Reinforcement Learning to Solve Snake with Randomized Obstacles

Antoine Bargé[1] and Victor Zhang[2]

Abstract— **In this report, we present six different algorithms to play a modified snake game with. These include: three baseline methods, one search method based on A*, and two reinforcement learning methods, Q-Learning and Sarsa. The latter used two different exploration strategies. We conducted experiments to compare their performance and explain their differences. Furthermore, we anticipate possible improvements as future work.**

I. INTRODUCTION

This report is the final report of our project for the CS221 course. Our project is focused on the implementation of AI strategies for a snake game.

We implemented several AI algorithms for the game. In this report, we are going to present our models and describe how we have adapted them to our snake game. We have used two reinforcement learning approaches, Q-Learning and Sarsa, as the problem can be modeled as a Markov Decision Process. The goal was to compare them to our oracle, A*.

We show the difference between the different approaches and quantitatively demonstrate their performance with several experiments.

II. PROBLEM STATEMENT

A. Environment

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

Our snake game is slightly modified in that there are 10 obstacles, randomly placed in the grid at each round. Throughout the whole project, we use the Pygame library to visualize a given simulation. We aim to determine which algorithms would be more effective, not only to play the game, but also to compete with human players.

[1]Antoine Bargé is with Stanford University, Department of Management Science and Engineering abarge@stanford.edu

[2]Victor Zhang is with Stanford University, Department of Mechanical Engineering zhangvwk@stanford.edu

B. Framework

Our implementation consists of the snake moving on the square grid, trying to eat as many apples as possible without eating itself or hitting a wall or an obstacle.

A player controls the snake by selecting the direction in which the snake moves. To score a point, one has to direct the snake to an apple. In our game, there is only one apple at a time and it appears randomly. Eating an apple increases the snake's length by one unit. The games is over when the snake runs into one of the boundaries, one of its body parts, or one of the obstacles. Additionally, there is a clock and the game automatically stops when it runs out.

III. FIRST APPROACHES

In this section we present three baselines strategies and one AI method to play snake. The AI algorithm belongs to a heuristic search and baseline methods were developed intuitively.

A. Baselines

Several simplistic strategies can serve as baselines. We chose to implement the following three:

- Random: the snake moves randomly on the grid but avoids the boundaries and the obstacle.
- Vertical: the snake moves up until the boundary edge, then goes down, and so on, all the while going around the obstacle upon encountering it.
- Greedy: this strategy is similar to a Best-First Search algorithm, as it only considers one move at a time. It directs the snake to the position of the grid closest to the apple, in terms of the Manhattan distance, chosen for its accuracy and fast computation (given it is only two integer comparisons).

B. Oracle

Our oracle is an A* algorithm.

A* incorporates a heuristic. Before taking action, it considers not only the distance to the apple, but also the current state it has searched so far. We chose to use the Manhattan distance from the head to the apple as a heuristic and the number of steps as the cost so far. A* is an improvement of the greedy search because it finds a full path to the apple and not only stops at the first move.

A* also has the advantage of rarely getting stuck at a dead end on the way to the apple. This can happen if the snake is very long. Furthermore, A* is guaranteed to find the optimal path to the apple if it exists.

C. Metrics

In this section we present our preliminary results. We used the following metric of success. The final score is determined by the number of apples that have been eaten and depends on the length at the time of death or when the clock runs out. Therefore, we use the length of the snake when the game ends as a metric of success.

We implemented and tested our baselines and oracle. In the following table, one can see the percentage of deaths and the average scores after 100 games.

TABLE I
BASELINES / ORACLE COMPARISON

	Random	**Vertical**	**Greedy**	**Oracle**
Average score	0.03	0.1	0.3	55.7
Deaths (%)	98	100	100	5

The average scores for the baselines (random, vertical and greedy) are all very low because of the presence of obstacles, as they are not even taken into account.

IV. REINFORCEMENT LEARNING METHODS

A. Modeling

Reinforcement learning involves an agent, a set of states and a set of actions per state. By performing an action the agent transitions from a state to another state. By executing an action in a specific state, the agent gets a reward. The goal of the agent is to maximize the sum of its future rewards. It is calculated as the sum of the maximum reward attainable and the reward of current states.

We modeled our snake game as a Markov Decision Process (MDP), the agent only needs to remember the last state's information. The MDP is defined as follows: model = $(S, A, P, R, \gamma, \mu)$, with S the set of all possible game states, A the finite set of actions, P the probability transition matrix, R the reward function, γ the discount factor and μ the set of initial probabilities of the states.

