Group name: Kung Fu Pandas

Group members:

Siow Woon Kang – 18042135

Wong Yuen-Yi – 17112723

Wong Wei Chean - 18000133

Yash Aubeeluck - 19059401

Introduction

Snake game is a popular game where the goal of the player is to control the movement of a snake to collect food on a given map, where the location of the food is randomly generated [1]. As the snake collects food, it extends in length and the player gains a score point each time it collects a food. The game ends when the path of the snake crosses its own body, or when it hits a wall.

In this assignment, we are tasked to create two artificially intelligent agents to play the snake game. The most appropriate algorithm to implement to achieve the best results, or best scores in the game, in the most efficient manner is search algorithms, as they are commonly used and are path-finding type algorithms [2]. Hence, we have implemented two AI agents using two types of search algorithms, which are informed search and uninformed search.

Problem Formulation

We have identified the problem of AI agents in Snake game. The state space of Snake game is a maze or map in which the snake agent moves. In our Snake game we have chosen the maze size to default to 10x10. The initial situation of our Snake game is the initial location of the snake agent in the maze. The snake can take four actions, that is to move in the north, south, west and east directions, as long as it does not hit a wall or cross the path of its own body. Each action that the snake agent takes will add a value of 1 to the path cost, while the player receives 1 score point with each food collected. As for the goal test, the algorithm must check that the current snake location is the location of the food.

For each iteration of the algorithm, the goal of the snake agent is to move from its current location to the food location safely. In our assignment, the minimum score required for each agent to achieve is 15.

Search Algorithms

In artificial intelligence, search algorithms are a universal problem-solving mechanism [1]. There are two types of search algorithms, namely uninformed search and informed search. For our assignment, we have implemented two algorithms in the form of agents, or “players”, to play the Snake game automatically, that is using -uninformed search- and greedy best-first search. In this section, the two types of search algorithms used are explored in detail.

Uninformed Search

Uninformed search algorithms are also known as blind search. A search algorithm can be classified as an uninformed search when no specifications or knowledge is provided in advance [2]. Some examples of uninformed search include breadth-first search (BFS), depth-first search (DFS) and uniform-cost search (UCS). For this assignment, we have chosen to use Breadth first search to solve the problem of AI Snake game.

Breadth First Search

In breadth first search, the expansion of the search tree begins at the root node. It expands the search tree by generating one level of the tree each time, until the goal node is found. Therefore, it is known as a level-by-level traversal technique. The goal test is performed during generation of the nodes. In implementation of breadth first search, it is most preferable to use a First In First Out (FIFO) data structure, where the FIFO queue first contains only the root node. Then, the node at the head of the queue is removed and expanded, and its children nodes are added to the tail of the queue. The time complexity for breadth first search is $O(b^d + 1)$ and the space complexity is $O(b^d + 1)$, where b is the branching factor and d is the depth of the solution.

Breadth first search always searches for the shortest path to a goal, because it explores all the nodes at the shallower levels of the search tree before generating more deeper level nodes [3]. This means that breadth first search gives the least number of steps to reach the solution when applied in any search tree. It also guarantees that the goal node is reached in the search tree.

Therefore, we can say that the following are the properties of breadth first search.

Completeness: Yes
Optimal: Yes
Time complexity: $O(b^d)$; where b is branching factor and d is depth of the solution.
Space complexity: $O(b^d)$; where b is branching factor and d is depth of the solution.

The pseudocode for breadth first search can be summarised as below:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Implementation of the Breadth First Search Algorithms.

In this section, we will explain the search algorithm Breadth First Search used in the snake game.

By making use of the breadth first search in the game, it is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
It explores the node though a search tree by exploring the nodes in the first level and then expands in the second level to reach the food, which it the goal. It is known as level transversal technique. By using this technique all the solutions for each node are found out. This ensures an optimal solution. We will explain the function of the breadth first used in this game.

The list pq is the priority queue list which is used to store the snake's location coordinates and the ID number. The all_location list is used store all the visited coordinates of the snake and the next_id_count is used to increment the ID number. The exp_seq, expansion sequence number is used to the construct of the search tree for the game. It is initialized as -1 as if it is not the path selected, the value will be inserted as -1.

```
pq = [[snake_loc[0], snake_loc[1], 1]] #last num is id number
```

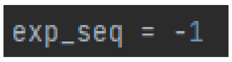
Figure 1.1: shows the pq list.

```
#to store all possible snake location without removing
all_locations = [[snake_loc[0], snake_loc[1]]]
```

Figure 1.2: stores all the snake locations in all_location list.

```
next_id_count = 1
```

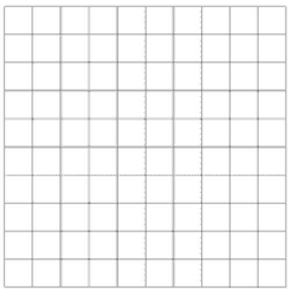
Figure 1.3: shows the initialization of the next_id_count.



```
exp_seq = -1
```

Figure 1.4: shows the initialization of the expansion sequence number.

