Universiteit Leiden
The Netherlands

Opleiding Informatica

Monte Carlo Methods for the

Board Game PATCHWORK

Teun Bergsma

Supervisors:
W.A. Kosters & J.M. de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 25, 2024

Abstract

In this thesis an altered version of the board game PATCHWORK is analyzed using various (Monte-Carlo) methods. PATCHWORK is a 2-player board game where players have to fill their own 9×9 grid board as efficiently as possible by purchasing tiles from a set of polyominoes. The altered version analyzed in this thesis is a variant where randomness and the second player are removed. This is done as the goal is to find a score as close to optimal as possible using various techniques, but namely Monte Carlo Methods. Our findings show that the huge search space of the game means that runtime is a large problem. For the pure Monte Carlo algorithm this limited playouts due to long runtimes, but for the Monte Carlo Tree Search algorithm this led to a bottleneck caused by memory usage. This is why we have used a number of techniques that limited the search space, such as the use of a heuristic and a Reinforcement Learning framework. To do more extensive testing a simulated version of the game is implemented, where tiles no longer need to be placed on the player board. Our results showed that for the real version of the game we were able to achieve a score of 74 using a player that combines multiple Monte Carlo methods and brute force and for the simulated player we were able to achieve a score of 84 using Monte Carlo Tree Search and brute force. The action sequence that led to a score of 84 can be altered to be possible in the 2-player game with a score of 81.

Contents

1 Introduction

Board games have been a popular pastime for hundreds of years due to the fun they offer. They can be enjoyed by people young, old, and everything in between, because they offer a fun challenge for all. The nature of this challenge varies greatly per board game, but for most games this challenge stems from the ability to understand the mechanisms of a game and then using these mechanics to find an optimal strategy given your situation. This topic lends itself very well to the discipline of computer science, as optimization is a task in which computers excel. Many popular board games such as CATAN [SCS09], MONOPOLY [Kot12], GO and CHESS [SHS$^+$18] have already been thoroughly researched using algorithms to find optimal strategies and much more about these games.
The board game that will be analyzed in this thesis is called PATCHWORK [Ros14a]. The game is quite popular and has won a number of awards [Gee], yet only a single other scientific paper [Lag20] describes research in regards to the game and that paper focuses mostly on the tile laying component. The game is a 2-player game where the players both have a 9×9 grid which they have to fill as efficiently as possible by taking turns and purchasing Tetris-piece like shapes (polyominoes). Scoring is based on the number of spots filled on their board and currency left at the end of the game. This means that the challenge here comes from a finely balanced set of polyominoes with different costs and shapes that each differ in quality at certain points in the game. This leads to an interesting puzzle where the players have to critically assess what polyominoes to purchase at what point in time to achieve the highest score, by maximizing their earnings and efficiently filling the open spaces.
This thesis will focus on an altered version of the game in which the game is only played by one player and is thus more of a puzzle. An example of a single-player game in progress can be seen in Figure 1. In this figure the player board is on the right, which the player is trying to fill as efficiently as possible by repeatedly purchasing pieces with different costs and shapes.

