AI Agents for the Board Game Santorini

Project for Introduction to AI (67842) taught by Professor Jeffrey Rosenschein 2021

Omer Rothenstein Adi Meroz Ravid Kaplan Arthur Oxenhorn

The Rachel and Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem

Omer.Rothenstein@mail.huji.ac.il Adi.Meroz@mail.huji.ac.il
Ravid.Kaplan@mail.huji.ac.il Arthur.Oxenhorn@mail.huji.ac.il

Abstract—Santorini is a deterministic turn-based game, played by two players. In this paper we explore various approaches for building artificial intelligence (AI) agents for the game.

I. Introduction

In this work we set to build Artificial intelligence agents for the game Santorini. We implement Minimax, Alpha-Beta Pruning, Monte Carlo Policy Evaluation and Q-Learning. Our aim is to compare the two former search algorithms with the two latter approximating algorithms.

II. Santorini and How to Play It

The standard version of Santorini is played by two players on a 5×5 board, on which players build towers. Each player has two worker pieces, and they take turns positioning them at the start of the game. Once the workers are positioned, the players take turns to play the game. Each turn is comprised of two phases:

1) Move Phase: The player moves one of the workers to an adjacent tile. The new position cannot be occupied by another worker. The worker can move down any number of levels, but cannot climb more than one level in a single turn. It is impossible to reach level four of a tower.
2) Build Phase: The player builds a tower in a tile adjacent to the worker which has been moved. Building increases the height of the tower in that tile by one level, up to a maximum height of four.

A player that moves a worker to level three wins the game. A worker that has no legal move or build actions loses. Because building towers up to level four prevents move and build actions, and because workers cannot climb more than one level in a single turn, strategies that aim to prevent the enemy from moving or building can be utilized to win the game.

Santorini is thus a perfect-information, deterministic game. There are a finite amount of discrete states, and the game concludes either at a state that has a worker piece at level 3, or one which has no succeeding states. Note that the latter can occur only during the move phase: during build phase, the tile from which a worker was moved is adjacent to the worker's new position, and is at most two levels high.

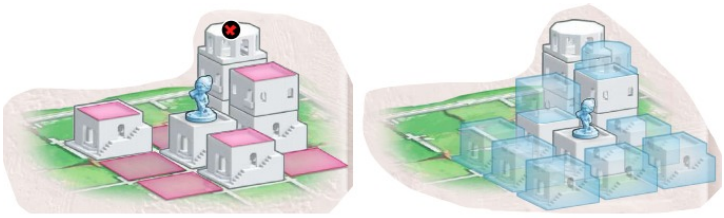

Fig. 1. Tiles available for moving and building.

III. How to Use Our Code

To begin, please install all the required packages specified in 'requirements.txt'. In order to run a game using the agents, one needs only to run the file 'santoriny.py' with python, along with the names of the competing agents and the values of any non-default parameters. A full description of these parameters is provided in the help prompts of the CLI. This utility initializes the agents – including training, where applicable – and running any number of games.

Should one wish to venture into the code itself, the important files to notice are 'game.py', which contains the infrastructure of running a game within the 'GameEngine' type, the types 'MinMax' and 'AlphaBeta' in the file 'minimax_agent.py', and the types 'MonteCarloAgent' and 'QLearningAgent' in 'monte_carlo_agent.py' and 'reinforcement_learning_agent.py' respectively. These types contain the logic for the various implemented algorithms.

IV. Algorithms

The four algorithms we implemented are Minimax, Alpha-Beta Pruning, Monte Carlo Policy Evaluation and Q-Learning. Minimax, and also Alpha-Beta Pruning which is its optimization, is a search algorithm which arrives at the optimal solution, utilizing the fact that Santorini is a perfect-information game. The latter two algorithms approximate the solution – Monte Carlo by sampling the

search space, and Q-Learning by attempting to learn the rewards of performing some action at some state.

A. Minimax and Alpha-Beta Pruning

1) Description: Minimax is a common backtracking algorithm, mostly used in decision making for games of two players – as is Santorini. By performing a depth-first search of the game's search tree, the algorithm can discover the optimal action for the player at a given turn, when playing against another optimal opponent.

