AI ASSIGNMENT 1: SNAKE GAME

MAY 2021

Group: Four Stooges

Group members:
Amrita Menon (17029596)
Marcus Teow King Yumn (19024371)
Muhammad Hafiz (16027344)
Liew Tze Chen (18032730)

Table Of Contents

Description.

1. Problem ~~Formulation~~

How can a simple AI achieve designated scores in different modes of the “snake” game using different types of search algorithms?

The aim of this project is to create 2 agents, one using uninformed and one using informed search, to pilot the snake in the snake game and achieve minimum scores in 3 modes of the game, creating 3 challenges to overcome:

1. One food at any time; Non-increasing snake length; Snake length of one; reaching at least 15 points.

2. One food at any time; Increasing snake length with food; Starting snake length of one; reaching at least 10 points.

3. Two food generated when no food is available on the maze; Increasing snake length with food; Starting snake length of one; reaching at least 10 points.

2. Method

Due to goal 3 possessing the requirements of all the other goals combined, a decision was made to base the algorithms on goal 3, as doing so would satisfy the requirements for the other 2 goals as well. For the sake of efficiency, A* search and breadth-first search were chosen as the informed and uninformed search agents’ algorithms, respectively. This is because A* search is essentially a modified version of breadth-first search, as shown in the code where removing line 119 of the A* search algorithm makes it into the breadth-first search algorithm. Furthermore, they are suitable algorithm types to be used in this scenario due to their effectiveness.

2.1 Shared requirements

This is part of problem formulation.

2.1.1 Achieving Goal State

The general purpose of the agents must be achieved, this is done by the playing of the game by the agents. The snake game requires a piece of generated “food” to be eaten by the snake. 1

piece of eaten food provides the player with 1 point. Thus, the agent must be able to pilot the "head" of the snake onto the coordinate where the generated food is located. Then, this must continue until the accumulated points achieve the mentioned scores in said situations. In addition, extra considerations must be taken in order to properly achieve the goal state in each challenge. These are:

1. For goal 1, there is no need for consideration of the length of the snake as there is no chance that the game will end due to the snake "eating" itself. The only extra consideration for the algorithm to achieve goal 1 to achieve the 15-point goal minimum is to not hit the wall of the snake game.

2. For goal 2, the agent is required to consider that the snake's body grows longer the more food it consumes. Hence, the agent should be able to "know" where the body is positioned and if it would potentially connect with itself. Therefore, the considerations to take to achieve goal 2 are that the snake must not bite itself, not connect with the wall, and be informed of the position of its own body.

3. For goal 3, the requirements are similar to goal 2 except that there are 2 pieces of food instead of 1. Hence, the agent must be able to determine which food is in a more efficient position and make the decision to prioritize it.

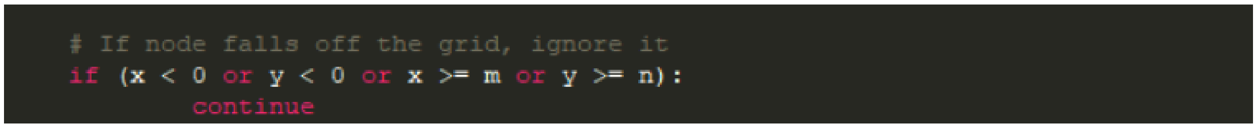
This is discussed further in the Method section.

2.1.2 Preventing Loss in Game

To ensure that the agents do not end the game prematurely, the ways of losing the game must be addressed. The game ends when:

1. The snake touches the wall of the environment.
2. The snake touches its own body

To ensure scenario 1 does not take place, the expanded nodes which may fall out of the grid are removed from consideration. This removes the option for the agent to attempt to pilot the snake into the wall as the only remaining valid nodes are located within the grid. This is implemented as such:



```
# If node falls off the grid, ignore it
if (x < 0 or y < 0 or x >= m or y >= n):
        continue
```