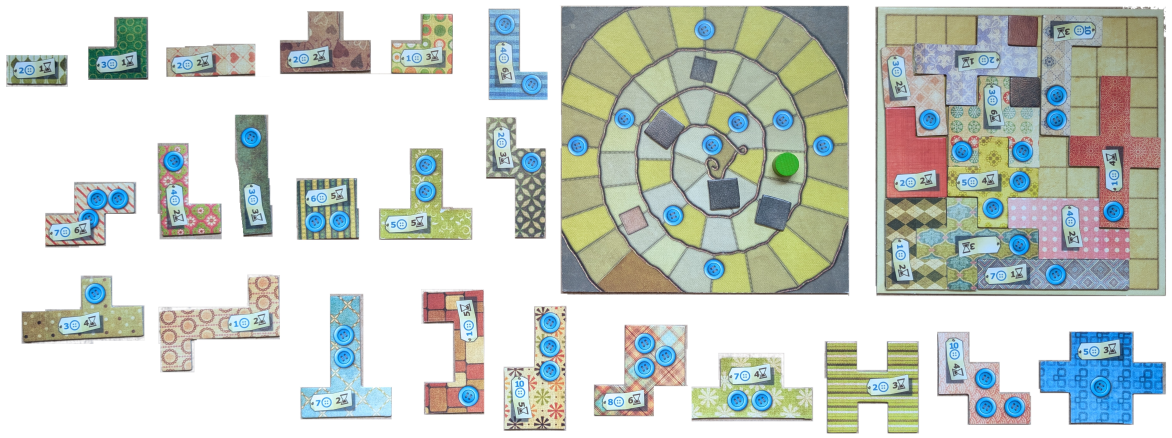


Figure 1: An overview of a single-player game in progress.

Thesis Goal

For this thesis, the goal is to achieve a score as close as possible to the maximum achievable score in the single-player game (which we presume to fall in the [84-90] range based on results from our experiments), and finding the action sequence that leads to this optimal score. As mentioned previously we look at an altered version of the game to do this. This altered version will only have one player and thus has slightly altered rules (that have been adapted for the single-player version) compared to the original game. This altered version also removes any randomness present in the game (this mechanic is normally present during tile selection) as the goal is finding maximum possible scores. This is done to reduce the amount of computing power required (also the second player is presumably irrelevant in finding an optimal score). We can then verify if this score is also achievable in the real 2-player game. While these changes are not major for determining an upper limit for a possible score in the 2-player game, they do alter the game significantly and thus it would be interesting for future work to analyze the actual 2-player game of PATCHWORK without the removed randomness and second player.
We aim to achieve our goal of finding a maximum score by using a variety of methods such as human like strategies, but we will be mainly focusing on Monte Carlo based methods. Monte Carlo methods are based on a large number of random playouts used to gauge the effectiveness of a move at that time. We will specifically use a pure Monte Carlo method, a greedy variant of this pure Monte Carlo method, Monte Carlo Tree Search and a greedy variant of Monte Carlo Tree Search. Using these methods we aim to achieve a score as close to perfection as possible.

Thesis overview

This bachelor thesis was written under the supervision of Walter Kosters and Jeannette de Graaf at Leiden University for the Leiden Institute of Advanced Computer Science.
We start with an introduction present in this section; Section 2 contains other work related to the methods or subject of this thesis; Section 3 contains the rules of the game, some examples and the single-player variant to be analyzed in this thesis; Section 4 explains the used algorithms and various other techniques that are utilized; Section 5 shows the results gathered from experiments based on the various methods used; Section 6 concludes our findings and discusses potential future work.

2 Related Work

In this section we will be looking at research focused on (board)games with similar mechanisms to the game PATCHWORK, games which have been experimented on using similar methods to the ones used in this paper and methods similar to the ones used in this paper.

The field of game theory as mentioned in the introduction is quite interesting for computer scientist as the rule based systems are a good fit for optimization algorithms. Some highly skilled games such as GO, SHOGI and CHESS have been played using a general algorithm which has been based on Reinforcement Learning principles as in [SHS+18]. Additionally, the authors of [CWvdH+08] give a detailed overview of the Monte Carlo Tree Search method and it is used to play the game GO.
In [DI22] the goal is to greedily play an altered version of TETRIS as quickly as possible. This altered version exclusively uses rectangular pieces (instead of the polyominoes present in normal TETRIS, such as an L shape), the goal is to greedily find a spot for a piece while minimizing the gained height from dropping this piece as efficiently as possible. This paper features an impressively large list of references to work related to algorithms playing the game of TETRIS by utilizing all kinds of machine learning and artificial intelligence techniques. In addition to knowledge on computers playing the game their references also include a number of papers regarding the way a human plays the game of TETRIS.
In [ZZN11] the game of TETRIS is analyzed using a Monte Carlo method. The authors use a bandit-based Monte Carlo planning method (specifically using Upper Confidence Bound Trees). To save on computing time they use a clever trick based on the fact that Monte Carlo Tree Search does not keep information of game states from previous iterations. A new tree is formed for each new decision, but in TETRIS a game state can be revisited. This is why they decided to create a method to store this information that allows it to be re-used. They also use a bandit algorithm to handle the trade off made between exploration and exploitation and to guide the planning process.
Lastly in regards to (board) games, [SCS09] analyzes the very popular game of CATAN using Monte Carlo Tree Search.

