Module 6 - Snake AI Game

Authors
Tommy Lin S1840932
Kevin Singpurwala S2205858

December 6, 2022

Contents

1 A* Search

1.1 Abstract search space modelling

We first started working on a basic A* search which did not consider any obstacles or the snakes body itself. To accomplish this, first the start and goal would have to be determined. The start is given through head_position, and to get the goal, another function had to be made. This would loop over board, and add any spaces with GameObject.FOOD to an array which would keep track of the food.This loop would then later keep track of other important objects.

```
for i in range(0, len(board), 1):
    for j in range(0, len(board[0]), 1):
        if (board[i][j] == GameObject.EMPTY):
            empty.append((i,j))
        elif (board[i][j] == GameObject.WALL ):
                obstacles.append((i, j));
        elif board[i][j] == GameObject.FOOD:
                food.append((i, j));
```

1.2 Transition function

With the start and goal now clear, a simple A* search could now be implemented. We made use of the astar library from networkx, as the challenge mentioned that you were allowed to use existing software on the internet. to save time. This library could determine the best path based on a graph of nodes, a start, a goal, and a weight for the edges of the nodes. Initially we did not make use of the weight function, as we first wanted to focus on making an agent which could play the game. The graph in this case was a simple 25x25 node-graph, with all weights for the edges being 1.
This function returns a list of tuples. This represents the shortest path from the current_position to the goal_position. The second tuple in this list is the next_move that we need to make from our current position. When subtracting the coordinates of the current_position from the next move, the change in coordinates can be determined. Using this information we can compute the desired direction we want to be in. Subtracting the desired direction with the current direction, which is given as parameter in get_move as 'direction', a value can be found, as the directions are based on an enum. In our system, if this value was 0, the move would be straight, if the value%4 is 3 it returns left, and every other case it goes right.

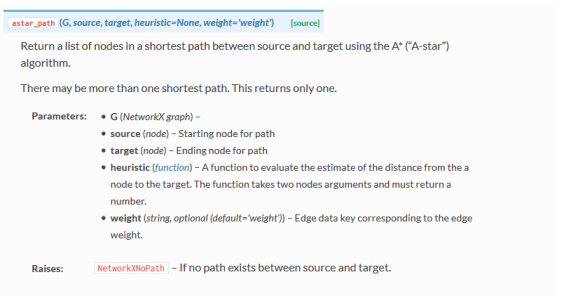
It worked but the snake quickly died due to running into an obstacle. Then obstacle detection was implemented with the prior mentioned loop, but in this case it would add the tuples with obstacles. Then from the 25x25 node-graph, any nodes and edges containing the obstacle-tuple would be removed, and replaced with the same tuples except with a weight = 1000. This would prevent the A* algorithm from using coordinates with obstacles on them. This proved to work, however, the scores were still limited as the snake still died quickly due to running in to its own body.

After that we detected the snakes body. This was done similar to the obstacles, by checking the coordinates and replacing the nodes and edges with a body_part for the same coordinate except weight = 1000. The program then ran into trouble after such a point that there was no available route from starting point to the goal. Thus the path array returned an error. To solve this we used a waiting or "stalling" function until a path previously blocked by the snakes body became available or no free spaces where left. This was done by checking for empty neighbours to the snake head. If there were, and there was no path to the goal, the snake would fill empty spaces. This would either cause a path to the goal to open up, or commit suicide if the snake had no more empty spaces.

1.3 Multiple food

To deal with multiple food spawns, we made use of a for loop when generating the shortest path. This would calculate the shortest path from the current position to one of the food spawns, and iterate for all of

the food. The shortest path to any food would be kept and used in further calculations to determine the correct move. This might not be the most elegant solution, as it would not keep in mind that the snake also has to move once it has the food, i.e. it would prefer 1 food spawn that was close as opposed to 3 food spawns next to each other but only a few more tiles away.



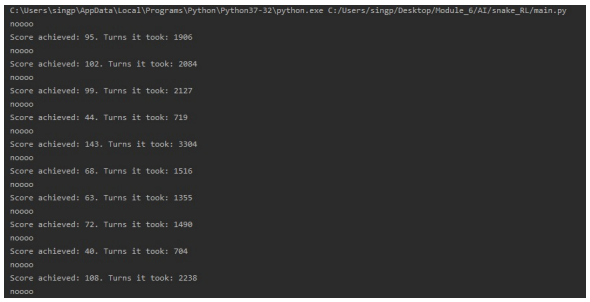
astar_path(G, source, target, heuristic=None, weight='weight') [source]

Return a list of nodes in a shortest path between source and target using the A* ("A-star") algorithm.

