Surakarta AI

Artifical Intelligence Final Project

Mike Greenbaum, Guy Levy, Idan Orzach, Harel Rotem*

August 16, 2020

Abstract

In this project we get deep into the Surakarta game, an ancient Indonesian game for two players, played by the most experienced strategy players. We will try our best to create the ultimate agents to get beyond the human capabilities in this strategy game. In particular, we will create agents based on Minimax and Monte-Carlo Tree Search algorithms. These algorithms based on heuristics we will design based on our own experience with this game. We will compare the performance of our agents and draw conclusions.

Contents

1 Introduction

1.1 What is Surakarta

In this game, two players are given a board and 12 pieces each. The goal is simple - to eat all pieces of the opponent, but the board and moves are quite unique as shown in the figure below.

*Hebrew University students

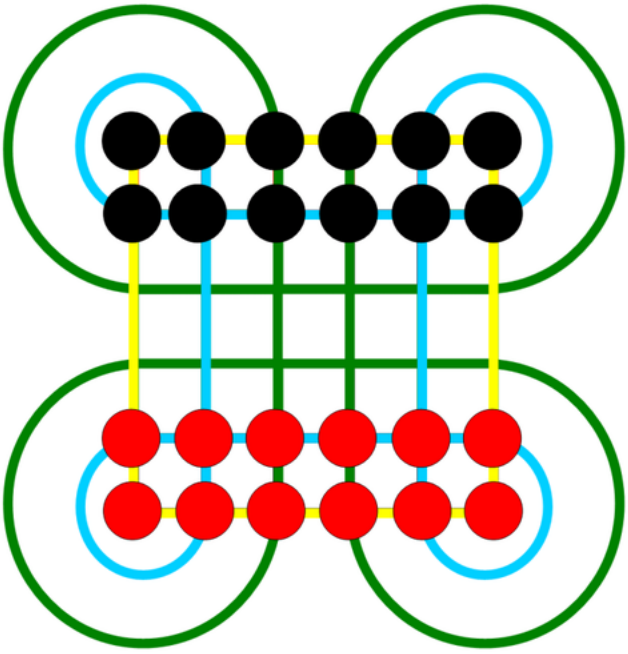


Figure 1: standard Surakarta board with the initial position of the pieces

The rules [2]:

- The players have to decide which one is first.
- In each turn there are two kinds of valid moves:
  - Regular move: one step of a piece to each direction on the grid (forwards, backwards, sideways, or diagonally) - to an **unoccupied point**.
  - Capturing move: the only way to eat an enemy's piece. A capturing move consists of traversing along an inner or outer circuit around at least one of the eight corner loops of the board, followed by landing on an enemy piece, capturing it. Captured pieces are removed from the game.
- End of game: A game is won when a player captures all 12 of the opponent's pieces. If neither side can make headway, the game is ended by agreement and the winner is the player with the greater number of pieces in play.
- There are versions of Surakarta where each turn is limited by time.

1.2 Agent's Quality

To determine that an agent is "better" than other agents we had to come up with scoring methods.

We decided to score an agent in three aspects:

- Game Score: based on how many pieces the agent ate - as the agent eats more - as better the agent.
- Win Rate: the ratio of victories against the other agents.
- Time Performance: the less the agent thinks - the better it is.

As it can be understood, there is a tradeoff between the time and scoring - the more time the agent invests to think - the higher probability the agent will make more sophisticated moves and get a high score. On the other hand, if the time is limited - the agent has to think fast and still get better results. These two aspects are crucial for human players.

It is worth mentioning that the scoring above is a better evaluator for score than the following scoring method: the player's amount of pieces left on the board. The reason is that because our proposed evaluating method actually gives more weight on doing progress moves in the game, as opposed to the second evaluating method where players can avoid eating each other and get a high score.

One point that is worth mentioning - the score and the win rate of an agent are dependent but it is not equivalent.

The second point that is worth mentioning is that there is a difference between the performance of an agent that makes the first move and an agent that waits for the first move - therefore in each measurement of agent1 vs. agent2 we are about to do two kinds of games - one game where the first player is agent1 and second game where the first player is agent2 - the performance of each agent will be the average of both games – it's logically correct to do so because we have an interest to compare agents in general - where the case of being the first player or the second is always possible.