The game of PATCHWORK features two key decisions: what tiles to purchase and where to place the purchased tile. The placement part has been researched thoroughly as this is a polyomino placement problem (a problem where a minimal amount of space is given and polyominoes have to be placed to fill this space as efficiently as possible) which has been analyzed using many different algorithms. Online solvers [Mea] for this problem have also been created, which can use a variety of methods to solve them such as: Algorithm X [Knu00] (Dancing Links), which reduces the problem to an exact cover problem, and an algorithm which reduces the problem to a SAT (Boolean satisfiability) problem to be solved.
The authors of [TYA22] improve upon a method used in previous studies to solve the polyomino placement problem. This method works by embedding the polyomino puzzle in a quadratic unconstrained binary optimization (QUBO) problem, "where the objective function and constraints are transformed into the Hamiltonian function of the simulated Ising model". This method aims to be improved by introducing new constraints such as removal of bubbles (1×1) gaps and introduction of new guiding terms to encourage favorable rotations and polyomino pairs.
With regards to placement [CRM+13] discusses the polyomino placement problem using a genetic algorithm. A genetic algorithm is an approach that is inspired by natural selection and it works by

representing a possible solution to a problem as a chromosome (collection of genes), these then go through numerous iterations of selection, crossover and mutations to potentially find a good solution. In this paper this is done by using a procedure named the “snowball” algorithm which is based on a binary genetic algorithm. Here the genes represent the rotation of a piece (but not its position) and a chromosome represents a collection of polyominoes and their associated rotations. To actually place said pieces the “snowball” algorithm starts from the center of the grid and repeatedly builds outwards. All the potential solutions (chromosomes) then get a rating and based on this, selection is performed meaning that the population (current set of available chromosomes) improves with each iteration.
Lastly in [Lag20] the PATCHWORK board game is used to research “State representation and Polyomino Placement”. In the paper constraint programming, a solving technique that involves specifying a set of constraints that must be satisfied in order to find a valid solution, is used. Regular constraints are used to “specify the required placement of a patch on the board”. This means that all placement options for a tile are encoded by a regular expression and these are evaluated later on to find the seemingly optimal choice.

3 Patchwork

Patchwork [Ros14b] is a tile-laying game where two players face off against one another by attempting to get the highest score (see scoring formula in Section 3.1) through the repeated selection of tiles and their placement. The tiles are polyominoes, meaning they are shaped similarly to Tetris [DI22] pieces. The players have to place these tiles as efficiently as possible on their own player board consisting of a 9×9 grid. On the player's turn, they can either skip a turn to earn currency and give up some time or they can purchase a tile with the two currencies present in the game, time and buttons. The players repeatedly take turns until they have spent enough time to reach the end of the "walking board", and once both players reach this point (space 53 on the "walking board") the game ends and the player with the highest score wins.

3.1 The game

In this section we mention the different game elements and their interplay to give an overview of the players goals and the mechanics of the game. Some elements of the original game will be left out, namely the turn order, limited selection and the random setup, as these things are not relevant for this thesis. For a full overview of the original rules of the game the reader is referred to the rule book [Ros14b].

Tiles

The game's goal is to maximize the player's score, which is achieved by filling their board as efficiently as possible with the available tiles. There are 38 tiles in total, where 5 of those tiles are 1×1 tiles which are awarded upon passing certain spaces on the time board (these are not purchasable and are explained in more detail below). The other 33 pieces are of varying shapes and sizes covering 2 to 8 squares on the player's board. Each of the other 33 tiles has a unique combination of the following attributes:

Button cost B_{C} : The number of buttons required to purchase the tile.

Time cost (T_{C}) : The number of steps taken on the "Walking board" upon purchase.

Buttons in piece (S_{B}) : The number of buttons this tile awards the player upon each passing of a button spot on the "Walking board".

