



CSC3206 Artificial Intelligence April 2021

Assignment 1

Group Name: Novice AI

Group Member:

Lim Bing Xuan 17075748

Ow Yeong Mun Yin 17076431

Tan Wen Kang 17060955

Arun Alagusunthram a/l Venkatachalam 17035668

Table of Contents

Introduction	3
Rules in snake game	4
Problems to be solved by the Algorithm.	5
Problem formulation in snake game.....	6
Search algorithm implementation	7
Input and output of the algorithms	8
How the algorithms deal with multiple food in the snake game.....	8
Dependencies of the algorithm	8
Node Structure of the algorithm	9
Breath-first search	10
Pseudocode of the algorithm.....	10
Greedy Best-first search.....	11
Proposed heuristic function	11
Pseudocode of the algorithm.....	20
Results of both types of algorithms	21
The performance of the agents/players	22
What causes death in the proposed search agents?	23
Informed agent	23
Uninformed agent.....	23
Appendix	24

Introduction

This project focuses on developing agents that can play the snake game through the application of artificial intelligence concepts. The aim of the agents is to achieve the highest possible score without the snake colliding with its body or against a wall. The agents should be capable of playing the snake game in multiple modes which include:

- Game mode 1: Snake game with non-increasing snake length and one food spawned at a time.
- Game mode 2: Snake game with increasing snake length and one food spawned at a time.
- Game mode 3: Snake game with increasing snake length and two food spawned at a time.

Search techniques in artificial intelligence will be used as the core of the problem-solving agents. In this project, we will develop two agents with different types of search techniques which are uninformed search and informed search, and a comprehensive comparison will be done between both agents on the performance of the algorithm.

To increase the difficulty of the project, multiple challenges are defined to evaluate the agents. The challenges are defined as below:

- Achieve at least 15 points for Game mode 1.
- Achieve at least 10 points for Game mode 2.
- Achieve at least 10 points for Game mode 3.

Rules in snake game

Prior to the problem formulation, the rule of the game needs to be defined. This is because the agents must obey the game rules while playing the game. Without providing game rules to the agents, the solution may not take game rules into consideration and may lose the game unwarily.

The general rules of the game are:

1. The game will end when the snake collides with its body or tail.
2. The game will end when the snake hits the wall.

The specified rules for each game mode are:

Snake game with non-increasing snake length and one food spawned at a time.

- The length of the snake will not increase when a food is consumed.
- One food will be generated on a random empty space when all food is consumed.

Snake game with increasing snake length and one food spawned at a time.

- The length of the snake will increase by one when a food is consumed.
- One food will be generated on a random empty space when all food is consumed.

Snake game with increasing snake length and two food spawned at a time.

- The length of the snake will increase by one when a food is consumed.
- Two food will be generated on two random empty spaces when all food is consumed.

Problems to be solved by the Algorithm.

The algorithm will ^{is} as an agent to play the snake game using search algorithms.

The problems that need to be solved by the algorithm are inherent to the rules and behavior of the game as stated in the previous section.

These problems include:


1. Generate a path from the snake's head's current location to the location of the food.
2. Prevent the snake from hitting the wall.
3. Prevent the snake from colliding with its tail or body.

The algorithm needs to solve the above problems while trying to achieve the highest possible score in the game. The snake's increasing body length is also a problem that needs to be taken into consideration by the algorithm as it might lead to the death of the snake if there is insufficient empty space allocated for the growth. If this factor is ignored, the snake's head might collide with its body or tail after the snake's body length increases after the consumption of food.

These problems are described in the form of a problem formulation in the next section to better illustrate how the different parameters of the problem fit into the algorithm.

Problem formulation in snake game

A problem must be formulated before the development of the algorithm. This is because problem formulation provides a general idea to the agents on how the search tree could be expanded to find the goal.

Problem formulation	
<u>Path Cost</u>	There is no path cost in snake game. Each cost of the actions are the same. 
<u>State:</u>	States are the current situation of the game. In the snake game, the state refers to the snake's current head and body position in the game and its head's direction.
<u>Action</u>	<p>Actions are possible decision that can be made when expanding a state to find the goal.</p> <p>Possible actions in the game are:</p> <ul style="list-style-type: none">- Move East ('e')- Move South ('s')- Move West ('w')- Move North ('n') <p>The possible actions vary depending on the current state.</p> <ol style="list-style-type: none">1. The action that is opposite to the current direction of the snake's head will be removed as the snake is not allowed to move backwards in the game. For example, if the current direction of the head of the snake is 'e', then the possible actions will be 'e', 's', 'n'.2. The action that will lead to collision with the snake's body will be removed. For example, if the next action will cause a collision between the snake's head with its body, the action will be removed from the list of possible actions as the game will end when the snake bites itself.3. The action that will lead to the between the snake collision with the wall will be removed. For example, if the next action will cause the snake to collide with the wall, the action will be removed from the list of possible actions as the game will end when the snake hits the wall.
<u>Goal</u>	The goal state is the food location in the game. This is because the aim of the agents is to increase game score by eating a food in the game.

Through the problem formulation, the input required for the algorithm are defined as below:

Initial state: The starting state of the snake's head and body in the game and its head's current direction.

Goal state: Food location.

Search algorithm implementation

This section will discuss the implementation of two types of search algorithms in this project. The search techniques implemented in the project are:

- **Breath-first Search (Uninformed Search)**
Considering the absence of path cost in the snake's problem, breath-first search is an ideal choice for the uninformed search in this project. This is because breath-first search can obtain the solution without the need of known cost ($g(x)$).
- **Greedy Best-first search (Informed Search)**
Similarly, Greedy Best-first search is chosen as the informed search of our project because it only requires heuristic cost ($h(x)$) to find the solution. We will propose a heuristic function that can evaluate the state and search for an optimal solution that can maximize the game score while minimizing the average steps needed to consume the food.

To better explain the algorithm, this section will be divided into three parts.

The first part of this section covers topics that are applicable to both the algorithms which include:

- Input and output of the algorithms
- How the algorithms deal with multiple food in the snake game
- Dependencies of the algorithms
- Node structure

The second part of this section discusses about the Implementation of Breath-first search while the third part is about the Implementation of Greedy Best-first search.

Input and output of the algorithms

The inputs and outputs of both breath-first search algorithm and greedy best-first search are the same.

There are two inputs required for the algorithm which are:

- Initial state of the game

The initial state of the game are required as the inputs.

An example of the input is:

```
{  
  "snake_locations": [[0,1],[0,2]], # Array of snake body coordinates in sequence (head to body)  
  "current_direction": 'e',         # Current direction of the snake's head  
  "food_locations": [[1,2],[2,2]],  # Array of all food coordinates in the game  
}
```

- Goal state of the game

The goal state of the game are required as the input.