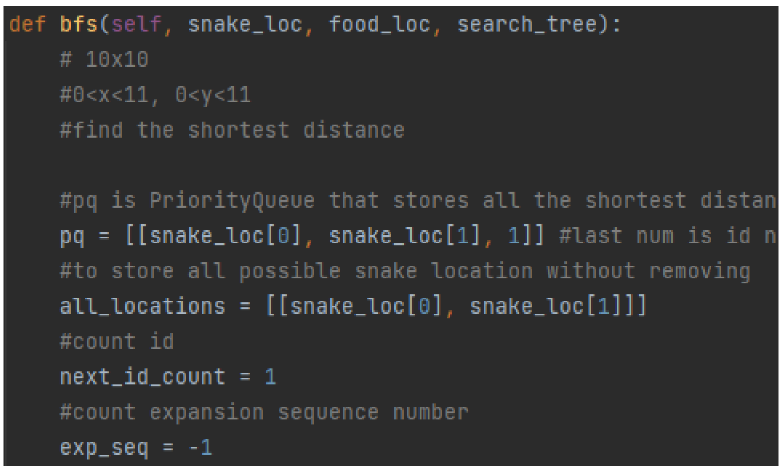
A while loop shown in Figure 1.6 is used to guide the snake to the food location by comparing the snake location and the location of the food.
The last location is then equal to the snake location.



The four IF statements inside the while loop are used to prevent the snake from collision to the four walls of the maze. The IF statements consisting of the following,

- In each IF statements the next_id_count is incremented by 1.
- We then check the condition if the direction the snake is heading is a loopy path or not, and if not pq either move to north, east, south or west according to the if statement.
- The direction variable is used to store the direction from the latest location of the snake to the new location.
- The location used by the search_tree function to fill the location in the search tree.
- The search_tree then uses the function add_dic to construct the search tree.
- The all_locations list is then used to append the visited locations.

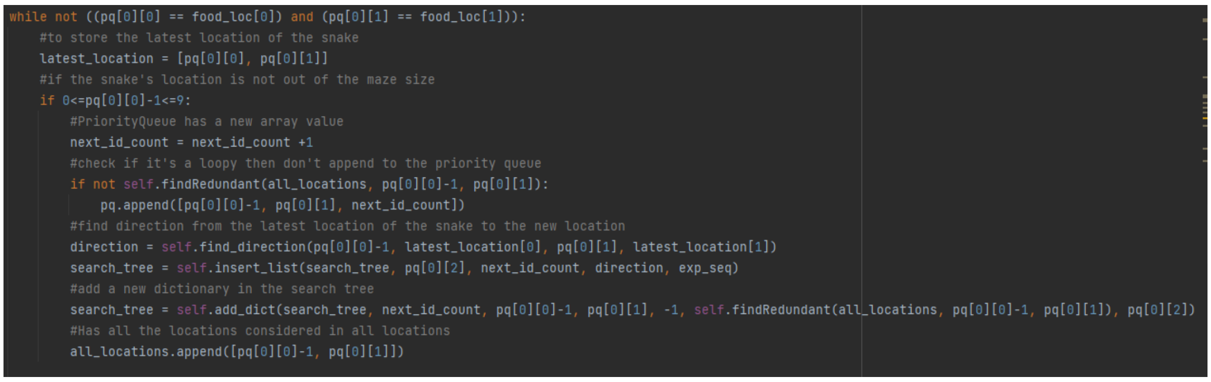
At the end of the four IF statements the first element is removed in pq.
The solution is produced by using the findsolution function.
The findsolution uses a while loop to find the parent, children, actions and expansionsequence and then returns the solution and the search tree.



```
def bfs(self, snake_loc, food_loc, search_tree):
    # 10x10
    #0<x<11, 0<y<11
    #find the shortest distance

    #pq is PriorityQueue that stores all the shortest distan
    pq = [[snake_loc[0], snake_loc[1], 1]] #last num is id n
    #to store all possible snake location without removing
    all_locations = [[snake_loc[0], snake_loc[1]]]
    #count id
    next_id_count = 1
    #count expansion sequence number
    exp_seq = -1
```