1.3 Complexity

Let's start by giving a lower bound to the number of possible states in a game. First, we can see that without capturing each piece from each color we can get to any unoccupied point on the grid - this way a lower bound to the number of possible states is the number of placing 12 red pieces on the 36 points of the grid and then placing the 12 black pieces on the remaining 24 points of the grids - which is

$$\left(\begin{array}{c} 36 \\ 12 \end{array} \right) \times \left(\begin{array}{c} 24 \\ 12 \end{array} \right) \approx 3 \times 10^{15}$$

states! we can keep going with calculating this lower bound of stated after eating pieces. If checking a state is taking 1 microsecond($= 10^{-6}$[sec]) - than it will take us at least $3 \times 10^{15} \times 10^{-6} \times \frac{1}{60 \times 60} = 8.3 \times 10^{6}$ hours to get over all the states. So far, the game hasn't been solved yet, and in fact, takes place in the Computer Olympiad - therefore it is interesting and challenging to design agents that perform better than the average.

1.4 Relaxation

In fact, the game can be played to infinity - for example, repeated moves that not make progress in the game. This kind of possible games are problematic for scoring - we can't score a game that never ends. Therefore, based on the referenced article [4] we made the following assumption: if passed 80 moves since a piece was last eaten or the game began - the game is over and the winner is the player that ate more pieces of the enemy.

2 The Agents

2.1 Discussion

When you think of a strategy game between two players - the first thing that comes up is the Minimax algorithm and Monte-Carlo Search Tree. Given limited time, there is a tradeoff between these two algorithms - the depth of the Minimax will be lower than the depth of the MCST - while MCST will lose accuracy that is in the Minimax algorithm.

We used a popular version of Monte-Carlo Search Tree shown here [3].

We used the Minimax algorithm with the optimization of alpha-beta pruning which yields the same results as Minimax with smaller runtime most of the time[1].

Both algorithms use heuristics given by the designer.

MCST besides the heuristic and the depth gets an additional parameter - the number of simulations. Due to limitations in time, we wanted to avoid from comparing a lot of agents, therefore using our experience - we decided the number of simulations in our MCST to be $2^{\text{depth}+1}$ - which logically makes sense - we don't want to as the size of the search tree - which is the advantage of MCST over Minimax, but we want to search enough to get in high probability good results - therefore the number of simulations that exponential in the depth logically may be enough.

2.2 Heuristics

Here is the heuristics we used in the MCST and Minimax algorithms.

2.2.1 H1: Attack Heuristic

In this heuristic we want the agent to eat as many pieces of the enemy as it can aggressively - without putting attention to its own state - meaning this agent is ready to sacrifice pieces to eat more of the enemy. The trivial

implementation of this heuristic is to return

$$-\#\text{enemy's pieces}$$

(minus the number of enemy's pieces).

2.2.2 H2: Defense Heuristic

In this heuristic we want the agent to do whatever it can to keep its pieces alive - meaning it can avoid eating the enemy when possible to avoid being eaten. The trivial implementation of this heuristic is to return

$$\#\text{player's pieces}$$

2.2.3 H3: Position Heuristics

The general idea of this kind of heuristic is to guide the player on how to place it's pieces on the board to get, heuristically speaking, a better position. One way to guide the agent using a heuristic to a specific position is to give greater weights for points on the board we want the player to control. We proposed two different heuristics of this kind:

- Control loops' shared points - we give more weight to points that share the inner loop and the outer loop.

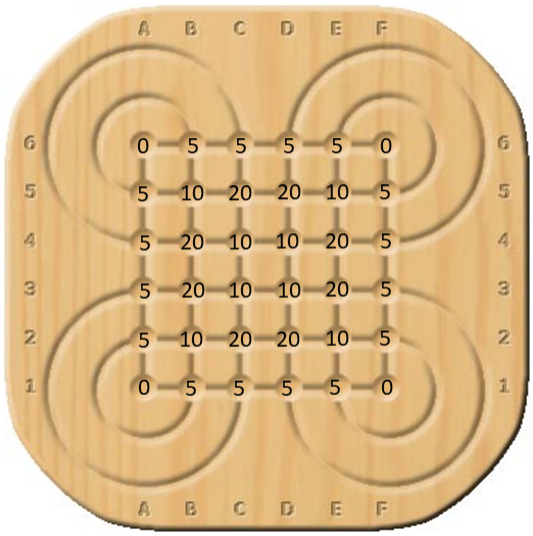


Figure 2: the weights we gave to each point in our first position heuristic

- Control each loop individually - we want the pieces to control all loops and each piece will focus on controlling its loop - this way we avoiding from getting attack from 2 loops simultaneously.