An example of the input is:

```
[1,1] # goal coordinate (usually food)
```

The output of the algorithm are:

- List of actions to the goal state

If the goal state has been reached, the algorithm will return the list of actions from the initial state to the goal state in order. If the goal state has not been found after searching through all possible states, a random solution is returned to avoid the frontend snake game from being stuck while waiting for a solution from the python script.

An example of solution is:

```
['e','s','w','n']
```

- Search tree

The search tree is a tree that contains all the nodes expanded by the search algorithm. Similar to the output of the solution, the search tree will be returned if the goal state is found during the search and a random search tree is returned if the goal state is not found after searching through all the possible states.

How the algorithms deal with multiple food in the snake game

The algorithms will automatically choose the first food location element as the goal state of the search in the food locations array provided by the snake game frontend.

Dependencies of the algorithm

The algorithm makes use of external libraries shown below for calculating the number of detached empty regions in the snake game maze:

- Numpy
- cv2

Node Structure of the algorithm

The node structure is defined as a python class as shown in Figure 1. A node object will store all relevant information of the node in the search tree. To avoid confusion, both Breath-first search and Greedy Best-first search will use the same node structure to implement the algorithm.

By storing all relevant information of a node in an object, we can:

- Easily obtain the list of actions that leads to this state.
- Easily generate the information of a search tree required by the snake game frontend.

```
class Node:
    def __init__(self, id=1, actionLeadtoThisState=None, state=None, parent=None, expansionSequence=-1, heuristicCost=0, removed=False):
        self.id = id
        self.actionLeadtoThisState = actionLeadtoThisState
        self.state = state
        self.parent = parent
        self.childrens = []
        self.expansionSequence = expansionSequence
        self.heuristicCost = heuristicCost
        self.removed = removed

    def getSnakeHeadCoordinate(self):
        return self.state["snake_locations"][0] or None

    def getAllSnakeBodyCoordinate(self):
        return self.state["snake_locations"]

    def getNodeDisplayableInfo(self):
        # This Function will format the node object to the required format by the frontend
        #
        # {
        #   "id": 2,
        #   "state": "5,8",
        #   "expansionSequence": 2,
        #   "children": [5,6,7],
        #   "actions": ["n","s","w"],
        #   "removed": False,
        #   "parent": 1
        # }
        nodeInfo = {
            "id": self.id,
            "state": ','.join([str(coor) for coor in self.getSnakeHeadCoordinate()]),
            "expansionSequence": self.expansionSequence,
            "children": [],
            "actions": [],
            "removed": self.removed,
            "parent": getattr(self.parent, 'id', None)
        }

        # Generate children information
        for child in self.childrens:
            nodeInfo["children"].append(child.id)
            nodeInfo["actions"].append(child.actionLeadtoThisState)

        return nodeInfo
```

Figure 1: Node structure

Breath-first search

In this project, we implemented a classic breath-first search to find a solution from the initial state to the goal state.

Pseudocode of the algorithm

To provide a better understanding of the algorithm, the explanation will be provided in pseudocode. The steps of the algorithm are as below:

1. Read the input of initial state and create a Node object as shown in Figure 1.
2. Create a frontier list and add the initial state into the list.
3. Create an explored list to keep track of explored states.
4. Loop the following until the goal is found or all frontier has been expanded:

 Get the first node in frontier.

 Obtain all possible actions for the state except for the action that is in the opposite direction of the snake head's current direction

 Loop through each possible action:

 Generate the new state after executing the action

 If the new state does not cause the snake to collide with its body or against the wall:

 Create a Node object for the new state

 If the new state is unexplored:

 Perform a goal test.

 Add the Node for the new state to the frontier list.

 Add the Node for the new state to the explored list.

5. After the loop, check if the goal has been found. If the goal is found, return the solution and the corresponding search tree. If the goal is not found, return a random solution and a random search tree.



Greedy Best-first search

In this project, we have implemented a greedy best-first search to find a solution from the initial state to the goal state.

Proposed heuristic function

To ensure great performance, an effective heuristic function needs to be used to estimate the heuristic score of each action. A heuristic function is a function which how close a state is to the goal, the lower the heuristic score, the better the solution. Generally, greedy best-first search will prioritize expanding the node/state with the lowest heuristic score ($h(x)$).

Characteristic of a good state

To develop a function that can accurately estimate the score of a solution, a comprehensive analysis needs to be done on the snake game to determine factors that can be used to evaluate the state. The goal of our heuristic function is to minimize the chances of losing the game. A low heuristic score would imply that it is a good solution. Prior to developing the heuristic function, we have concluded a list of characteristics of a good state in the snake game as the core of our heuristic function which are:

- The presence of a path from the snake's head to its tail

If a path from the snake's head to its tail exists, the agent would not lose the game as the snake's head would not collide with its body.

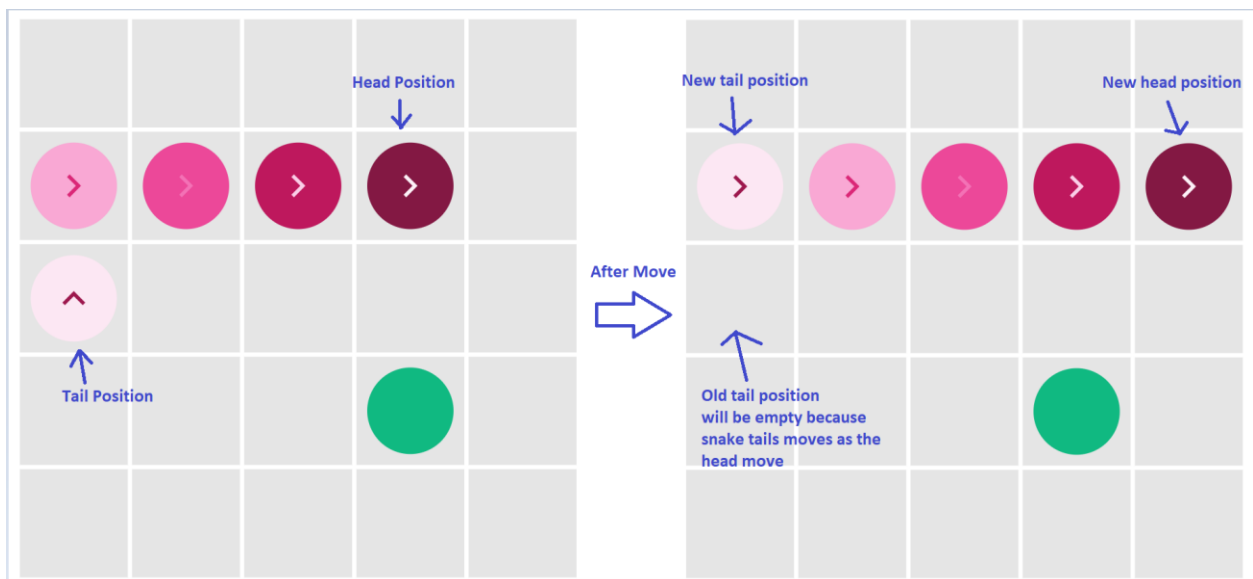


Figure 2: Example: when head moves, tail will also moves

This is because the snake's tail will move as our its head moves. Therefore, if a path from the snake's head to its tail exists, the agent would not lose the game as the space occupied by the tail would be vacated once its head moves.