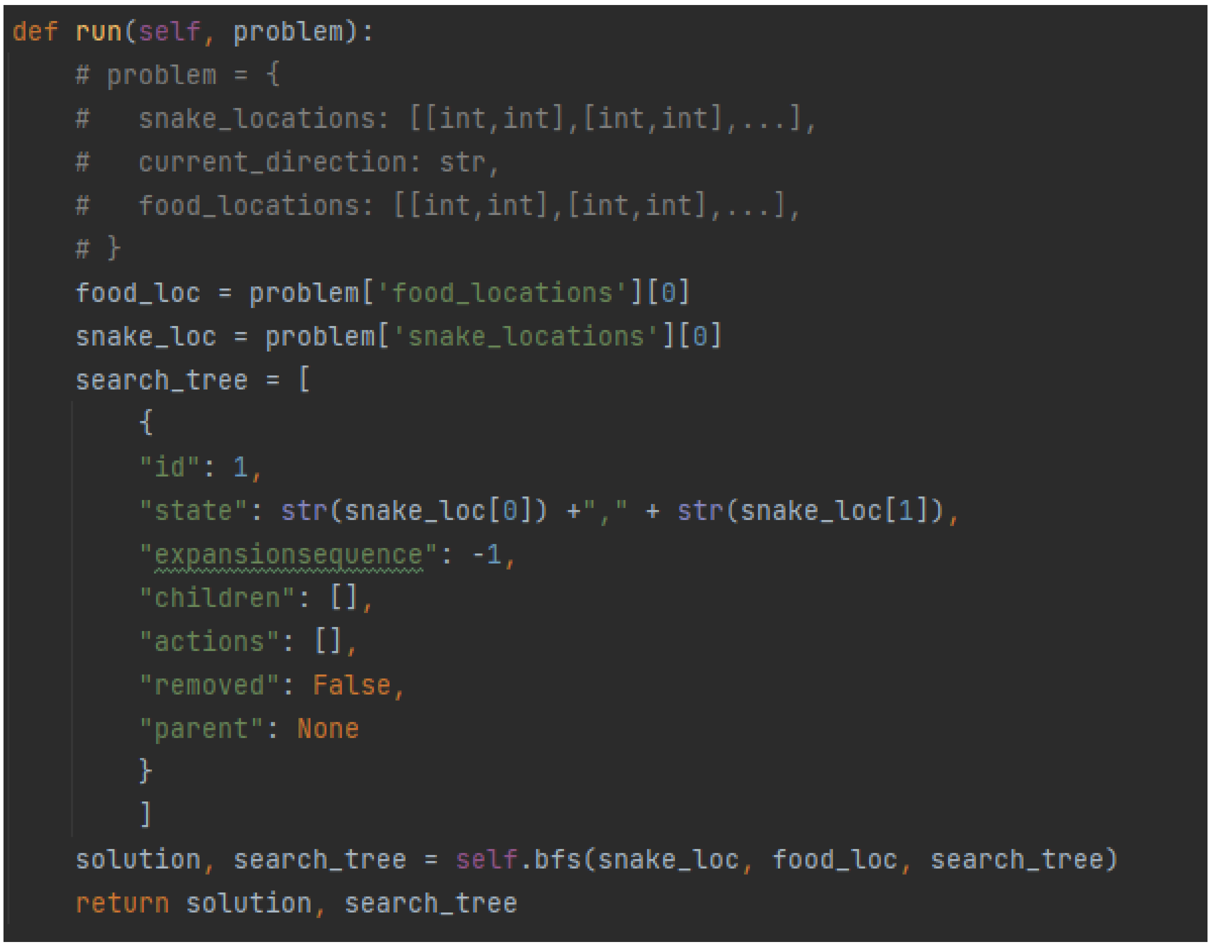
Figure 1.5: BFS fuction.



```python
while not ((pq[0][0] == food_loc[0]) and (pq[0][1] == food_loc[1])):
    #to store the latest location of the snake
    latest_location = [pq[0][0], pq[0][1]]
    #if the snake's location is not out of the maze size
    if 0<=pq[0][0]-1<=9:
        #PriorityQueue has a new array value
        next_id_count = next_id_count +1
        #check if it's a loopy then don't append to the priority queue
        if not self.findRedundant(all_locations, pq[0][0]-1, pq[0][1]):
            pq.append([pq[0][0]-1, pq[0][1], next_id_count])
        #find direction from the latest location of the snake to the new location
        direction = self.find_direction(pq[0][0]-1, latest_location[0], pq[0][1], latest_location[1])
        search_tree = self.insert_list(search_tree, pq[0][2], next_id_count, direction, exp_seq)
        #add a new dictionary in the search tree
        search_tree = self.add_dict(search_tree, next_id_count, pq[0][0]-1, pq[0][1], -1, self.findRedundant(all_locations, pq[0][0]-1, pq[0][1]), pq[0][2])
        #Has all the locations considered in all locations
        all_locations.append([pq[0][0]-1, pq[0][1]])
```

Figure 1.6: The while loop with one of the if statements.

The food_loc is set with food_location in the problem object and the snake_loc is set with the snake location in the problem object. The search tree is constructed with default values and default variable names. The solution and search_tree is then used to run the bfs function with snake_loc, food_loc and search_tree as the parameters.



```python
def run(self, problem):
    # problem = {
    #   snake_locations: [[int,int],[int,int],...],
    #   current_direction: str,
    #   food_locations: [[int,int],[int,int],...],
    # }
    food_loc = problem['food_locations'][0]
    snake_loc = problem['snake_locations'][0]
    search_tree = [
        {
        "id": 1,
        "state": str(snake_loc[0]) +"," + str(snake_loc[1]),
        "expansionsequence": -1,
        "children": [],
        "actions": [],
        "removed": False,
        "parent": None
        }
        ]
    solution, search_tree = self.bfs(snake_loc, food_loc, search_tree)
    return solution, search_tree
```

Figure 1.7: The solution of the game.

There are other functions used in this algorithm but there are also used by the Greedy Best-first Search algorithm. The other algorithms used are explain below at **Other functions used**.

Informed Search

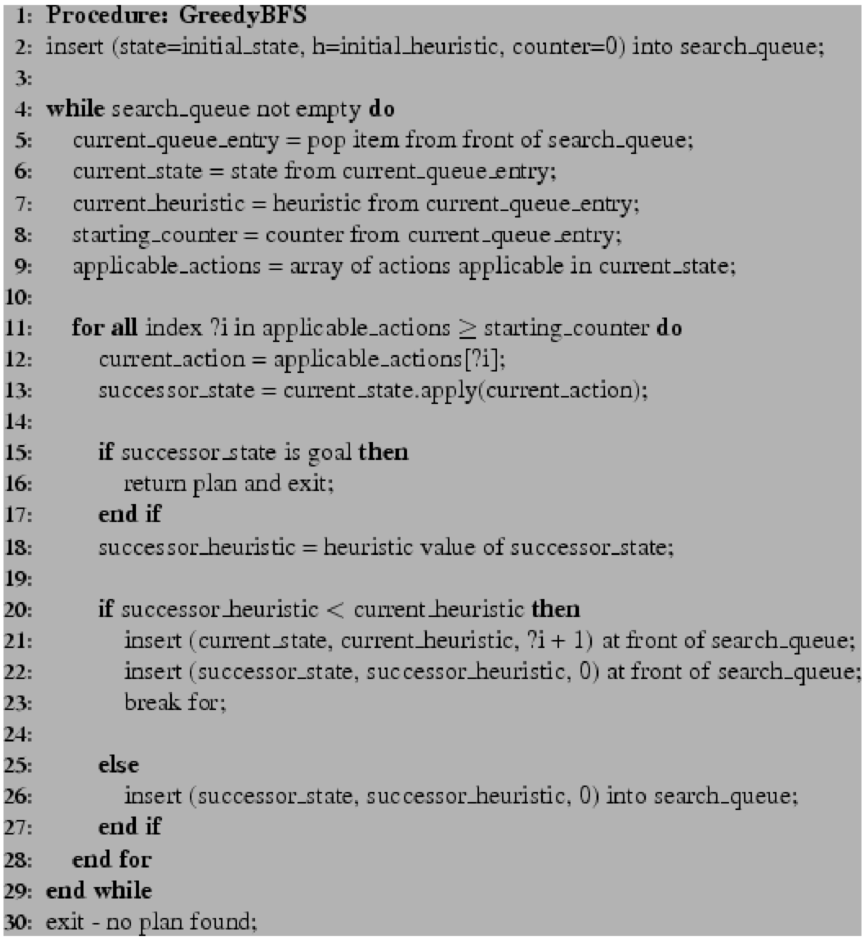
Informed search algorithms are also referred to as heuristic searches. These algorithms are categorised by the fact that some heuristic function, or prior knowledge or specification, is provided to the algorithm beforehand [2]. Due to this heuristic function, the performance of these algorithms usually is higher than uninformed search algorithms. There are many popular informed search algorithms, such as A* search, best-first search, greedy best-first search and Dijkstra's algorithm. For our assignment, we have chosen to use greedy best-first search to solve the problem of AI Snake game.