Alpha-Beta Pruning is a variation of this algorithm which, instead of running an exhaustive search of all the search tree, limits the amount of branches explored. It does so by utilizing the fact that the algorithm won't choose a smaller value during the "max" phase, and won't choose a larger value during the "min" phase. Thus, while the result of the algorithm doesn't change, runtime is greatly improved.

2) Heuristics: Because the search space is prohibitively large, the algorithms searches along the tree only up to a certain depth. It then utilizes heuristics in order to estimate the quality of the states. The heuristic implemented in the code relies on five metrics.

- Piece Height: Since the goal of the game is to be the first player to climb to height 3, actions in which the player gains height, or the opponent is prevented from gaining height, should be given priority. Thus the value of this metric is calculated as:

$$\begin{aligned} h(player, opponent) = \quad & \text{height}(player_{worker_1}) \\ & +\text{height}(player_{worker_2}) \\ & -\text{height}(opponent_{worker_1}) \\ & -\text{height}(opponent_{worker_2}) \end{aligned}$$

- Tile Values: Some tiles are preferable to others, when it comes to worker placements. For example, a worker placed in a corner will have at most three adjacent tiles, whereas a player placed in the board's center will have eight. Thus, pieces in corners are more susceptible to being blocked by the opponent, and we assign a higher weight for tiles closer to the center of the board.
- Available Adjacent Tiles: As in 'Tile Values', states that provide workers with more freedom of movement should be prioritized. Furthermore, tiles that have an adjacent tile that is one level higher are preferable since they provide a "climbing potential" which can be utilized by the player.
- Distance Between Workers: When a player's worker is sufficiently distant from the opponent's workers, the chances of interference are lower. On the other hand, it won't be able to interfere in the opponents actions and prevent it from winning. Thus, this metric calculates the distances between the opponent and player workers, aiming to maintain a balance between being to close and too far.
- Winning State: Naturally, winning states are assigned a high value.

In order to calculate the value of state, all the components described above are combined. That is, the final value of the heuristic is given by:

$$\begin{aligned} & \alpha_1 \cdot \text{Piece Height} \\ + \quad & \alpha_2 \cdot \text{Tile Values} \\ + \quad & \alpha_3 \cdot \text{Available Adjacent Tiles} \\ + \quad & \alpha_4 \cdot \text{Distance Between Workers} \\ + \quad & \alpha_5 \cdot \text{Winning State} \end{aligned}$$

Where the values of $\alpha_1, \ldots, \alpha_5$ reflect the importance of each component of the heuristic function. This was tested empirically by setting two agents with different α_i values to compete. The values which were found to improve the heuristic value and thus improve the algorithm as a whole are $\alpha_1 = 100, \alpha_2 = 10, \alpha_3 = 20, \alpha_4 = 70, \alpha_5 = 200$.

3) The Search Space: The game has three phases: Setup, Move and Build. Each Move-Build sequence is considered an action, and its result is considered to be a single node in the tree. The placement of the two pieces during the setup phase also constitute an action. Consider then the size of the search space: During the setup phase, the first player has $\frac{25 \cdot 24}{2} = 300$ possible actions to place the two figures. The second player has then $\frac{23 \cdot 22}{2} = 253$ possible actions. The first player now has at most sixteen options to move and eight options to build. Thus, The number of possible states after Setup phase and the first player's Move and Build phase is $300 \cdot 253 \cdot 16 \cdot 8 = 9715200$.

Note that most of the options derive from the various placements of the workers. Should the Setup phase be excluded, searching to depth 2 becomes more feasible: three levels becomes $(16 \cdot 8)^2 = 16384$ nodes, which is more reasonable. By introducing an upper bound of the depth of the search in the Setup phase, the amount of nodes will not "explode" and the runtime of the algorithm will be vastly improved, while allowing deeper exploration of the two main phases of the game.