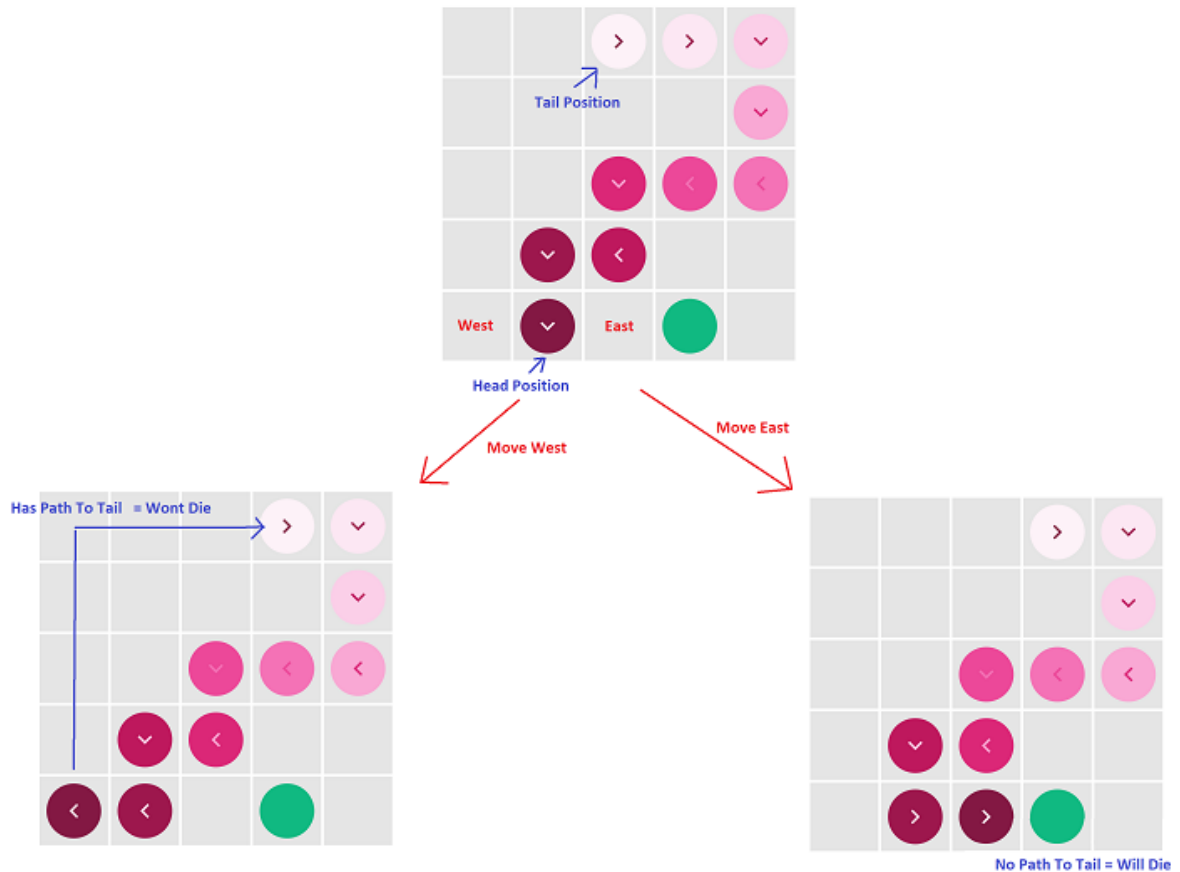


Figure 3: Example: A state with path to tail is better

Consider the example in Figure 3, the initial state has two possible actions which are move 'east' or move 'west'. If the snake moves west, a path from the snake's tail to its head exists. However, if the snake moves east, there is no path from the snake's tail to its head, in other words, this will lead to a dead end and the game will end. Therefore, the presence of a path from the snake's head to its tail is a crucial factor that should be taken into consideration for the heuristic function.

- The presence of a path to a food and the presence of a path from the food to the snake's tail.

The presence of a path between the snake's head and its tail does not suffice as such a path would only ensure that the snake would not die. Hence, there is a need to check for the presence of a path from the current state to the food and the presence of a path from the food to the snake's tail. There will be chances where multiple new states contain a path to the snake's tail and have a similar heuristic score. Therefore, we will need to further evaluate the action to allow the algorithm to make a better decision. The ideal action would be that a path from the snake's current state and the food exists while there is a path from the snake's head to its tail after the snake consumes the food. If the action meets this ideal situation, the heuristic score will be lower.

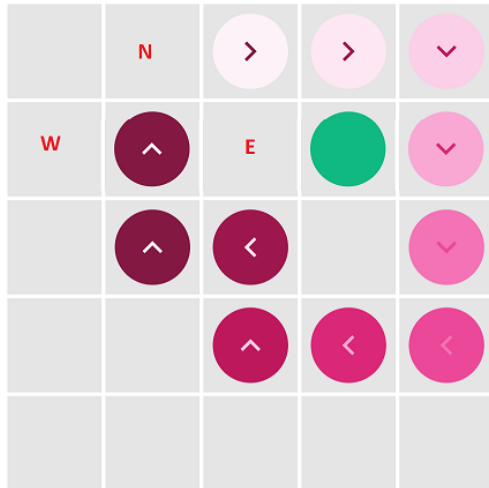


Figure 4: Example: Presence of path to food and path to tail after eating food.

As shown in Figure 4, there are 3 possible actions that can be made in the initial state which are move 'east', move 'west', or move 'north'. Although all new states of moving 'east', 'west' and 'north' contain a path from the snake's head to its tail, only the new state after the snake moves 'west' and the new state after the snake moves 'north' meet the aforementioned ideal situation. The new state after the snake moves 'east' does not meet the ideal situation because there will be no path from the snake's head to its tails after the snake consumes the food.

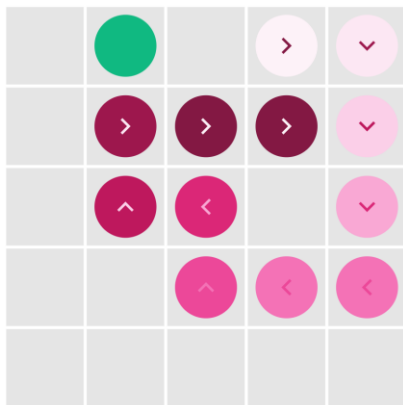


Figure 5: Example: Moving 'east' does not meet the ideal situation

Figure 5 illustrates that there will be no path from the snake's head to its tail after the snake consumes the food if the snake moves 'east'.