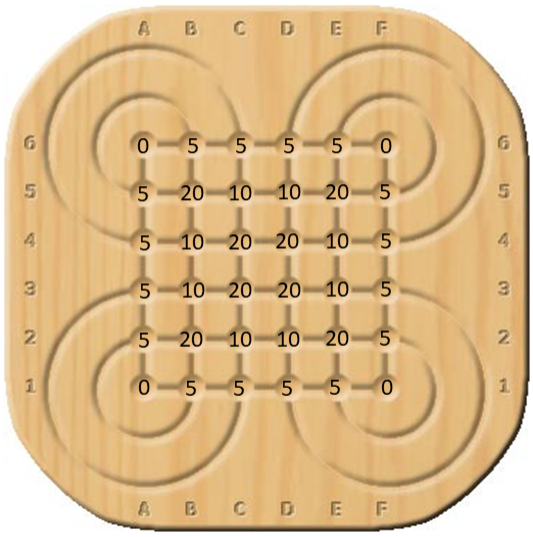


Figure 3: the weights we gave to each point in our second position heuristic

From experiencing both heuristics above, we found the superiority of the first one over the second one – therefore from now on we treat "Position Heuristic" as the first heuristic mentioned above.

2.2.4 H4: Attack-Defense Heuristic

In this heuristic, we combine the ideas of eating more as you can and avoid getting eaten as long as you can - and trying to find a balance between these two ideas. We designed a basic heuristic that combines this ideas in a trivial way: the heuristic returns

$$\text{Attack Heuristic} + \text{Defense Heuristic} = \#\text{player's pieces} - \#\text{enemy's pieces}$$

2.2.5 H5: Complex Attack-Defense-Position Heuristic

In this heuristic we combined the all three ideas mentioned above with the first position heuristic - but this time, instead of just summing up the heuristics we tried few games and gave weights to each heuristic:

$$\text{Attack Heuristic} + \text{Defense Heuristic} + 0.1 \times \text{Position Heuristic}$$

3 Measurements

The next step in our journey of searching for the superior agent of Surakarta is to restrict ourselves to a finite set of agents, and this way to be able to compare each agent with each other.

3.1 What Agents to Measure

First, we restricted ourselves to the following agents we want to measure:

- Monte-Carlo Search Tree:

```
for d in [2,4,6,...,14]:
    for h in [H4, H5]:
        MCST(depth = d, #simulations = 2^(d+1), heuristic = h)
```

- Minimax alpha-beta pruning:

```
for d in [1,2,3,4,5]:
    for h in [H4, H5]:
        Minimax(depth = d, heuristic = h)
```

From now on we will call Minimax the algorithm minimax with the optimization of alpha-beta pruning which reduces the runtime most of the time.

The reason we limited ourselves with the depths above is due to our limitations in time and we wanted MCST with high depths (which is the advantage of MCST over Minimax).

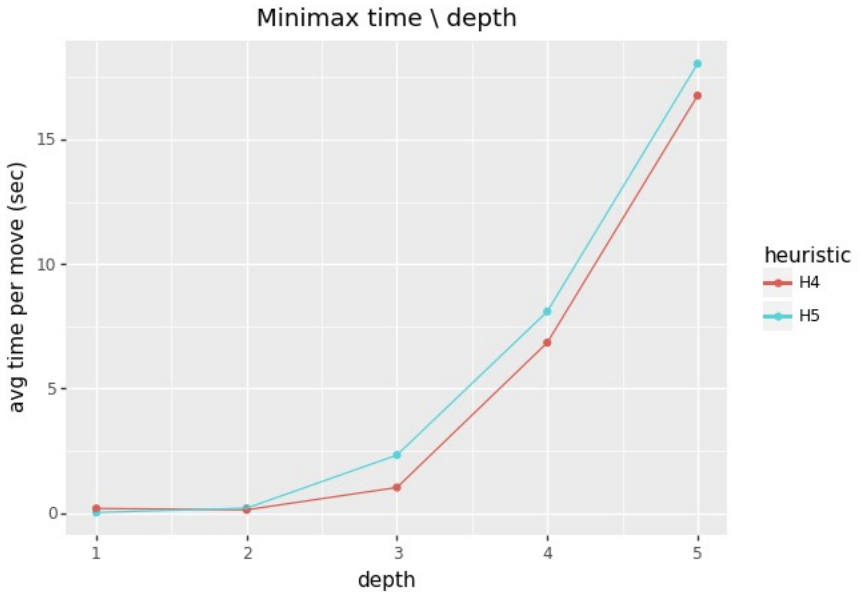
We also limited the heuristics we use - because the first three heuristics H1, H2, H3 are very simple ideas while H4, H5 combine these ideas to a more sophisticated and successful heuristic (from experiencing them).