1) State space: The finite set of all possible game states depends on the state representation. Our state space representation is a tuple composed of five elements $(f_l, f_r, f_s, q_x, q_y)$.

- For dir $\in \{\text{left}, \text{right}, \text{straight}\}$:

$$f_{\text{dir}} = \begin{cases} \text{-1} & \text{if going dir results in death} \\ \text{+1} & \text{if going dir results in an apple} \\ 0 & \text{if going dir results in nothing} \end{cases}$$

- For $i \in \{x, y\}$, q_i is the quadrant of the i coordinate of (snake's head - apple).

The above representation and directions are relative to the position of the snake's head. A quick analysis of the state space $\mathcal{S}$ shows that $|\mathcal{S}| = 153$. We start off with $3^5 = 243$, from which we subtract $9 + 27 \cdot 3$ (invalid states) which brings the number of states to 153.

2) Action space: The action space $\mathcal{A}$ is also relative to the snake's direction, therefore

$$\mathcal{A} = \{\text{go straight, go left, go right}\}$$

3) Reward model: We adopted the following reward model, when taking action a in state s:

$$R(s, a) = \begin{cases} l >> 0 & \text{if the snake eats an apple} \\ d << 0 & \text{if the snake dies} \\ c < 0 & \text{else} \end{cases}$$

The only positive reward l occurs when the snake eats an apple. A large penalty d happens whenever the game stops, i.e. whenever the snake hits a wall, an obstacle or eats itself. Note that a small penalty c for every other situation is there

to encourage the snake to go towards the apple as soon as possible. For now, we have fixed (l, d, c) to $(50, -10, -0.1)$, but we can use different values of l and d to evaluate the model performance in different environments.

B. Q-Learning

Q-learning is a reinforcement learning technique whose goal is to learn a policy. The policy tells an agent what action to take under given circumstances.

Q-learning uses a Markov Decision Process model to control and make decisions. It consists of a Q table that is constantly updated. The Q table is matrix mapping $(s, a) \in \mathcal{S} \times \mathcal{A}$ to its corresponding $Q(s, a)$ value, according the algorithm described below:

- initialize $t \leftarrow 0$ and state s_0
- initialize Q (e.g. 0 on $\mathcal{S} \times \mathcal{A}$, or a positive value if we are optimistic)
- iterate the following steps:
  1) choose an action a_t based on Q and some exploration strategy;
  2) observe reward r_t and next state s_{t+1};
  3) update $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_t + \gamma \max_{a \in \mathcal{A}(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t)\big]$

Note that there are a couple of trade-offs that need tuning to get the best possible training result:

1) the learning rate α, which we expect to be high in the beginning for fast changes, and lower as time progresses: so far, our learning rate is constant but we intend to try out a time-dependent version to see how it might alter the performance of the algorithm;
2) the discount factor γ: since we don't want a shallow look-ahead, for now we have fixed $\gamma = 0.99$;
3) the exploration strategy: not only which one to use but also exploration parameters (see IV. D.).

C. Sarsa

Sarsa is very similar to Q-learning. The difference lies in the way the Q-Table is updated. With Sarsa, we use the following formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha.(r + \gamma Q(s', a') - Q(s, a))$$

Sarsa agent will update the policy based on the actions taken, its an on-policy learning algorithm. Q-learning is off-policy.

D. Exploration strategies

The most difficult challenge is to devise a suitable exploration strategy for our problem. The trade-off between exploration and exploitation is an inherent dilemma to reinforcement learning, and we carried out experiments with the following three different strategies: ϵ-greedy, decaying ϵ-greedy and Softmax.

1) ϵ-greedy: This is the most basic approach, consisting in selecting with probability $1 - \epsilon$ the action yielding the largest Q value in a given state, and a random action with probability ϵ. Formally, in state s, the agent selects an action according to the following rule:

$$\pi_\epsilon(s) = \begin{cases} \text{Random}(a) & \text{with probability } \epsilon \\ \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \end{cases}$$

2) Decaying ϵ-greedy: One way to make the ϵ-greedy approach less brutal is to reduce it over time: that way, the agent explores more in the beginning and less as time goes on. Formally, in state s and iteration n_{iter}, the agent selects an action according to the following rule:

$$\pi_{\epsilon, n_{\text{iter}}}(s) = \pi_{\epsilon'}(s)$$

where $\epsilon' = \epsilon / d^m$ with $m = \lfloor n_{\text{iter}} / n_{\text{threshold}} \rfloor$, meaning we divide ϵ by d every $n_{\text{threshold}}$ steps.