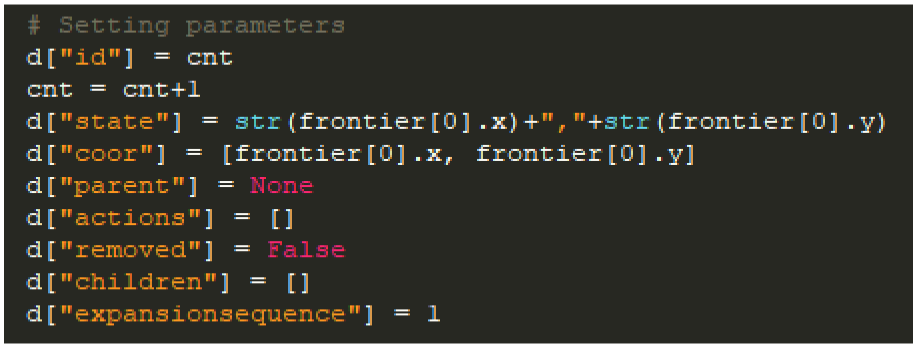
Figure 2.1.2 (Do not touch wall implementation)

To ensure scenario 2 does not take place, expanded nodes which consist of the snake's body are removed. Furthermore, they are not pushed to the frontier queue. Similar to the solution to scenario 1, this removes the option for the agent to pilot the snake into its own body. This is implemented in both scripts and can be seen in lines 157-159 and lines 170-178 of the Breadth-First Search script.

2.1.3 State-space and search tree generation

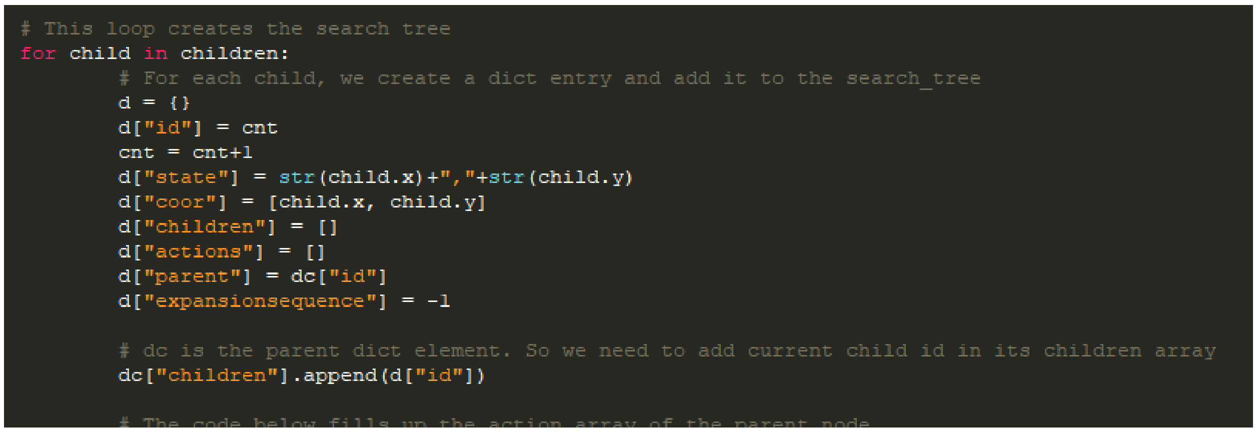
Another requirement of the project is to generate a search tree when the game is played by the agents. To accomplish this, a state-space must be made available. To create a search tree for the decisions made by the agents while accomplishing their goal in this game, the multiple or continuous "goal" states must be considered.

Therefore, because the positioning of the "goal" (fruit) resets every time the snake eats it, the available coordinates and the visited coordinates should reset as well. If not, the snake will run out of non-visited coordinates to manoeuvre through the grid, making it unable to move. To counter this issue, the "graph" or "grid" of the game is created via a dictionary. To add on the keys, with their values in the dictionary, a for loop is used. The information set in the dictionary, helps with the process of creating the search tree, as the id's and other values are automatically placed when the game is running.



```
# Setting parameters
d["id"] = cnt
cnt = cnt+1
d["state"] = str(frontier[0].x)+","+str(frontier[0].y)
d["coor"] = [frontier[0].x, frontier[0].y]
d["parent"] = None
d["actions"] = []
d["removed"] = False
d["children"] = []
d["expansionsequence"] = 1
```

Figure 2.1.3 (a) (Dictionary)

