AI agent to solve the Snake game

Artificial Intelligence Final Project Report

Abdalrhm Kharubi - Abdalrhm.Kharubi@mail.huji.ac.il
Ahmad Okosh - Ahmad.Okosh@mail.huji.ac.il
Kareem Abbasi - Kareem.Abbasi@mail.huji.ac.il
Waseem Abu Leil - Waseem.AbuLeil@mail.huji.ac.il

***Abstract* - In this project we explored different approaches to solve the Snake game. First we used a path finding algorithm (A*) with a number of different heuristics. Then we used a reinforcement learning algorithm (Q-learning). The goal is to compare the performance of a path finding agent with an agent that uses reinforcement learning and to see whether it is possible to achieve better results using reinforcement learning.**

I. Introduction

The Snake game can be dated back to arcade game Blockade which was developed by Gremlin in 1976. The game was popularized with Nokia mobile phones after a variant was preloaded on the phones in 1988.

Snake is a computer game, where the player has to control a snake and move it to collect food (we will refer to it as fruits) which appears in random locations on the game board without hitting it's body nor the walls and obstacles. Each time the snake eats, it's body gets longer by one unit (block). The goal of the game is to make the snake eat as many fruits as possible without hitting any obstacles or making the snake bite itself. When this happens, the game ends and a final score is returned.

II. Gameplay

In this section, we will clarify some basic characteristics about our game implementation.

The game starts with a board of size $n * n$ blocks and a fruit in a random position on the board. To make the game harder, the game generates obstacles at random 4×4 sections on the board with probability p. For example, if $p = 1$, the board will always have obstacles in each 4×4 section. In a similar way, if $p = 0$, the board will not have any obstacles in any section, Therefore it will be a standard board. The reason it is done this way is to guarantee that the snake can never be surrounded and can always reach the fruit.

Similar to other Snake games, the snake will start at a random position on the board and can move in four directions; up, down, left and right. The body of the snake will increase in length by one block when the snake eats a fruit. If the snake hits its body or an obstacle it will die and the game will end. The final length of the snake will be the score of the game.

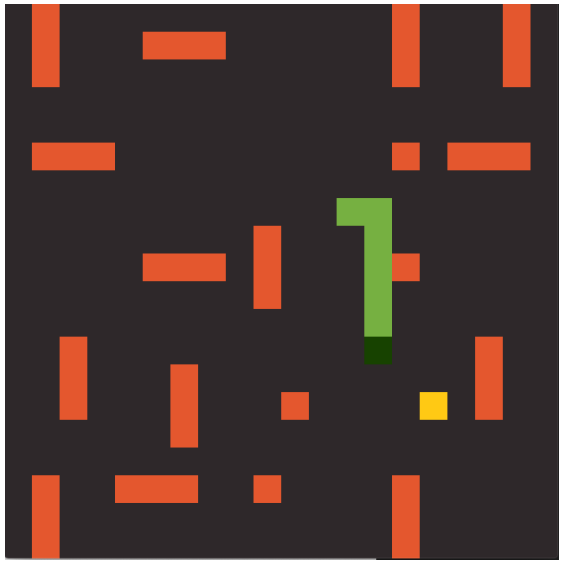

Figure 1 - A picture of the Snake game.

III. Approach

On the one hand, a $n \times n$ snake game can be represented by a graph $G = (V, E)$ which represents the game board where $V = \{v_i\}$ is a set of vertices where each vertex is a block on the board and $E = \{e_{ij} | v_i$ is adjacent to $v_j\}$. The snake itself is the path $\{u_1, ..., u_k\}$ where u_1 is the head and u_k is the tail. Because of this analogy, we can use a search algorithm to find a path from the head (starting node) to the fruit (goal node).

On the other hand, a snake game consists of a number of states and actions that when performed can lead to the goal. Therefore we can use reinforcement learning to learn these actions.

IV. Method

In this section we will describe the different methods we used to create the AI agent that would solve the Snake game.

I. *A** Algorithm.

A* is an informed search algorithm. Starting from a specific starting node, it finds a path to the given goal node having the smallest cost. A* selects the path that minimizes the following equation

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of the path from the start node to n, and $h(n)$is a heuristic function that estimates the cost of the cheapest path from n to the goal. We tried a number of different heuristic functions and compared between them.

