Snake Challenge

Group 71

Giel Heldens
Akhheel Bandi

The goal of this assignment is to use both an A* search algorithm and reinforced learning to design the Agent for the game: Snake. The game will be played multiple times under different circumstances and all the results will be gathered. The one rule that is always obliged by is that the snake can only turn Left, Right, or keep going Straight. The changing factors include for example the amount of food spawns and wall blocks.

Assignment 1: A* Search

The A* Search method is an informed search method that expands each subsequent state based on the cost from the start state to that current state, and a heuristic function that estimates the cost of the remaining path from the current state to the goal state.

State space model (of the agent)

To describe every state of the state space the networkx library from python is used. This library allows us to create a graph in which we treat each node as a state. To describe each state each node has a name/label and a dictionary of attributes that looks as follows. Node: ((label_x, label_y), {‘path’, ‘ c’,’ body’} where,

(label_x, label_y)	: x and y position of the snake head
‘path’	: list of labels that describing the shortest path from start to current state
‘c’	: cost to reach current state, equals the length of path
‘body’	: list of (x, y) tuples that describe the body parts positions in that state

To retrieve the heuristic for each state, the algorithm stores a dictionary with the heuristic value for each state. Apart from that, the state space also keeps track of the ‘unpacked’ key which can be set to True and is otherwise None for each state, depending on whether that state has been expanded. Additionally the agent defines the direction at each state by looking at the direction of the snake head in the previous state and the relative position of the snake head in the current state and previous state. This variable ‘d’ for direction is set as attribute of the edge that connects the previous state with the current state

Transition function

The transition function is used to update the list of possible next states and determines which next state should be expanded first. Initially just the start state is added to the graph and so it is the only one that we can expand.

Possible moves:
Before a state (or node) is expanded, the agent checks for possible next states depending on the current direction of the snake head and current game obstacles. The game obstacles are the wall block positions (included in input of the agent), the side walls, and the snake body parts (retrieved from the current state). The current direction is either set at North, for the start state, or retrieved from the edge attribute that connects the current state to its previous state. This leaves the agent with a list of at max three potential nodes that neighbour the current node.

Expanding nodes:
Each node in the list of potential nodes is added as a state to the graph with the corresponding attributes described in 'state space model'. Subsequently these nodes are connected to the current node by edges that have 'd'=the next direction as attribute.

Next state:
All unexplored states ('unpacked' is None) at the frontier of the graph are potential next states. The agent sorts these states first by their expected total path cost – the heuristic value 'h' plus the path cost 'c' – and then by path length. The total path cost is ascending and path length is descending. It then simply takes the first state node in this list, defines it as its next state, and sets 'unpacked'=True for that node. This is still considered as A* search since it first sorts by 'heuristic + cost', meaning it will always expand nodes with the lowest estimated total costs first. We decided the algorithm sorts nodes that have the same estimated total cost by the longest path, meaning it expands nodes furthest from the start state first. This will speed up the process of finding the shortest path to the food.

Assignment questions

- **How you model and deal with the multiple instances of food**.
  The algorithm stores the heuristic value for each square on the board using a dictionary. For multiple instances of food the heuristic dictionary will be simply a dictionary of dictionaries that will look like this: h = {0: {}, 1: {} , ..} where the keys of h are the instances of food that address the dictionary of heuristic values, for each square, to that specific food instance.
  When the algorithm is deciding what state to expand next, it will search the dictionary for the heuristic values for each instance of food and choose the minimal heuristic.
- **How you model the body of the snake and how the position of the body is updated.**
  Each state keeps track of the body of the snake. The node for the state has an attribute: 'body' that is a list of tuples of all coordinates occupied by body parts. Initially the list of body parts is empty. As the snake eats its first piece of food, the body parts remain

empty, its only at the state after the snake ate its food that the body part (coordinates of previous state) is added to the body parts.
In the case the snake is just moving across the board, in between the food pieces, the body parts for each new state are always the previous state coordinates plus the previous body parts excluding its last body part.