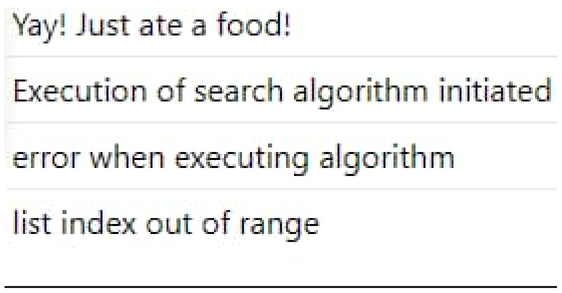


```
# This loop creates the search tree
for child in children:
        # For each child, we create a dict entry and add it to the search_tree
        d = {}
        d["id"] = cnt
        cnt = cnt+1
        d["state"] = str(child.x)+","+str(child.y)
        d["coor"] = [child.x, child.y]
        d["children"] = []
        d["actions"] = []
        d["parent"] = dc["id"]
        d["expansionsequence"] = -1

        # dc is the parent dict element. So we need to add current child id in its children array
        dc["children"].append(d["id"])

        # The code below fills up the action array of the parent node
```

Figure 2.1.3 (b) (For Loop to create Search Tree)

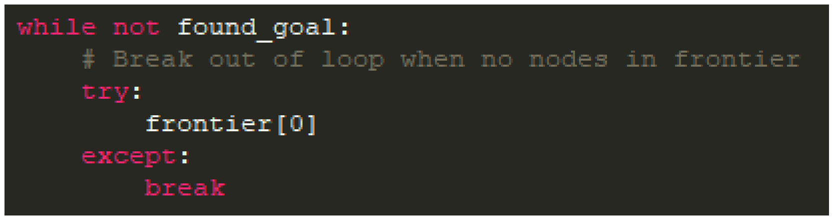
2.1.4 Exception Handling

Upon running the code, there was a recurring error message that always seemed to show up every time the snake ran out of space to move (e.g: was about to eat itself). From the error message, we could determine that the issue lies within one of the lists in the code. After numerous attempts, our group managed to figure out the source of the problem, which is that the loop doesn't break when there are no nodes left in the frontier. A try/except function was used to counter it.



```
Yay! Just ate a food!
Execution of search algorithm initiated
error when executing algorithm
list index out of range
```

Figure 2.1.4 (a) (Error Message)



```
while not found_goal:
    # Break out of loop when no nodes in frontier
    try:
        frontier[0]
    except:
        break
```

Figure 2.1.4 (b) (Try and Except Statement)

2.2 The Algorithms

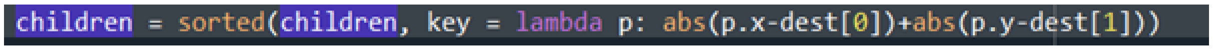
2.2.1 A* Search

A* search is an informed search algorithm. This means that it has information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state. In this algorithm, this is represented by line 119, which is the basis of the algorithm's ability to make decisions, with the help of a heuristic called the Manhattan Distance.

This Manhattan Distance is the sum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates. This is also known as h, which is the distance between the current point to the goal. This heuristic is an approximation heuristic, which saves time to calculate the value of h compared to an exact heuristic. This heuristic was chosen because there are only 4 directions available for the snake to move, satisfying its condition for use.

Next, the working of the A* Search Algorithm in general. The A* search algorithm works based on the estimation function; f(n) = g(n) + h(n) where g(n) is the cost of the path from the starting point to a coordinate on the grid.

First, the agent takes note of the initial coordinates of the snake. Then, the neighbors of the starting point of the snake are identified. Once this information has been collected, it is processed and used to solve the above function. Since this algorithm is part of a computer program, it requires a method to provide it with a reason to choose certain options over others. Thus, the function in figure 2.2.1(a) is required to sort the available child nodes by ascending order, because the smaller sum of f(n) is considered more efficient than a larger sum due to it being the cheapest path. This is done so that the algorithm can choose the ideal path.



```
children = sorted(children, key = lambda p: abs(p.x-dest[0])+abs(p.y-dest[1]))
```

Figure 2.2.1 (a) (A* Implementation)

Then, the agent moves the snake to take the chosen path. Finally, this process is repeated until the goal state is achieved. The process of decision making can be illustrated by the search tree in Figure 2.2.1 (b).