A. *Manhattan distance*

The first heuristic function we used is the Manhattan distance between the current node and the goal node which is the fruit. The Manhattan distance between two point (x_1, y_1), (x_2, y_2) can be calculated using the following equation:

$$ManDistance = |x_1 - x_2| + |y_1 - y_2|$$

B. *Combination of Heuristics*

The goal of the snake game is trivial, which is to follow the fruit without hitting yourself. Our first heuristic function - manhattan distance - was trivial, where the agent simply follows the fruit. We didn't expect this heuristic function to get good results so some adjustments were needed to improve our results. We looked for the best techniques to play the game and as a result we used these techniques to improve our results:

a. ***Squareness*** **-** How much snake shape resembles a square.
b. ***Compactness*** **-** Number resembling how close the snake is to itself.
c. ***Dead End*** **-** This is an indication of the size of the area that can't be reached by the snake in the current state due to the body of the snake blocking any pathway to that area.

d. ***Connectivity -*** The same as "Dead End" but here we take a random space in the board and count the spaces that can't be reached from this randomly chosen space due to the body of the snake blocking any pathway to that area.

We used two versions, one is the sum of the heuristics and the other is a linear combination of these heuristics where we assigned different weights to each one of them (as shown in the following equation), and used the result in as the heuristic function for the A* algorithm.

$$Result = w_1 * M + w_2 * S + w_3 * P + w_4 * D + w_5 '$$

Where M is the manhattan distance heuristic, S is squareness, $P$is compactness, D is dead end and C in connectivity. w_i represents the assigned weight for each one of them. After a lot of experiments, we got the best scores when using these weights:

$$Result = 1 * M + 3 * S + 3 * P + 2 * D + 2 * C$$

II. Hamiltonian cycle

A Hamiltonian cycle is a closed loop on a graph where every node is visited exactly once.
A loop is just an edge that joins a node to itself, so a Hamiltonian cycle is a path traveling from a point back to itself, visiting every node on the path.

In case of the snake game, a Hamiltonian path will visit each block on the board exactly once. This means that it is guaranteed that when the snake uses this path, it will always reach the fruit without hitting itself. To put it differently, using the Hamiltonian cycle in a board with no obstacles will always lead the snake to filling the board and therefore to victory.

One downside to following a Hamiltonian cycle is that the snake has to go over all of the path in order to reach the end point (the fruit). To overcome this, our Hamiltonian agent calculates if it is safe to take a shortcut (skip some blocks) depending on the length of the snake and the number of the free blocks between the head of the snake, it's tail and the fruit. If it is 100% safe, the agent takes a shortcut and saves sometime by not going over the full path.

III. Reinforcement Learning agent using Q-Learning algorithm.

A reinforcement learning algorithm learns the optimal action in a set of states by trial and error. This is done by rewarding the agent with a positive reward when performing a good action and rewarding it with a negative reward when it performs that causes a negative outcome.

One of the obstacles in using a reinforcement learning algorithm is the enormous size of the state space. Let's say that the board is of size $n \times n$, then each block has four different possibilities, which can be one of {contains fruit, contains head of the snake, contains an obstacle (a wall or the body of the snake), empty}. This means that the size of this state space is $|S| > n^8$. To avoid this problem we defined our state space in the following way: We only record the contents of the four blocks surrounding the head of the snake (up, down, left and right) and the direction of the fruit relative to the head of the snake (8 directions from the head of the snake, which are NW, N, NE, E, SE, S, SW, W). Therefore the state is of the form

$$\{w_l, w_u, w_r, w_d, f\}$$

where for example w_l describes the contents of the block to the left of the head of the snake. And f describes the direction of the fruit relative to the

head, $f = 01000000$ means that the fruit is to the north of the head.
This reduces the size of the state space to $3^4 \cdot 8 = 648$ states and the performance would be improved in the learning process.

