Reinforcement Learning 2020 : Assignment 2
Adaptive Sampling

Amin Moradi
me@mamin.io

Martijn Swenne
martijnswenne@hotmail.com

Bartosz Piaskowski
b.piaskowski@umail.leidenuniv.nl

March 27, 2020

Abstract

One of the major aspects of an intelligent agent is its ability to traverse and look ahead for future possible moves. In the previous reports, we examined some of the characteristics of the Alpha-Beta algorithm. This is reports aims to take a deep look into the random search method for traversing a game tree. The Monte Carlo Tree Search algorithm brought on one of the major successes of Reinforcement Learning and surpassed all of its preceding dominant traversing algorithms.

Introduction

In this assignment, we had to make an agent that is based on the Monte Carlo Tree Search (MCTS) and can play the Hex game. This agent needed to be tested against an Alpha-Beta based agent. Then we were asked to perform a hyper-parameter analysis of the two hyper-parameters of MCTS, namely the maximum number of iterations and the exploitation/exploration variable C_p. We partly re-used our user interface, but we added a lot of new improvements too:

- Firstly, we rewrote the Alpha-Beta code into a class in a separate file, and also the PlayGame code was rewritten into a class in a separate file. For the new MCTS agent we made two classes in a separate file, one for the MCTS agent and one for the nodes that are used by the MCTS.
- We also made a utils file, which contains functions that are used by multiple files.
- Lastly, we made a main file, which handles every aspect of the assignment. The text interface walks the user through a series of questions which allows the user to choose which part of the assignment needs to be executed.

1 MCTS Hex

In this section, we are going to explain how we created the MCTS based Hex engine and the problems we encountered with it.

We started by making a class `MCTS`, which contains the variables C_p, `itermax` and `max_time`, which are initialized when creating an MCTS object. The function `makeMove` uses MCTS to choose and play a move on the Hex board. We started with implementing the basics of the MCTS, based on the pseudo code from the book (Plaat (2019)). Instead of using a for-loop over all iterations, we used a while loop which ends when we reach the maximum number of iterations `itermax` or when we exceed the given time limit `max_time`. Inside this loop we do the four MCTS stages:

1. Selection: We call the function `selectPromising` using the root node of the tree. This function uses the UCT formula to determine a path to traverse the tree. Eventually it returns our current node to the variable `curr`.

2. Expansion: If the board is game_over we make no child, so our variable `child` is set to be `curr`. If the board is not game_over we call the function expand on `curr`, which makes a random move and returns the child node to the variable `child`.

3. Playout: we call the function `playout` on `child`, which makes random moves until the board is game_over and returns the result to the variable `result`.

4. Backpropagation: We do a while-loop if `child` is not `None`, where we update the state of `child` and set `child` to the parent of itself. Since the parent of the root is `None`, it will update all nodes until the root.

When the while statement becomes false, because it exceeded `itermax` or `max_time`, it will get the most visited child of the root node and return the board with the chosen move on it. We also clean up after our process, which means we delete our tree before returning the board.

At first we struggled a really long time with the MCTS, because the MCTS agent was not getting the results we expected. The agent always lost to the Alpha-Beta based agent even when the MCTS agent made the first move. We had gone over every stage of the algorithm several times, when we eventually found out it was a problem in our return statement. We were not returning the node with the most visits, but the child which had the best UCT value. Once we changed this to the correct statement, we saw the MCTS agent being somewhat equal to the Alpha-Beta agent on small board sizes.

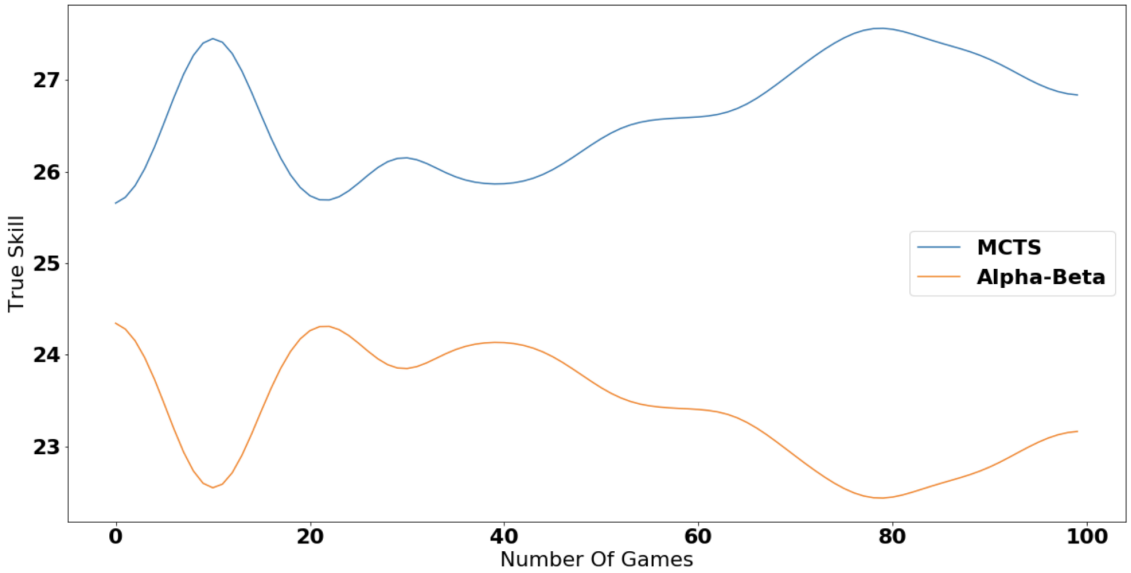

Figure 1

2 Experiment

We implemented the experiments of this section in `main.py`. It makes use of the classes previously described. We first create two bots, an Alpha-Beta agent which uses Iterative Deepening and Transition Tables, and a MCTS agent. For both agents a maximum time is set in which they have to choose a move to make on the board. The MCTS agent is also initialized with a C_p value and a maximum number of iterations. We ask the user to give a board size, after which 10 games will be played, where each round the beginning player is switched. This is because beginning a game has a significant advantage on smaller board sizes and we want realistic results from these experiments.

