Combining Evolutionary Algorithms with Monte Carlo Tree Search for Game AI

(Łączenie Algorytmów Ewolucyjnych z
Monte Carlo Tree Search dla sztucznej inteligencji w grach)

Dominik Kowalczyk Bartosz Stefaniak

Praca inżynierska

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

19 lutego 2022

Abstract

Rolling Horizon Evolutionary Algorithm and Monte Carlo Tree Search are two famous and widely used algorithms. They are often used for playing optimization games. Ways in which they explore space of possible game states are very different from each other. It is not obvious if they could be combined and whether such combination would make sense. In this thesis, we will present our method to combine these two algorithms and results using selected games as the example.

Rolling Horizon Evolutionary Algorithm i Monte Carlo Tree Search to dwa znane i powszechnie używane algorytmy. Często znajdują zastosowanie do grania w gry optymalizacyjne. Sposoby w jakie przeszukują przestrzeń możliwych stanów gry znacznie się od siebie rożnią. Nie jest oczywistym w jakis sposób możnaby używać ich jednocześnie i czy takie połącznie miałoby sens. W tej pracy prezentujemy nasz sposób połączenia tych algorytmów i osiągnięte wyniki na przykładzie wybranych gier.

Contents

Chapter 1

Introduction

First attempts to use Artificial Intelligence (AI) to play games trace back to 1950s when Arthur Samuel wrote a series of programs that played checkers. In 1997 Deep Blue became the first program to beat world champion at chess - Garry Kasparov. Since then chess AI became so strong that no human can defeat it. Although beating strongest humans in specific games is great achievement, in the meantime researchers started looking into problem of being able to play any game. To pursue that interest new research area was born: General Game Playing (GGP), formally established by Stanford University in 2005 (Love et al., 2008; Genesereth et al., 2005). In 2013 another research area was founded: General video game playing (GVGP) which focuses on playing video games (Levine et al., 2013).

Two popular methods for developing Game AI are Monte Carlo Tree Search (MCTS) and Rolling Horizon Evolutionary Algorithm (RHEA). They are used to play many different games. Often in a game where MCTS works well, RHEA performs badly and vice versa. Both of them have different strengths and weaknesses, therefore method that could combine advantages of these algorithms seems promising. There were some attempts to make hybridization for MCTS with RHEA (Horn et al., 2016). We will propose and test some approaches to combine these two algorithms. We chose two single-player optimization games in order to test them.

Chapter 2

Background

2.1 Algorithms

2.1.1 RHEA

Rolling Horizon Evolutionary Algorithm (RHEA) is approach that uses evolutionary algorithms to play games that was proposed by (Perez et al., 2013). It uses sequences of moves as individuals that build up population. Initially, each individual in the population receives random action for each gene. A new population is then generated from the population. Firstly it promotes E best individuals to the next population, through elitism. Then it generates $P - E$ new individuals, where P is size of the population. Each individual is the product of crossover between two individuals from the previous population, selected through some method (common choice is a tournament, where T individuals are selected from population and the best one out of them is used), and then mutated. Individual's fitness value is usually a heuristic evaluation of the game state after the moves are played. At the end of the turn, RHEA chooses the first action in the individual with the highest fitness as move to play. After that it removes the first action from each individual and adds new one at the end. The name "Rolling Horizon" comes from this behaviour.

2.1.2 MCTS

Monte Carlo Tree Search (MCTS) is tree search algorithm that builds game tree by sampling the search space. It has been applied to a large variety of games and other tasks with great success (Browne et al., 2012). Moreover, it has been used in General Game Playing (e.g., (Bjornsson and Finnsson, 2009)) winning all the AAAI GGP competitions since 2007.

It typically consists of four phases sequence: Selection, Expansion, Rollout, and Backpropagation. It repeats the following four phases until computation time runs

out.

During the *Selection* phase, a *Tree Policy* is used to traverse the tree until a leaf node is reached. Usually policy is based on game scores and numbers of visits. A very common policy is UCB1 (Kocsis and Szepesvári, 2006). This policy balances between exploitation and exploration through the value of parameter C.

During the *Expansion* phase, to the previously selected node, which was a leaf, new successors nodes with empty statistics are added. Then, the *Rollout* phase starts. A *Rollout Policy* is used to play out the game until reaching the end of the game or predefined depth. This policy can be uniformly random, but could also use heuristics (Schadd et al., 2012). Finally the *Backpropagation* phase updates the values of all nodes traversed during the *Selection* phase with the result of state reached at the end of the *Rollout* phase.

In order to adapt MCTS to single-player games, in addition to values stored by MCTS's nodes we also keep the highest score ever seen in that node's subtree, similiar to (Schadd et al., 2012). When choosing next move, we choose a child with the highest *best score*.

2.1.3 MCTS RHEA hybrids

There were some attempts to make hybridization for MCTS with RHEA.

Some of them were proposed by (Horn et al., 2016) to be used for General Video Game AI. The main ones are:

RHEA, then MCTS for alternative actions: EAaltActions – after running the RHEA, MCTS is ran, but it is not allowed to make the first move chosen by the RHEA. Then best moves found by both algorithms are compared and better one is chosen.

RHEA with rollouts: EAroll – it takes over the RHEA algorithm and extends evaluation with *rollouts*. After all moves from individual are simulated, instead of evaluating it, a predefined number of rollouts with parametrized length is performed, and the fitness is the average of the RHEA part and the MCTS part.

In addition, several more variants were proposed, each basing on *EAroll*. *EAroll* turned out to be the best, compared to other versions and basic RHEA or MCTS.

2.2 Games

2.2.1 Search Race

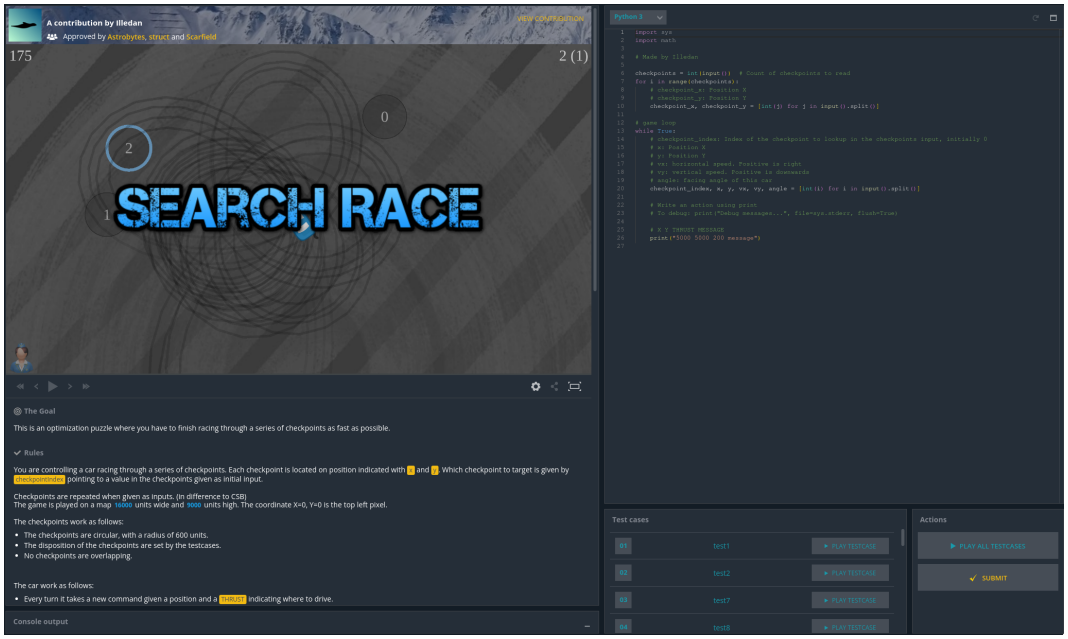

Figure 2.1: Playing Search Race in the CodinGame online IDE. Top left – game visualization, bottom left – information about current turn and debug, top right – agent source code, bottom right – test cases for testing.

Search Race[1] is a programming game available on the CodinGame platform (Fig. 2.2). Everyone can write his agent using an online CodinGame IDE (Fig. 2.1) and test it on a public arena that currently contains about 1,000 players. Most successfully approaches are: genetic algorithms and deep learning.

Game Rules. *Search Race* is single player optimization game. The game is played on map 13,000 units wide and 9,000 units high. Player control a pod racing through a series of checkpoints. The checkpoints are circular, with a radius of 600 units. Positions and order of checkpoints is fixed for test case. To enter a checkpoint, the center of a pod must be within 600 units of the checkpoint center. Every turn player can change the thrust of the pod, and change the angle that the pod is heading by a maximum of 18 degrees.

Finally the goal of the game is to finish racing through a series of checkpoints as fast as possible. If a player use more than 600 rounds, they lose.

At CodinGame there are 50 test cases, but many of them are similar to each other. We removed the ones that were too similar or too easy. Then for efficiency reasons, we end up picking 15 of them.