- **The used cost function and heuristic function**.
The cost function is defined as the length of the path from start state to the current state. The heuristic function is the Manhattan distance between the current state and the goal state. For both functions, up, down, left, or right has a value of 1.
- **What if the A* algorithm returns a failure (meaning no food reachable, at least to the A* algorithm) and can this be solved?**
If no food is reachable according to the A* search, the agent returns None. This will be processed by the snake.py algorithm reset the board and print the score.
In short, no! If we stick to A* search, strictly speaking, there is always a possibility that the snake locks itself in. You would have to plan ahead to make sure the snake never locks itself in and that is not a clean A* search algorithm anymore.
One way to drastically improve the A* search when the snake is locked in is as follows:
    - For each state, track if the state is neighbouring at least two body parts blocks: the body part block that follows the head plus any other.
    - Now, for each of these states we have to find the shortest path that brings us a new ‘unpacked’ state. So basically the algorithm takes an already ‘unpacked’ state, sets up a list of all possible paths to that state and for each path, checks whether, when reaching that state, an undiscovered move is now possible (the body part that blocked the way has moved away).
    - Another simple solution when A* cannot find a path would be to simply make the snake move in a straight line until it finds a new path or eats itself.

Performance

For this section the performance of the A* search algorithm was tested depending on several factors. Statistical data and depending factors will be mentioned with each result.

A* search performance in test setup:
First three tests were conducted in the test setup of the algorithm:
- number of walls spawns = 2
- number of food spawns = 1

For these three cases, the (lowest total cost - heuristic + path cost) was used as the main sorting factor for next possible states.

The first test was conducted by sorting on the longest path and thus expanding nodes further from the start state, this is mainly to decrease computation time.

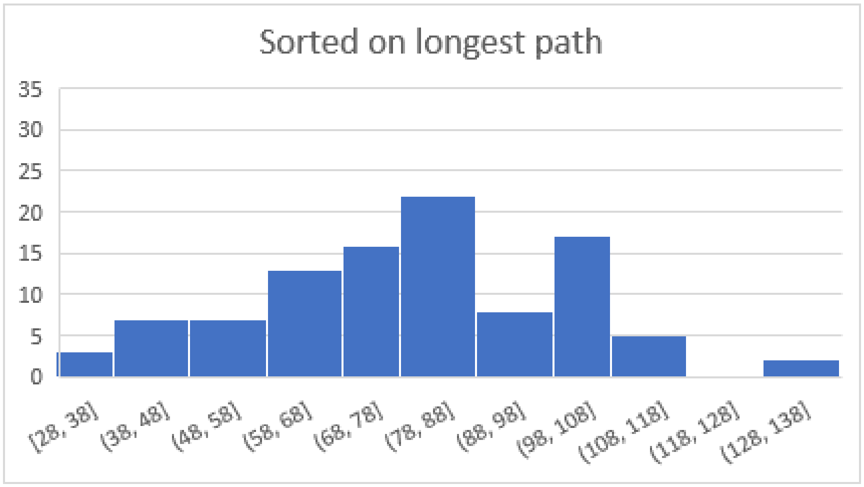


Figure 1: histogram of 100 tries, longest path is expanded first. Frequency/occurrence (y-axis) against range of averages (xaxis).

The second test was conducted by first sorting on the shortest distance to the tail and then the longest path from the start state.