3.2 Stay with Reasonable Time Agents

Next, we wanted to stay only with the agents that do a move in a reasonable time and check each agent against the random agent with a constant seed.

Why a game against a player with random moves is enough for estimating the time it takes to an agent to make a move? for MCST it's a necessarily good estimation - because for every state the agent will do the same amount of calculations (and the same kind of calculations). The time for the Minimax algorithm depends on the pruning - which means that against the random player the estimation is less representative - therefore we did more than one game (4 games) to hopefully get a better estimation of the time it takes the agent to make a move.

In the following graphs, the average time for an agent to make a move is shown as a factor of the depth for each algorithm. Our interest in see the data this way is to make a check on our algorithms – we know that for MCST - greater depths with a greater amount of simulations will cause the agent to use more time to give a decision on a move. For Minimax we know the greater the depth - the greater the lower bound of operations the agent has to do - so it's not necessarily will be monotonically increasing - but most likely.



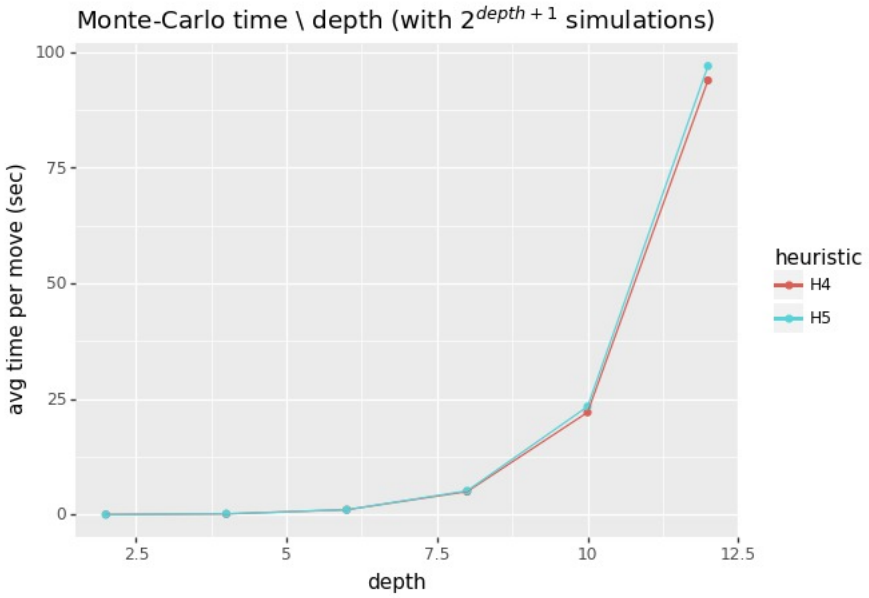


Figure 4: Minimax and MCST average time per move over the depth of the agent.

The measurement of the agents MCST(depth = 14, h in [H4, H5]) was omitted due to making the graph less readable.

After running all of the agents, we found that on average a game takes 20 minutes. To get a representative estimation of the variables we are looking for (scoring, win rate) for each estimation we need to repeat each game as much as we can (*The Weak Law of Large Numbers*). Assuming we have n agents with an average time per game t and m is the number of repetitions we want - we get

$$\binom{n}{2} \times m \times t\,[\text{minutes}]$$

we have $n = 24$ and $t = 20$ and for $m = 4$ we get measurements that will take $15\frac{1}{3}$ days! Due to our limited time - we choose to focus on the following 4 agents.