[1] `https://www.codingame.com/ide/puzzle/search-race`

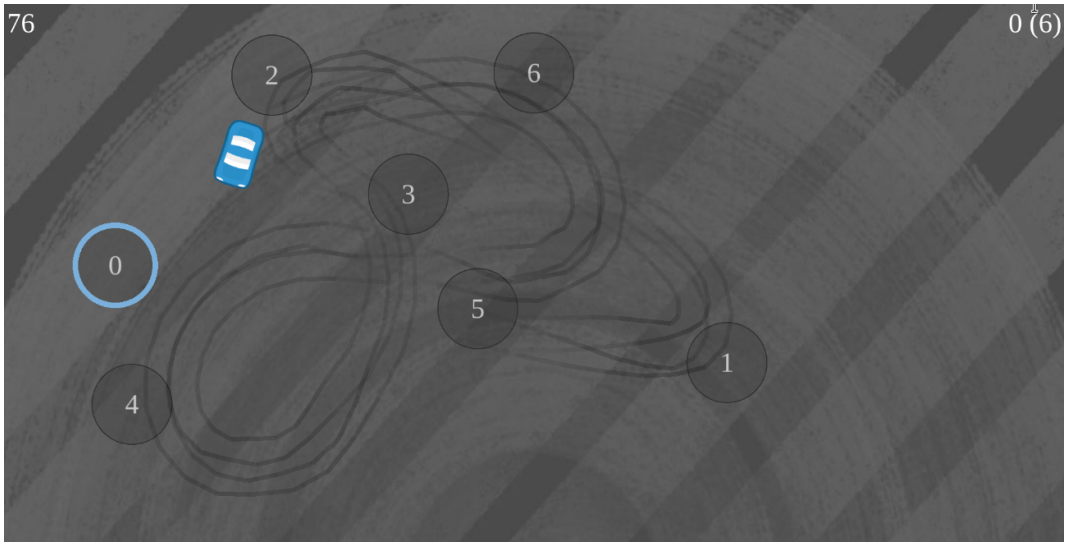

Figure 2.2: A visualization of the Search Race game.

2.2.2 SameGame

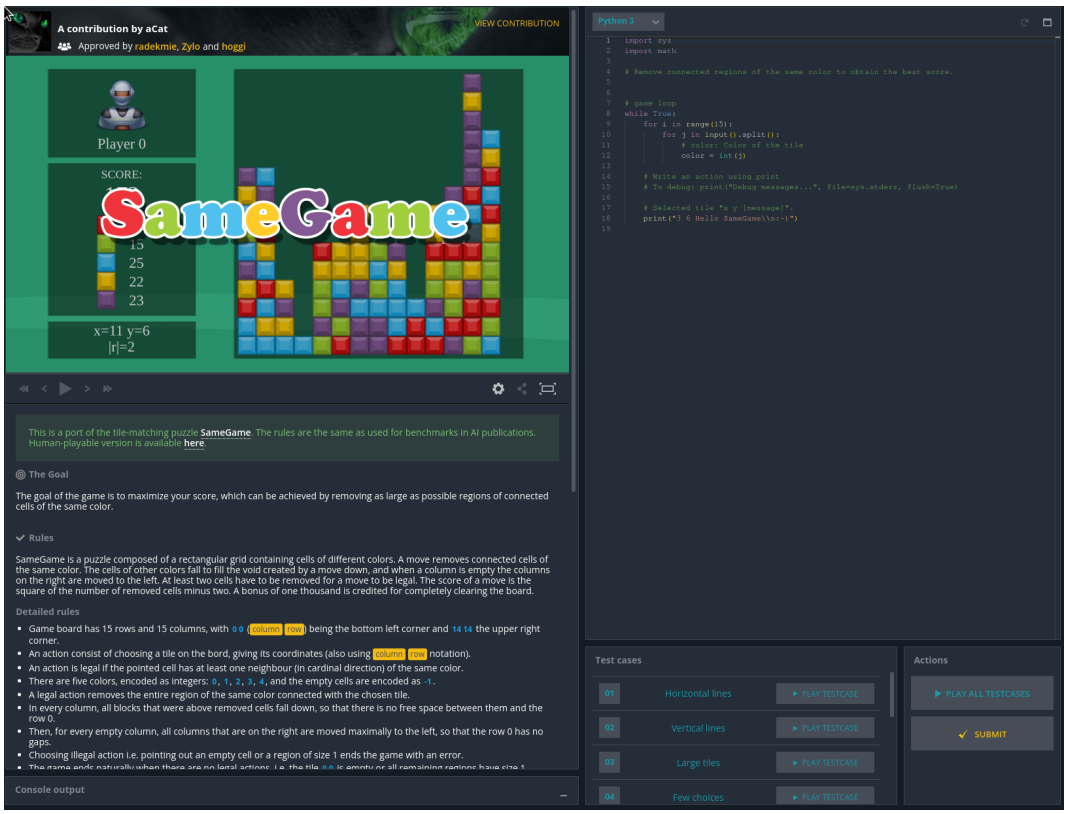

Figure 2.3: Playing SameGame in t he CodinGame online IDE. Top left – game visualization, bottom left – information about current turn and debug, top right – agent source code, bottom right – test cases for testing.

Game Rules. *SameGame*[2] is single player game played on rectangular board made of colorful squares. We say that the squares are connected if they share an edge. On each turn player can select any group that consist of at least two connected squares and remove it from board. After that if there are any squares that don't have any other square or board border underneath they are moved down, afterward if any column is empty, all columns to the right of it are moved left. In our case, the game is played on a board 15 units wide and high, after each turn player receives points equal to $(n-2)^2$, where n is size of removed group, additionally if after the move board is empty player receives bonus 1000 points. Game ends when no move is available or board is empty, goal of the player is to maximize earned points.

For SameGame we will be using 20 standard boards used in many papers (Schadd et al., 2012; Negrevergne and Cazenave, 2017).

[2] `https://www.codingame.com/ide/puzzle/samegame`

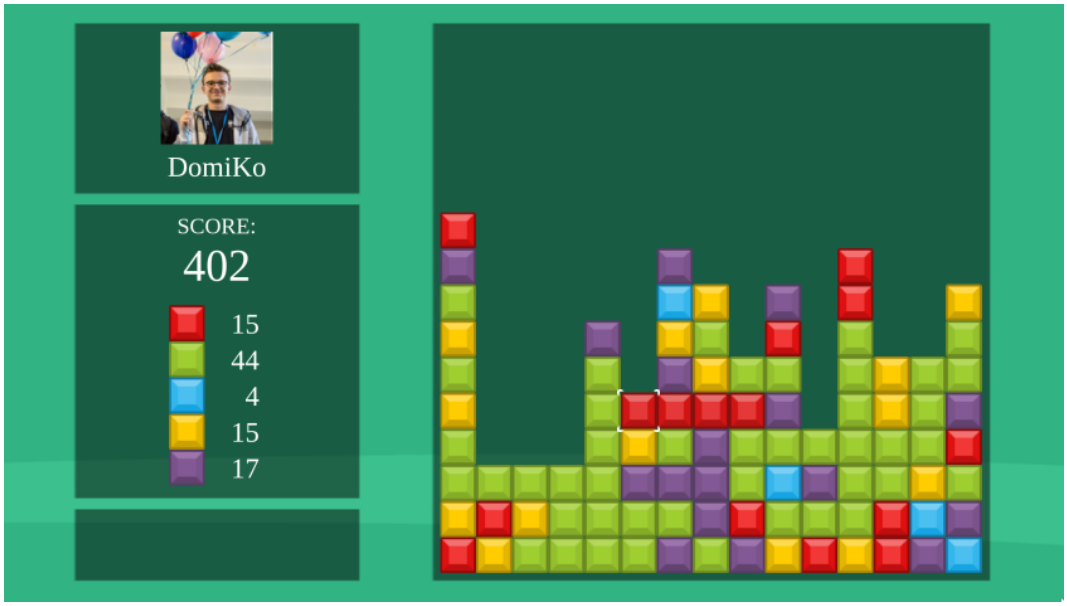

Figure 2.4: A visualization of the SameGame game.

Chapter 3

Combining RHEA with MCTS

3.1 General idea of combining

The idea is to supply each algorithm with knowledge acquired by the other one. In MCTS, we balance its behaviour between exploration and exploitation through the value of parameter C. We thought that we could guide the search even more by applying results found by RHEA. This can be done by pruning certain areas of the tree that are considered not promising or adding paths made of individuals. In RHEA, we guide the search by modifying population that is used in further iterations.

Our idea is to design a method, so in any moment we could switch the currently running algorithm between RHEA and MCTS. But we are going to switch them exactly once per turn. We distinguish two orders of running them each turn: *RHEA-MCTS* and *MCTS-RHEA*. The first algorithm starts computation at each turn, and the second one decides which move to make.

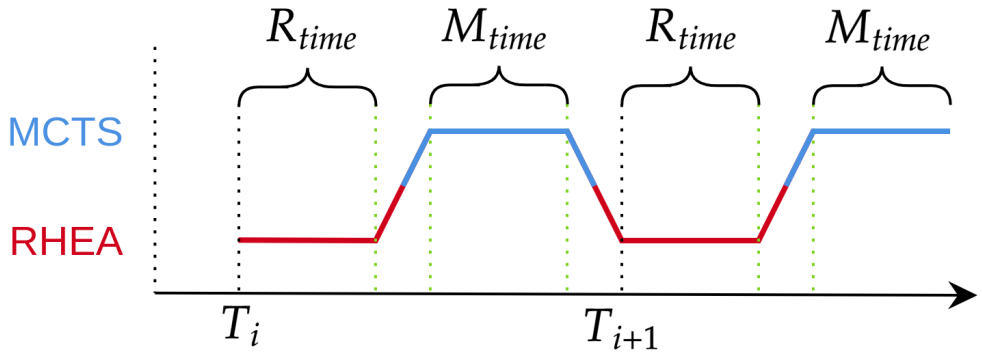

Figure 3.1: Example timeline for turns i and $i+1$ in *RHEA-MCTS*. R_{time} describes computation time assigned to RHEA, and M_{time} describes computation time assigned to MCTS.