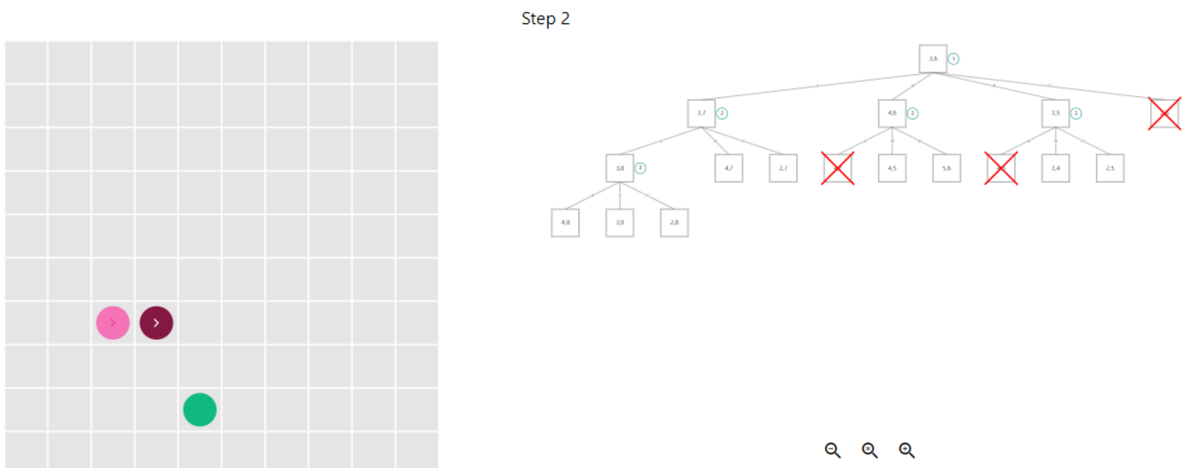
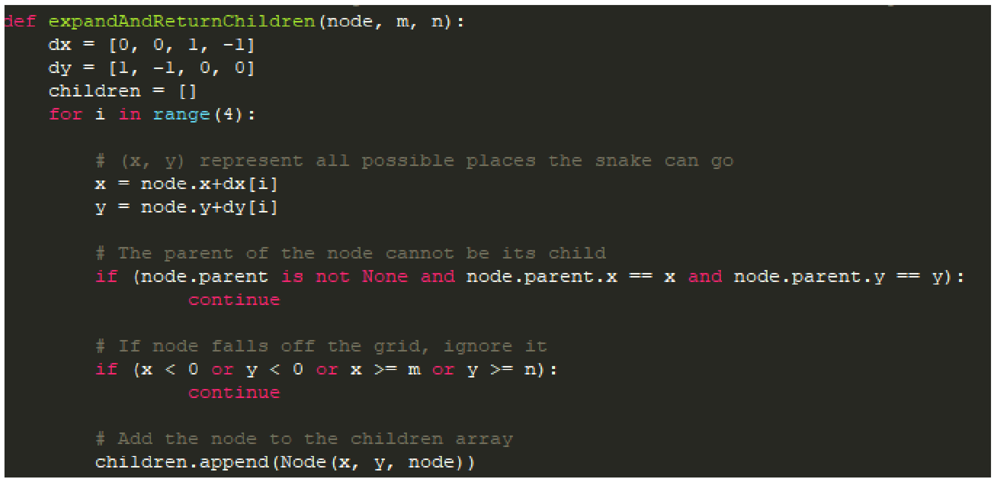


Figure 2.2.1 (b) (A* Search Tree)

2.2.2 Breadth-First Search

In this assignment, Breadth-First Search was chosen to be the ideal uninformed search algorithm. BFS is ideal because it uses the shortest possible path to reach the goal node. This algorithm works via level-by-level traversals in which it goes through the deepest node, followed by the adjacent nodes.



```
def expandAndReturnChildren(node, m, n):
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    children = []
    for i in range(4):

        # (x, y) represent all possible places the snake can go
        x = node.x+dx[i]
        y = node.y+dy[i]

        # The parent of the node cannot be its child
        if (node.parent is not None and node.parent.x == x and node.parent.y == y):
            continue

        # If node falls off the grid, ignore it
        if (x < 0 or y < 0 or x >= m or y >= n):
            continue

        # Add the node to the children array
        children.append(Node(x, y, node))
```

Figure 2.2.2 (a) (Function to determine plausible moves)

After the initialization of the parent and children array for the nodes, the children of parent nodes are determined by examining the nodes adjacent to the snake's head. This is implemented through the function in figure XX.
But first, the initial coordinates are appended into the frontier array and after, set as the explored coordinates. The non-initial coordinates in the frontier array are then popped out one by one and set as explored coordinates, while the adjacent coordinates are pushed into the frontier array. This step is conducted repeatedly until the goal is reached, in this case, the food. Once the food coordinate is reached, it stops and returns the traced path (solution).