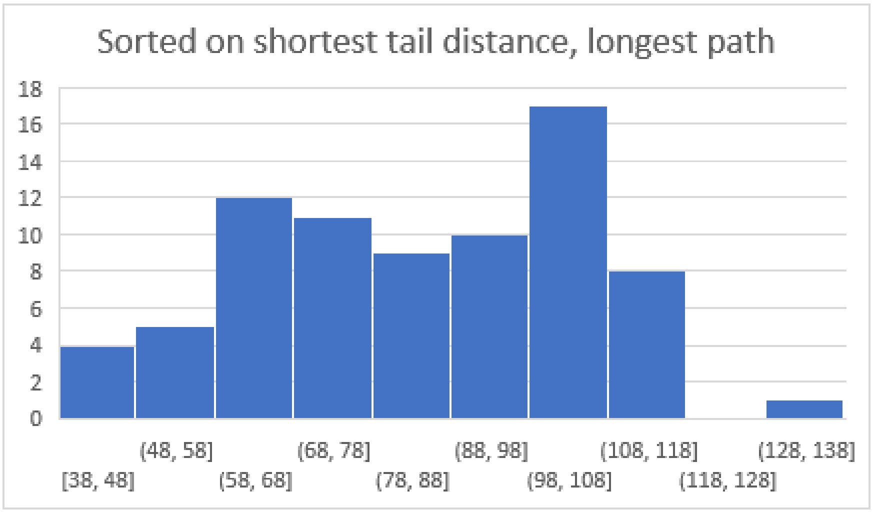


Figure 2: histogram of 77 tries, shortest tail distance expanded first. For equal tail distance the longest path is expanded first. Frequency/occurrence (y-axis) against range of averages (xaxis).

The third test was conducted by first sorting on the shortest distance to the tail and then the longest path from the start state.

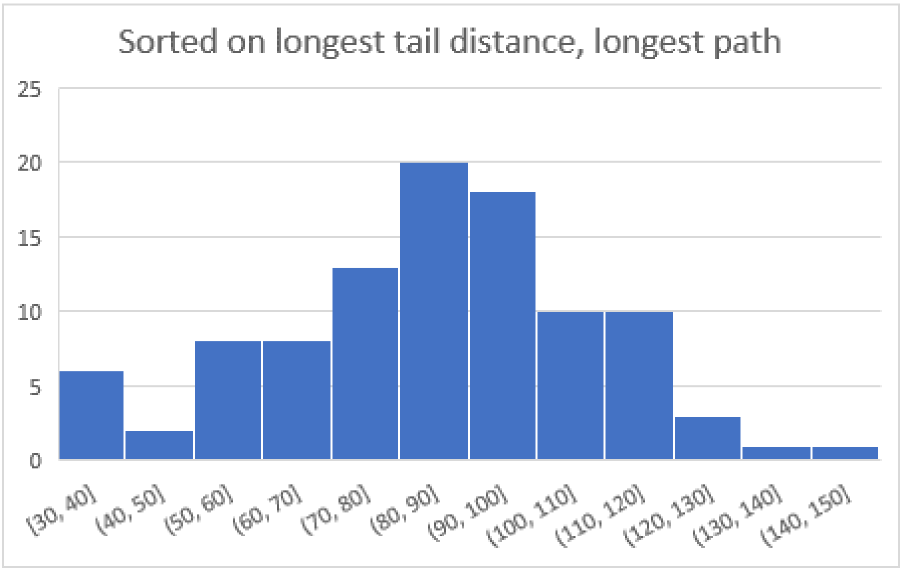


Figure 3: histogram of 100 tries, longest tail distance expanded first. For equal tail distance the longest path is expanded first. Frequency/occurrence (y-axis) against range of averages (xaxis).

Table 1: average and variance of test 1 to 3.

Test	Mean	Variance
1	79.5	471
2	84.6	225
3	85.4	676

A* search performance in non-test setup:
This section includes the results for the test on the snake game with the following configurations:

- 2, 3, and 4 food block spawns
- 5, 10, 15 and 20 wall block spawns

For each configuration the algorithm will use the ‘longest tail distance and longest path first’ technique. Furthermore, all results will be based on 30 test samples.