3) Softmax: The problem with the two strategies above is that they are precisely "greedy", in that only the one action that yields the largest Q value is considered, when another one might actually result in a better policy. One way to overcome this is to rank all the actions in a given state and assign a weight to each of them. Formally, in state s, the agent chooses action a with probability $\pi_{\text{sm}}(s, a)$:

$$\pi_{\text{sm}}(s, a) = \frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}$$

where the parameter T is tuned. A large T results in all actions being treated equally and a

small T results in greedy selection.

V. RESULTS

In this section, we present the results obtained with the reinforcement learning methods. We compare the different methods themselves and also study the importance of the hyperparameters.

We tested several exploration strategies, as explained in the previous section. We chose to present two types of graphs:

- The first one gives the average score, $\overline{\text{score}}$, at each iteration, meaning it is averaged over all the former iterations. Formally, it is computed as follows:

$$\overline{\text{score}} = \frac{1}{n_{\text{iter}}} \sum_{k=1}^{n_{\text{iter}}} \text{score}_k$$

  where score_k is the score at the k^{th} iteration.
- The second one represents the moving average of the real score at each iteration. In all the following graphs, we chose a moving average of 500.

Let us first recall that our oracle, the A* algorithm, has an average score of 55.7. The reinforcement learning approaches do not perform as well. Shown on Figures 2, 4 and 6 are the moving averages for Q-learning (using ϵ-greedy and Softmax) and Sarsa (using ϵ-greedy). Even though the highest scores obtained were high, we decided not to show the immediate scores because they are not as representative as the average and the moving average graphs.

A foremost and common observation is that all reinforcement learning methods (regardless of the hyperparameters we used) outperform our best baseline (the greedy one) by a factor greater than 10, in terms of average score. Second, there are notable differences between the average and the moving average results, that are discussed in the subsections below.

We decided not to show the results of using the standard ϵ-greedy approach as it yielded very suboptimal results, compared to the other two approaches.

A. Q-Learning

1) ϵ-greedy exploration: One first interesting of Q-Learning to tune is the ϵ of the ϵ-greedy exploration strategy. On Figures 1 and 2, one can see the effects of ϵ. We achieved a better average score without any exploration ($\epsilon = 0$). In other words, our Q-Learning algorithm does not need much exploration. This is due to our very compact state representation. However, one caveat for using $\epsilon = 0$ is that the snake sometimes gets stuck in local optima. That was overcome by putting a limited training time. Also, to further contrast using no exploration with using exploration, we capped the lowest possible ϵ at a value of 0.01 for the decaying ϵ-greedy strategy. That is, we keep diving the value of ϵ by 2 every 500 training iterations as long as it is greater than 0.01. The value of 0.01 was tuned empirically: we chose the one that gave just enough exploration so that the snake eventually gets out of its

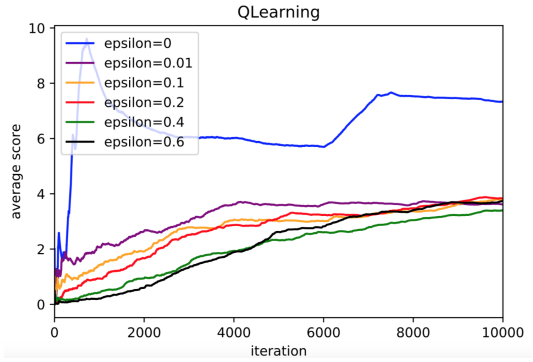

Fig. 1. Average score for 10000 iterations, using Q-Learning with a decaying ϵ-greedy exploration strategy.

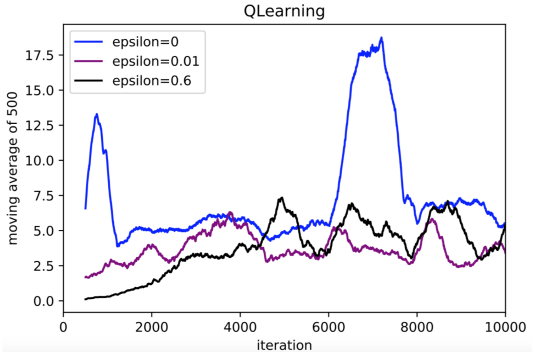

Fig. 2. Moving average of the score at each iteration, using Q-Learning with a decaying ϵ-greedy exploration strategy.