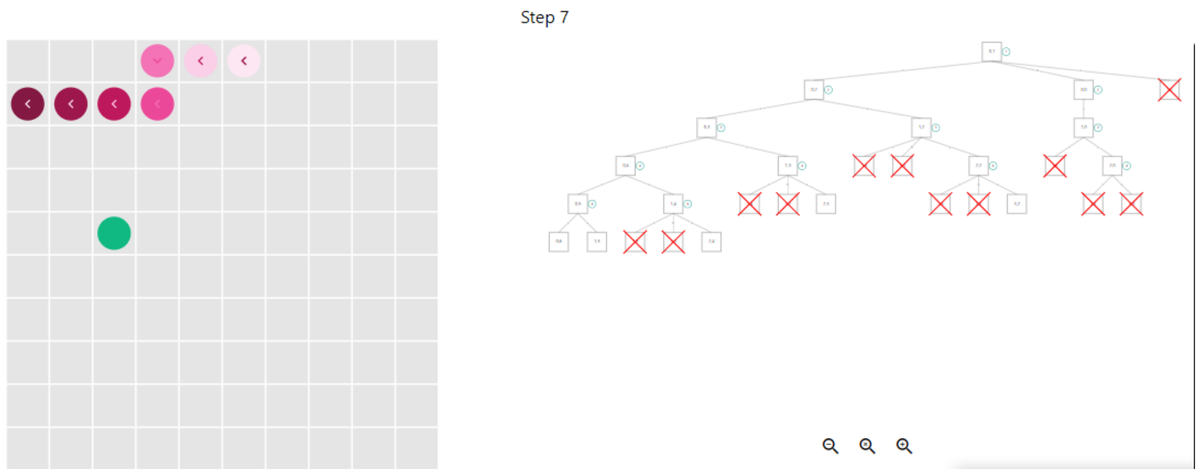


Figure 2.2.2 (b) (BFS Search Tree)

2.3 Results

The results of the two agents are as follows:

Breadth-First Search	Average	High Score
Goal 1	20 (∞)	20 (∞)
Goal 2	27	38
Goal 3	24.2	29

Table 1 (Breadth-First Search Results)

A* Search	Average	High Score
Goal 1	20 (∞)	20 (∞)
Goal 2	27.2	38
Goal 3	25.2	29

Table 2 (A* Search Results)

These results were obtained after 10 trials of each condition by each agent. For goal 1, it can be reasonably said that due to the static length of the snake, the game would go on infinitely. Hence, the scores of the attempts of this condition are limited to 20.

The agents have shown that they can consistently achieve the goal points in each condition. Thus, it can be said that the agents are successful in achieving the aim of the project.

3. Discussion

The performance of the agents in achieving goal 1 is optimal, as they are consistently able to hit scores above what is reasonable to record. There were no issues with the agents' abilities to achieve the goals repeatedly. It is also impossible to record the variance in the results of the agents' achieved scores as the upper range of the scores of the agents in this mode is theorized to extend into infinity. Furthermore, the number of steps taken to achieve the goal state is heavily influenced by the random position of the generated food, making it an inaccurate standard for measurement.

The main characteristics and issues with the algorithms can be seen in their attempts to achieve goals 2 and 3.

The ideal snake AI would be able to determine the most strategic method to reach the goal without 1- eating itself and 2- running into a wall. However, the agents as they are currently, are not optimal as it “gives up” the moment its body encloses the food, leaving it with no straightforward escape route. This is because it has run out of solutions, leading to it removing all possible outcomes and then not having the ability to make any more decisions. For example, it is not intelligent enough to manoeuvre in a zig-zag motion to make the most efficient use of the empty space within its body to create an exit for itself so it stops movement completely. To solve this problem, further modifications to the current agents’ algorithms must be carried out. For example, including a function to specifically make use and search for an escape route once it traps itself or a modification to the search algorithm which would allow it to not trap itself in the first place.