In Q-learning, the algorithm tries to learn the best policy from its history of interactions with the game environment. The algorithm only needs the last state information in order to learn. We define an experience as the tuple

$$< s, a, r, s' >$$

where s, s' are the current and the next state respectively, a is the current actions and r is the reward from performing the action a in state s. Then using this experience, the algorithm will learn by updating the Q-values of the state using the function

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \cdot \max_{a'} Q(s',a') - (\text{}$$

where α is the learning factor, which determines how much the algorithm will learn from the new data, and γ is the discount factor which has the effect of valuing rewards received earlier higher than those received later.

To train the agent, we used an ϵ- greedy algorithm. This means that the agent chooses a random move with probability ϵ and chooses the move with the highest Q value with probability $1 - \epsilon$. At the beginning of the training process, we set $\epsilon = 0.8$ and after every 1000 rounds we reduce it in half. This allows the agent to explore the board at the beginning and then learn from the moves it did in the training process.

We tested a number of different combinations for rewards and at the end used the rewards in Table 1.
We added a -10 penalty each time the algorithm doesn't reach the fruit which causes it to learn to reach the fruit faster.

Event	Eat fruit	Hit wall or body	Other
Reward	+50 * score	-100	-10

Table 1: Rewards Table

V. Results

We started with the case where we have 0 obstacle chance (standard board) then we compared different obstacle chances.

1. Q-Learning results

We wanted our Q-learning agent to learn from the new information it receives while training, therefore we used a higher learning rate $\alpha = 0.9$.

As mentioned before, the discount factor has the effect of valuing rewards received earlier higher than those received later. So in order to find which discount factor to use we ran the game and found which parameter got the best results. We got the results in Figure 2. We found that using $\gamma = 0.85$gets the best results. Any γ bigger than that will get us very similar results.

After finding the parameters for our agent, we trained it by running 50000 times as we described before. Figure 3a shows the results after training on a 12×12 board. We can see that after around 17000 rounds, the agent starts hitting a plateau, and the average score only increases by a small amount.

When training the agent on a board with obstacles, we get a similar learning curve, but the results are significantly lower. Figure 3b shows the training results on a 12×12 board with obstacles.

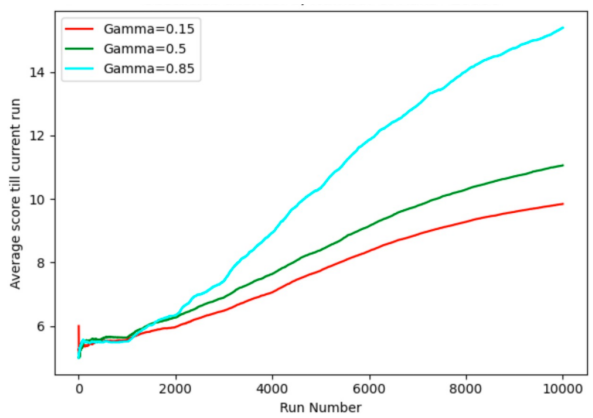

Figure 2 - Results from running the game 10000 round on board size 12 × 12 with different discount factors.

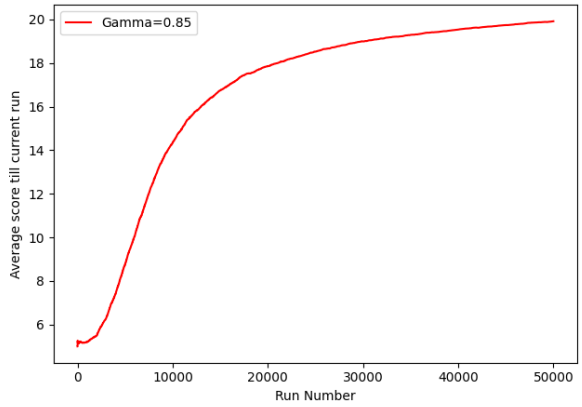

Figure 3a - The results after training the Q-learning agent 50000 rounds on a 12 × 12 board without obstacles.