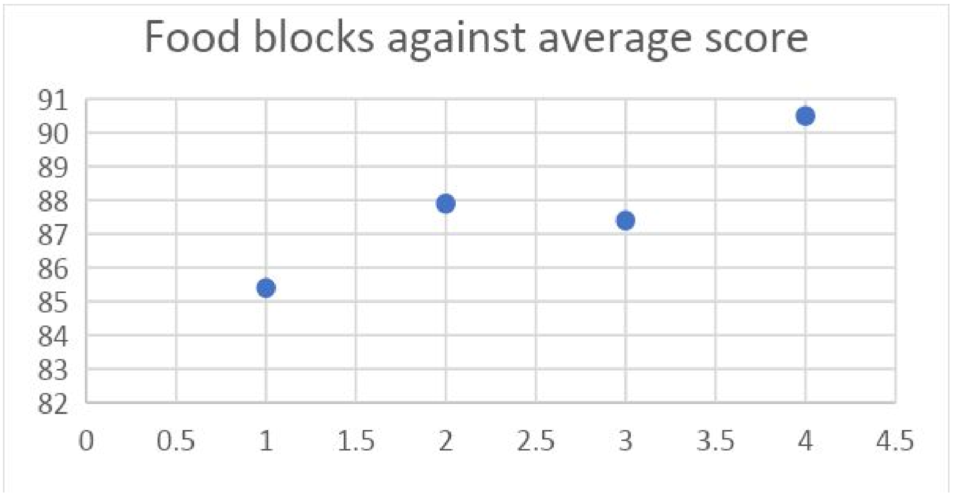


Figure 4: scatter plot of 30 tries, number of food spawns against average score.

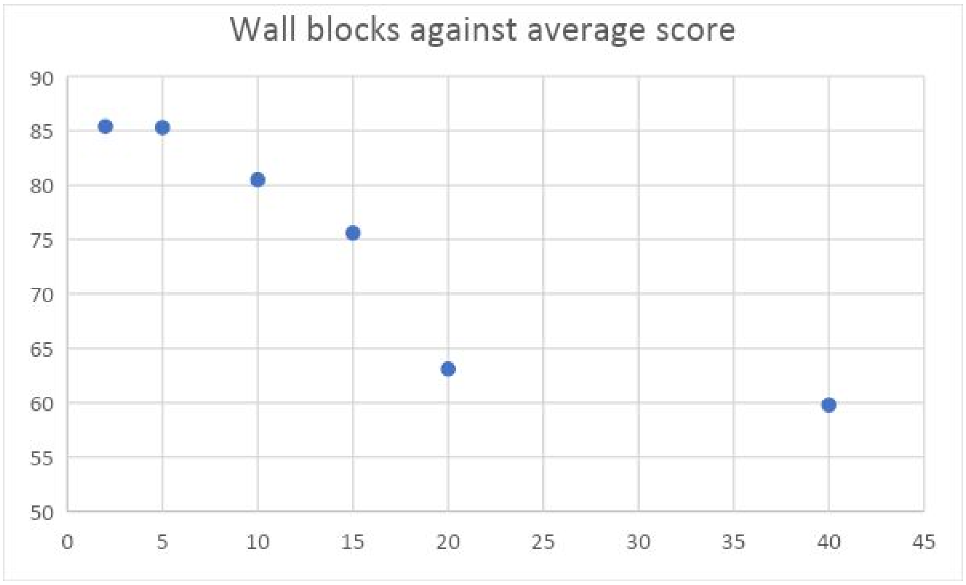


Figure 5: scatter plot of 30 tries, number of walls spawns against average score.

For both instances it is hard to conclude the actual dependency of average score on the changing factor but in general it seems to increase linearly.

Assignment 2: Complexity of A* Search

For this assignment we had to investigate the time complexity of our implementation of A* search for different sizes of the board and numbers of obstacles.

Part a

The theoretical time complexity of A* search depends on the heuristic and belongs to the polynomial class.

In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution which is the shortest path d. So the time complexity would be **O(b^d)**, where b is the branching factor which is the average number of successors per state. If the goal state is not reachable or the state space is infinite, the algorithm will not terminate.

The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes that an uninformed search will expand.

In the case with our algorithm, we have O(3^d) since every node has 3 valid moves and d is the shortest path from the snake head to food. In the snake case, we could assume that on average d is equal to the board size.