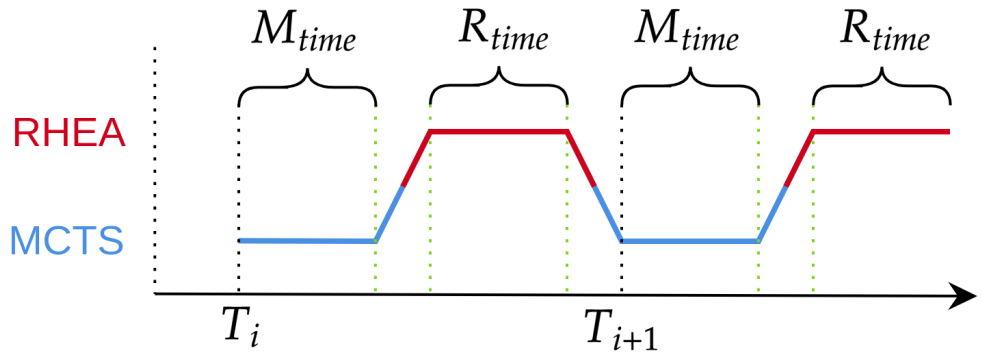

Figure 3.2: Example timeline for turns i and $i+1$ in *MCTS-RHEA*. R_{time} describes computation time assigned to RHEA, and M_{time} describes computation time assigned to MCTS.

In the next sections, we will explain exactly how we make transitions between RHEA and MCTS.

3.2 Adjusting games to be used by both RHEA and MCTS

RHEA can be used regardless of whether player's move is represented discretely or continuously as opposed to MCTS that requires them to be discrete, because of representation of move as edge in tree. In order to use MCTS in *Search Race* we had to discretize available moves, so we represent set of available moves as $SRM = [-18, -9, 0, 9, 18] \times [0, 200]$, where the first element is angle change, and the second one is chosen thrust.

The moves in *SameGame* are represented as a pair (row, col), marking connected group of squares' position, there is only one pair representing single group. The moves are already discrete, but unlike *Search Race* moves, not every move is legal in current state of the game, which raises problems. Individuals representing sequences of moves are changed during crossover and mutation phase of RHEA algorithm. To make them represent legal sequences of moves, after new one is created, during its evaluation if at some point there is chosen move that is currently not legal we find closest group of connected squares that can be removed and set our move to this group.

3.3 Transition from RHEA to MCTS

3.3.1 Pruning

Population from RHEA could be used to build new MCTS tree each turn. We can look at each individual as a path and make the tree from them. We allow MCTS to only create new nodes at the tree levels deeper than length of the individual. We can additionally supply nodes with information about scores calculated by RHEA and a certain number of visits. But that way we lose information calculated in previous turn by MCTS, which is the main reason why MCTS works. What we described can be seen as pruning tree to a certain depth, as only at deeper levels of the tree we would allow MCTS to expand the tree. In order to keep acquired knowledge, we decided to use RHEA's results to prune the MCTS tree and keep information gathered in nodes that stayed in the tree. Pruning too deep could be a problem. To prevent that, we simply cut each individual from RHEA at a certain depth (rather small) and then prune the tree (Fig. 3.3).

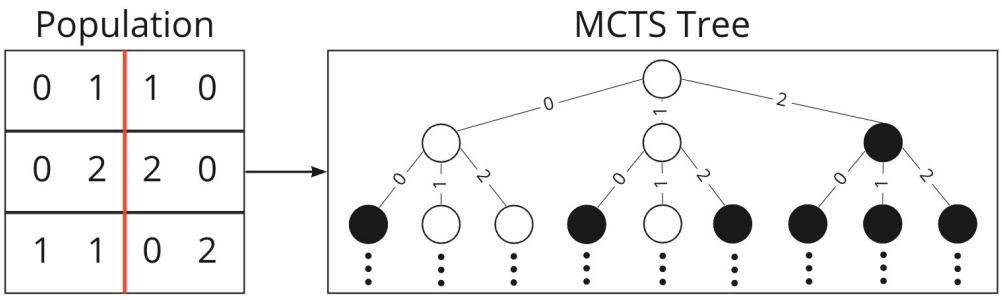

Figure 3.3: Example of pruning two first layers of MCTS, when RHEA individuals have length 4. Legal actions are 0, 1 and 2. White nodes are kept in tree, black ones are removed.

3.3.2 Carrying more information

Although pruning could be useful, often sequences of moves represented by RHEA's individuals are longer than depth of the tree and lead to promising results, so additionally we insert those paths into the tree. To have a little more impact on the tree's future traversal, we can consider each individual from RHEA's population as a path visited N times. That way score of that individual will have greater impact on the average score of nodes on that path.

3.4 Transition from MCTS to RHEA

After the phase of running MCTS, we need to decide how RHEA's population will look the next time it will be used. The population should consist of individuals with high scores, but if each one is similar to the other ones, then RHEA could converge quickly. In order to prevent that, we have to maintain two things: diversity of population and individuals with high scores.

Firstly, let us look at the RHEA algorithm, how it maintains individuals with high scores. It promotes some individuals through elitism to the next generation. So we will take the one, with the highest score from the MCTS tree, and put it into the RHEA population, which is similar to elitism with size 1. We do that by going down the tree starting from root, choosing each time edge that leads to node with highest *max-score* ever seen. We keep going down the tree until we are in the leaf node or the length of the individual is enough for RHEA.

Now we need to generate more individuals with roughly high scores, but not too similar to the individual with the highest score. We can sample individual by traversing the tree from the root node. We go down the tree unless we are in the leaf node or the length of the individual is enough for RHEA. When going down we choose edge according to probability distribution determined by the softmax function applied to values calculated as linear combination of *mean-score* and *max-score* of current node's children. In our case, it will be the average of these two values.

We have to consider the fact that the length of individuals in RHEA could be much longer than the depth of the MCTS tree, so individuals sampled from the tree could be too short. We can add random moves to them, to match the length of individuals in the RHEA. Even though we have some first moves from the MCTS tree, we still can have solution with score much lower than *max-score* in that leaf node, because MCTS could earn all the points in the rollout. Since rollouts could be random, scores of two rollouts may vary a lot. Thus adding some random moves to that beginning could lead us to the solution with low score. We can think about this that way: MCTS knows which first moves are good, so it will be exploiting them more, but it is not able to transfer that knowledge to RHEA.

To address that problem, and do not lose information about good solutions in transition from MCTS to RHEA, we will save some rollouts. That way we can always create individuals that do not have to be completed with any random moves to match the length of the RHEA's individuals, and they are guaranteed to have proper scores.

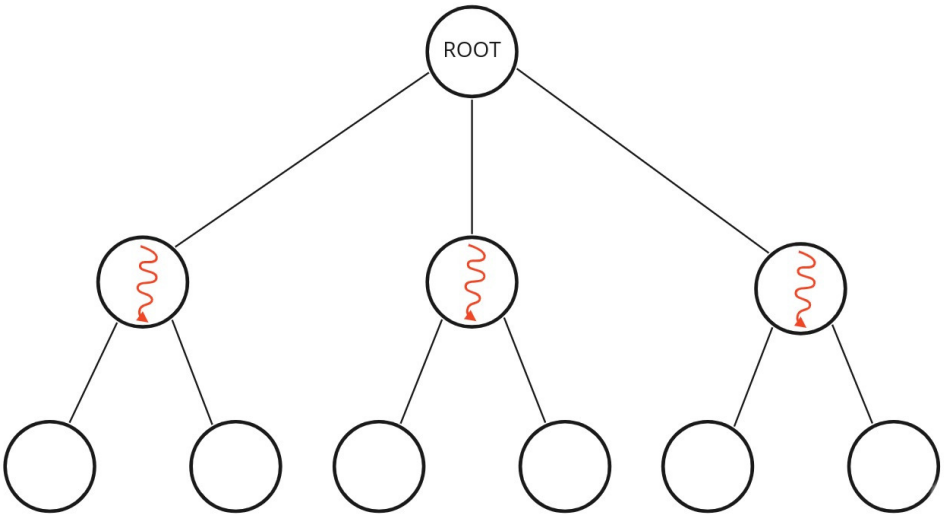

Figure 3.4: Rollouts are stored in the children of the root.

Many MCTS variations save the best rollout, and we will do something similar, but we will save a bit more. For every possible next move, we want to store the best rollout that begins with that move (Fig. 3.4).

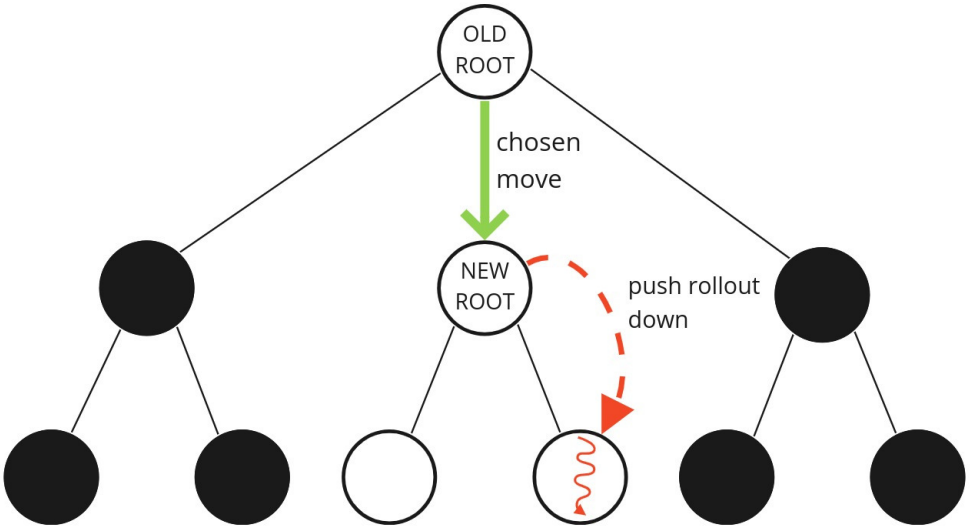

Figure 3.5: Rollout is pushed down when the tree's root is being changed after game turn.

This approach works when the turn order is *MCTS-RHEA* however, there is a problem with such solution. When MCTS is deciding about the chosen move (turn order is *RHEA-MCTS*), we will have only one stored rollout (Fig. 3.5).