B. Q-Learning

The Santorini problem can be seen as a Partially Observable Markov Decision Process (POMDP), where the algorithm does not know the reward of picking a certain action from a given state. Q-Learning is a reinforcement learning algorithm that attempts to learn these rewards by running a certain number of games, moving from state to state and updating the values of the Q-Function:

$$Q(s_t, a_t) += \alpha \left(r_t + \gamma \cdot \max_a \{Q(s_{t+1}, a\} - Q(s_t, a_t) \right)$$

When training, besides updating the Q-Function values, the agent also has an ε probability of providing a random action instead of the approximated best action.

The parameters $\alpha, \varepsilon, \gamma$ control the algorithm. α controls the learn rate, as a value of $\alpha = 0$ means the algorithm relies soley on the previous values of the Q-Function and no learning occurs, while a value of $\alpha = 1$ means

the algorithm relies soley on the current value of the Q-Function and it learns only from this last sample. ε controls the rate of exploration, since by exploring actions that are not necessarily the best, the algorithm can escape local minima. Finally, the γ value controls the discount factor – i.e., the importance assigned to future rewards.

In essence, Q-Learning fills a table of size $|S| \times |A|$ – S being the set of states and A the set of actions – where the cell corresponding to a state-action pair is the expected reward of performing the action at the state. In many problems, such as Santorini, the size of S and/or A is too large to make Q-Learning feasible, which is what we wanted to see for ourselves.

C. Monte Carlo Policy Evaluation

As with Q-Learning, Monte Carlo Policy Evaluation is another algorithm which approximates the optimal policy. In a given episode it samples the search space by picking a random action, then simulating the game to completion, aggregating its rewards into a Q-Table. The algorithm repeats this process for N episodes, before picking the action that achieved the most rewards. Our implementation treats wins as rewards, and so the algorithm returns the action with the highest win count once all N episodes have finished running. Given a large N which is order of magnitude larger than the number of available moves, the action chosen by Monte Carlo Policy Evaluation is statistically more likely to lead to a victory.

Since at any given time the algorithm only keeps track of a single game, its complexity is much lower than that of search algorithms such as Minimax – which must keep track of the entire search tree. Since it samples the search space at random, it is also less likely to converge on a local minimum.

V. Performance Analysis

A. Methodology

In order to evaluate our algorithms, we have two main metrics. The first is the win rate, which directly indicates which algorithm comes closer to solving Santorini. The second metric is the algorithm's runtime, which is interesting for practicality's sake. However, it should be noted that the amount of time it takes an algorithm can yield biased results. For example, an algorithm that wins Santorini easily may take a shorter amount of time to run compared to another algorithm due to requiring less steps, not because of each step's time complexity.

To begin, we show Q-Learning's failure to converge. Then we compare Minimax with Alpha-Beta Pruning, providing us with a rational to use only Alpha-Beta Pruning for the remaining comparison with Monte Carlo. Finally we compare Monte Carlo and Alpha-Beta Pruning, as representatives of Approximation and Search algorithms.

B. Q-Learning sucks

As explained theoratically above, it is hard for Q-Learning to work well on domains where the amount of states is very large, as is the case for Santorini. When limiting the training episodes, due to either memory or time constraints, the algorithm experiences severe underfitting. This can be seen in its failure to improve its win rate, even against a random agent, the most naive opponent, as can be seen in Fig. 2.

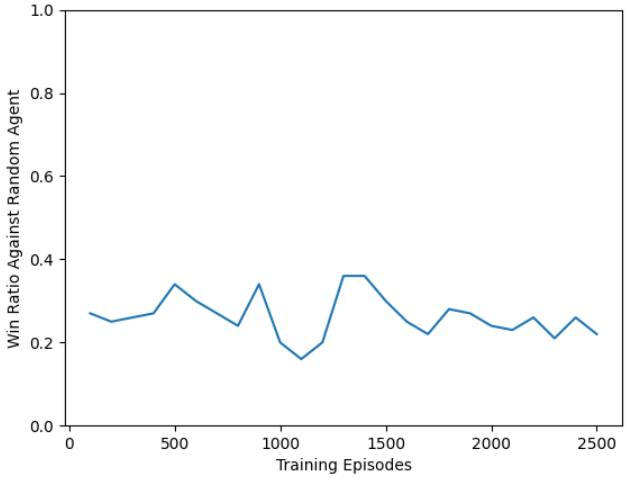

Fig. 2. Win rates of a Q-Learning agent versus a Random agent as a function of training episodes.