- Monte-Carlo Search Tree(depth = 8, #simulations = 512, heuristic = H4)
- Monte-Carlo Search Tree(depth = 8, #simulations = 512, heuristic = H5)
- Minimax(depth = 3, heuristic = H4)
- Minimax(depth = 3, heuristic = H5)

We chose those 4 agents after taking a cutoff of 5 seconds per move and taking the most complex agents from the remaining agents (complexity based on depth).

3.3 Monte-Carlo Search Tree Agents

Before moving on to the main measurements we wanted to be sure that Monte-Carlo with smaller depth (and the number of simulations determined by $2^{\text{depth}+1}$) is "weaker" than Monte-Carlo with higher depth, given for both agents a reasonable heuristic.

We did the following measurements: for each heuristic in [H4, H5] we check MCST agents from depth [4, 6, 8] against each other (each game is between agents with the same heuristic) and repeat games 10 times (5 games where the agent do the first move, 5 games where the agent wait for the first move) and take the average on the score and win rate on the matches.

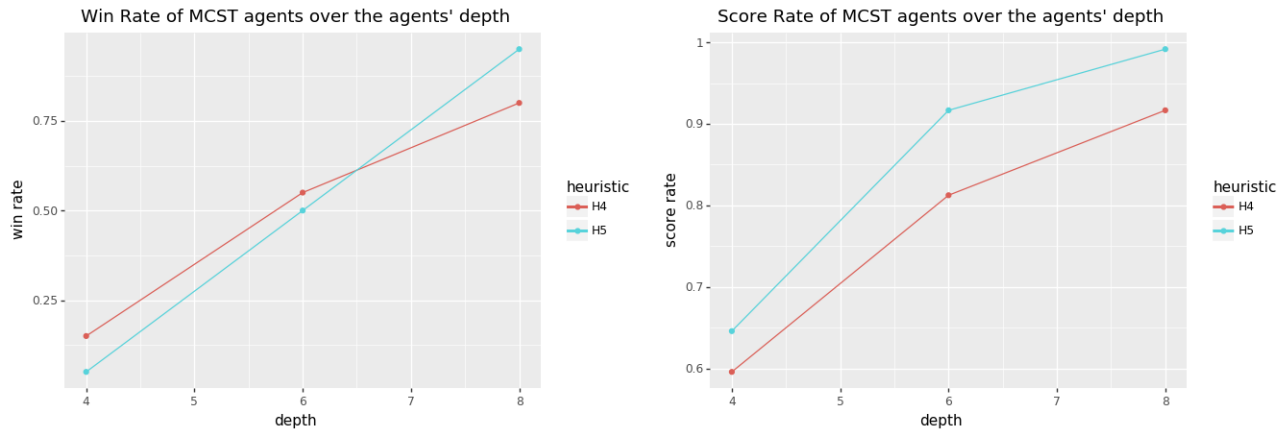


Figure 5: the win rate and score rate of MCST agents with [4, 6, 8] depths with heuristics [H4, H5]

As expressed by the graphs, in our scale of depths our assumption that greater depth on the same heuristic results better agents turns to be true.

3.4 Minimax Agents

We did the same check on the Minimax agents: for each heuristic, we check if for greater depth the agent gives better results.