To solve that, we will store rollouts at depth two, when MCTS is the second algorithm. That way for each first move in the next turn we can make individual for RHEA's population.

Now we can describe how exactly we will create RHEA's population. Firstly we will create one individual from the path from the tree with the highest score. Then for each possible first move we create one individual from saved rollout. Remaining ones are sampled from the tree.

Chapter 4

Experiments

4.1 Technical setup

Experiments were performed on PCs with Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz processors belonging to Institute of Computer Science, University of Wrocław.

4.2 Algorithms' parameters

MCTS can be parametrized by the C value, rollout or heuristic function $H(s)$.

RHEA's parameters are: population size, offspring size, mutation probability, mutation operator, crossover operator, individual length, number of elite parents, tournament size, and heuristic function $H(s)$.

Combination's parameters are: order of running algorithms, time given for each algorithm, type of transitions between algorithms.

4.3 Heuristics and MCTS rollouts

4.3.1 SameGame

The heuristic function is simply scored points divided by large value to make it normalized for MCTS.

$$H(s) = \textit{scored points}/5000$$

As a rollout we use *TabuColor* strategy proposed by (Schadd et al., 2012), meaning that in the beginning we chose color that is most frequent on the board. We are not allowed to remove groups of squares of that color unless no other move

is allowed. Other than that, we simply perform random moves until the end of the game.

4.3.2 Search Race

We are using known heuristic from CodinGame. That uses following values:

- C – completed checkpoints,
- T – in-game time elapsed,
- MAX – Number checkpoints in whole game $\times$ 100,
- D – euclidean distance from pod to (next checkpoint $-$ 3 $\times$ pod's velocity)

$$H(s) = \begin{cases} (100 \times (C - (T/600)))/MAX, & \text{when map is completed} \\ (100 \times C - 5 \times log(D))/MAX, \end{cases}$$

MCTS has problems with this game for several reasons. One of them is that the turn limit is 600. Furthermore, the pod to drive optimally should have smooth movement. Generally, it should maintain direction or slow down and take turns. There is very little chance that many random moves will make smooth movements, so it would be better to use good heuristic and make movement less chaotic. Considering this, we end up using two other rollouts for this game: the score is evaluated game state after performing 10 random moves or game state is evaluated without making any additional moves.

4.4 Results

4.4.1 SameGame – algorithms' parameters

MCTS parameter C is set to 0.5.

RHEA's parameters are set to:

- population size – 100
- offspring size – 100
- mutation probability – 0.6
- mutation operator – Uniform
- crossover operator – One-point
- individual length – 30
- number of elite parents – 3
- tournament size – 1

4.4.2 SameGame – Running RHEA and MCTS separately

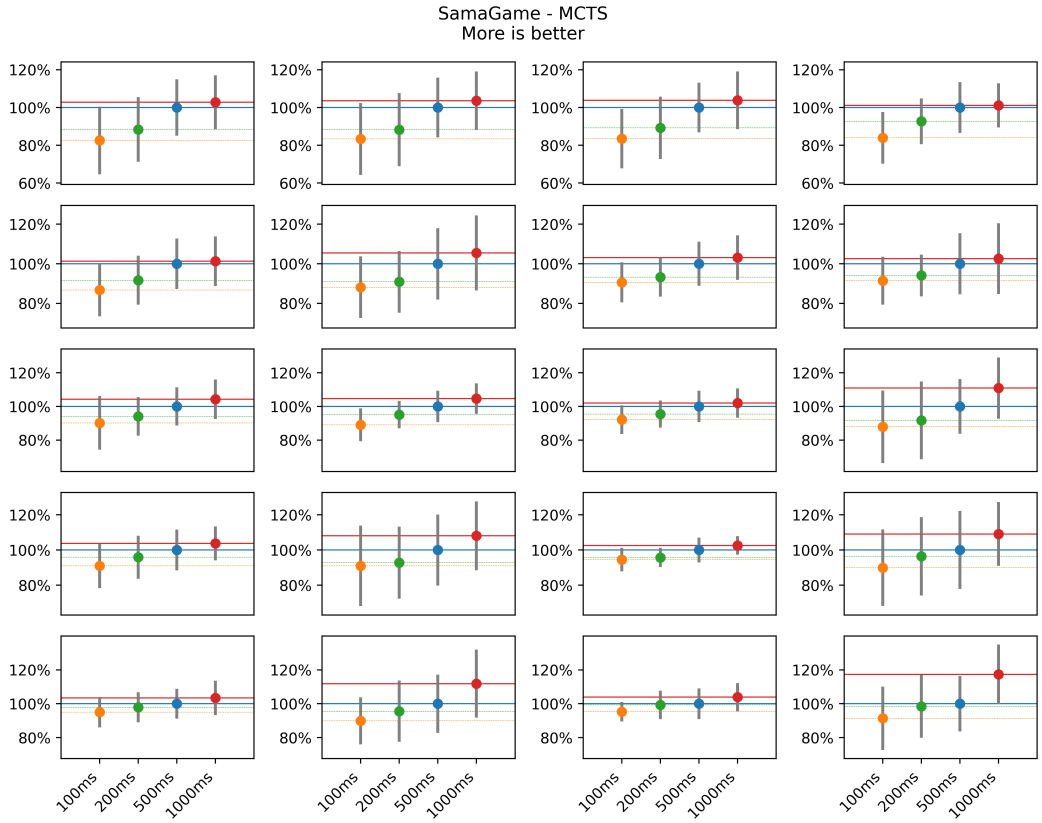

Figure 4.1: Results of MCTS algorithms with given time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms.

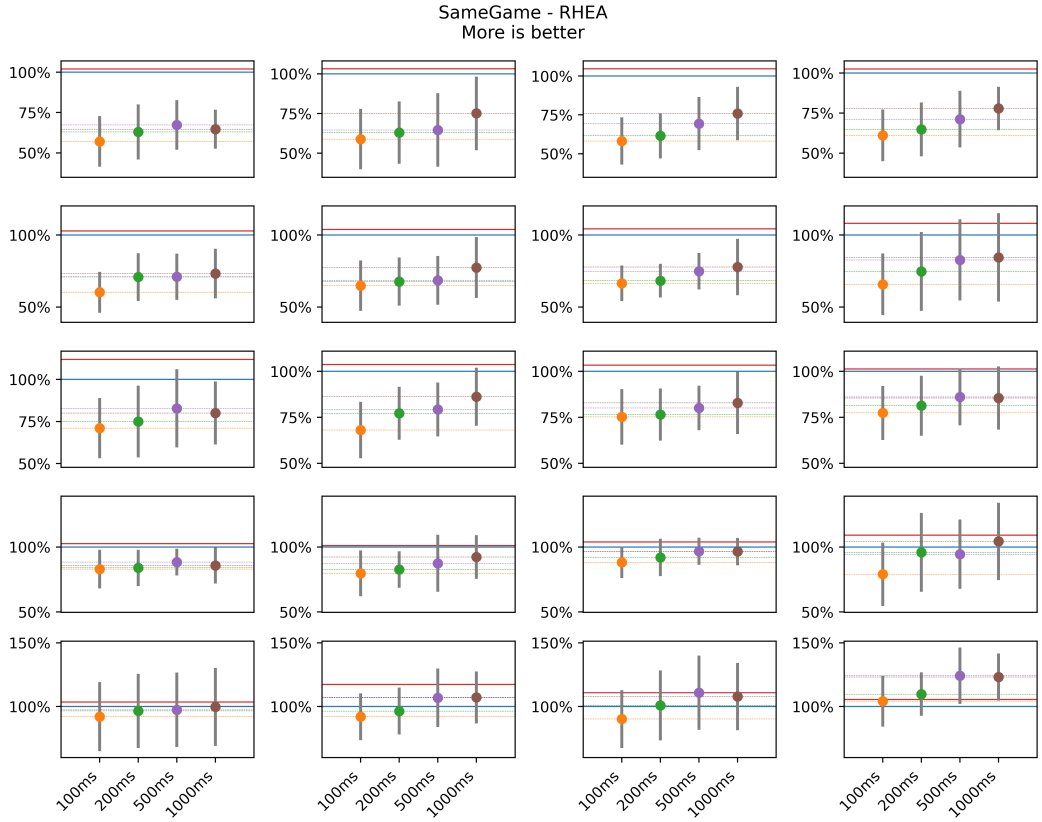

Figure 4.2: Results of RHEA algorithms with given time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

It is clear that MCTS outperforms RHEA in most test cases, even when RHEA has twice the time limit (Fig. 4.1, 4.2).