Greedy Best-first Search

In this informed search strategy, the goal is targeted by trying to expand the closest node to the goal node [2]. The heuristic function used for node evaluation is as follows:

$$f(n) = h(n)$$

This heuristic function tells us the estimated cost from that node to the goal node.
Greedy Best-first search is a special case of the best-first search and is a combination of the breadth-first search and depth-first search algorithms. While best-first search uses the evaluation function of $f(n)$, greedy best-first search includes the $h(n)$ heuristic function and values to obtain the most optimal solution, although it is not guaranteed that it is able to arrive at a solution due to the fact that it is a recursive algorithm. Hence, care must be taken when implementing this algorithm to prevent an infinite loop.
Figure 1.8 shows the pseudocode for implementing the greedy best-first search strategy.



```
Procedure: GreedyBFS
insert (state=initial_state, h=initial_heuristic, counter=0) into search_queue;

while search_queue not empty do
    current_queue_entry = pop item from front of search_queue;
    current_state = state from current_queue_entry;
    current_heuristic = heuristic from current_queue_entry;
    starting_counter = counter from current_queue_entry;
    applicable_actions = array of actions applicable in current_state;

    for all index ?i in applicable_actions $\geq$ starting_counter do
        current_action = applicable_actions[?i];
        successor_state = current_state.apply(current_action);

        if successor_state is goal then
            return plan and exit;
        end if
        successor_heuristic = heuristic value of successor_state;

        if successor_heuristic $<$ current_heuristic then
            insert (current_state, current_heuristic, ?i + 1) at front of search_queue;
            insert (successor_state, successor_heuristic, 0) at front of search_queue;
            break for;

        else
            insert (successor_state, successor_heuristic, 0) into search_queue;
        end if
    end for
end while
exit - no plan found;
```

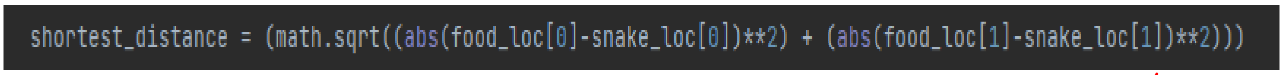
Figure 1.8: Pseudocode for greedy best-first search algorithm.

The following are the properties of greedy best-first search.
Completeness: No
Optimal: No
Time complexity: O(b^m); where b is the branching factor, and m is the maximum depth of the tree.
Space complexity: O(b^m)

Implementation of the Greedy Best First Search Algorithms.

In this section, we will explain the search algorithm Greedy Best First Search used in the snake game.

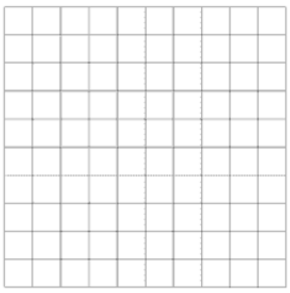
Greedy Best First Search algorithm expands the node which is the closest to the goal, it makes use of an estimation function F(n) = H(n).
H(n) calculates the shortest distance between food location coordinates and the snake location coordinates in the game as shown in figure 2.0.

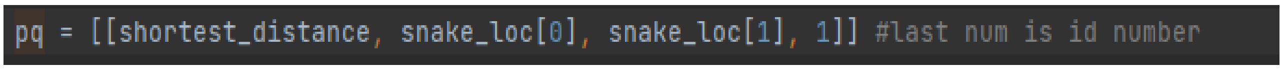


```
shortest_distance = (math.sqrt((abs(food_loc[0]-snake_loc[0])**2) + (abs(food_loc[1]-snake_loc[1])**2)))
```

Figure 2.0: shows the formula to calculate the shortest distance.

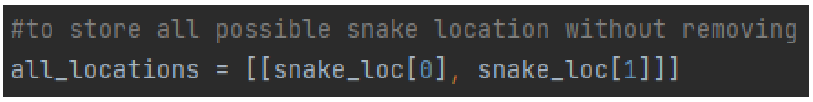
The list pq is the priority queue list which is used to store all the shortest distance, the snake's location coordinates and the ID number. The next_id_count is used to increment the ID number. The all_location list is used store all the visited coordinates of the snake and the exp_seq, expansion sequence number is used to the construct of the search tree for the game.





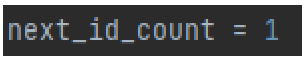
```
pq = [[shortest_distance, snake_loc[0], snake_loc[1], 1]] #last num is id number
```

Figure 2.1: shows the pq list.