We also initialize two trueskill[1] ratings, one for each agent. Each round it will print who wins and the update the trueskill ratings. At the end it will print the resulting ratings. We tested the trueskill on a different number of playouts. As seen in Figure 1 we looked at the ratings of the MCTS and Alpha-Beta agents after $1 - 100$ playouts. We can see that around $20 - 30$ playouts the ratings stabilize a bit. On the trueskill webpage it says a 2P free-for-all needs at least 12 games to stabilize. As seen in our results this is not really the case. We assume this is because we change the starting player each playout, which essentially should doubles the number of playouts. And since we estimate the minimal number of playouts for the ratings to stabilize is $20 - 30$, where $2 * 12 = 24$ playouts, seem reasonable. In order to empirically evaluate our implementation from a different angle, we also decided to test it's strength by playing against it manually. Using the same set of hyper-parameters as in the experiment mentioned above, we played around 10 games for smaller board sizes of $3 - 5$ and, in order to have a good overview, 3 games for board sizes of $6 - 7$. The final version of the code was able to perform exceptionally well, leaving no room for the human player to make mistakes, by making consecutive optimal moves.

[1] https://trueskill.org/

Although the Monte Carlo method proved to be working and performing really well, it was still possible to outsmart the agent by utilizing its sporadic mistakes.

3 Tune

Tuning was one of the most power-consuming tasks for this assignment. To perform tuning we create a grid search space to test and evaluate the parameters of our MCTS agent. We then iteratively performed evaluation test between two MCTS agents to traverse our grid space and pick the agent with the highest number of wins. We compared each pair of hyper-parameter 10 times and accumulated the winning results. As we were testing and tuning our agent, we took one step further and implemented the MCTS ensemble method while taking advantage of parallelization ability in MCTS structure. You can run the tuning experiments in the file tune.py. For running the following tasks and experiments below we used the $fs.dslc.liacs.nl$ cluster. To set up the cluster you can follow this link[2].

CN	N	Par-Iter	Win rate
0.4	100	50	9
0.7	400	50	9
1.3	400	50	8
1	700	50	7

Table 1: Hyper-parameters for MCTS agent. Iteration is fixed for parallelization tasks.

3.1 Ensembles and parallelization

Parallelization is one of the ways to increase the performance of MCTS, as it consists of many tree traversings that can be calculated independently. We took advantage of the ensemble strategy to pick the best move for our agent. For each move selection of the MCTS, we parallelize multiple runs from the root and store all the optimal moves that each of the runs generates. After completion of all the parallel tasks, we again perform a selection operation on the highest number of visits. Doing so enables our algorithm to perform significantly better compared to the base model as well as the Alpha-beta algorithm. We ran and tested our methods and fine tuned parameters on a large search space. As mentioned Mirsoleimani, Herik, Plaat, and Vermaseren (2018) and Plaat (2019), by running the model with the ensemble approach we can set the C_p value very small and let each parallel task explore less and exploit in its own region. This fact enables the algorithm to have a balance in the final result as it performs exploration from root and exploitation on lower depths of the search-tree space. As can be seen in the table 1 we showcased 5 of our best values that had the highest win rate.

[2]https://github.com/maminio/dsctl

4 Discussion

In this report, we went through some of the different aspects of the MCTS algorithm and examined a tuned and optimal ensemble of the MCTS agents performing in a game of Hex. We also did a hyper-parameter analysis of the two variables C_p and N (the number of iterations). During our MCTS development, we came across different implementations of the UCT formula. As the UCT value has a very important role in our algorithm we think having a more adaptive UCT value in a paralleled ensemble approach would result in a higher performance and win rate. We did test this approach while setting different values of the UCT for each parallel task. The problem was that having a low UCT value naturally resulted in higher visits and did not necessarily mean a better move. While testing and tuning our algorithm we came across one of the very powerful hyper-parameter optimization python packages, `Hyperopt`, that can increase the efficiency of tuning MCTS. We did not forget about our Alpha-Beta algorithm and boosted its performance by refactoring it for parallelization. The results were interesting as it was performing as good as our MCTS algorithm.

5 Conclusion

Heuristic planning algorithms, such as Alpha-Beta, have advantage over the base model of the MCTS algorithm. Nevertheless, our implementation of MTCS in Hex has proved to be able to go head to head with the previously implemented Alpha-Beta solution, making it a comparable competitor. The reason behind the Monte Carlo method being initially outperformed by it's opponent was due to the fact that its implementation requires more refining and upgrading. Inclusion of certain enhancement procedures such as parallelization, and accurate hyper-parameter adjustments is needed for it to perform in an efficient and satisfying way.

6 How to run the code

The source has been attached to this submission. Please navigate to the root of the project and make sure you have all the required packages listed in **requirements.txt**.

You can run all parts of the assignment by running `python main.py`. The textual user interface will guide you through the different parts of the assignment by asking questions. All the source code has been commented and explained line by line.

References

Mirsoleimani, S. A., Herik, H., Plaat, A., & Vermaseren, J. (2018, 01). Pipeline pattern for parallel mcts. In (p. 614-621). doi: 10.5220/0006656306140621

Plaat, A. (2019). *Learning to play* (1st ed.). Leiden, The Netherlands.