4.4.3 SameGame – Running combined RHEA and MCTS

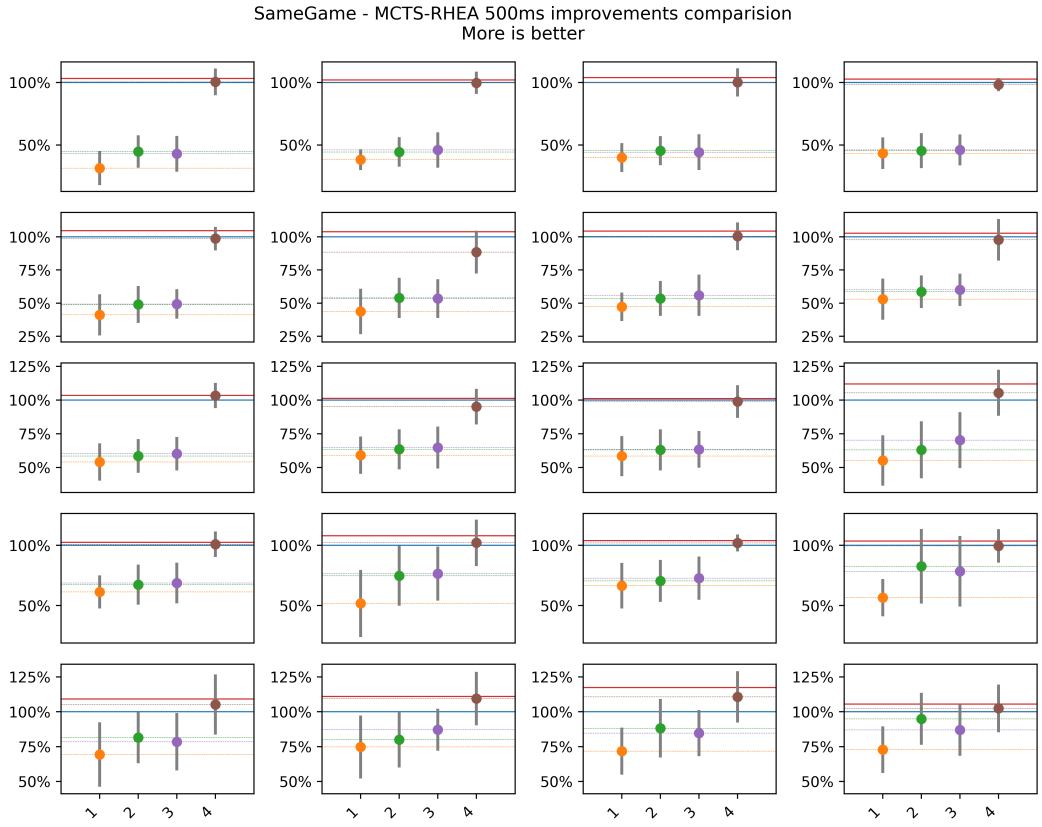

Figure 4.3:
1 – new tree with pruning,
2 – prune tree from previous round,
3 – *2* + add population to tree,
4 – *3* + save rollouts.
100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

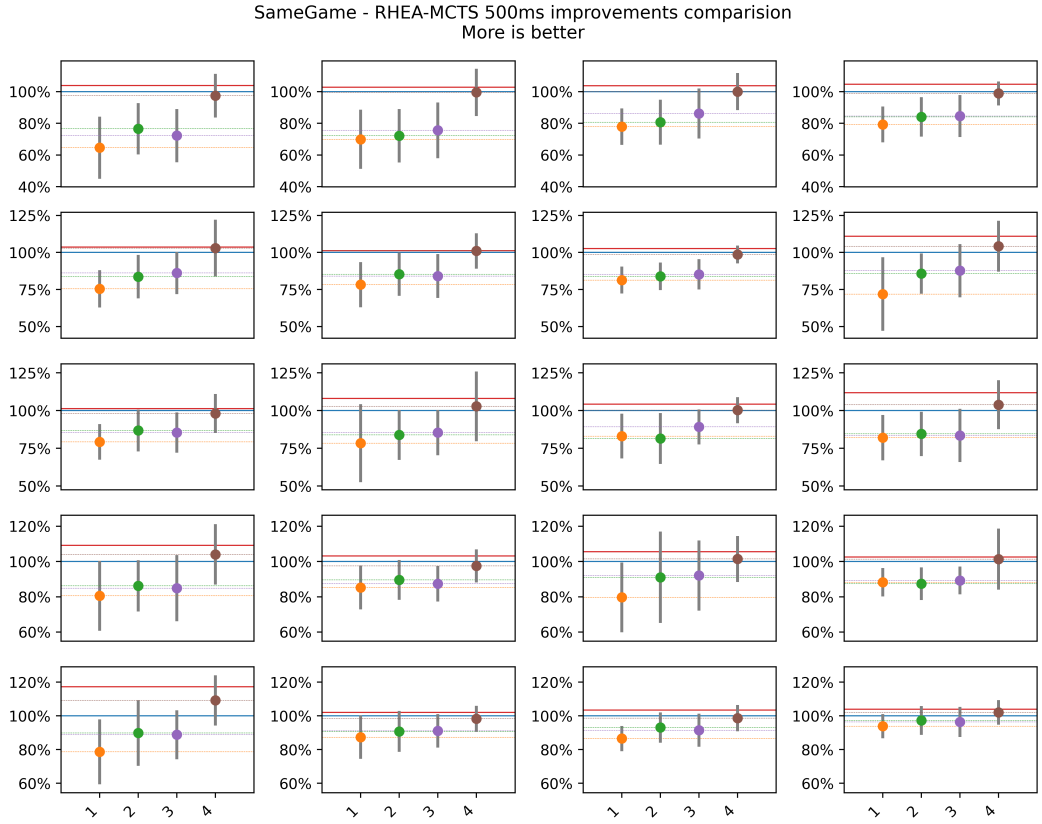

Figure 4.4:
1 – new tree with pruning,
2 – prune tree from previous round,
3 – *2* + add population to tree,
4 – *3* + save rollouts.
100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

As we can see at figures (Fig. 4.3, Fig. 4.4), saving and pruning tree from previous round helped a lot in almost all test cases. Adding population to tree helped in some test cases, but not in every. And finally the breakthrough. Saving the best rollouts to give them to RHEA always improves our scores, even up to nearly 3 times! Now our combination is reaching some promising results.

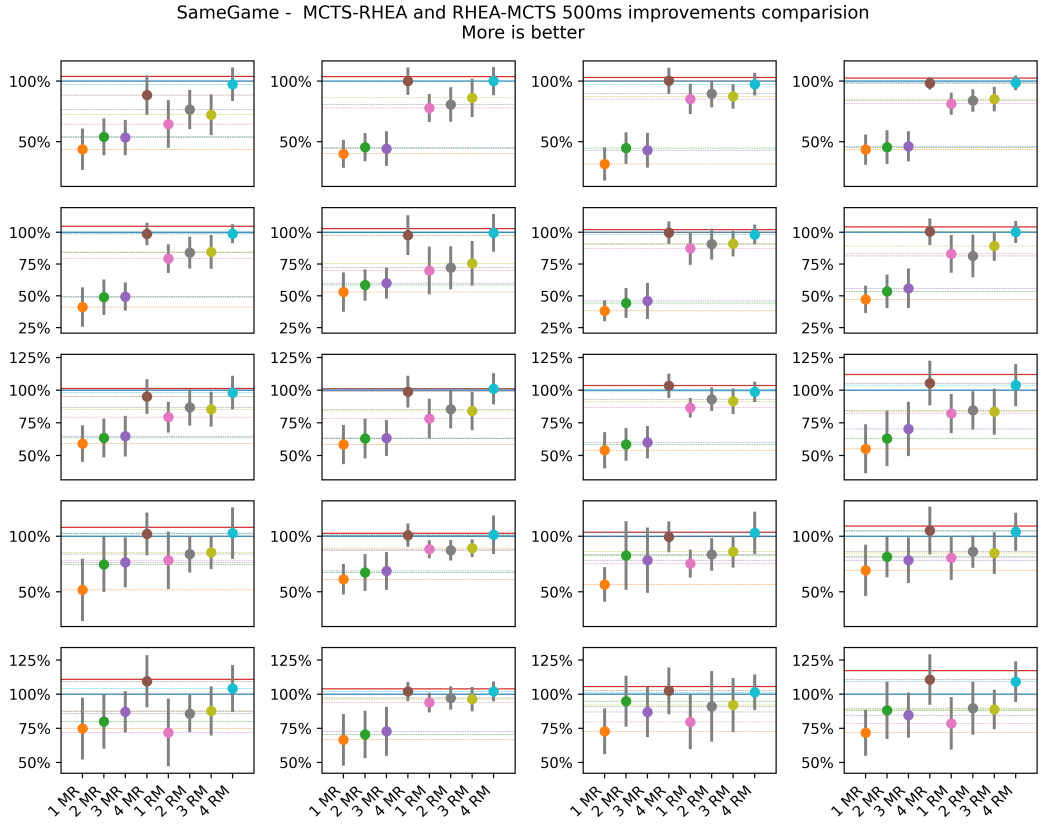

Figure 4.5:
RM – RHEA-MCTS,
MR – MCTS-RHEA
100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

Now let us take a closer look at the comparison between when MCTS is the first algorithm and when is the second one (Fig. 4.5). In versions 1, 2, and 3 we can see that, when MCTS decides about the move, the overall score in most cases is much greater. However, that does not hold in version 4. Although MCTS is much stronger alone than RHEA, scores of *RHEA-MCTS* and *MCTS-RHEA* are close to each other. Even in some cases, *MCTS-RHEA* has a greater average score. Additionally, we can already see, that in some cases versions 4 are above the blue line and close to the red one. Meaning that the average result is close to the MCTS with twice the time limit.