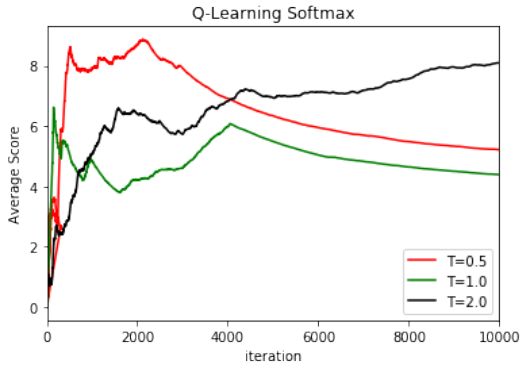

Fig. 3. Average score for 10000 iterations, using Q-Learning with a Softmax exploration strategy.

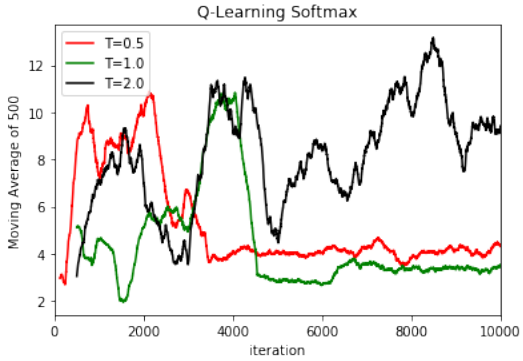

Fig. 4. Moving average of the score at each iteration, using Q-Learning with a Softmax exploration strategy.

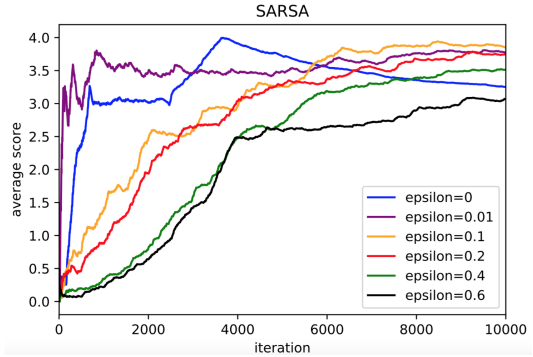

Fig. 5. Average score for 10000 iterations, using Sarsa with a decaying ϵ-greedy exploration strategy.

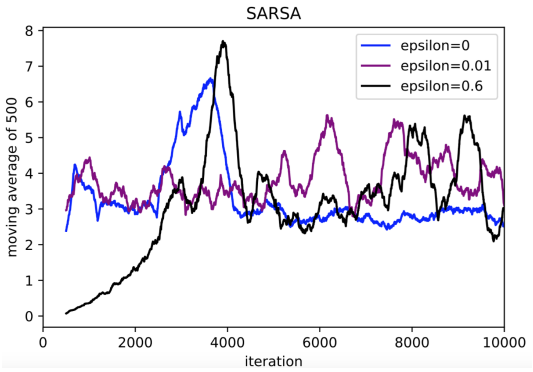

Fig. 6. Moving average of the score at each iteration, using Sarsa with a decaying ϵ-greedy exploration strategy.

local optimum before the maximum training time.

2) Softmax exploration: The same approach was conducted with the Softmax exploration. We tested different values for the parameter T and one can see the results on Figures 3 and 4.

When comparing the first four figures, one can see that the Softmax exploration strategy yields better results than the ϵ-greedy strategy. The average scores are higher with the Softmax and the moving average shows much higher scores when $T = 2$, which makes sense as larger T corresponds to less greedy selections.

B. Sarsa

The second reinforcement learning method we implemented is Sarsa. As before, we decided to show the effects of different explorations on the score. One can see the results on the Figures 5 and 6.

An interesting fact is that the exploration strategy has more impact, in our case, for Sarsa than for Q-Learning. It is because the former is an on-policy algorithm as opposed to the latter which is off-policy. Thus, one can see here that higher scores are achieved with more exploration: in fact the moving average with largest peak is obtained for the highest exploration probability ($\epsilon = 0.6$). That was the opposite with Q-Learning, for which the largest peak was obtained with no exploration at all ($\epsilon = 0$).

C. Discussion and error analysis

1) Error analysis: The main issue that was observed is the deterioration of the learning process whether it be using Softmax or ϵ-greedy with a low ϵ. The accuracy of the Q values actually seems to decrease at some point, which might be counter-intuitive at first. The situations in which the agent is stuck in local optima seem to increase in number over the training period. By local optima, we

denote a situation in which the snake navigates round and round without dying but without getting anywhere close to the apple. Note that the cautionary word "seem" is employed here to remind that these are empirical observations only.