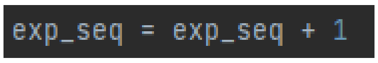
```
#to store all possible snake location without removing
all_locations = [[snake_loc[0], snake_loc[1]]]
```

Figure 2.2: stores all the snake locations in all_location list.



```
next_id_count = 1
```

Figure 2.3: shows the initialization of the next_id_count.



```
exp_seq = exp_seq + 1
```

Figure 2.4: shows the initialization of the expansion sequence number.

A while loop shown in figure 2.5 is used to guide the snake to the food location using the priority queue list and as the snake will move towards the food the first item in the priority queue list will be removed, in other words, we are decrementing the priority queue list at the end of the while loop. When the shortest_distance in the priority queue list is 0, this means that the snake has reached the food location.

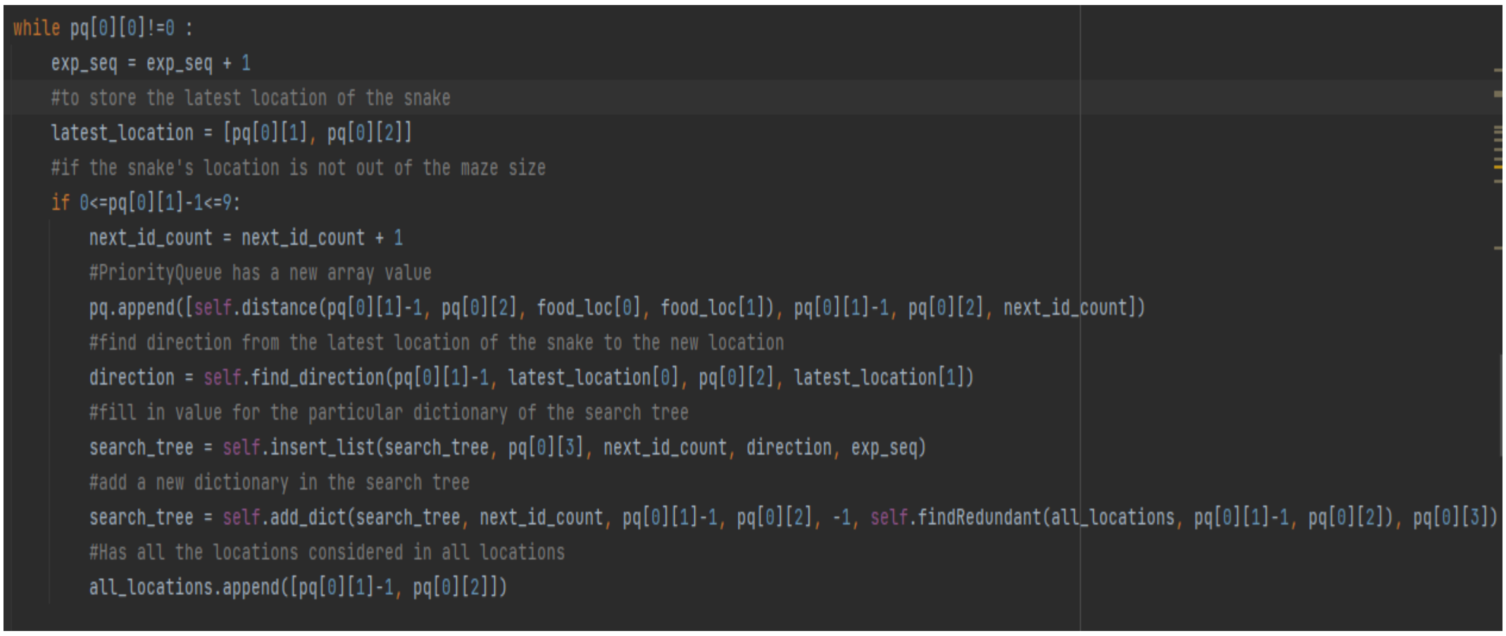
Each time the while loop is executed the exp_seq is incremented by 1, it is indexing each child node that has the shortest distance in the search tree. As the snake moves the latest_location list is used to store the last location coordinates of the snake.

The four IF statements are used to prevent the snake from collision to the four walls of the maze. The IF statements consisting of the following,

- In each IF statements the next_id_count is incremented by 1.
- The pq is appended with a new calculated distance, next snake value and the incremented of next_id_count.
- The find_direction finds the latest location of the snake to the new location.
- The search_tree fill in the value for the dictionary of the search tree for the node that was explore.
- The next search_tree adds a new dictionary in the search tree.
- all_location is appended with all the locations of the snake movements.

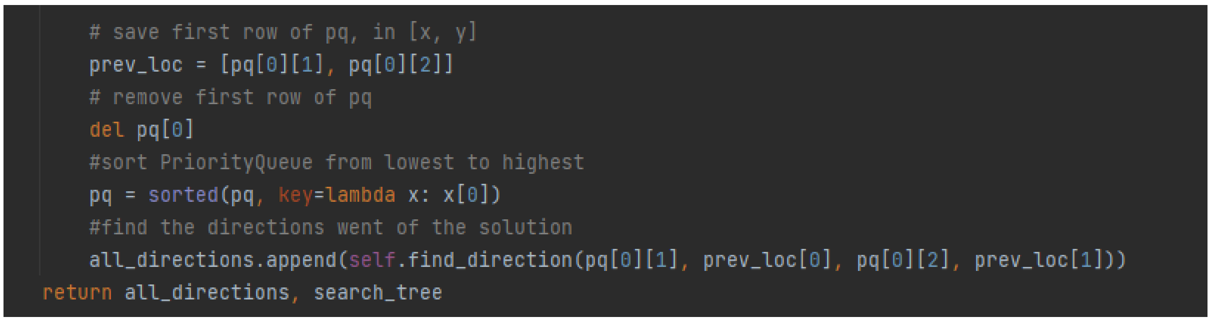
At the end of the four IF statements the prev_loc list is used to save the first row of pq in [x,y]. The first row of pq which is the calculated shortest distance is removed