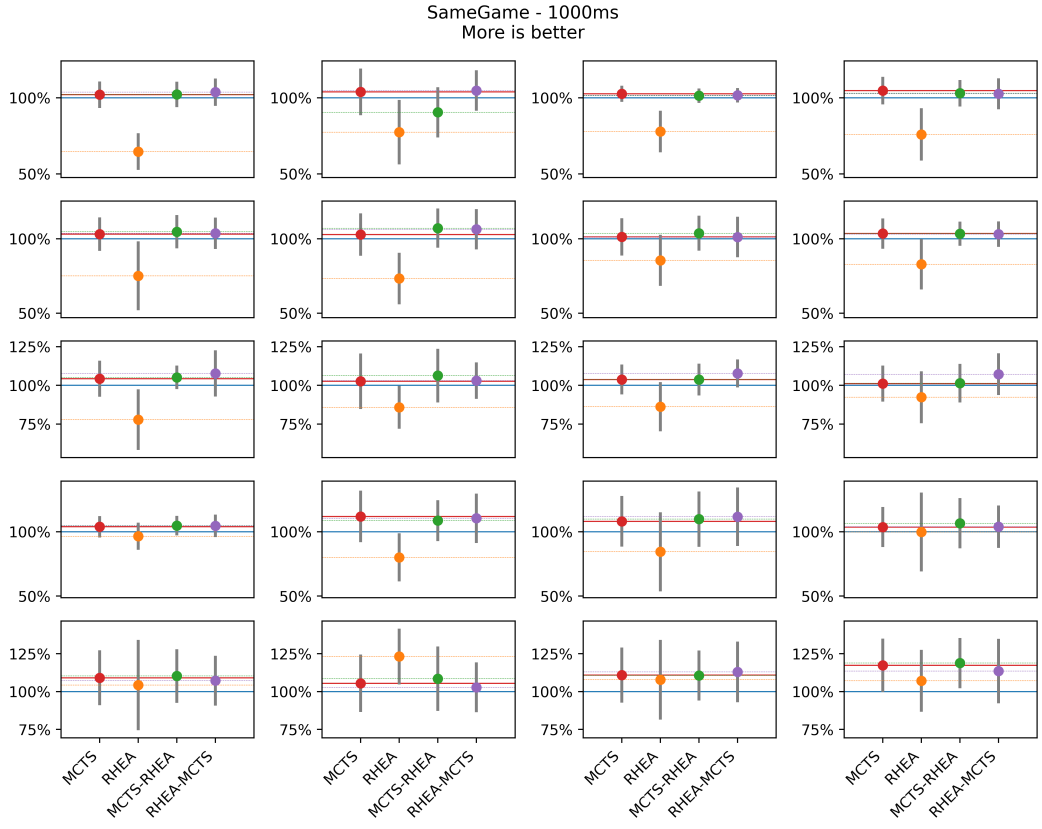

Figure 4.6: Results of algorithms with 1000ms time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

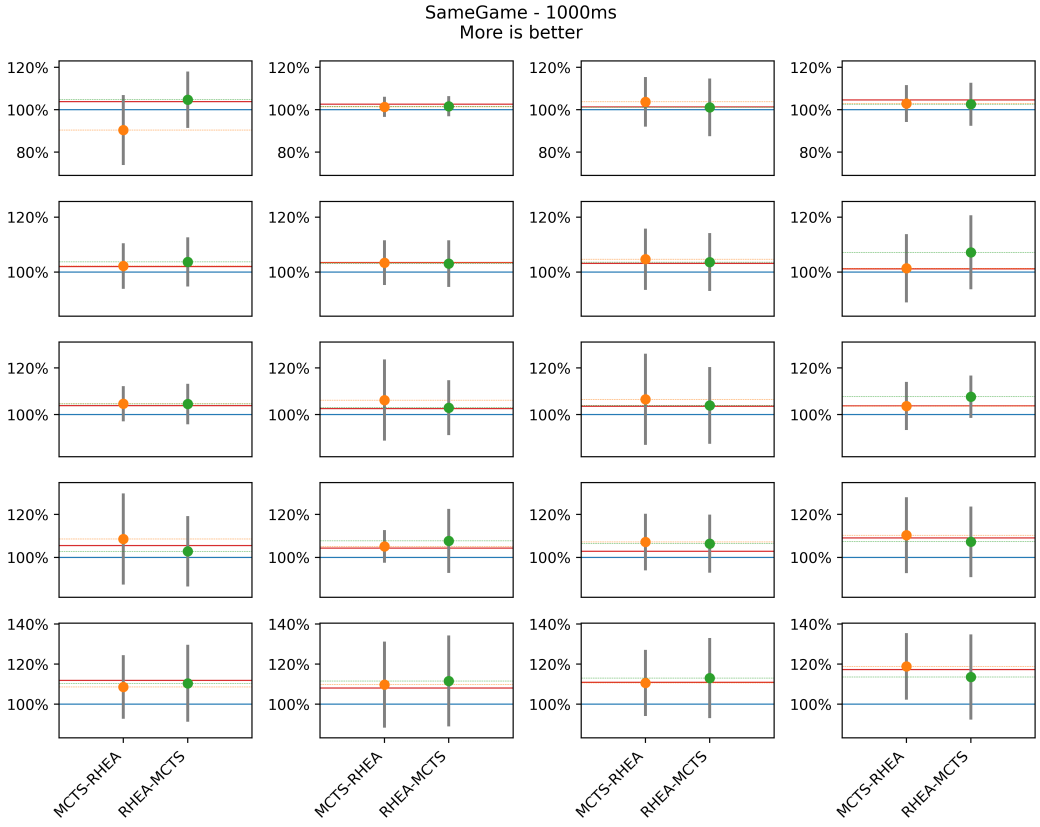

Figure 4.7: Results of algorithms with 1000ms time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

As we can see at Fig. 4.6, even when RHEA score is lower than 80% of MCTS's score both of ours combinations can improve score compared to MCTS, or at least be similar. There is one test case, where RHEA is much better than MCTS, and neither of our combinations is close to that score. This could indicate that we can improve MCTS performance, but we can not do the same for RHEA.

At Fig. 4.7 we can see, that at least one combination improved result of the MCTS at 16 of 20 tests cases. In a few of them, improvement is similar to the difference between MCTS with time-limit 500ms and 1000ms.

As we already know combination could improve results, but until now time was equally distributed for both algorithms. Now we want to test how splitting time in different ratios would affect scores.

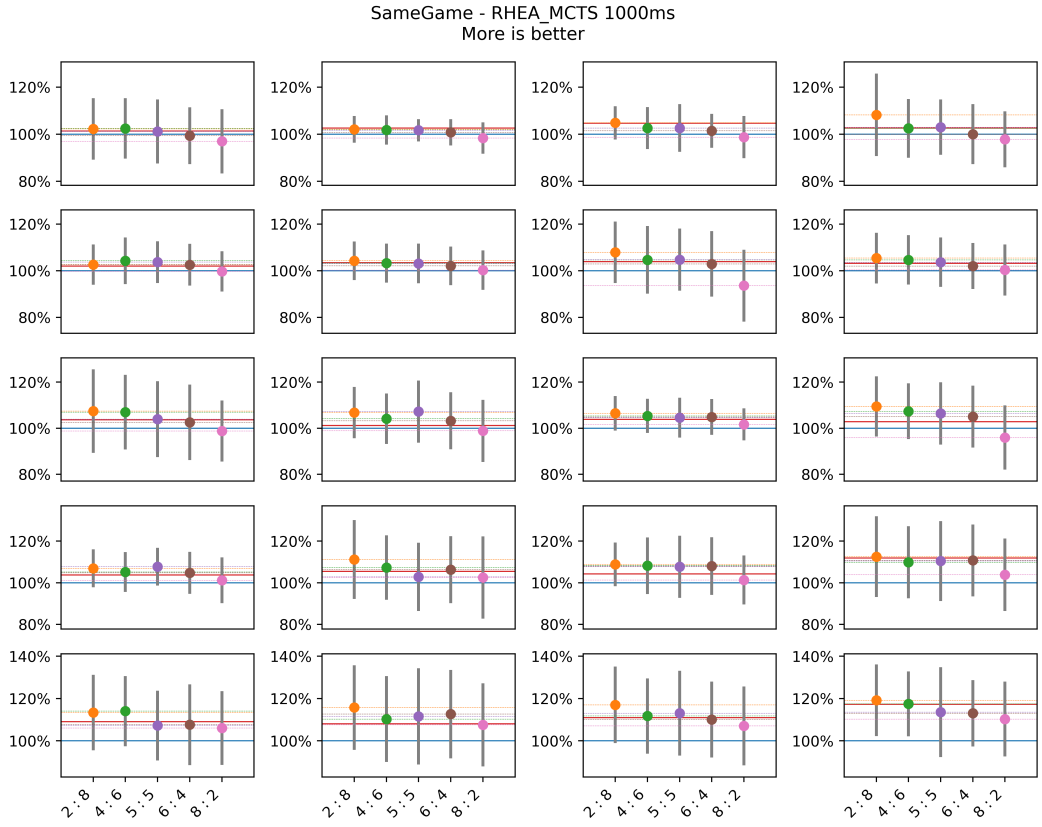

Figure 4.8: Results of algorithms with 1000ms time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms. $R : M$ describes time division between RHEA and MCTS.

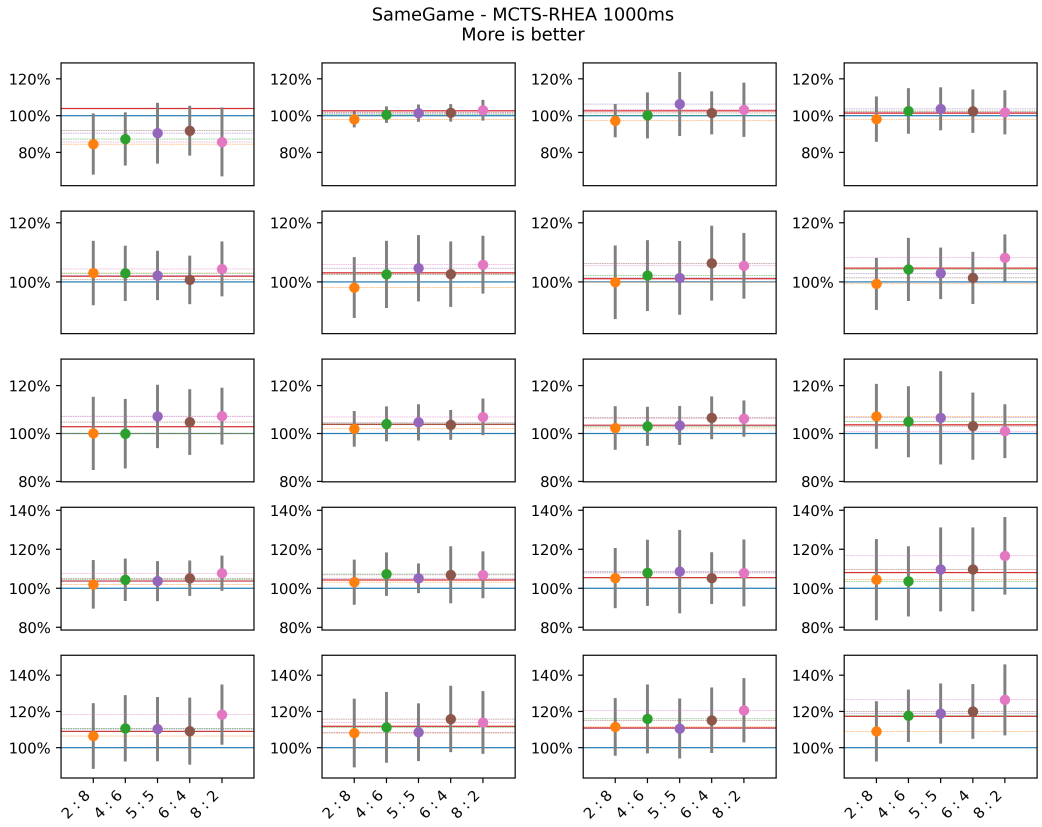

Figure 4.9: Results of algorithms with 1000ms time limit for each turn. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms. $M : R$ describes time division between MCTS and RHEA.