- Less detached empty region in the snake game maze

Hamilton cycle is one of the most popular methods implemented for the snake game. However, the efficiency of the solution is low as the average number of steps for a snake to eat a food in the game is extremely high because a Hamilton cycle requires every vertex once to be visited once. Besides, the computational complexity of a Hamilton cycle search algorithm is high. After analyzing the

Hamilton cycle, a characteristic of a Hamilton cycle discovered is that there will only be one detached empty region in the maze while the snake navigates around the Hamilton cycle. Therefore, a new concept where the lower the number of detached empty spaces in the maze will allow our algorithm to take advantage of the benefit of using a Hamilton cycle while minimizing the average number of steps for the snake to eat a food. A detached empty space refers to an empty region on the maze that is separated by the snake body. Through some experimentation it is concluded that:

If there are many detached empty spaces in the game, there will be a possibility that some detached spaces cannot be utilized by the snake to eat a food and will cause the snake to come to a dead end.

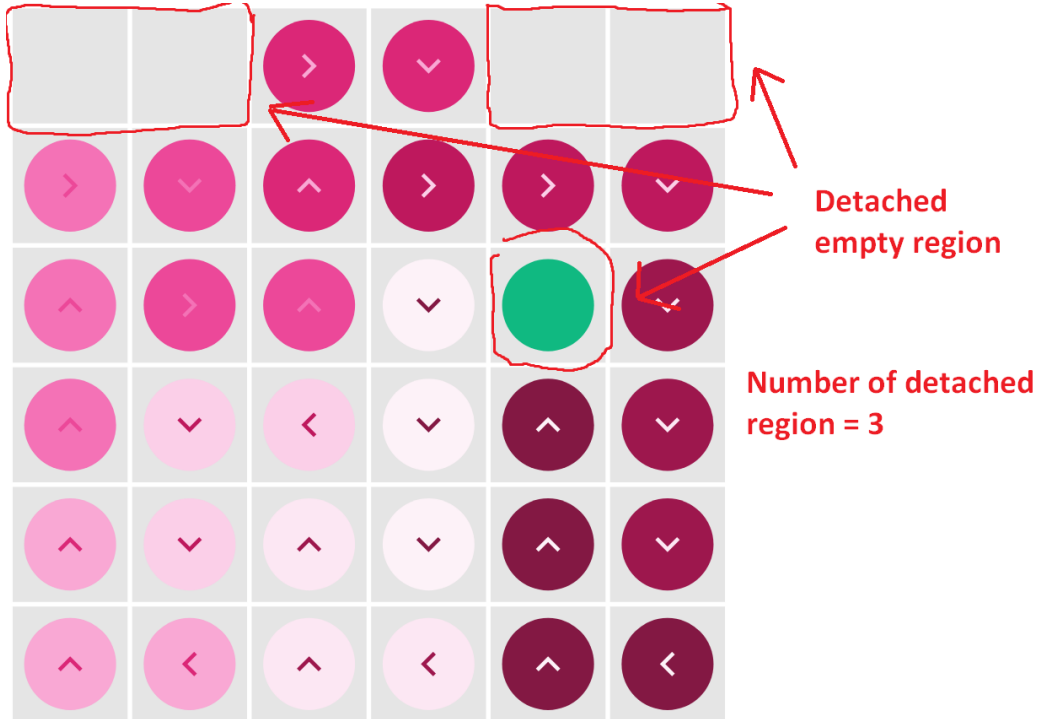


Figure 6: Example: detached region cannot be utilized for a snake to eat a food

Figure 6 depicts an example where the snake cannot find any space to utilize for eating the food because of the high number of detached regions. This situation can be avoided if we have only one detached region.

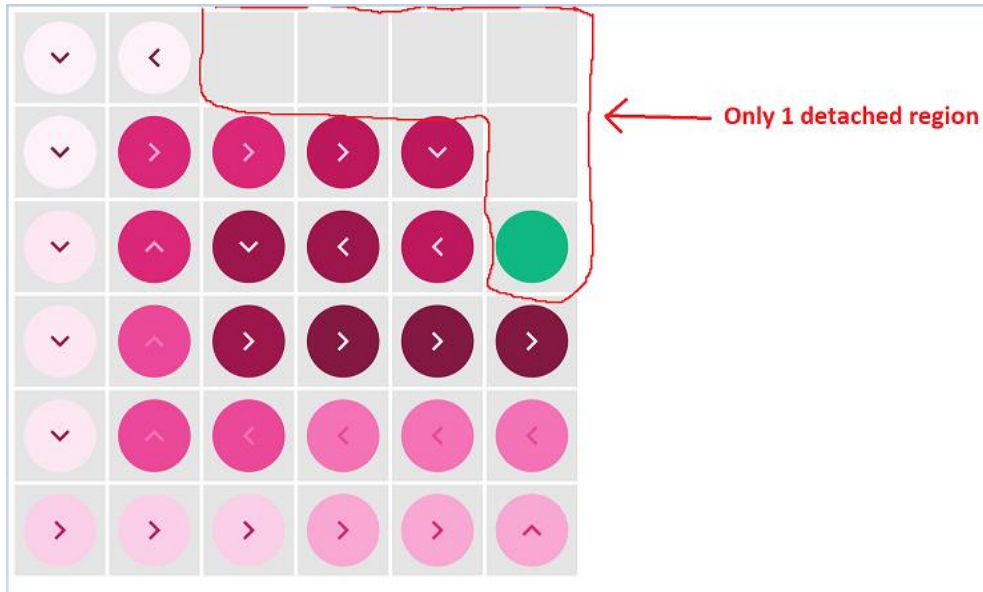


Figure 7: Example: If we keep our number of detached region to 1, we will be able to utilize all empty space

Figure 7 depicts the results when we keep the number of detached empty regions to one. If we keep the number of detached empty regions to one, the snake will be able to utilize all empty spaces.

- Maximum number of steps to snake's tail

Other than the three aforementioned characteristics, a higher number of steps for the snake's head to reach its tails is also a characteristic of a better action. This is because a higher number of steps for the snake's head to reach its tail would minimize the chance of losing the game. In the snake game, food is generated in a random empty space, if the number of steps for the snake's head to reach its tail is low, there is a limited number of empty spaces for the snake to utilize before its head reaches its tail. However, if the food is spawned in front the snake at this point of time, we will lose the game because the snake length will increase by one after the snake consumes the food, and the number of empty spaces is insufficient to accommodate the increased tail length.