When looking at the moving average graphs, one can also observe that the Q-learning algorithm is not very stable. The scores indeed tend to vary, even after a long period of training. It seems as though the agent goes through certain phases: at times, given certain "favorable" configurations, it manages to learn very well and improve its policy (peaks), and at other times, given certain "unfavorable" configurations, it starts to update its Q values in a way that is less and less optimal.

One possible explanation for this strange phenomenon is that the compactness of the state-space representation is a double-edged sword: on the one hand, the agent learns very fast in just a few hundreds iterations, but on the other, updating too much the Q values may decrease their accuracy. To illustrate this point, we extracted the policy after a 300-iteration training using $\epsilon = 0$, and tested it on 1000 new testing iterations without updating the Q table. The average score obtained was 18.73 compared to an average of 7.39 when using the policy extracted after 10^4 training iterations. Therefore, there seems to be a threshold $n_{\text{iter}_{\text{thresh}}}$ in the number of training iterations after which the agent doesn't learn any better and might even update its Q table in a way that results in a worse policy.

A closer look at our state space representation shows that out of the 153 possible states, only 84 are actually reached by the agent. The states that aren't reached are typically absurd cases where we wouldn't expect the agent to get into (e.g. the agent is stuck between three walls and the apple is outside of them). This even smaller state space explains why the agent manages to obtain high scores in only a few iterations, but why do we observe a decrease in accuracy in the Q table? Is this due to an incomplete state-space representation or something else? In the next section, we try to answer these questions by experimenting on the presence of the obstacles.

2) Discussion: In this subsection, we compare the performances of our reinforcement learning algorithms. The learning curves for Q-learning and Sarsa can be found in the previous section. We can observe that the performance of agent with Q-learning get improved faster than that of agent with Sarsa in the beginning. This is especially the case with the Softmax exploration or without any exploration at all. This means that in a short run, the agent with Q-learning algorithm outperforms the agent with Sarsa. However, as the number of iterations increases, the performance of the agent with Q-learning does not improve much, while the performance the one with Sarsa still gets improved. Q-learning does well when the training period is short. But in the long period, the two approaches are very similar in terms of performance.

The exploration strategy has an influence on the performance of Q-learning. The Softmax exploration gives better score than the ϵ-greedy. Nevertheless, all these scores remains lower than those obtained with A* Search, and so no matter how long the training period is.

D. Effect of obstacles

One important factor that we need to address is the presence of obstacles. How does adding obstacles to the problem affect the learning process of the reinforcement learning agent? We ran our Q-learning agent using a Softmax exploration strategy with $T = 1$ on configurations with: no obstacle, fixed obstacles and random obstacles.

1) No obstacle: This configuration corresponds to the standard Snake game: there are no obstacle whatsoever.

2) Fixed obstacles: Let us now add n_{obs} fixed obstacles. Our experiments were run using $n_{obs} = 10$. One can see the combined results on Figure 7. It is interesting to see that the presence of fixed obstacles doesn't seem to have any influence on the score, compared to a configuration with no obstacle at all. This is due to the fact that 10 obstacles on a 20 by 20 grid leaves large free spaces for the snake to safely navigate in, and also that since they are fixed, we can simply consider them to be an inherent part of the map. The scores are also much higher than with obstacles displayed in a random way at each

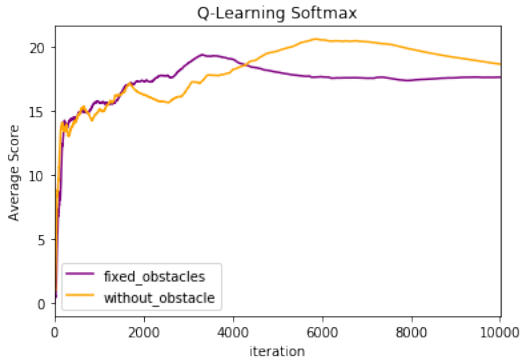

Fig. 7. Average Score for 10000 iterations, using Q-learning with a Softmax exploration strategy.

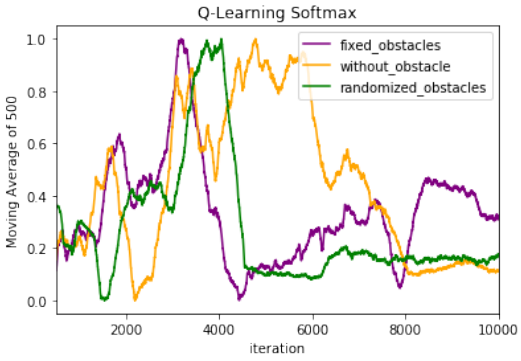

Fig. 8. Normalized moving average of the score at each iteration, using Q-Learning with a Softmax exploration strategy, for the three different configurations.

turn.