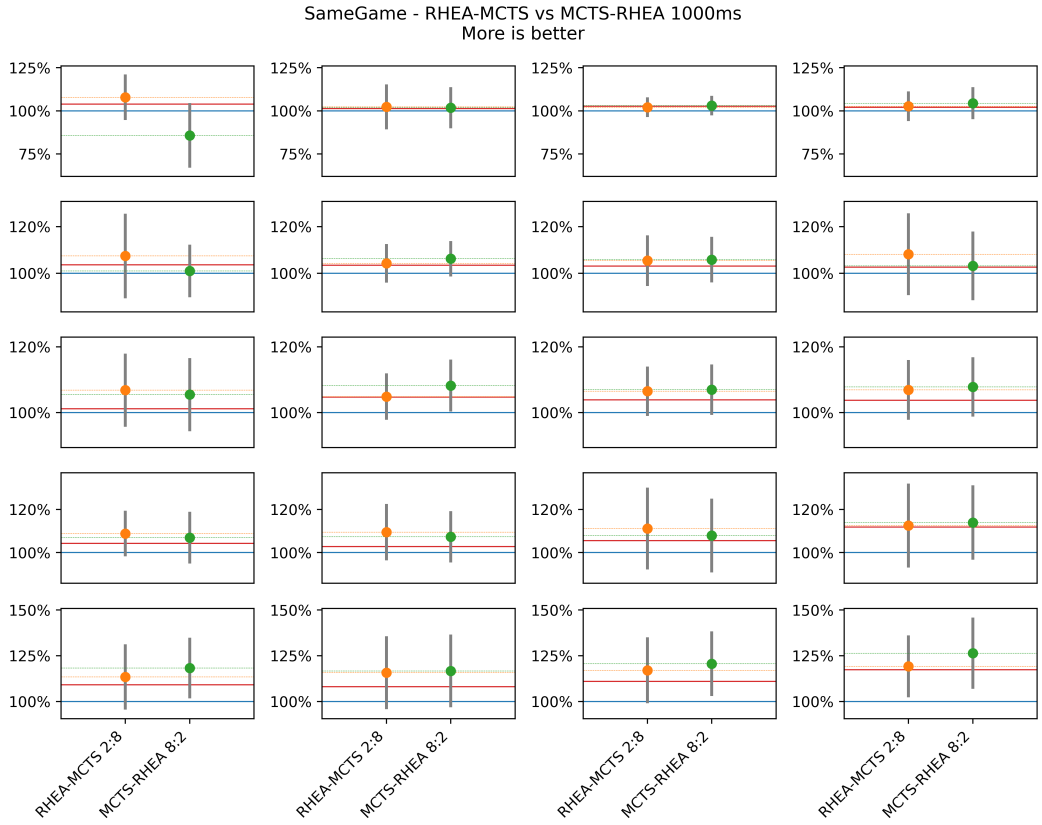

Figure 4.10: Comparison between *RHEA-MCTS* and *MCTS-RHEA*. RHEA runs for 200ms in each of them and MCTS runs for 800ms in each of them, but order in which they are ran differs. 100% is the average score for baseline, here MCTS with time limit equal to 500ms. Red line shows the average score of MCTS with time limit equal to 1000ms.

It is hard to say which ratio is the best one. Usually the more time for MCTS the better. We can not determine which order of running is better. *RHEA-MCTS* performs better in some test cases and worse in the other ones. Often the average scores are similar but sometimes the difference is significant.

4.4.4 Search Race – algorithms' parameters

MCTS parameter C is set to 1.5, rollout is set to evaluate current game state.

RHEA's parameters are set to:

- population size – 100
- offspring size – 300
- mutation probability – 0.15
- mutation operator – Diversity
- crossover operator – Uniform
- individual length – 20
- number of elite parents – 20
- tournament size – 5

4.4.5 Search Race – Running RHEA and MCTS separately

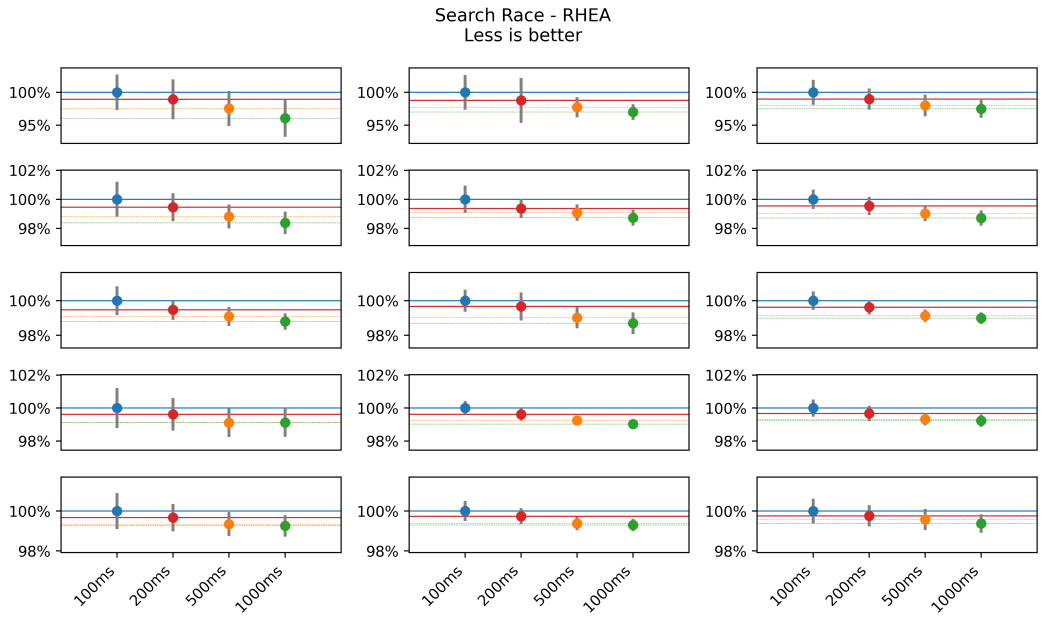

Figure 4.11: Results of RHEA algorithms with given time limit for each turn. 100% is the average score for baseline, here RHEA with time limit equal to 100ms.

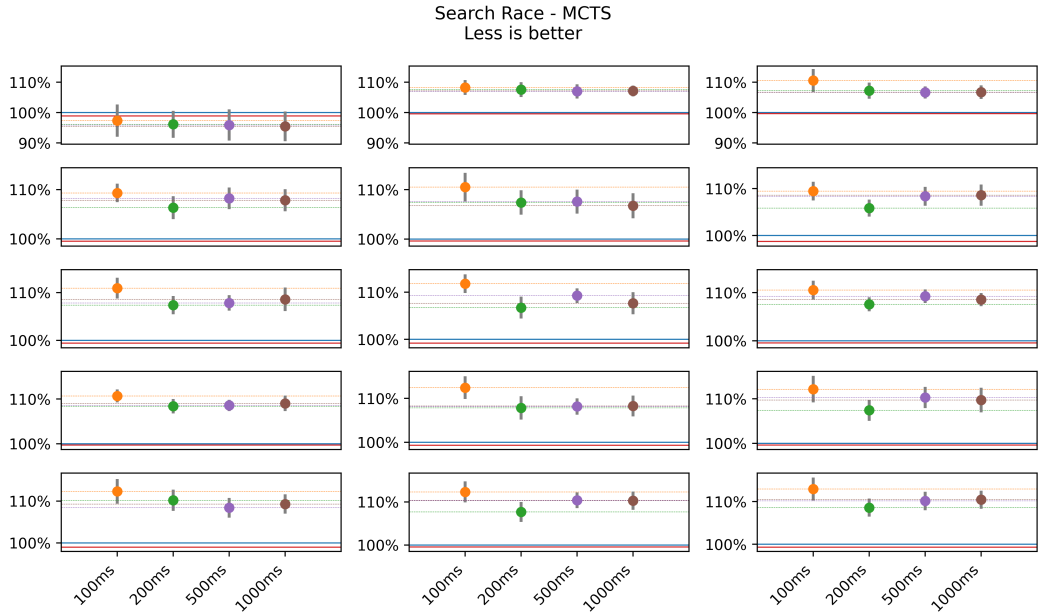

Figure 4.12: Results of MCTS algorithms with given time limit for each turn. 100% is the average score for baseline, here RHEA with time limit equal to 100ms. Red line shows the average score of RHEA with time limit equal to 200ms.

It is clear that RHEA outperforms MCTS in every test case except one, by about 10% (Fig. 4.11, 4.12).