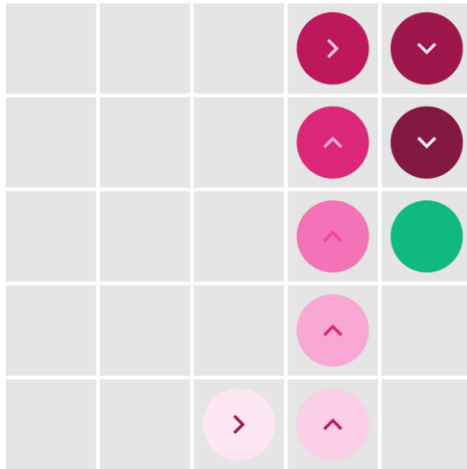


Figure 8: Example: Situation where we just leave just right amount of step to our tail.

Figure 8 illustrates a situation where there is just a sufficient number of steps for the snake's tail to grow after consuming the food without resulting in the collision between the snake's head and its tail.

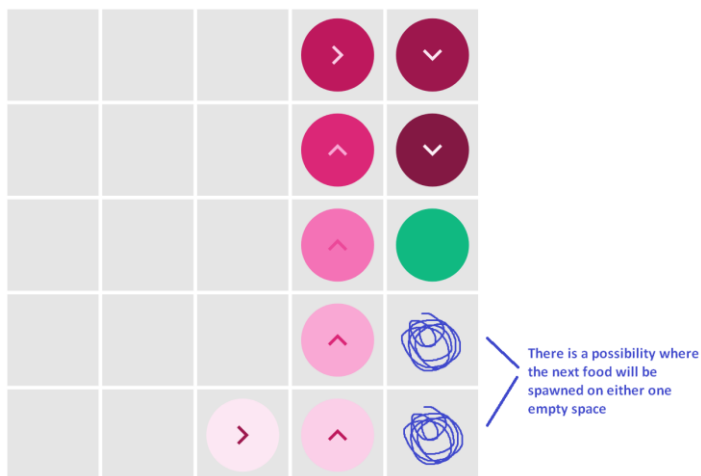


Figure 9: Example: Location of next food that will lead to a collision on our snake tails

Please note that there is a possibility that food will be spawned at one of the two blue space highlighted in Figure 9. If the next food spawns at one of the blue space highlighted, the snake will consume the food and its tail's length will increase, resulting in the collision between its head and tail. To avoid this possibility, the algorithm will prioritize choosing a path that has the maximum number of steps from the snake's head to its tail.

Prioritization of each characteristic

The sequence of prioritization of the characteristics mentioned is shown below with decreasing priority:

1. The presence of a path from the snake's head to its tail
2. The presence of a path to a food and the presence of a path from the food to the snake's tail.
3. Less detached empty region in the snake game maze
4. Maximum number of steps to snake's tail

This prioritization sequence will be reflected in their respective weightage in the calculation of the heuristic score using the heuristic function. The higher the priority of the characteristic, heavier its weightage in the calculation of the heuristic score.

Characteristic 1 has a higher level of prioritization because it may lead to a dead state if the characteristic is not met. Characteristics 2, 3 and 4 will only increase the chances of losing the game if the characteristics are not satisfied.

Implementation of heuristic function

This section will provide a detailed explanation of the implementation of the heuristic function in the algorithm.

Finding the number of steps from the snake's head to its tail

Characteristic 4 will be examined using breath first search. Breath first search is an appropriate method for this purpose because we are looking for the path with the least number of steps for the snake's head to reach its tail. To allocate higher score for a path the least number of steps, a subtraction will be performed between a constant and the number of steps from the snake's head to its tail. The constant value used for the subtraction will vary based on the maze size, to allow the algorithm to accommodate to varying maze sizes. For example, for a snake game with a 5x5 maze size, the constant value will be 25.

If the number of steps from the snake's head to its tail is 5, the calculation of heuristic score will be: $25 - 5 = 20$.

If the number of steps from the snake's head to its tail is 3, the calculation of heuristic score will be: $25 - 3 = 22$.

The example above show that the heuristic score is decreases with the number of steps from the snake's head to its tail.

Determination of the existence of a path from the snake's head to its tail.

Breath first is also implemented to determine if a path from the snake's head to its tail exists. If a path from the snake's head to its tail is not found, the heuristic score will be extremely high as this is the top prioritization. To ensure an extremely high heuristic score over other characteristics, the mazeSize will be multiplied by 100. For example, if a path from the snake's head to its tail is not found in a snake game with a 5x5 maze, the calculation of the heuristic score will be: $25 * 100 = 2500$.

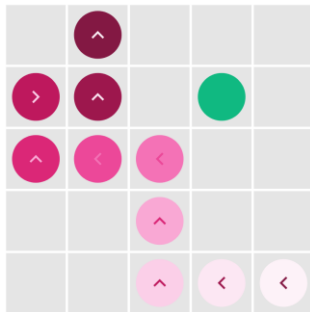
Determination of the existence of a path to a food and the presence of a path from the food to the snake's tail.

Breath first search is again implemented to determine the existence of a path to a food and the presence of a path from the food to the snake's tail. If a path to a food or a path from food to snake's tail is not found, the heuristic score will be multiplied by 2.

Finding the number of detached empty regions

To find the number of detached empty regions, the connected-component labelling will be implemented. The cv2.connectedComponents function from the OpenCV library will be utilized. The steps to calculate the heuristic score of detached empty regions is described as below:

1. Generate a current snake game situation in an array



For example, the snake game situation shown above will be generated in an 5x5 array

Value 0 will be used to represent the snake's body

Value 1 will be used to represent other empty spaces in the game

```
[  
  [ 1, 0, 1, 1, 1],  
  [ 0, 0, 1, 1, 1],  
  [ 0, 0, 0, 1, 1],  
  [ 1, 1, 0, 1, 1],  
  [ 1, 1, 0, 0, 0],  
]
```

2. Perform cv2.connectedComponents function and find the number of detached empty region

The Cv2.connectedComponents function will assign a label for each detached region. The function will return the number of labels in the array and will be used to find the number of detached empty regions in the game.

For example, after implementing the cv2.connectedComponents function, the array above will become:

```
[  
  [ 1, 0, 2, 2, 2],  
  [ 0, 0, 2, 2, 2],  
  [ 0, 0, 0, 2, 2],  
  [ 3, 3, 0, 2, 2],  
  [ 3, 3, 0, 0, 0],  
]
```

Number of labels returned will be 4.

The final number of detached empty regions will be $4 - 1 = 3$. This is because we will need to exclude the label generated by the snake body.

3. Calculate the heuristic score based on the number of detached empty regions

The contribution of this characteristic to the heuristic score is calculated by:

$(\text{mazeSize}) * (\text{number of detached empty region})$

In this case,

$$25 * 3 = 75$$

The number of detached empty regions has a higher priority over the number of steps from the snake's head to its tail. Therefore, the mazeSize is used as the base heuristic score and is multiply with the number of detached empty regions. This ensures that it has a higher contribution to the heuristic score if the characteristic is not met.