Part b

The table below shows the time complexity of our A* search algorithm with respect to the board sizes ranging from 5 to 50. The number of walls and food spans are set at default which are 2 and 1 respectively. The given times are taken in seconds as an average over 30 games.

Board size	**Time**
5x5	0.0007
10x10	0.002
15x15	0.003
20x20	0.005
25x25	0.011
30x30	0.020
35x35	0.026
40x40	0.037
45x45	0.049
50x50	0.075

As we can see from the average times in the table, the size of the board affects the time complexity of A* search. This correlates with the theory in ‘part a’ because as the size of the board increases, the depth of the solution path ‘d’ increases**.** Which will increase the upper bound on time complexity O(3^d).

Part c

For this exercise we had to investigate the time complexity of our A* search algorithm with respect to the number of obstacles. We kept the board size 25x25 and number of food spawns at 1 which are by default. The table with the number of obstacles and it’s respective time is shown below. We took the range of obstacles from 4 to 40 with intervals of 4. The given times are taken in seconds as an average over 30 games.

Number of obstacles	**Time**
4	0.012
8	0.011
12	0.010
16	0.010
20	0.011
24	0.011
28	0.010
32	0.011
36	0.010
40	0.010

As we can see from the average times in the table, the number of obstacles on the board does not affect the time complexity of A* search as long as the size of the board remains the same. Which again correlates with the theory in ‘part a’. The time complexity O(3^d) does not depend on the number of obstacles or goal states. It only depends on the number of successors per state which is 3 in our game and the depth of the solution path.

Assignment 3: Applying Reinforcement learning

The goal of this assignment is to apply Reinforcement Learning to the Snake game.

Part a

The state space U is created by enforcing the Bellman equations locally, updating U_(s) as:

$$U^{\pi}(s) <- U^{\pi}(s) + \alpha(R(s) + \gamma(U^{\pi}(s') - U^{\pi}(s))$$

where,
R(s) is the learning rate, U(s) the state space, α is the learning rate and γ the discount factor. We can enforce convergence by letting α decrease over time. The convergence speed is defined as the amount with which the learning rate decreases each loop. At learning rate = 0 the learning stops and the algorithm chooses a moves. Normally, action selection still needs to take p(s'|s, a) into account but our algorithm can simply neglect this since its success rate for any given move is 1 (or 100%). It is also for this reason that we can simply stick to U-learning. Q-learning would otherwise be used so that one does not have to model p(s'|s, a), but in our case it is always 1. The reward function R(s) is defined as 0 for any non-goal state and 1 for the goal state.

Below is a table with average learning speeds with respect to different values in learning parameters. In this part of the assignment, we set a 5x5 board and the snake doesn't grow when it eats food.

Learning parameters	**Learning time**

discount_factor = 0.1
learning_rate = 0.1
convergence_speed = 0.1
	0.00012172150611877441

discount_factor = 0.5
learning_rate = 0.1
convergence_speed = 0.1
	0.00017198491096496582

discount_factor = 1
learning_rate = 0.1
convergence_speed = 0.1
	0.00024017167091369628

discount_factor = 0.5
learning_rate = 0.5
convergence_speed = 0.1
	0.0007267160415649415

discount_factor = 0.5
learning_rate = 1
convergence_speed = 0.1
	0.001192702054977417