4.4.6 Search Race – Running combined RHEA and MCTS

In order to use combination's improvement that saves rollouts we have to change previously used rollout to the one that performs 10 random steps and then evaluates game state. The rollout is changed only in fourth tested version.

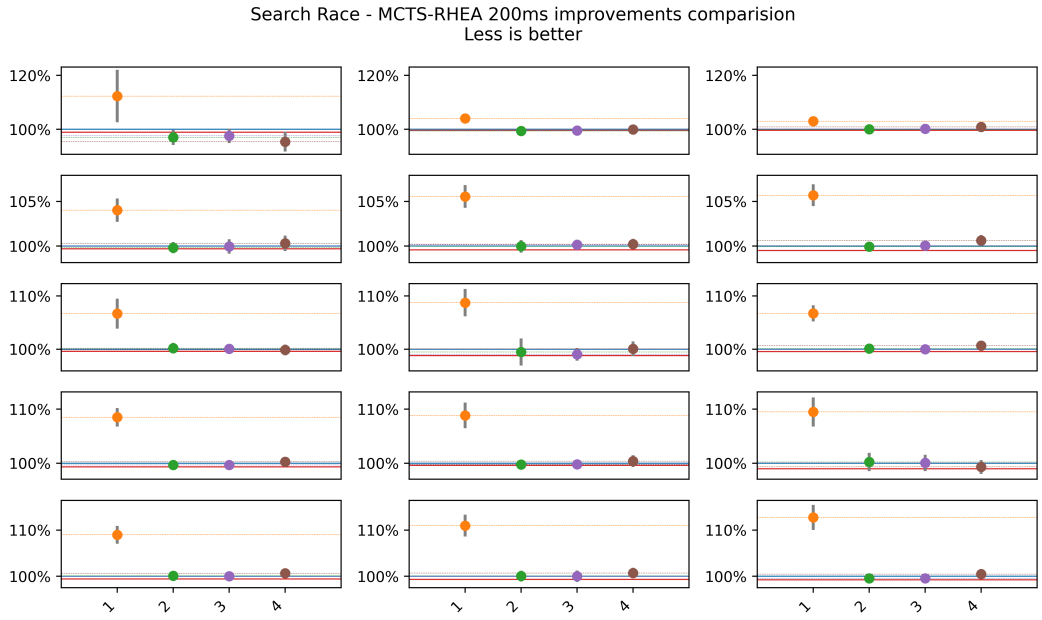

Figure 4.13:
1 – new tree with pruning,
2 – prune tree from previous round,
3 – *2* + add population to tree,
4 – *3* + save rollouts.
100% is the average score for baseline, here RHEA with time limit equal to 100ms. Red line shows the average score of RHEA with time limit equal to 200ms.

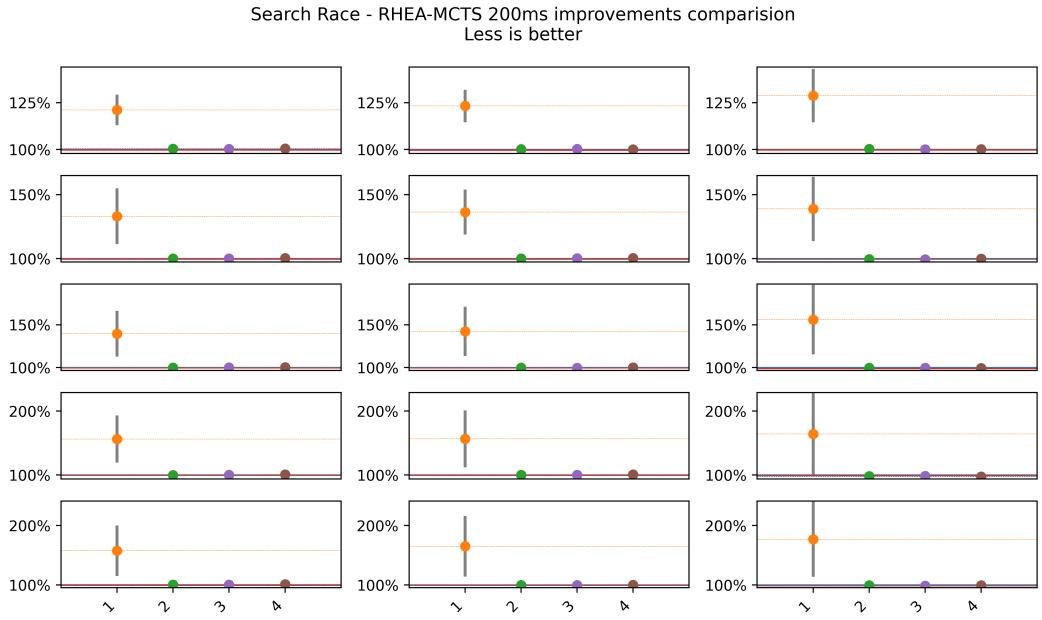

Figure 4.14:
1 – new tree with pruning,
2 – prune tree from previous round,
3 – *2* + add population to tree,
4 – *3* + save rollouts.
100% is the average score for baseline, here RHEA with time limit equal to 100ms. Red line shows the average score of RHEA with time limit equal to 200ms.

As we can see at figures (Fig. 4.13, Fig. 4.14), there is hardly any difference in performance of any of the improvements. Unlike improvements tested on *SameGame* fourth one did not make much impact.

In order to compare *RHEA-MCTS* with *MCTS-RHEA* we will not show first versions as it is far worse that any of other versions.

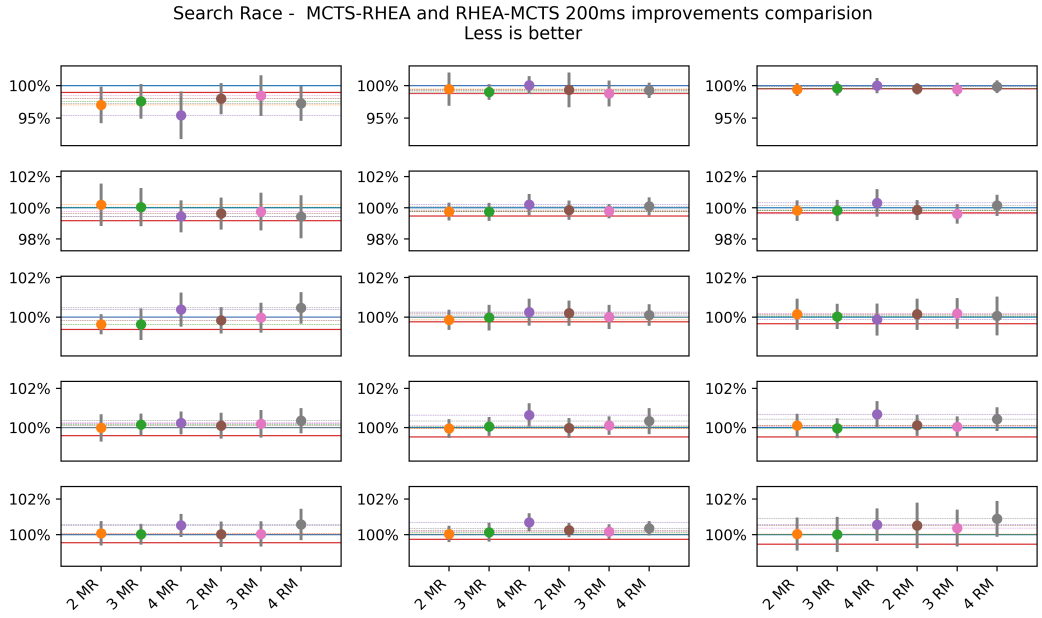

Figure 4.15: Comparison between *RHEA-MCTS* and *MCTS-RHEA*.
100% is the average score for baseline, here RHEA with time limit equal to 100ms. Red line shows the average score of RHEA with time limit equal to 200ms.

As we can see at figure (Fig. 4.15) there are some differences, but we can not determine one version as better than the other ones. Additionally, we can see that besides one test case all versions are worse than RHEA with time limit 200ms. Furthermore they often are outperformed by RHEA With time limit 100ms.

Chapter 5

Conclusion

We have presented several methods of combining MCTS with RHEA that led to improved scores in *SameGame* game. However, they did not succeed in doing the same for *Search Race*. From our experiments, the proposed methods seem to be able to improve score in games in which MCTS performs better than RHEA by default. It should be noted that the most benefiting version in *SameGame* that used MCTS's rollouts, could not be used directly for *Search Race*. Rollout that performed best in *Search Race* was evaluating current game state, thus not producing long rollouts that would be valuable for RHEA's population.

5.1 Future work

Proposed methods of combination could be tested on games in which MCTS performs better by default to ensure our conclusion. We could test as well games that favour RHEA and benefit from long rollouts. Games chosen by us were single player, both RHEA and MCTS could be applied to multiplayer games, thus combination could be as well. Switching between algorithms more than once per turn could be also tested.

Bibliography

Bjornsson, Y. and Finnsson, H. (2009). Cadiaplayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

Genesereth, M., Love, N., and Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI Magazine*, 26:62–72.

Horn, H., Volz, V., Pérez-Liébana, D., and Preuss, M. (2016). Mcts/ea hybrid gvgai players and game difficulty estimation. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Levine, J., Bates Congdon, C., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., and Thompson, T. (2013). General video game playing.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. (2008). General Game Playing: Game Description Language Specification. Technical report, Stanford Logic Group.

Negrevergne, B. and Cazenave, T. (2017). Distributed nested rollout policy for samegame. In *Workshop on Computer Games*, pages 108–120. Springer.

Perez, D., Samothrakis, S., Lucas, S., and Rohlfshagen, P. (2013). Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 351–358.

Schadd, M. P., Winands, M. H., Tak, M. J., and Uiterwijk, J. W. (2012). Single-player monte-carlo tree search for samegame. *Knowledge-Based Systems*, 34:3–11.