Pseudocode of the algorithm

To provide a better understanding of the Greedy Best-first search algorithm, the explanation will be provided in a form of pseudocode. The steps of the algorithm are as below:

1. Read the input of initial state and create a Node object as mentioned in Figure 1.
2. Create a frontier list and add the initial state into the list.
3. Create an explored list to keep track of explored states.
4. Loop the following until the goal is found or all frontiers have been expanded:
 - Sort the frontier list by the heuristic score in ascending order.
 - Get the first node in the frontier list.
 - Perform a goal test
 - Obtain all possible actions for the state except for the action that is in the opposite direction of the snake head's current direction.
 - Loop through each possible action:
 - Generate the new state after executing the action
 - If the new state does not cause the snake to collide with its body or against the wall:
 - Calculate the heuristic score of the new state.
 - Create a Node object for the new state.
 - If the new state is unexplored:
 - Add the Node for the new state to the frontier list.
 - Else:
 - Retain the state with lower heuristic score in the frontier list between the old state and the new state.
5. After the loop, check if the goal has been found. If the goal is found, return the solution and the corresponding search tree. If the goal is not found, return a random solution and a random search tree.

Results of both types of algorithms

The score achieved by both agents in each game mode are displayed as below:

	Informed Search agent	Uninformed Search agent
<p>Game mode 1</p> <p>Snake game with non-increasing snake length and one food spawned at a time.</p> <p>Challenge: Achieve at least 15 points</p> <p>**Note: For this game mode, both agents do not lose the game due to the non-increasing length of the snake. Therefore, an approximate score is recorded.</p>	<p>Round</p> <ol style="list-style-type: none"> 1. More than 15 points 2. More than 15 points 3. More than 15 points 4. More than 15 points 5. More than 15 points <p>Average Score: -</p>	<p>Round</p> <ol style="list-style-type: none"> 1. More than 15 points 2. More than 15 points 3. More than 15 points 4. More than 15 points 5. More than 15 points <p>Average Score: -</p>
<p>Game mode 2</p> <p>Snake game with increasing snake length and one food spawned at a time.</p> <p>Challenge: Achieve at least 10 points</p> <p>Screenshot of result are included in appendix section.</p>	<p>Round</p> <ol style="list-style-type: none"> 1. 99 points 2. 99 points 3. 99 points 4. 99 points 5. 99 points <p>Average Score: 99 points</p>	<p>Round</p> <ol style="list-style-type: none"> 1. 18 points 2. 28 points 3. 8 points 4. 38 points 5. 6 points <p>Average Score: 19.6 points</p>
<p>Game mode 3</p> <p>Snake game with increasing snake length and two food spawned at a time.</p> <p>Challenge: Achieve at least 10 points.</p> <p>Screenshot of result are included in appendix section.</p>	<p>Round</p> <ol style="list-style-type: none"> 1. 98 points 2. 98 points 3. 98 points 4. 98 points 5. 98 points <p>Average Score: 98 points</p>	<p>Round</p> <ol style="list-style-type: none"> 1. 34 points 2. 37 points 3. 30 points 4. 18 points 5. 6 points <p>Average Score: 25 points</p>

The results show that the informed search agent has successfully achieved all the challenges in the three-game modes mentioned earlier while uninformed agent only achieve the challenge in game mode 1. This shows that the informed search agent is capable of solving the problems it faces to win the game, hence it can be said that a solution has been obtained.

The performance of the agents/players

Both the informed search and uninformed search agents successfully return the solution needed for the frontend. A comprehensive comparison is done between both agents proposed in this project.

	Informed search	Uninformed search
Score achieved	Higher	Lower
Speed of algorithm	Slower	Faster

In terms of the scores achieved by both agents, the informed search agent is capable of consistently achieving a higher score in the game than the uninformed search agent. This is probably because the informed search agent takes all the characteristic mentioned earlier into account. Therefore, the informed search agent can avoid dead corners and minimizing the chances of losing the game. The uninformed search agent on the other hand is a blind search which only takes the number of steps to reach its destination into consideration. Hence, there will be a high possibility where the uninformed search agent may cause the snake to get into dead corners, resulting in losing the game. Besides, it can also be seen in the results that the informed search agent is more consistent in its ability to minimize the probability of losing the game. The score achieved by the uninformed agent is highly reliant on the location where the food spawns which causes it to have an inconsistent score. If the food is spawned in an undesirable location where the path closest to the food will lead to a dead corner, the uninformed agent will undoubtedly select the path as it only takes the number of steps from the snake's head to the food into consideration.

In terms of the speed of the agents, the uninformed search agent outperforms the informed agent by producing a quicker response. This is probably because the computation complexity of the uninformed agent is lower which reduces the amount of processing required. In contrast, the informed search agent has a slower speed due to the tremendous number of calculations required to be done to calculate the heuristic score.

All in all, the informed search is a better solution for the snake game despite having a downside in speed of the algorithm. This is because the score achieved is the main priority for the snake game.

What causes death in the proposed search agents?

The limitation of both agents is an important factor to be discovered to further improve the algorithm. Hence, an analysis is done on both proposed agents on the cause of losing a game by the agents.

Informed agent

Although the informed agent can achieve a high score, there will still be some situations where the informed agent may lose the game. The situations are listed below:

- The informed agent will try to find the action with a maximum number of steps from the snake's head to its tail. However, the informed agent may lead to a dead state, if the food is spawned in front of the snake's head for multiple consecutive iterations until the number of steps from the snake's head to its tail reserved to utilize is insufficient. For example, the agent reserves 5 empty spaces before the snake's head reaches its tail. However, if the food is unluckily spawned on the 5 consecutive empty spaces in front of the snake's head, then it will lead to a dead state as every time the snake consumes a food its length grows by 1. However, this situation happens rarely.

Uninformed agent

The uninformed agent will may lose the game when:

- The nearest path to a food will lead to a dead corner. This is because the uninformed agent always chooses a path that is nearest to the food.

Appendix

Result of Uninformed Search agent (BFS)

Game Mode: Snake game with increasing snake's length and one foods are spawned at a time.