In terms of performance, the agents performed extremely well, attaining higher scores than required consistently. Higher scores would be possible if not for the reason mentioned above. In terms of the comparison between informed and uninformed search, informed search lived up to its reputation of being more effective than uninformed search by outperforming it, as seen in the results section, with higher average scores in all possible conditions. The high scores of the agents in the 2 conditions are the same. This could be due to the length of the snake’s body becoming too long at such high scores, severely limiting the movement of the agents. This could possibly be its upper limit. To further investigate this, more trials need to be run.

A* Search was chosen as the most efficient informed search algorithm for this due to its properties of being a complete search algorithm. This is because it not only provides the shortest path, like other simple heuristic approaches, but also considers that said path is the most optimal overall. It is resource-intensive, which makes it more suited for smaller state spaces such as that of the snake game. Hence, other informed search algorithms would not be advised.

Breadth-First Search, in comparison to other uninformed search algorithms, provides a definite solution given that such a solution exists unlike an algorithm like depth-first search, which depends on luck to find a solution if the depth of the tree is very large. It is also resource intensive, given that it also requires the storing of each level of its search tree in order to expand to a deeper level. However, other uninformed search algorithms could possibly perform equally well or better. For example, although difficult to implement, bidirectional search could possibly provide faster solutions with less resources. Uniform-cost search could also provide an efficient solution, due to its property of always choosing the lowest cost path. However, its drawback of possibly being stuck in an infinite loop is highly detrimental to the aim of the project.

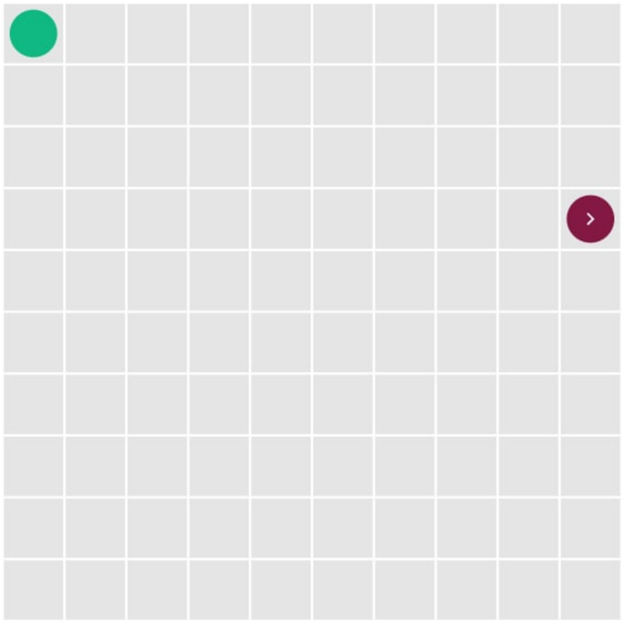
Conclusion

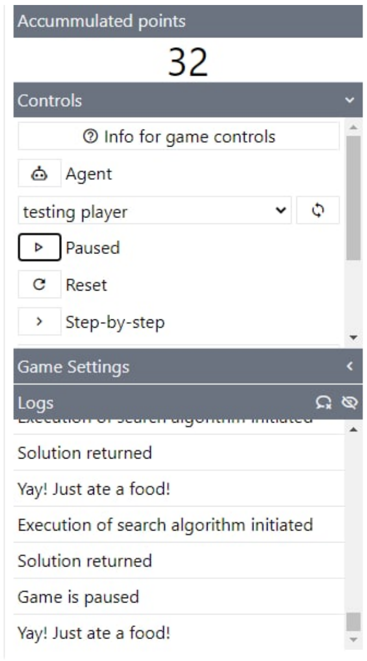
The 3 challenges have been overcome by the created agents with flying colours. The results align with the expectations of the algorithms as compared to each other, but completely exceeded them in terms of individual performance. This has shown that the two types of search algorithms were correctly deemed suitable or optimal for the task at hand and have been implemented well, which shows how artificial intelligence can be used to excel in tasks such as playing the snake game.