There may be more than one shortest path. This returns only one.

Parameters:
- G (NetworkX graph) –
- source (node) – Starting node for path
- target (node) – Ending node for path
- heuristic (function) – A function to evaluate the estimate of the distance from the a node to the target. The function takes two nodes arguments and must return a number.
- weight (string, optional (default='weight')) – Edge data key corresponding to the edge weight.

Raises: NetworkXNoPath – If no path exists between source and target.



```
C:\Users\singp\AppData\Local\Programs\Python\Python37-32\python.exe C:/Users/singp/Desktop/Module_6/AI/snake_RL/main.py
noooo
Score achieved: 95. Turns it took: 1906
noooo
Score achieved: 102. Turns it took: 2084
noooo
Score achieved: 99. Turns it took: 2127
noooo
Score achieved: 44. Turns it took: 719
noooo
Score achieved: 143. Turns it took: 3304
noooo
Score achieved: 68. Turns it took: 1516
noooo
Score achieved: 63. Turns it took: 1355
noooo
Score achieved: 72. Turns it took: 1490
noooo
Score achieved: 40. Turns it took: 704
noooo
Score achieved: 108. Turns it took: 2238
noooo
```

1.4 Snake body and cost ("weight")

The snakes body is initialized via self.body_parts = body_parts, which is given as parameter in get_move. In a similar way that we detected obstacles we will detect body parts. We avail of networkx.alogorithms.tree.branchings. In this library by networkx we can get the edges of each node. If the game object is a wall or snake body we replace the edges with a weight of 1000. In this way any path from start to goal that contains body or wall will not be the optimal as the cost will be very high. The cost of a normal empty block is by default 1. Now we guarantee that the optimal path will be without obstacles and thus the snake will not die.
As this function is part of get_move, this will be updated every time the snake moves. Since the body only follows the snake head, this should not cause the path to change once a clear start and goal have been found.

1.5 A* failure

When A* fails this means that there is currently no possible way to the goal. We use a try except block. In the try we find the path with networkx as in the picture above and in the except we get snake head position and get its four corresponding neighbours. If a neighbour is of node type empty we go to it. This was done by comparing the empty array and an additional neighbours nested list containing each neighbour. If there was a match we go to that neighbour[i] as the next move. From here we calculate as before, current position = snake head. This will be repeated until the A* algorithm can find a path from start to goal.

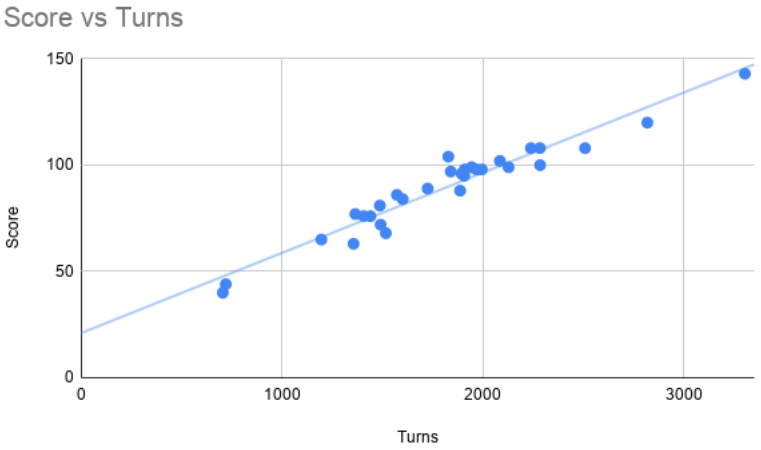
1.6 Heuristic

We used a heuristic function which availed of Pythagoras's theorem to estimate the cost from snake head to each food.The snake goes to the goal which is estimated to be the cheapest. $c^2 = a^2 + b^2$ where a is start , b is goal and c is what the heuristic returns. The heuristic from networkx expects a function so;
return = $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ This is a sound heuristic as the estimate of the diagonal line from start state to goal state is always an underestimate of the true cost due to the fact that the shortest distance between two points is a straight line and thus this estimate can not be an overestimate.

However in subsequent versions with multiple food nodes this heuristic was not needed as this heuristic calculates cost from snake head to food[0] and if there is more than one food node this may not be the quickest path to food. Thus we checked for each food path the closest food (goal) and went to that. Then the heuristic was made null and was thus removed. We relied on weighted nodes which had a cost associated and the default networkx library heuristic. This is enough to find the optimal path however it increases the space time complexity as now we must calculate the optimal path to each food and then get the optimal path from those. The complexity is increased by a factor of the number of food nodes.