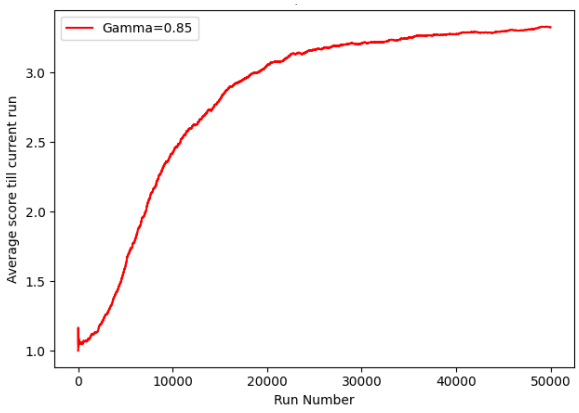

Figure 3b - The results after training the Q-learning agent 50000 rounds on a 12 × 12 board with obstacles chance = 1.

Since the snake only sees its immediate surrounding and the general direction to the fruit, it does not have any knowledge about the location of the rest of its body. As a result, this method won't achieve great results. As we mentioned later, we tried to give it more knowledge but the results didn't improve.

2. A* Results.

We ran the A* algorithm with the different heuristics that we used and got the following results.

a. Manhattan distance

The manhattan distance is a trivial heuristic that only takes into account the distance between the head of the snake and the position of the fruit, and doesn't try to avoid hitting the snake's body. The results we got were not good which was expected from this heuristic. Figure 4 shows the results from running this heuristic on a 12 × 12 board.

b. The heuristic combo

In contrast with the manhattan distance, these heuristics take into account the position of the snake's body at all times, and therefore all the goal of every step was not only to reach the fruit but also to avoid hitting the body of the snake. As a result we were able to achieve better results using them. We used two different combinations of these heuristics as we described before.

When using these heuristics to solve the 12 × 12 board, the agent was able to score 15-20 points more than the agent using manhattan distance. The results can be seen in Figure 4.

The heuristics combinations still outperforms the manhattan distance heuristic on a board with obstacles. But nevertheless, both get lower results in comparison with running with no

obstacles. Figure 5b shows the average scores of different methods on a 12×12 board with obstacle chance = 1.

3. Hamiltonian cycle

In a board with no obstacles, this achieves perfect results in every run. We explained why we got these results earlier. The results from running the Hamiltonian cycle heuristic on a 12×12 board can be seen in Figure 4.

The downside with this approach is that it does not work with obstacles, nor with $n \times n$ boards where n is an odd number. On a board with obstacles, there is no guarantee that there exists a hamiltonian cycle. In this case the algorithm will keep looking for a cycle and it will never find one.

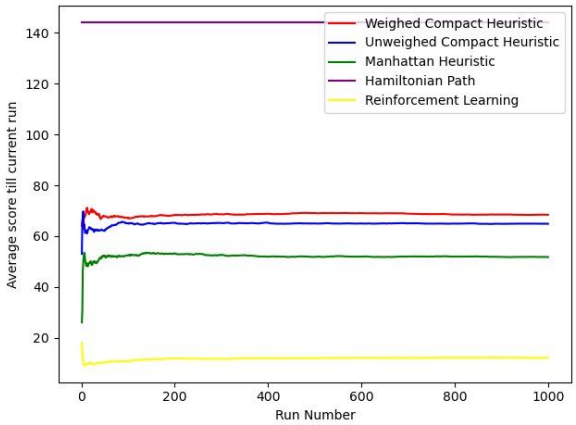

Figure 4 - Average scores from running each algorithm 1000 rounds on a 12×12 board with no obstacles.

4. Results from different board sizes

After seeing the general results for each algorithm, we ran them on different board sizes in order to see how they would perform. We wanted to check how much each method scored on average, and how which method is the fastest. From what we've seen from the previous graph, we see a convergence to the average after 200 runs approximately, so we see that 200 runs is enough for comparing agents.

A. Scores

We ran each one of the methods on 4 board of sizes 6×6, 8×8, 12×12 and 16×16, once with obstacles and once without. We saw that when increasing the board size, the A* heuristics will get higher scores, however they will fill a lower percentage of the board. Moreover, we start noticing the difference in performance between the different heuristics in bigger board sizes, especially in a board with obstacles. Additionally, we found that on a board with no obstacles, the Q-learning algorithm performs almost the same on all board sizes, and it is not affected by the board size as much as the other methods, whereas on a board with obstacles, the performance starts declining when we increase the board size. These results can be seen in Figure 5a and 5b.