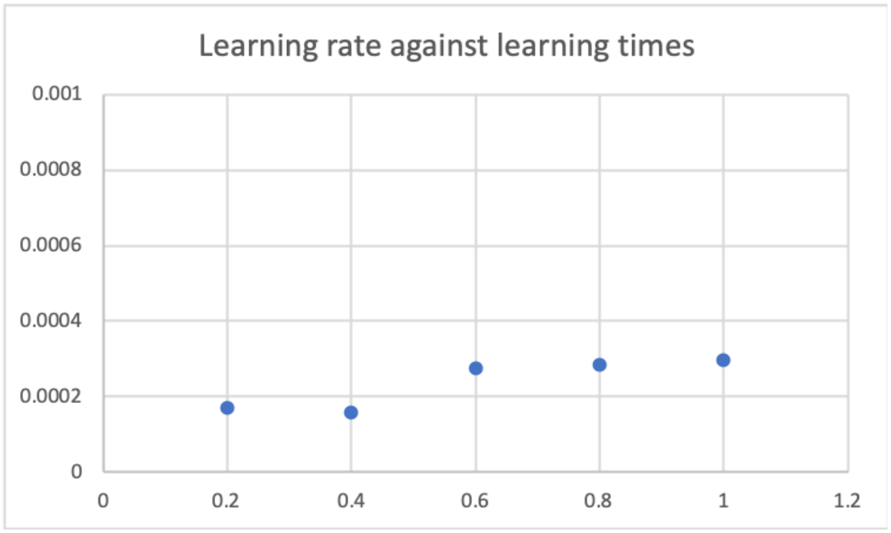
discount_factor = 0.5	0.00025581002235412596

learning_rate = 1
convergence_speed = 0.5	

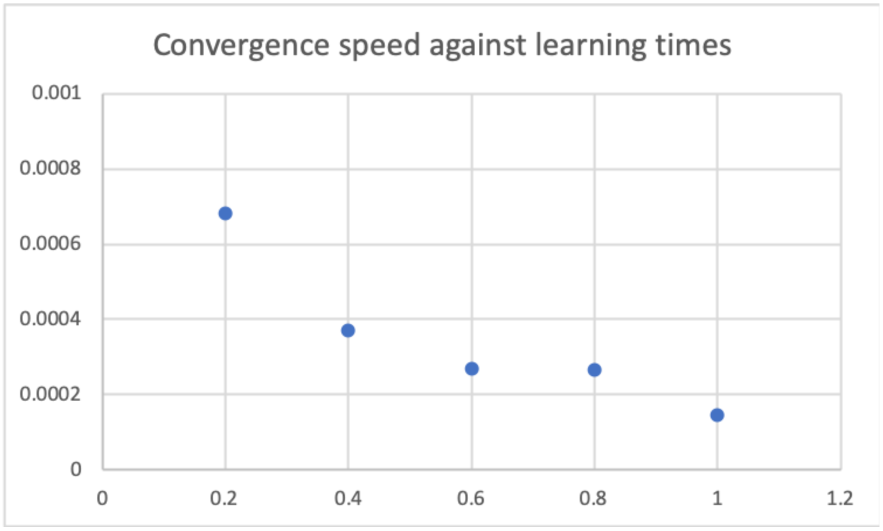
discount_factor = 0.5
learning_rate = 1
convergence_speed = 0.9
	0.0002802639007568359

We also have three scatter plots to show how the learning speed is influenced by three different learning parameters.

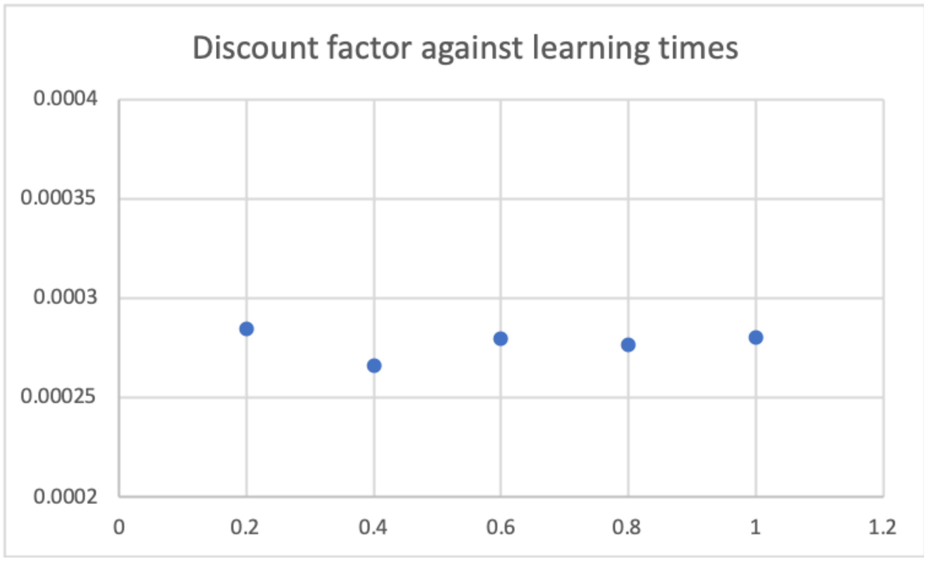
The first plot shows how learning time is affected by the learning rate. Here we kept the discount factor at 0.5 and convergence speed at 0.5 and learning speed is in the range 0.2 to 1 by steps of 0.2.



