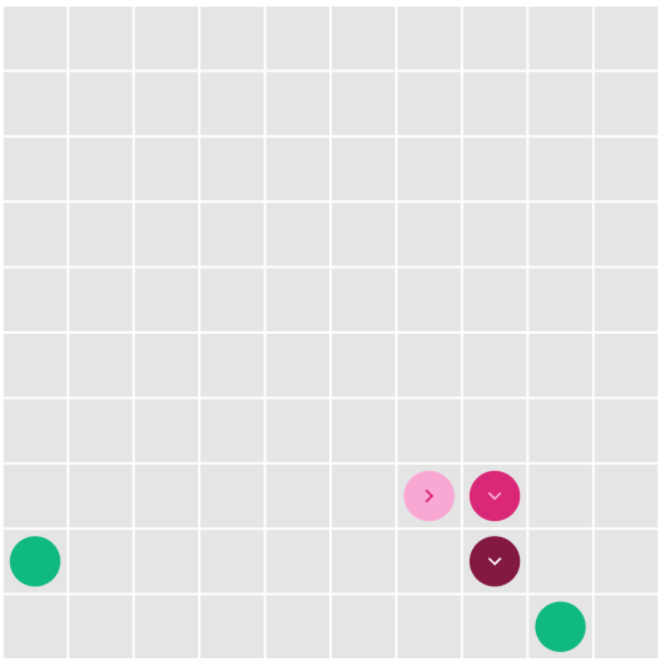


Figure 5.2.3.2 Example of current state

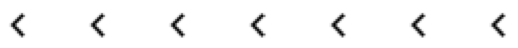
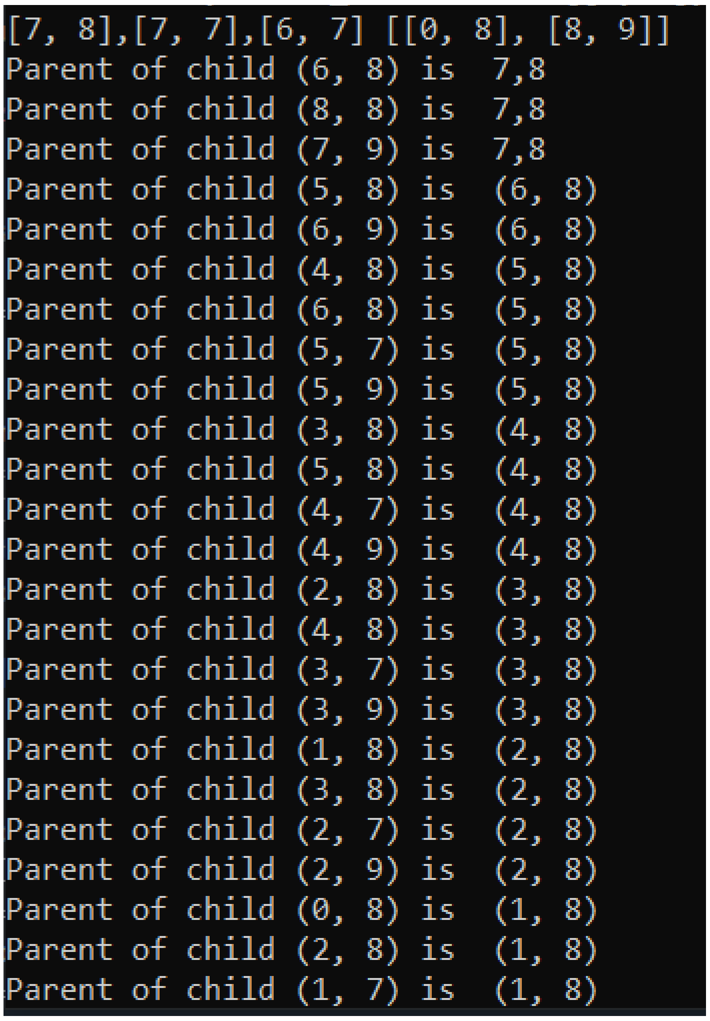


Figure 5.2.3.3 The Solution to the Example



```
[7, 8],[7, 7],[6, 7] [[0, 8], [8, 9]]
Parent of child (6, 8) is  7,8
Parent of child (8, 8) is  7,8
Parent of child (7, 9) is  7,8
Parent of child (5, 8) is  (6, 8)
Parent of child (6, 9) is  (6, 8)
Parent of child (4, 8) is  (5, 8)
Parent of child (6, 8) is  (5, 8)
Parent of child (5, 7) is  (5, 8)
Parent of child (5, 9) is  (5, 8)
Parent of child (3, 8) is  (4, 8)
Parent of child (5, 8) is  (4, 8)
Parent of child (4, 7) is  (4, 8)
Parent of child (4, 9) is  (4, 8)
Parent of child (2, 8) is  (3, 8)
Parent of child (4, 8) is  (3, 8)
Parent of child (3, 7) is  (3, 8)
Parent of child (3, 9) is  (3, 8)
Parent of child (1, 8) is  (2, 8)
Parent of child (3, 8) is  (2, 8)
Parent of child (2, 7) is  (2, 8)
Parent of child (2, 9) is  (2, 8)
Parent of child (0, 8) is  (1, 8)
Parent of child (2, 8) is  (1, 8)
Parent of child (1, 7) is  (1, 8)
```

Figure 5.2.3.4 The Solution shown in the terminal

6.0 Discussion and Evaluation

6.1 Uninformed Search Algorithm - BFS

For each challenge, the search tree, direction solution and the search tree it made in the terminal are evaluated below. The algorithm can run even with changes to parameters such as map size and number of food spawned. The search tree and solution in terms of direction is shown on the website it's hosted on. However, the search tree has issues of overlapping on the side sometimes even though the search tree should be correct such as Figure 5.1.3.3.