Size (S) : The number of squares the tile fills on the player board.

Shape : The structure of the tile.

For a structured overview of all purchasable tiles see Figure 2 (these are the same as the real game shown in Figure 1) and for an overview of how these tiles actually look like in the real game see Figure 1. Figure 1 shows each tile (seen on the player board and left of and below the boards) with its button cost in blue denoted with a blue button behind the number, and the time cost is shown with an hourglass after it. Shape and size speak for themselves and the number of sewn in buttons is based on the number of blue buttons depicted on the tile (not including the button icon denoting the number of buttons required to purchase the tile).

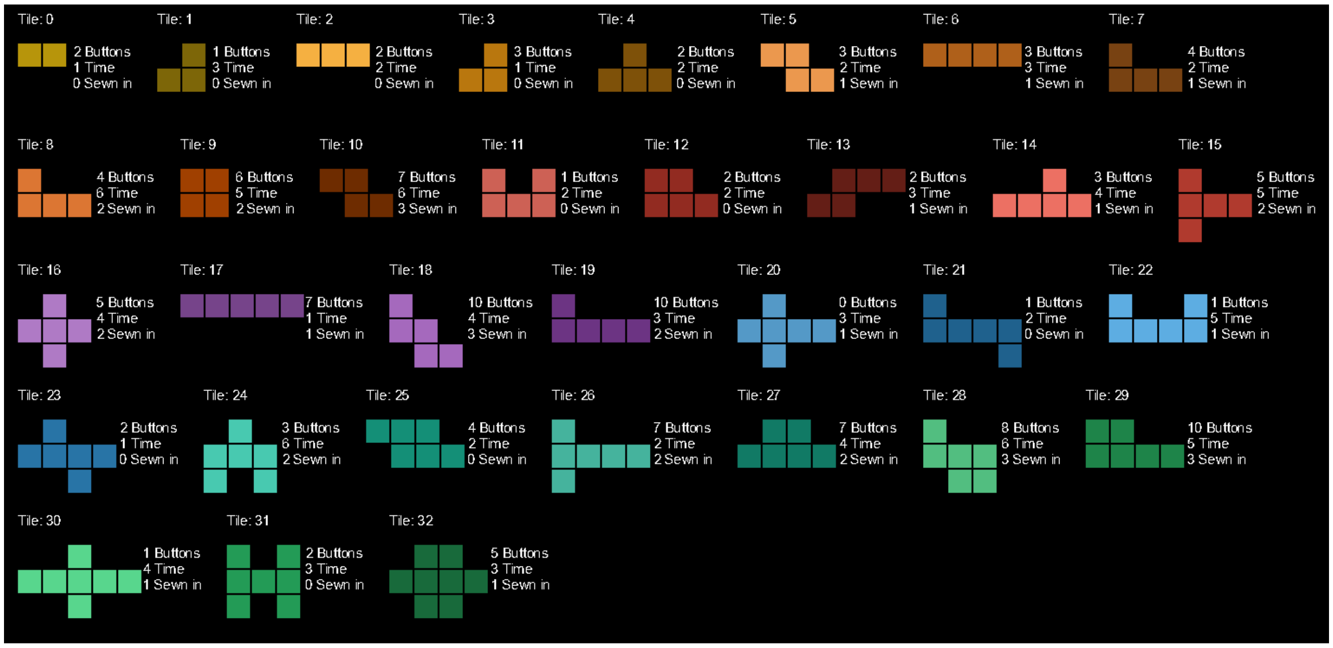


Figure 2: All the purchasable tiles in the game, containing their associated costs and the number of buttons they contain (are sewn into them).

Boards

As seen in Figure 3 there are three boards in the game:

- Two identical player boards: These boards show a 9×9 grid which the players try to fill by purchasing tiles and placing these on their board.
- A "walking board": This board has a finite number of steps (53) and players advance on this board by purchasing pieces. Most of the pieces have a cost in time, where 1 time unit represents a single step on this board. This means that if your purchase a tile with a time cost of 3, that after purchasing it you immediately take 3 steps on the "walking board". The board can also be advanced by deciding to pass the other player which awards the player with a number of buttons equal to the number of steps required to pass the other player. This gives the player an option to exchange time for buttons. This board also contains 9 spaces which award any player that goes past it (both players can earn buttons from the same spot) a number of buttons equal to the number of buttons sewn in all the tiles they own. Finally 5 spaces reward only the player that passes it first with a 1×1 tile that has to be placed immediately.