In the second graph, we plot convergence speed against learning times. Here learning rate = 1 and discount factor = 0.5 and the convergence speed ranges from 0.2 to 1 in steps of 0.2.



In the final graph, we plot discount factor against learning times. Here learning rate = 1 and convergence speed = 0.9 and the discount factor ranges from 0.2 to 1 in steps of 0.2.



As we can clearly see from the plots, discount factor and learning rate does not influence the learning time very much, but the learning time is highly dependent on the convergence speed and the higher the convergence speed, the lower the learning time.

Part b

The algorithm deals with the state explosion by only resetting the state space after it eats food. In between food instances it stores the state space and keeps updating it for every move. In this way it will initially not always move towards the food (depending on the learning parameters and the goal distance) but with every move be more likely to move towards the food.

Part c

In this part of the assignment we make the board 25x25 again and set the default parameters from assignment 1. The algorithm for solving the test situation (test_config = True), with growing on food is as follows.

- The first thing the agent does is checks whether it is either the first turn after eating food or the very start of the algorithm.
    - If this is NOT the case, it retrieves the state space from the previous state and continues learning on that state. In this way the algorithm does not need to start its learning from a blank state space after every move, but only after every time the snake ate.
    - If this is the case the agent initialises the state space using the board inputs. It creates a dictionary: Utility, for the state space, taking the (x, y) position of the

non_wall and food blocks as keys, with values 0 and 1 respectively.
The reward function is 0 for all non-wall blocks and 1 for the food block (the food block state is never updated however)

- Next the algorithm enforces the Bellman equation locally (see part 3 a) with learning rate α and discount factor γ for each state in the state space. It then achieves convergence by letting the learning factor decrease gradually to 0. The size of these steps if the convergence speed.
- Ultimately the agent, after it is done learning, chooses the optimal (highest utility) neighbouring state as its next move. After this the agent repeats the algorithm.

Below is a table with the learning times with respect to the different values of learning parameters which we took the same as in part b, so that there is consistency and we can compare the results.

Learning parameters	**Learning time**

discount_factor = 0.1
learning_rate = 0.1
convergence_speed = 0.1
	0.0030239267349243164

discount_factor = 0.5
learning_rate = 0.1
convergence_speed = 0.1
	0.003090662956237793

discount_factor = 1
learning_rate = 0.1
convergence_speed = 0.1
	0.003141897201538086

discount_factor = 0.5
learning_rate = 0.5
convergence_speed = 0.1
	0.016891170740127564

discount_factor = 0.5
learning_rate = 1
convergence_speed = 0.1
	0.027820087909698487

discount_factor = 0.5
learning_rate = 1
convergence_speed = 0.5
	0.00581368088722229

discount_factor = 0.5
learning_rate = 1
convergence_speed = 0.9
	0.005820889234542847

The game performance is noticeably worse than A* search. The average score achieved over 30 games with the default configurations on a 25x25 board are shown below. For reinforcement learning, the learning parameters used are as follows: discount_factor = 0.5, learning rate = 1, convergence_speed = 0.9

A* search	**Reinforcement Learning**
81.27	34.13