and the pq is sorted from lowest to highest. The all_directions are appended with the latest location of the snake.
As the first value of pq, which is the shortest_distance is 0 the while loop ends and return all_directions and the search tree.



```
while pq[0][0]!=0 :
    exp_seq = exp_seq + 1
    #to store the latest location of the snake
    latest_location = [pq[0][1], pq[0][2]]
    #if the snake's location is not out of the maze size
    if 0<=pq[0][1]-1<=9:
        next_id_count = next_id_count + 1
        #PriorityQueue has a new array value
        pq.append([self.distance(pq[0][1]-1, pq[0][2], food_loc[0], food_loc[1]), pq[0][1]-1, pq[0][2], next_id_count])
        #find direction from the latest location of the snake to the new location
        direction = self.find_direction(pq[0][1]-1, latest_location[0], pq[0][2], latest_location[1])
        #fill in value for the particular dictionary of the search tree
        search_tree = self.insert_list(search_tree, pq[0][3], next_id_count, direction, exp_seq)
        #add a new dictionary in the search tree
        search_tree = self.add_dict(search_tree, next_id_count, pq[0][1]-1, pq[0][2], -1, self.findRedundant(all_locations, pq[0][1]-1, pq[0][2]), pq[0][3])
        #Has all the locations considered in all locations
        all_locations.append([pq[0][1]-1, pq[0][2]])
```

Figure 2.5: The while loop is executed while the first row is not 0. The If statements to prevent the snake from collision.



```
        # save first row of pq, in [x, y]
        prev_loc = [pq[0][1], pq[0][2]]
        # remove first row of pq
        del pq[0]
        #sort PriorityQueue from lowest to highest
        pq = sorted(pq, key=lambda x: x[0])
        #find the directions went of the solution
        all_directions.append(self.find_direction(pq[0][1], prev_loc[0], pq[0][2], prev_loc[1]))
    return all_directions, search_tree
```

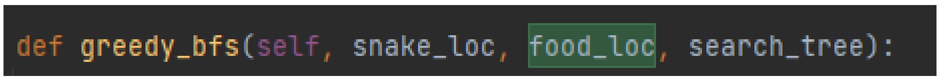
Figure 2.6: Returns all directions and the search tree when the while loop have been executed.

All the above coding were place inside a function call greedy_bfs.
The function requires three parameters to initialize, they are,

- Snake_loc
  It is the current location of the snake.

- Food_loc
  It is the current location of the food.

- Search_tree
  It is the for the construction of the search tree with the functions.

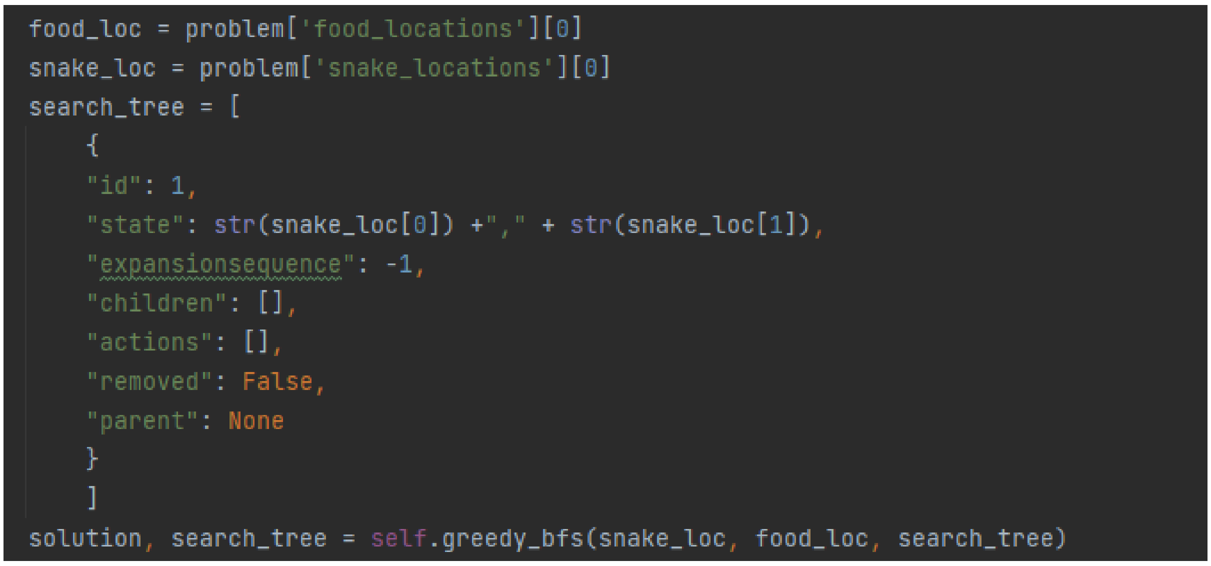


```
def greedy_bfs(self, snake_loc, food_loc, search_tree):
```

Figure 2.7: The name of the function with the parameters.

The food_loc is set with food_location in the problem object and the snake_loc is set with the snake location in the problem object. The search tree is constructed with the values executed inside the greedy_bfs function. We set the solution and the search_tree by initializing the parameters in the greedy_bfs function.

The below code figure 2.8 shows the initialization of the parameters.