Buttons

Each purchasable tile has a cost in buttons tied to it. To be able to purchase this piece the player has to be able to pay this many buttons otherwise they are not allowed to purchase it. Both players start with 5 buttons and there are two ways to get buttons during the game:

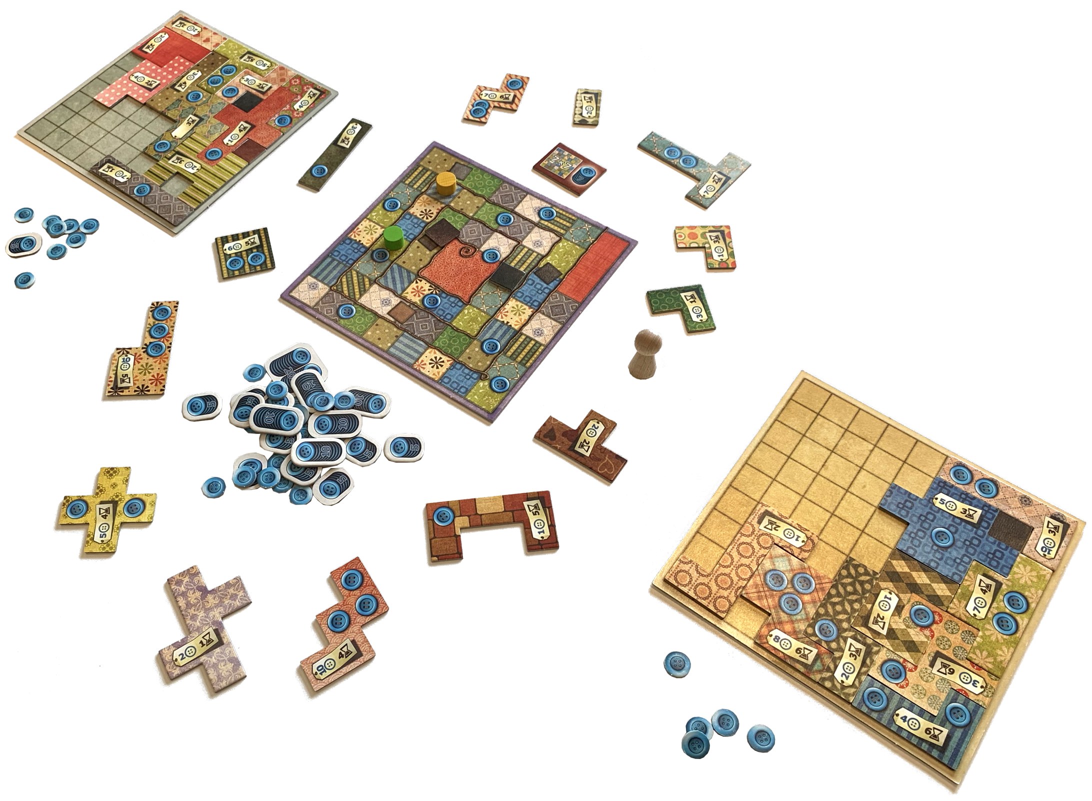


Figure 3: An overview of a 2-player game in progress [Lag20]. On the left and right the player boards can be seen which are partially filled with purchased tiles, and in the middle the "Walking board" is shown where the yellow and green circles represent what time the players are at.

- There are 9 spots on the "walking board" where upon entering a certain spot the player receives buttons equal to the number of buttons in the tiles they have collected. The exact spots are at timesteps 5, 11, 17, 23, 29, 35, 41, 47 and 53.
- On one's turn there is the option to take as many steps as it requires to pass the other player on the time board and one receives buttons equal to the number of steps taken. This is a way to exchange time for buttons. However, the opposite is not possible. This is not possible in the single-player version, but we will account for this by making some adjustments described later in this section.