3) Randomized obstacles: Let us now add randomness to the locations of the obstacles: every time the snake dies, a new set of n_{obs} randomly generated obstacles appears. Again, the experiments were run using $n_{obs} = 10$. The results can be seen on any figure of the previous section. Overall, the scores are much lower with randomized obstacles.

4) Comparison of learning curves in the different cases: To see whether or not the presence of random obstacles influences the learning process and if that might be the reason for the previously mentioned policy degradation over training steps, we normalized the moving average curves for each configuration (no obstacles, fixed obstacles and random obstacles) to focus only on the variations. The results can be seen on the Figure 8. All three have drastic negative variations. Therefore, it seems as though the randomization of obstacles is not what causes the policy degradation.

VI. FUTURE WORKS

A. Stability of Q-Learning

In some cases, the performance of our Q-learning algorithm is not very stable. That's why we implemented a decreasing exploration probability and although it improved the results, there are still some improvements to be made as the negative variations are at times drastic. Therefore, a first future work could be exploring methods to improve the stability of the Q-learning algorithm and understanding why the policy degrades over training steps. For example, one can think of adding various tuning parameters. It could improve convergence of the Q-learning algorithm.

B. Deep Q-learning

An amelioration of Q-learning could be Deep Q-learning. The general idea is similar to Q-learning but it doesn't use a Q table. The goal of Deep Q-learning is to efficiently approximate the optimal action-value function Q of the MDP, so that the agent could play the game with the following greedy policy:

$$\pi(s) = \arg\max_{a \in A} Q(s, a)$$

The network receives the state as an input and outputs the Q values for all possible actions. The largest output is chosen as the next action.

However, Deep Q-learning would need another state representation for it to be useful in our case. Our current state representation is indeed now very compact and doesn't really call for using neural networks, and Deep Q-learning would not necessarily be a big improvement. Therefore, it makes sense to use another state representation in parallel. Now, because of the use of two quadrants, our snake does not have a very safe approach for not hitting itself. A new representation could include relative positions of the snake's head to its tail, as well as its entire body positions. Another, more straightforward approach would simply be to consider a state to be the frame of the game. It would significantly enlarge the size of the state

space, but some pre-processing (such as grey-scaling) combined with Deep Q-learning could potentially yield better results.

C. Complexifying the game

To make the problem even more challenging, we could also add multiple snake agents to compete with each other for one apple at a time. We could therefore directly compare multiple strategies for reaching the higher score. Minimax could be a first possibility.

VII. CONCLUSION

In this project, we developed several methods in order to solve a complexified snake game. We showed implementations of three baselines and of an A* Search. Then, we described two reinforcement learning methods, Q-learning and Sarsa, with three different exploration strategies. We saw that the performances of Q-Learning and Sarsa are similar after a long training period, and that the Softmax exploration strategy yields better results. When comparing these two reinforcement learning methods with our A*, we found that that neither of the two agents could achieve the performance of the search algorithm. This is true even after a long training period. One question that remains is the policy degradation problem, which we are not certain how to explain.

We would like to thank Professor Liang and all the eaching staff for the CS221 course as well as our project mentor Jaebum Lee.

All our code (including GIFs demos) can be found here:
https://github.com/zhangvwk/modified-snake-reinforcement-learning

REFERENCES

[1] Andrew Barto and Richard S. Sutton, "Reinforcement Learning: An Introduction", *MIT Press*, 2015.
[2] Shu Kong and Joan Aguilar Mayans, "Automated Snake Game Solvers via AI Search Algorithms".
[3] Bowei Ma, Meng Tang, Jun Zhang, "Exploration of Reinforcement Learning to SNAKE".
[4] Patrick Merrill, "Snake: Artificial Intelligence Controller".
[5] Suraj Narayanan Sasikumar, "Exploration in Feature Space for Reinforcement Learning".
[6] Michel Tokic, "Adaptive ϵ-greedy Exploration in Reinforcement Learning Based on Value Differences".
[7] Risto Miikkulainen, Bobby Bryant, Ryan Cornelius, Igor Karpov, Kenneth Stanley, and Chern Han Yong, "Computational Intelligence in Games".