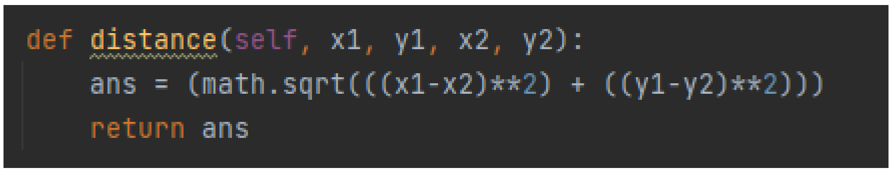
```
food_loc = problem['food_locations'][0]
snake_loc = problem['snake_locations'][0]
search_tree = [
    {
    "id": 1,
    "state": str(snake_loc[0]) +"," + str(snake_loc[1]),
    "expansionsequence": -1,
    "children": [],
    "actions": [],
    "removed": False,
    "parent": None
    }
    ]
solution, search_tree = self.greedy_bfs(snake_loc, food_loc, search_tree)
```

Figure 2.8

Other functions used

The following are the additional functions used:

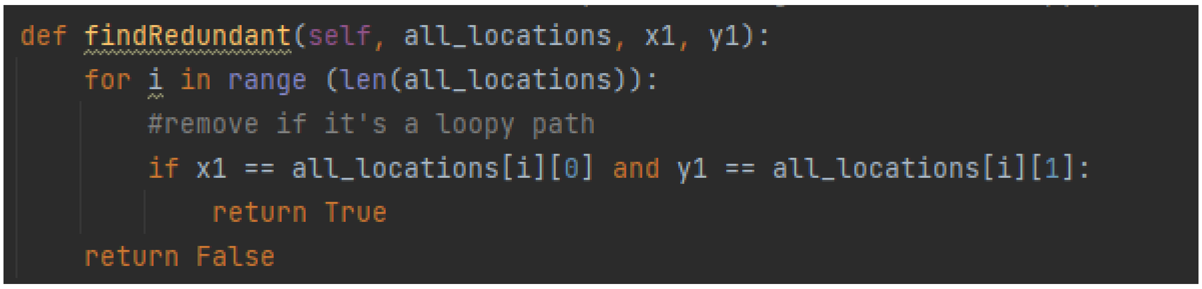
As shown in figure 2.9, the function is used to calculate the distance. We use the distance function in which we need to set four parameters to calculate the ans.



```
def distance(self, x1, y1, x2, y2):
    ans = (math.sqrt(((x1-x2)**2) + ((y1-y2)**2)))
    return ans
```

Figure 2.9

The function findRedundant is used to find the loopy path, which is the location that snake has been through, as shown in figure 2.10.

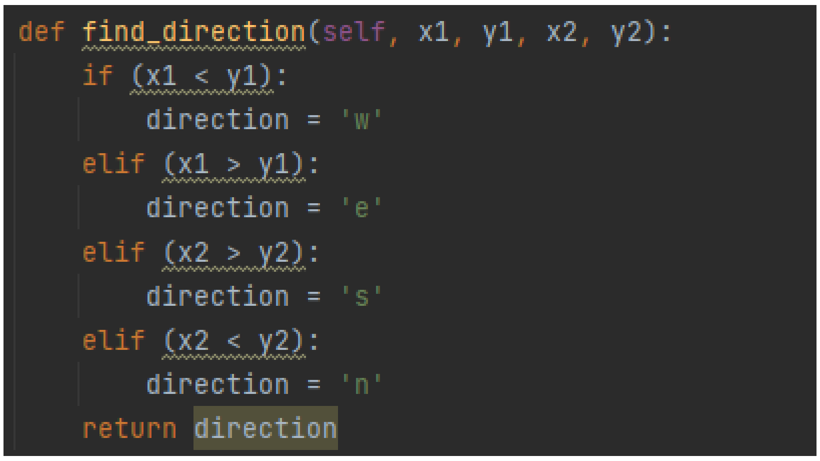


```python
def findRedundant(self, all_locations, x1, y1):
    for i in range (len(all_locations)):
        #remove if it's a loopy path
        if x1 == all_locations[i][0] and y1 == all_locations[i][1]:
            return True
    return False
```

Figure 2.10

In order to guide the snake in a direction we used the find_direction function as shown in figure 2.11. In this function it uses the compare method. For example, if the input of x1 and y1 represents the column value and x1 < y1 is true which means that it is heading to left.

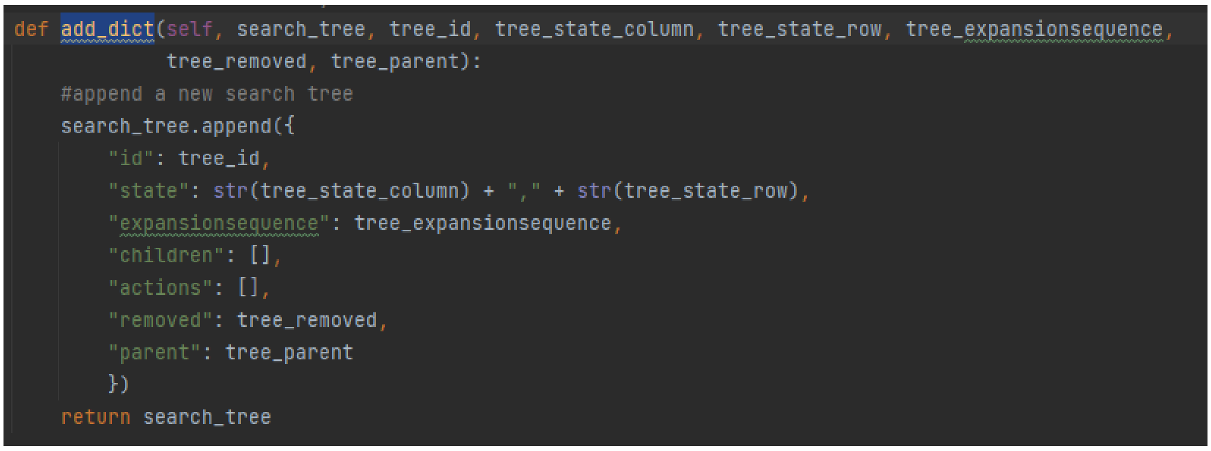
If the snake needs to move up it will return the direction ‘n’.
If the snake needs to move down it will return the direction ‘s’.
If the snake needs to move right it will return the direction ‘e’.
If the snake needs to move left it will return the direction ‘w’.



```python
def find_direction(self, x1, y1, x2, y2):
    if (x1 < y1):
        direction = 'w'
    elif (x1 > y1):
        direction = 'e'
    elif (x2 > y2):
        direction = 's'
    elif (x2 < y2):
        direction = 'n'
    return direction
```