Scoring

The final formula for deciding the achieved score is:

$$score(a) = \begin{cases} Buttons(a) + 7, & \text{if } OBS(a) = 0 \\ Buttons(a) - 2 \cdot OBS(a), & \text{otherwise} \end{cases} \quad (1)$$

Here a is the player and $Buttons(a)$ refers to the total number of buttons not spent at the end of the game by player a. Next, $OBS(a)$ refers to the number of Open Board Spaces on player a's player board that are not covered by any tile. As can be seen in (1) there is a bonus for entirely filling each space on the board to incentivize a well-structured board even more.

3.2 Single-player variant

With the rules of the regular PATCHWORK game explained we will now discuss any components that had to be altered for the single-player variant that we attempt to optimize for this thesis. The game remains mostly the same as there is very little direct player interaction other than the other player purchasing pieces and being able to get the 1×1 pieces. Some mechanics do however need to be altered such as:

The option to pass the other player to get buttons in exchange for time: We will allow the player to take as many steps (>0) on the time board as they want to get an equal number of buttons.

Limited tile selection and random setup in the base game limits the tile selection (tiles purchasable at that moment) at any time to a select number of tiles which updates and changes upon purchase of a tile (for the full rules on this the reader is referred to rule book [Ros14b] sections "Setup" and "Take and Place a Patch"). We aim to find a maximum score, thus wish to eliminate this randomness from the equation. To do this any tile will be purchasable at any time.

Single tiles have not been altered in any way, but it is worth noting that in a normal 2-player game it is highly unlikely (yet not impossible) for one player to get all the available single tiles in the game. This could be accounted for by limiting the number of available single tiles, but in order to find a maximum score we have kept them as is.

This leads to an optimization puzzle where an optimal sequence of actions consisting of purchases and skips (exchanges of time for buttons) has to be found to achieve a high score for this single-player variant. We can then also examine if this score is theoretically possible in a two-player game by attempting to find an action sequence for the second player that allows the first player to perform their optimal action sequence.

3.3 Examples

In this section we give some brief examples of actions in a single-player game in order to make the game clearer.

Example 1

In Figure 4 we can see a diagram depicting two game states and the steps taken to transition from the state on the left to the state on the right. This example gives an overview of the option to skip spaces (exchange time for buttons) in order to earn buttons. In this case the player lacks the buttons to purchase the piece they want to acquire (as can be seen in the left game state, denoted by "0 buttons" next to their board). This means that the player will have to skip a number of spaces. The player needs 3 buttons (the button cost of the tile the player has decided to purchase), but if they skip 2 spaces they will reach a button spot (as can be seen on the timeline below the player board). The player has 4 sewn in buttons in the pieces on their board (shown right of the board), meaning that skipping 2 spaces will award them 2 buttons (due to skipping 2 spaces) and 4 more buttons (due to them passing a button spot and having 4 sewn in buttons on their board). The player can then purchase and place the piece it has selected. This leads to the game state on the right.