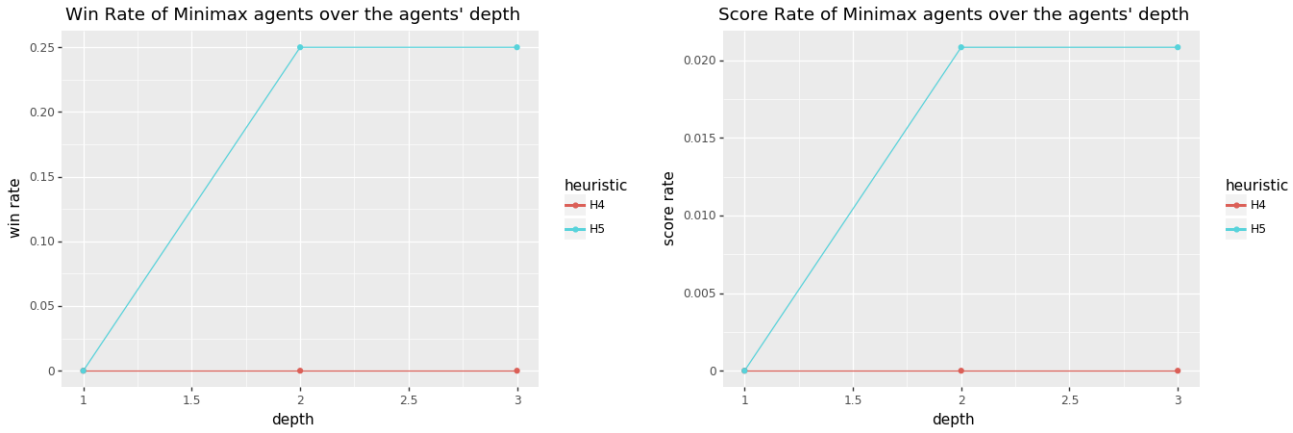


Figure 6: the win rate and score rate of Minimax agents with [1, 2, 3] depths with heuristics [H4, H5]

Here there is an interesting phenomenon of minimax where with heuristic H4 the agents do not eat at all and with H5 the agents barely eat. An explanation for this phenomenon is that Minimax is choosing a move by considering all cases in the search tree till the given depth – it is probable that this way the agents find that eating will put them in risk that doesn't worth the eating. We can also see in the graphs that the Position Heuristic in H5 makes a huge difference, comparing to H4, probably by "push" the agents to the strategic positions where eating may be more worthy sometimes.

3.5 Measure the Agents' Quality

As mentioned in the Introduction section three variables define the quality of an agent – the score it averagely gets, the win rate, and the time it averagely takes to think. We already took a cutoff of a reasonable time - and therefore we will focus on the score and win rate variables.

With respect to our limited time and also our desire for representative estimations we decided to repeat each game in our measurements 10 times (5 games where the agent does the first move and 5 games where the player wait for the first move).

We performed the following measurements: lets mark AGENTS = [MCST(8, 512, H4), MCST(8, 512, H5), MINIMAX(3, H4), MINIMAX(3, H5)]

```
for i = 0 to len(AGENTS) - 1:
    for j = i + 1 to len(AGENTS) - 1:
        let agent1 = AGENTS[i]
        let agent2 = AGENTS[j]
        repeat 5 times:
            do game(first = agent1, second = agent2)
        repeat 5 times:
            do game(first = agent2, second = agent1)
        keep the win rate of agent1 (over all 10 games) and the average score over all games of agent1
```

3.6 Results

By doing the above measurements we got the following results:

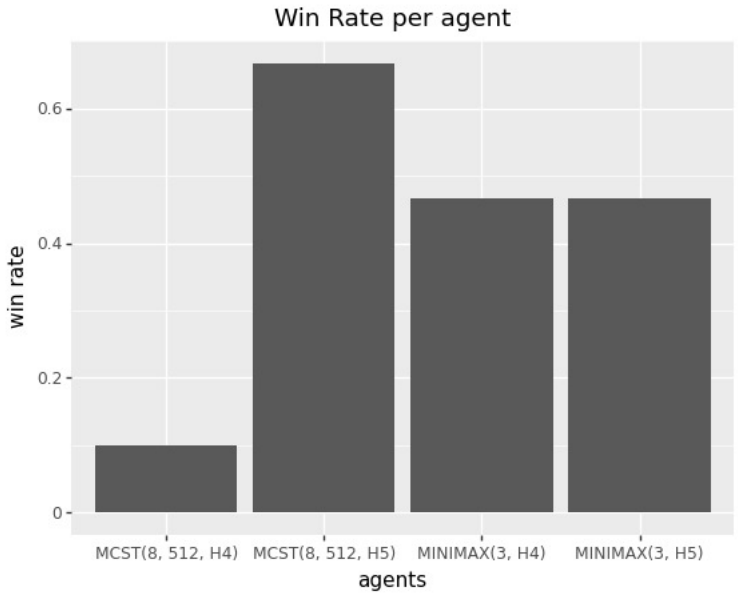


Figure 7: the measured win rate of each agent.