Figure 2.11

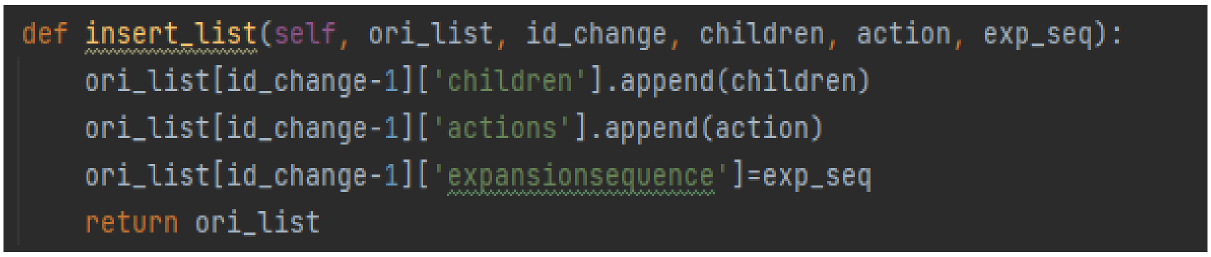
We used the add_dict function to add the coordinates of the nodes that were explored.
Each time we visit a node it will append a new dictionary to the search tree.



```python
def add_dict(self, search_tree, tree_id, tree_state_column, tree_state_row, tree_expansionsequence,
             tree_removed, tree_parent):
    #append a new search tree
    search_tree.append({
        "id": tree_id,
        "state": str(tree_state_column) + "," + str(tree_state_row),
        "expansionsequence": tree_expansionsequence,
        "children": [],
        "actions": [],
        "removed": tree_removed,
        "parent": tree_parent
        })
    return search_tree
```

Figure 2.12

The insert_list is a function that fills in the existing search tree values.



```python
def insert_list(self, ori_list, id_change, children, action, exp_seq):
    ori_list[id_change-1]['children'].append(children)
    ori_list[id_change-1]['actions'].append(action)
    ori_list[id_change-1]['expansionsequence']=exp_seq
    return ori_list
```

Figure 2.13

Results of Breadth first search and Greedy best first search.

We make 4 experiments of both algorithms with settings of snake length enable, which means that the snake will increase in length when it eats the food and another experiment where it will not increase in length when eating the food.
The locations of the food are generated at random.
The scores of each algorithm are found in the below tables.

Experiment	**Breadth first search**	**Greedy best first search**	**Length**
Experiment 1	**12**	**6**	**Dynamic**
Experiment 2	**8**	**7**	**Dynamic**
Experiment 3	**5**	**5**	**Dynamic**
Experiment 4	**4**	**5**	**Dynamic**

Table 1

Experiment	**Breadth first search**	**Greedy best first search**	**Length**
Experiment 1	**>100**	**>100**	**Fixed**
Experiment 2	**>100**	**>100**	**Fixed**
Experiment 3	**>100**	**>100**	**Fixed**
Experiment 4	**>100**	**>100**	**Fixed**

Table 2

From table 1, we can we that when the snake length is increase as it eats the food, breadth first search has a better performance in terms of search than greedy best first search. With the condition increase snake length enable we can conclude that breadth first search is more optimal than greedy best first search in the snake game.

From table 2, as the length of the snake remains static it can be seen that the performance of both algorithms is better than when the length is fixed.
Thus, it can be concluded that both algorithms are optimal when the snake length remains the same when eating the food.

Conclusion

The selected algorithms selected were compared with each other of performance when the snake size is static and dynamic when consuming the food. In the result section we conclusion that when the snake size is statics both algorithms are optimal, but the snake size is dynamic the Breadth search algorithm is more optimal. In different AL games different algorithms can be optimal depend on different factors based on the game environment. If the algorithms were compared to human agent, the results would vary depend on the performance of the individual playing the game. We demonstrated how to implement each algorithm, explained the codes. For this assignment we were asked to use one search algorithm using informed search and one search algorithm using uninformed search. For both informed and uninformed search, we have other searching methods. For uninformed search, there is Depth-First Search, Uniformed Cost Search, Iterative-Deepening Search and Bidirectional Search and for informed search there is, A* Search, Iterative deepening-A* Search. if comparing the different algorithms, it can lead to another algorithm being more optimal.

References

[1] R. E. Korf, “Artificial Intelligence Search Algorithms,” 2010. [Online]. Available: https://www.cs.helsinki.fi/u/bmmalone/heuristic-search-fall-2013/Korf1996.pdf.

[2] K. A. O. Algasi, “Uninformed and Informed Search Techniques in Artificial Intelligence,” Eastern Mediterranean University, North Cyprus, 2017.