Attempts to mitigate this focused on tuning the hyperparameters $\alpha, \varepsilon, \gamma$. While keeping two parameters constant, variations on the third parameter's values were tested. The goal was to differentiate between these values and so to manually find better settings, however all values reacted in much the same way, as can be seen in Fig. 3, Fig. 4, Fig. 5, all of which plot the change in the average rewards aggregated by the agent during 100 episode bulks.

In summary, while Q-Learning is theoretically sufficiently robust to solve Santorini, the amount of resources required to train it past a state of underfitting render this algorithm, as was hypothesised, untenable.

C. Minimax versus Alpha-Beta Pruning

Minimax and Alpha-Beta Pruning yield an identical win ratio against other agents, as was demonstrated in class. However, Alpha-Beta Pruning shows an impressive improvement in performance: when both run using a search depth of 2, the average runtime of Minimax against a Random agent was 5:19 minutes and the average runtime of Alpha-Beta Pruning was a mere 49 seconds. Due to the identical results at a vastly improved runtime, we can test only Alpha-Beta Pruning, trusting its results to be representative of also Minimax's abilities to win the game.

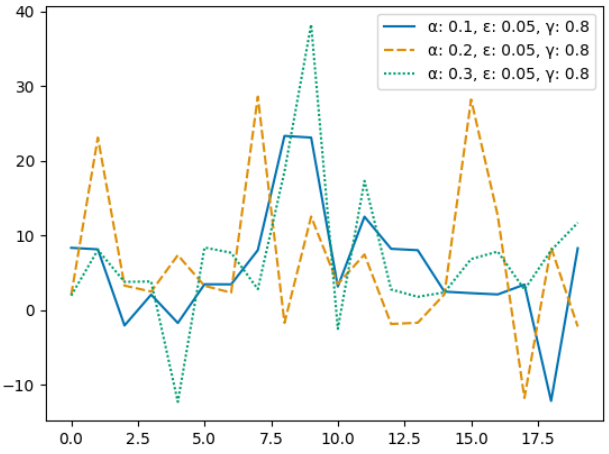

Fig. 3. Average rewards on 100 epsiode bulks, for various α values.

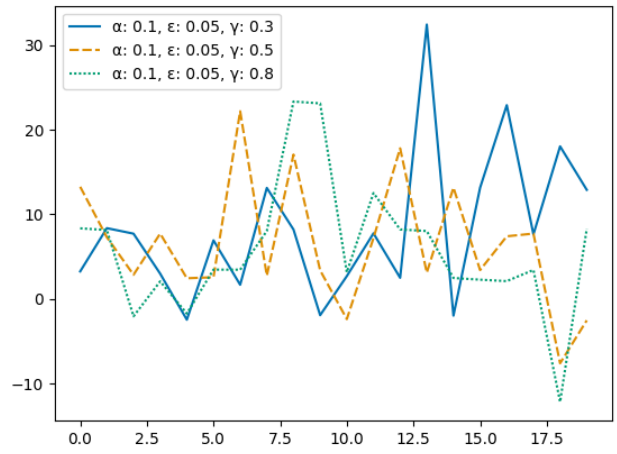

Fig. 5. Average rewards on 100 epsiodes bulks, for various γ values.

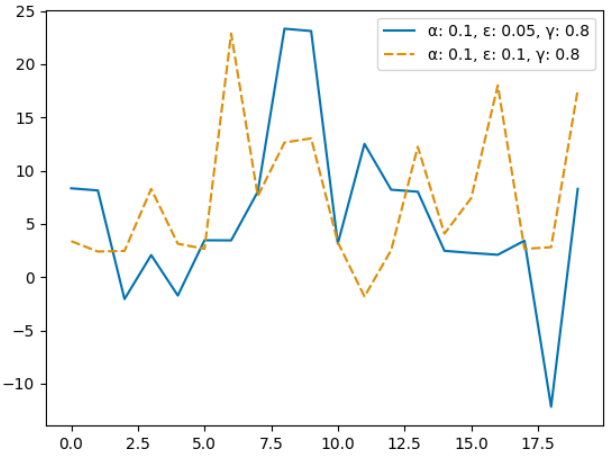

Fig. 4. Average rewards on 100 epsiodes bulks, for various ε values.