4. Appendices

A* Search

Goal 1 Achieved



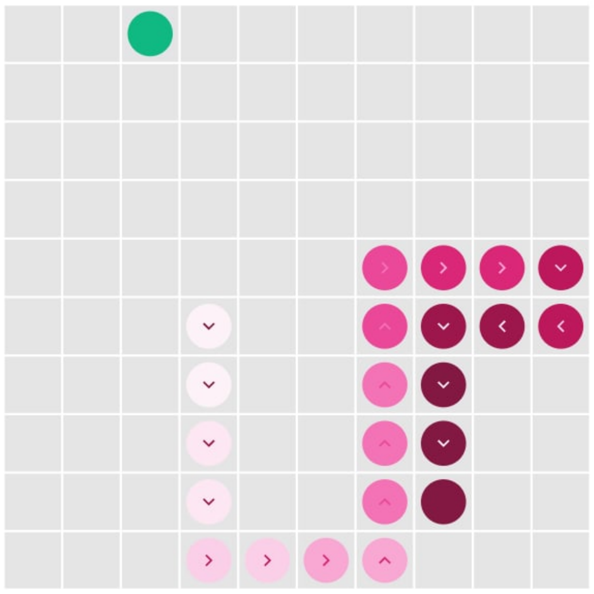


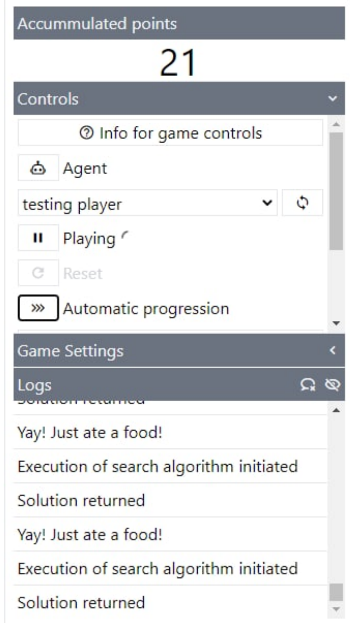
Accummulated points: 32

Agent: testing player

```
Solution returned
Yay! Just ate a food!
Execution of search algorithm initiated
Solution returned
Game is paused
Yay! Just ate a food!
```

Goal 2 Achieved





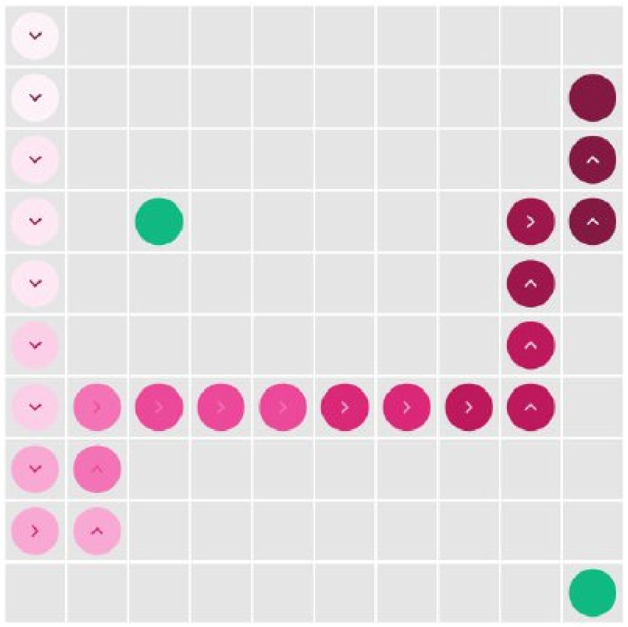
testing player

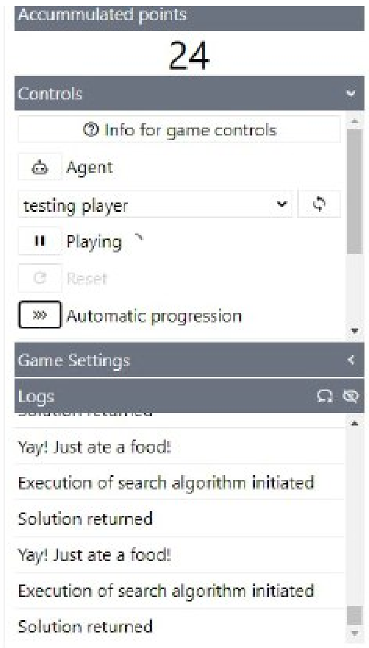
Playing

Automatic progression

```
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
```

Goal 3 Achieved





testing player

Playing

Automatic progression

```
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
```