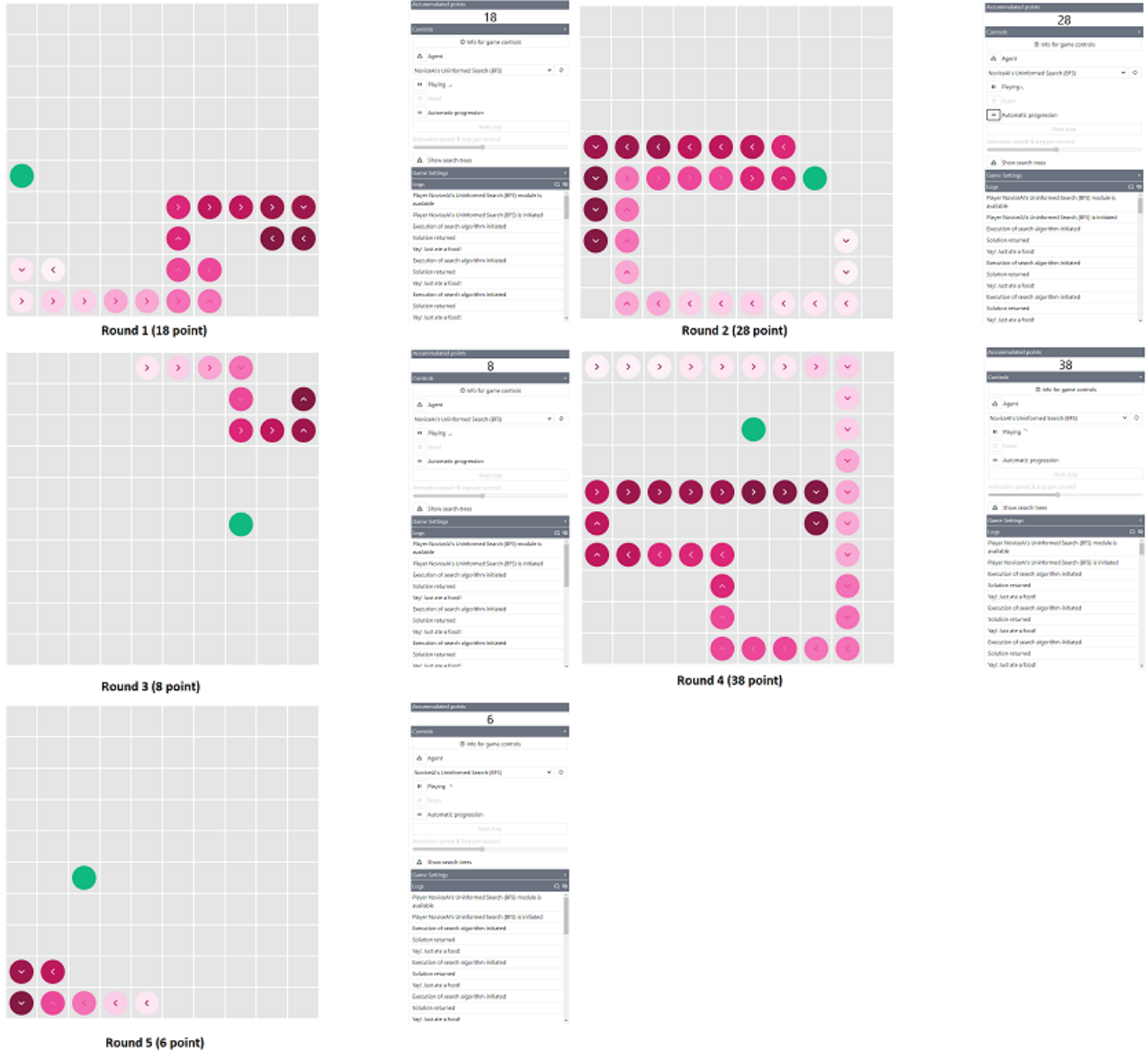


Figure 10: Screenshot of final result of uninformed search agent in game mode 2

Result of Uninformed Search agent (BFS)

Game Mode: Snake game with increasing snake's length and two foods are spawned at a time.

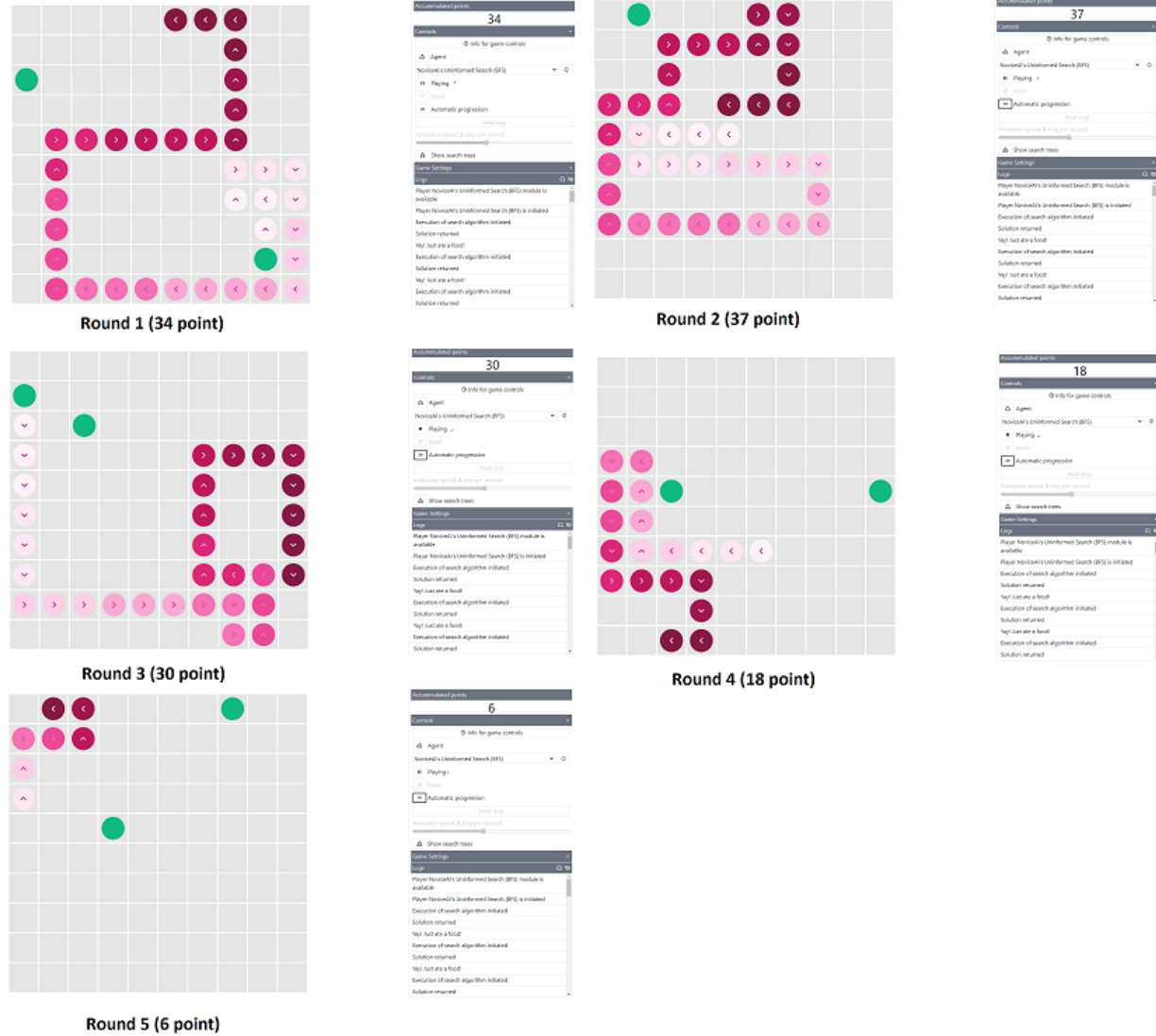


Figure 11: Screenshot of final result of uninformed search agent in game mode 3

Result of informed agent (Greedy Best-first search)

Game Mode: Snake game with increasing snake's length and one foods are spawned at a time.