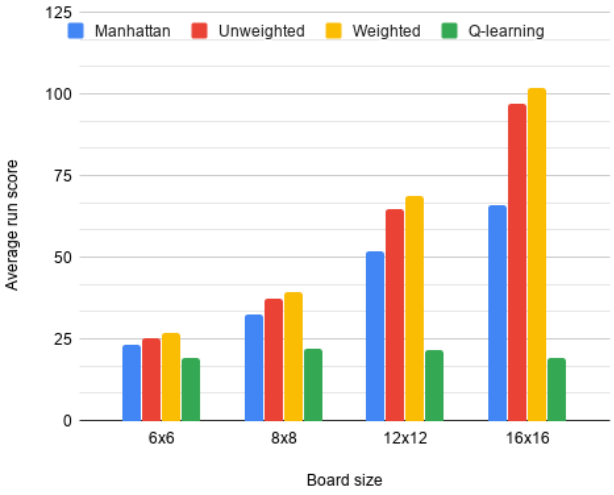

Figure 5a - The average score from running each method 200 times on different board sizes with no obstacles.

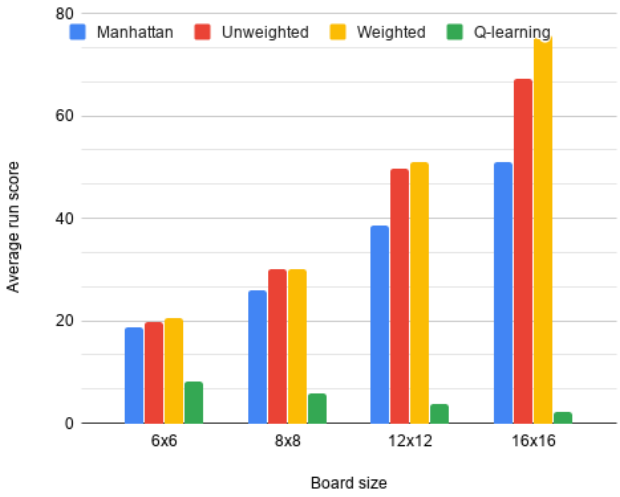

Figure 5b - The average score from running each method 200 times on different board sizes with obstacles.

B. Speed

Speed is an important part of the game. By speed we mean the number of steps taken to achieve the score. To calculate the relation between the speed and the score we used the following equation:

$$relation\ (r)\ = \frac{score}{number\ of\ steps\ till\ game\ end}$$

From this relation, which we will denote (r), we can understand how the agent performs. If r is big, this means that either the agent got a higher score, or the agent performed a lower number of steps.

We found that in a board with no obstacles the hamiltonian cycle is the slowest. This was as expected because it uses a path that traverses the entire board. Then came the weighted heuristics and that is due to the fact that they achieve the highest score and will take a bigger number of steps to do so. The manhattan heuristic is the fastest because as we mentioned before, its goal is to find the shortest path to the fruit, and it doesn't check if the snake gets trapped or not.

In a board with obstacles, the Q-learning agent gets the best results, that is because it gets the lowest scores, and as a result the number of steps taken is minimal. The heuristics perform similar to the board with no obstacles. Figures 6a and 6b show the speed results.

VI. Improvements and Experiments

We tried to improve the results from the Q-learning agent by increasing the size of the state space. This was done by enabling the snake to see its surroundings within a specified radius, but this resulted in worse results, which was unexpected. We also tried to give the snake the relative position of its tail and the fruit, but we got worse results as well. Figure 7 shows the results from other states configurations.

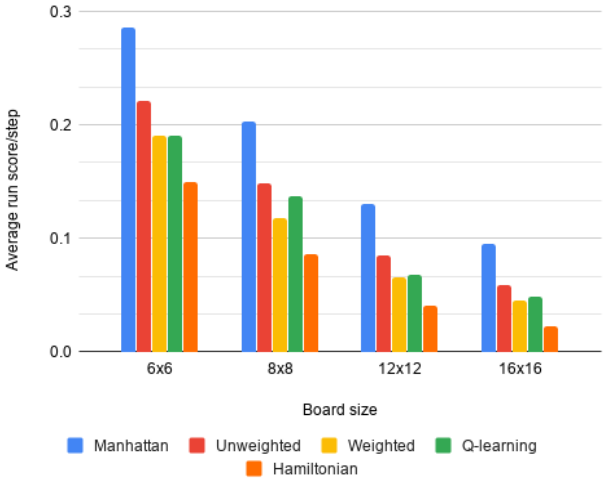

Figure 6a - The average score/steps from running each method 200 times on different board sizes with no obstacles.