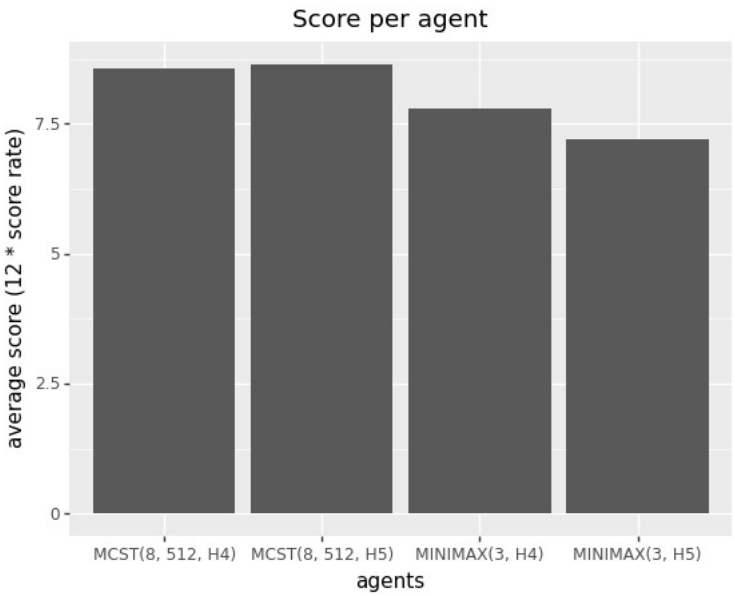


Figure 8: the measured score of each agent.

3.7 Discussion

From looking at the results we can see that indeed the win rate and the scoring rate are not equivalent, although the connection between these two rates. The agent MCST(8, 512, H4) got a higher score rate than MINIMAX(3, H4) and MINIMAX(3, H5) - while the win rate of this agent is much lower than any other agent rate. This phenomenon can be explained by for example the following behavior: while losing - eating a lot of pieces (closer to the winner) - this way, by losing to the agent with the highest score rate you can increase your score rate extremely without winning a lot of games and be with respectively high score rate.

We can see from the results that the Position Heuristic with a weight of 0.1 was extremely helpful (this is the difference between H4 and H5) to MCST(8, 512, H5) - this tiny difference was the difference between the smallest winning rate and the highest winning rate.

One additional point worth discussion is that our Monte-Carlo Search Tree agents are looking deeper in the Search Tree than our Minimax agents - and we see that it doesn't guarantee wise moves - it depends on "luck" (the search over the search tree include random moves) and good heuristic.

It turns out that in general MINIMAX(3, H4) agent performs better than MINIMAX(3, H5) (both agents have the same win rate and the first agent has a scoring rate better than the second agent) – on the other hand, H5 shows extreme superiority in the MCST algorithm. This difference is quite fascinating due to the fact that MCST and Minimax are quite similar algorithms.

4 Conclusions and Further Research

Our main goal in this project was to find the "perfect" agent to play Surakarta with. An agent's performance is measured by different kinds of variables - scoring, win rate, time, etc. Due to time limitations, we bounded ourselves from measuring agents that take more time to think and probably would do better moves. In the context of limiting our agents' time, we measured and compares the quality of agents by the score and win rate. We found that although the connection between these two variables, there is no guarantee that a higher score rate implies a higher win rate. We also saw that looking deeper in a search tree using MCST has no guarantee to do better than Minimax with small depth - and there is a huge impact on a better heuristic.

Eventually, our best agent was MCST(8,512, H5) - with a win rate of $\frac{2}{3}$. Out measurements is only over 4 agents in the end - therefore they are less representative in the scales of the score and win rates. It is interesting to find out these rates against more agents. Also interesting to investigate agents that think more time.

It is worth mentioning that after measuring and finding the best agent, we did a tournament against it and suffered a crushing defeat.

There are a few ways to expand our project:

- comparing more agents with different parameters.
- comparing agents with the respect to the runtime as a quality measure
- comparing agents based on more different and complex heuristics
  - for example, heuristics that give weight to the position of the enemy.
- comparing with agents based on q-learning (what we avoided from doing due to the large state space)

5 How to Run the Code

First make sure the packages `pygame` and `numpy` is installed on your computer. Next, type in the command line:

```
python3 Main.py
```

and follow the instructions on the screen.

References

[1] Alpha beta pruning. *Wikipedia*, August 2020.

[2] Surakarta (game). *Wikipedia*, May 2020.

[3] Rahul Roy. Ml | monte carlo tree search (mcts). January 2019.

[4] Guoyzo Zou and Chenming Wu. A heuristic monte carlo tree search method for surakarta chess. 2016.