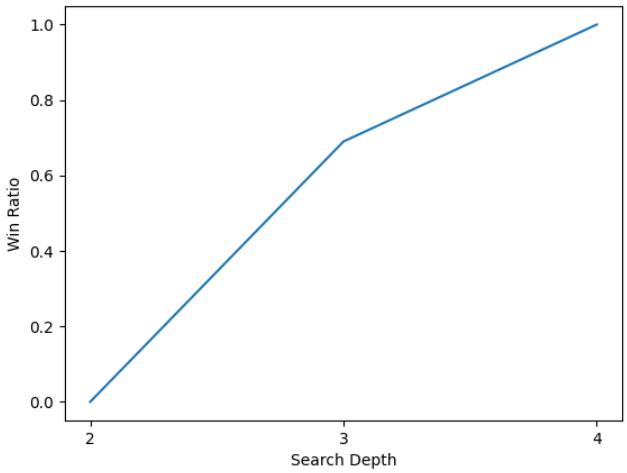

Fig. 6. Alpha-Beta Pruning win ratio against Monte Carlo Policy Evaluation running 500 episodes, by search depth.

D. Alpha-Beta Pruning versus Monte Carlo Policy Evaluation

In terms of runtime, while Alpha-beta Pruning outperforms Minimax it still lags considerably behind Monte Carlo Policy Evaluation. Running with a parameter $N = 500$, the average runtime of a Monte Carlo Policy Evaluation agent against a Random agent was under 3 seconds. Both algorithms easily win against a Random Agent. Running both algorithms against a Random Agent for 100 games, Alpha-Beta Pruning won 100 times, and Monte Carlo Policy Evaluation won 99 times.

More interesting is setting these two algorithms to compete against one another. While keeping Monte Carlo Policy Evaluation's number of simulated episodes at 500, it appears that Alpha-Beta Pruning is incapable of winning with a search depth of 2, and wins consistently when using search depth 4, as can be seen in Fig. 6.

On the other hand, while keeping Alpha-Beta Pruning's search depth at 3, increasing the amount of simulated episodes in the Monte Carlo Policy Evaluation agent increases its success, as can be seen in Fig. 7.

VI. Conclusion

We implemented and compared the performance of four AI algorithms – Minimax, Alpha-Beta Pruning, Monte Carlo Policy Evaluation and Q-Learning – on the board game Santorini. We confirmed that Alpha-Beta Pruning runs significantly faster than Minimax while providing the same win ratio, and we showed that Q-Learning fails to learn meaningful Q-Values for a problem of this size.

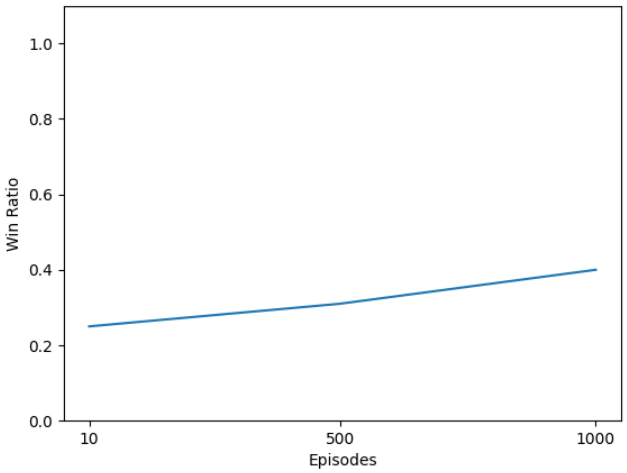

Fig. 7. Monte Carlo Policy Evaluation win rate against Alpha-Beta Pruning with a search depth 2, by episode count.

Having seen that both Alpha-Beta Pruning and Monte Carlo Policy Evaluation manage to win the game consistently, we compared the two algorithms, seeing how their performance changed as we modified their parameters: the search depth for Alpha-Beta Pruning, and the number of episodes for Monte Carlo Policy Evaluation.