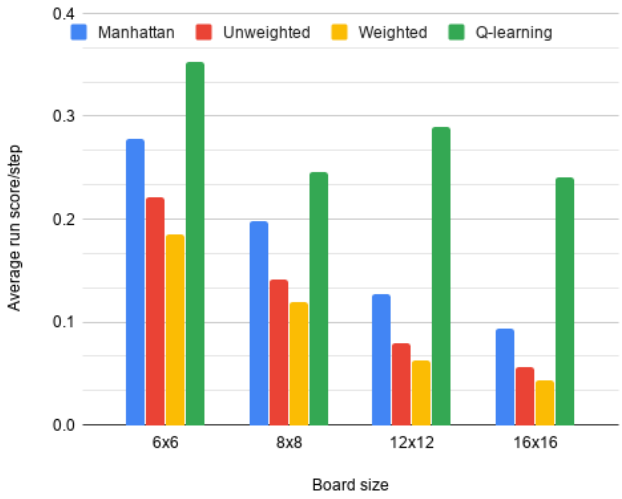

Figure 6b - The average score/steps from running each method 200 times on different board sizes with obstacles.

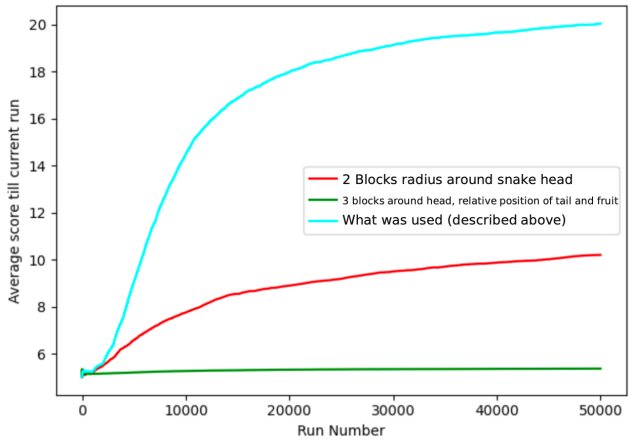

Figure 7 - Results from different states configurations for the Q-learning agent.

VII. Conclusions

In this project, we implemented a number of methods to solve the snake game. We found that using a path finding algorithm will lead to the best results whether the board is with or without obstacles. Moreover, we found that a Q-learning agent with a limited state space will not perform well compared to other methods. Increasing the number of states that the algorithm learns from might increase the score achieved, but this requires much more training time and more memory to store the huge Q-table (which has exponential size).

In addition, we've noticed that in general, agents that perform the best on a board without obstacles tend to perform the best with obstacles as evident from the graphs we've shown above.

VIII. Future Work

In future work, we can use a type of reinforcement learning to get the perfect weights for the combination of heuristics which will lead to the best results. Moreover, we can implement a deep Q-learning algorithm which might be able to achieve better results than the standard Q-learning algorithm.

IX. References

[1] Snake (video game genre) - Wikipedia https://en.wikipedia.org/wiki/Snake_(video_game_genre)

[2] Game AI: Snake Game - Hangry blog https://han-gyeol.github.io/fun/2017/07/08/Game-AI-Snake-Game/

[3] Hamiltonian Cycle: Simple Definition and Example - Statistics How To https://www.statisticshowto.com/hamiltonian-cycle/

[4] Nokia 6110 Part 3 – Algorithms - JOHN TAPSELL
https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/

[5] Exploration of Reinforcement Learning to SNAKE - Bowei Ma, Meng Tang, Jun Zhang
http://cs229.stanford.edu/proj2016spr/report/060.pdf

[6] AI learns to play SNAKE using Reinforcement Learning
https://www.youtube.com/watch?v=8cdUree20j4&t=70s