6.1.1 Challenge 1 Discussion and Evaluation

The code is runnable. It is able to collect all 15 foods easily. It is able to show the search tree for each solution as shown in Figure 5.1.1.3 and the solution direction in Figure 5.1.1.4. It is also able to show the explored, frontier and children during its search in the terminal as shown in Figure 5.1.1.5. Continuous running of the code allows it to constantly collect food without any error.

6.1.2 Challenge 2 Discussion and Evaluation

The code is runnable. It is able to collect all 10 foods easily without crashing into its own body. It is able to show the search tree for each solution as shown in Figure 5.1.2.3 and the solution's direction in Figure 5.1.2.4. It is also able to show the explored, frontier and children during its search in the terminal as shown in Figure 5.1.2.5. The code is able to run without showing exception errors. However, if the snake traps itself in a circle or the whole body of the snake blocks itself from the food, it will try to move to the nearest explored node until it either finds an opening or crashes into a wall or itself. ✓

6.1.3 Challenge 2 Discussion and Evaluation

The code is runnable. It is able to collect all 10 foods easily without crashing into its own body even when there are 2 foods spawned on the maze at the same time. It is able to show the search tree for each solution as shown in Figure 5.1.3.3 and the solution's direction in Figure 5.1.3.4. It is also able to show the explored, frontier and children during its search in the terminal as shown in Figure 5.1.3.5. The code can run no matter how many food is spawned at a time in the maze. The code is able to run without showing exception error. However, if the snake traps itself in a circle or the whole body of the snake blocks itself from the food, it will try to move to the nearest explored node until it either finds an opening or crashes into a wall or itself. ✓

6.2 Informed Search Algorithm - A* Search

For each challenge, the direction solution and the search tree it made in the terminal are evaluated below. The algorithm can run even with changes to parameters such as map size and number of food spawned. The direction solution is shown on the website it's hosted on.

6.2.1 Challenge 1 Discussion and Evaluation

When the code runs successfully, the snake can collect enough food to reach 15 points easily. The solution of the snake in terms of direction is shown in Figure 5.2.1.3 and the relation of parents and children during searching is also shown in the terminal in Figure 5.2.1.4. With the code, the snake consistently reaches every food without any error.

6.2.2 Challenge 2 Discussion and Evaluation

When the code runs successfully, the snake can collect at least 15 foods without bumping into its own body and it can be seen that the snake tends to move diagonally to the food. The solution of the snake in terms of direction is shown in Figure 5.2.2.3 and the relation of parents and children during searching is also shown in the terminal in Figure 5.2.2.4. With the code, the snake consistently reaches every food without any error. However, in some cases, A* Search makes the snake succeed in eating the food but traps itself within its own body or between itself and the wall. This is incurred by the property of A* Search that it only considers the current situation and how to reach the goal more efficiently without considering possible effects after achieving the goal. Although the A* Search algorithm may be trapped in a loop at times, it will break out of the loop after reaching the maximum number of nodes available. It will then proceed to the next node nearby and do a search again.

6.2.3 Challenge 3 Discussion and Evaluation

When the code runs successfully, the snake can collect at least 10 foods without bumping to its own body or walls. The solution of the snake in terms of direction is shown in Figure 5.2.3.3 and the relation of parents and children during searching is also shown in the terminal in Figure 5.2.3.4. With the code, the snake consistently reaches every food without any error. However, just like challenge 2, the snake will sometimes be trapped within its own body or between itself and the wall.

7.0 Conclusion

Both search algorithms are able to perform their tasks and complete the minimum requirement of each challenge. However, as they are just search algorithms and do not look ahead to see if the path taken will trap themselves, there are chances for the snake to trap itself in a circle and lose. On another note, the snake will not run into a wall or its own body unless there is no other pathway for it to go anymore. There are also distinct differences to the path they take when attempting to get the food. For the BFS search algorithm, the snake will take the shortest path with the minimum amount of turn made to reach the food. A* Search algorithm will take a snake-like pathing, with lots of turns towards the food after being diagonally set towards the food. Another difference is that BFS algorithm will not loop in itself, while A* Search algorithm has a chance to loop in itself at times, but the code will not loop indefinitely as a limiter is placed in it. During the looping, the code will take a long time before it can break out of the loop and proceed to its next neighbouring node and re-searching a new path to the food with a new snake's body and head coordinate given.