1.7 Game playing performance

```
Score achieved: 99. Turns it took: 1943
Score achieved: 76. Turns it took: 1406
Score achieved: 100. Turns it took: 2284
Score achieved: 86. Turns it took: 1571
Score achieved: 98. Turns it took: 1994
Score achieved: 98. Turns it took: 1967
Score achieved: 104. Turns it took: 1827
Score achieved: 98. Turns it took: 1909
Score achieved: 108. Turns it took: 2508
Score achieved: 89. Turns it took: 1724
Score achieved: 76. Turns it took: 1439
Score achieved: 108. Turns it took: 2283
Score achieved: 65. Turns it took: 1195
Score achieved: 120. Turns it took: 2818
Score achieved: 81. Turns it took: 1486
Score achieved: 97. Turns it took: 1839
Score achieved: 84. Turns it took: 1600
Score achieved: 77. Turns it took: 1364
Score achieved: 96. Turns it took: 1893
Score achieved: 88. Turns it took: 1886
```



All the trials were done with the standard testing scenario, i.e. 2 obstacles and 1 food spawn, on a 25x25 board.

From 30 trials we found that our average score was 89.4 and average turns 1812. This is sufficient, as the exercise mentioned an average of 80 being the minimum. Interesting numbers are that our lowest score was 44 while the highest was 143. This shows that there is a large variance in our scores, however, in general the points fall in a range of roughly 70-100. Furthermore, the relation between score and turns taken resembles a linear relation.

2 Complexity of A Search for the Snake game

2.1 Part A

A* Search is like Dijkstra's algorithm with an additional heuristic.This is an estimate from starting node (head) to goal node (food). The heuristic must never overestimate the cost if one wants the most optimal path (without dying). Thus we calculate this heuristic via distance to a neighbouring node + the remaining distance to go from that neighbour

The complexity of A* search is $O(|V|) = O(b^d)$. If we set heuristic as none. The complexity is the same as dijkstra. $O(V^2)$ In the worst case this results in big O notation . (The worst case) being n^2

2.2 Part B

DID NOT IMPLEMENT: Our actual time complexity would be a lot greater as in our implementation we have found the most optimal path to each food node and then from that decided to take the shortest of those food nodes. This was done via taking the smallest array from head to food as path. ie. sudo code; (min(len(food)). Thus our complexity of A* search is $O(|V|) = O(b^d) * len(food)$. In the worst case big O notation being (nlog$_{10}$n *$len(food)$)

where len(food) is the number of food nodes present on the map. Given len(food) > 0. If len(food) $= 0$ then complexity is undefined as no goal. Thus in reality our A*search should be

2.3 Part C

DID NOT IMPLEMENT: As obstacle complexity increases Time-space complexity decreases as the search space is reduced. Thus they have an inverse relationship. For example one could increase the obstacles from 5 to 30 in steps of 5.

3 Reinforcement learning

DID NOT IMPLEMENT: I did not do this section but I found some code online which implements Q learning in snake which I link below. Did not do but have answered:

3.1 Part A

DID NOT IMPLEMENT. Learning parameters:
α - the learning rate. Between 0 and 1.
γ - discount factor .Set as 0.95. Between 0 and 1.
Max α - The reward for taking the optimal action. Discount factor: $= 0.95$. The discount factor gives us the importance of a reward form that current state.
Reward: sudo code
if get food = positive , if die = negative , if long time without food = negative * .5
Q learning: I would if I were to choose between Q and U learning choose Q leaning. This is as it is model free. It seems easier to implement however Q learning requires some time for the program to learn. As the complexity increases , in general it would take longer for an agent to get good at the task.

3.2 Part B

DID NOT IMPLEMENT. The issue with state explosion is as the map grows in size the possibilities grow exponentially. To deal with state explosion we can make a max "view" around SNAKEHEAD. Thus the snake can only "see" regions surrounding its head in that radius. This prevents the problem of state explosion as the snake only considers locally visible states less than a distance x away from its head. The value of x you want to set depends on the processing time and power you have.

3.3 Part C

DID NOT IMPLEMENT.

4 References

- Library we used
  `https://networkx.github.io/`
- Q learning which we did not do. Only ran the code
  `https://github.com/danielegrattarola/deep-q-snake`