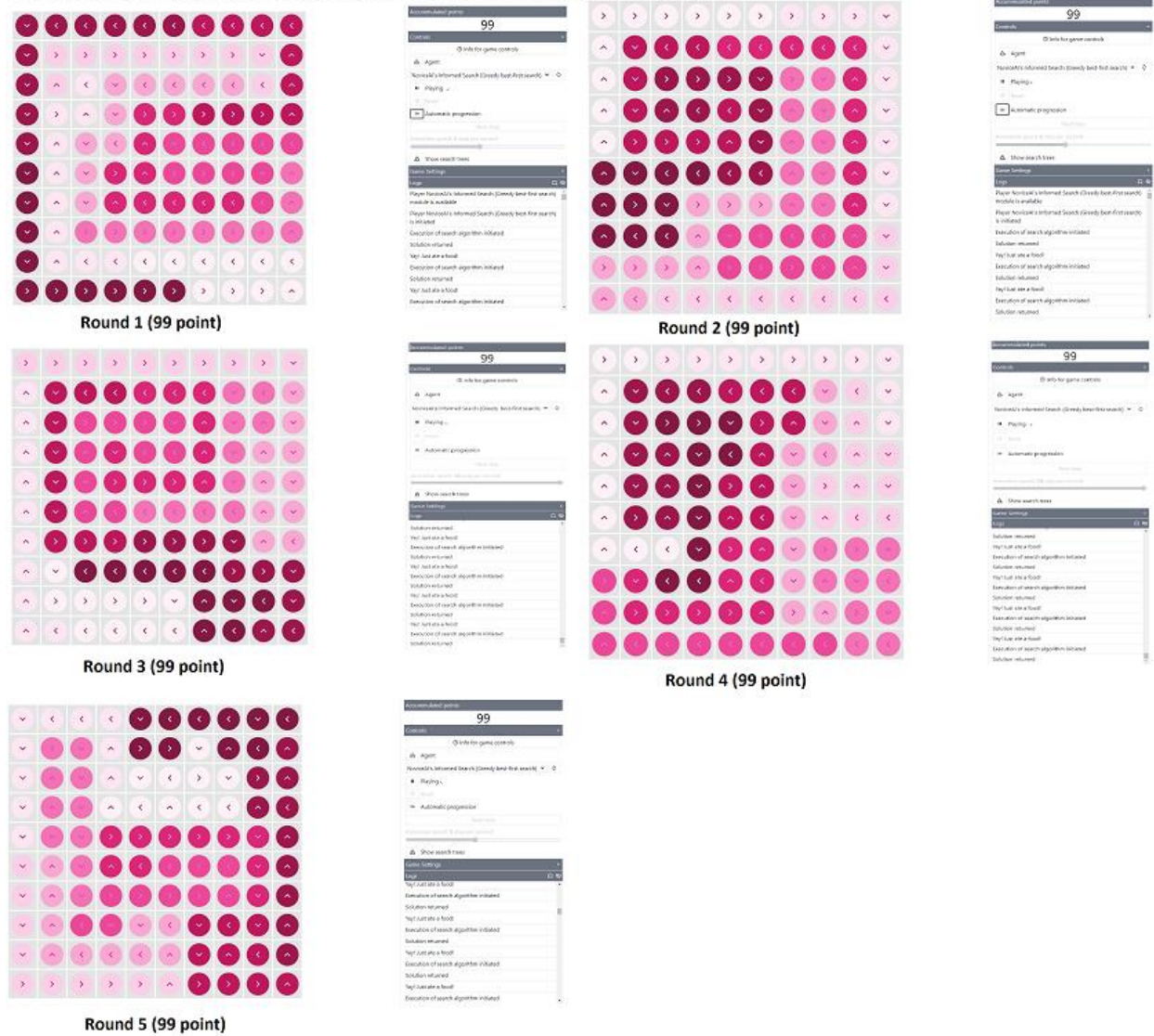
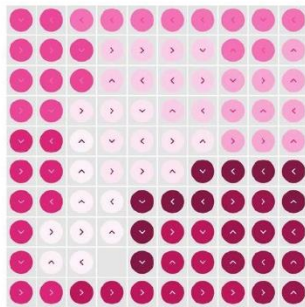


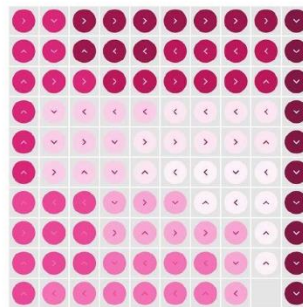
Figure 12: Screenshot of final result of informed search agent in game mode 2

Result of Informed Search Agent (Greedy Best First Search)

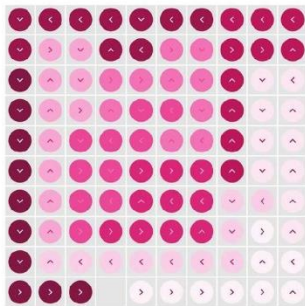
Game Mode: Snake game with increasing snake length and two food spawned at a time.



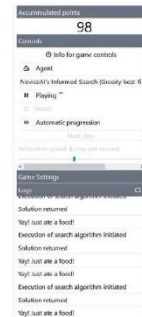
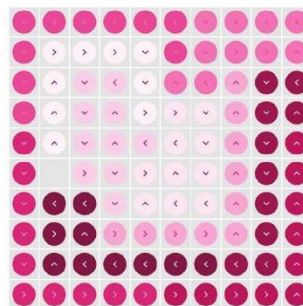
Round 1 (98 points)



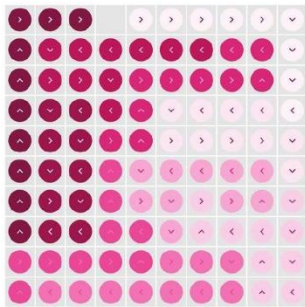
Round 2 (98 points)



Round 3 (98 points)



Round 4 (98 points)



Round 5 (98 points)

Figure 13: Screenshot of final result of informed search agent in game mode 3