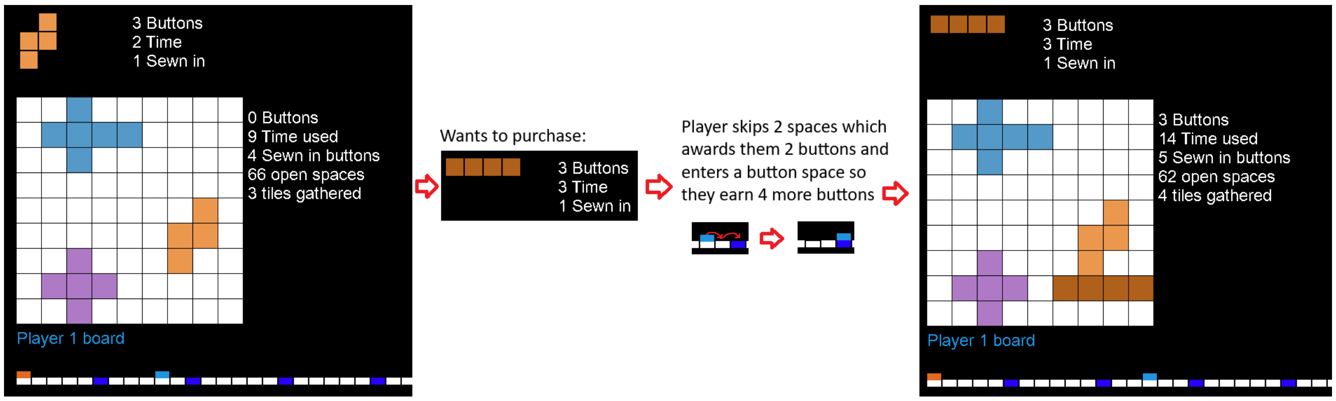


Figure 4: A diagram with two actions in the game, highlighting the option to skip spaces on the time board to earn buttons in order to be able to purchase a piece. In the left the game is in a state where the player possesses zero buttons. The player will need to skip a number of spaces to gather funds for the selected tile. The player can skip 2 spaces, earning them 2 buttons and 4 additional buttons due to them reaching a button spot. This means that they have 6 buttons once they reach that button space and then they are able to purchase the selected tile.

Example 2

In Figure 5 a diagram depicts two game states and their transition. This example gives an overview of the way that players can place one of the 1×1 tiles present in the game. In this case the player purchases a piece which makes them advance the time board by 3 steps (due to the tile having a time cost of 3). This causes the player to pass a single tile spot and this is then placed in a spot where no other tile would fit as seen in the game state on the right.

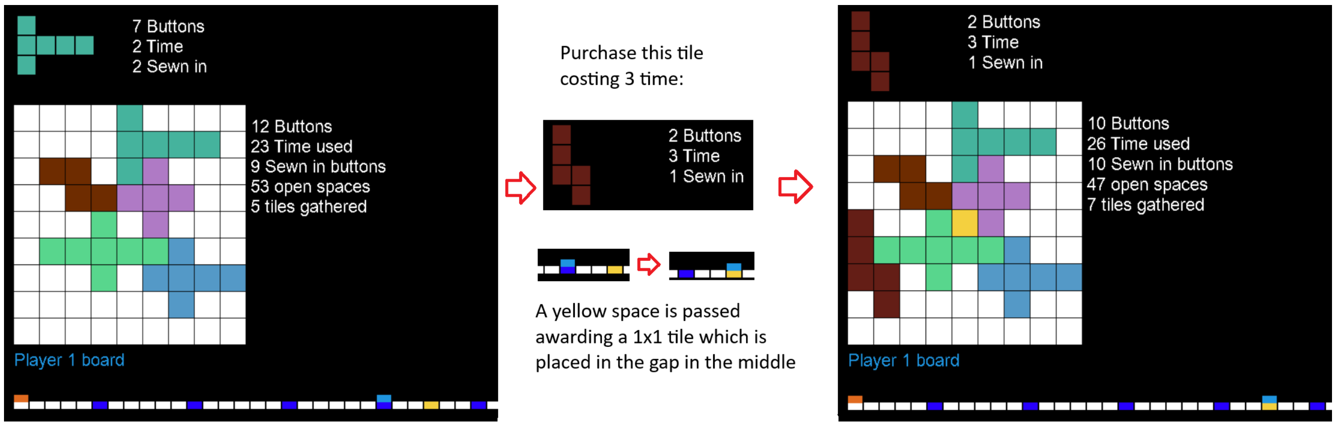


Figure 5: The figure depicts a diagram showcasing the purchase of the tile shown in the middle, which results in the player passing a 1×1 tile spot. Upon reaching one of these spaces the player is awarded a single 1×1 (yellow) tile which has to be immediately placed. In this case it is used to fill the hole in the middle of the player board.

4 Methodology

To optimize and research the game several methods have been used and these will be described in detail in this section. This includes methods directly solving the game, such as a simple random player or a Monte Carlo player as well as methods that aim to reduce the search space such as a heuristic. Additionally an overview of runtime optimization is discussed and a Reinforcement Learning framework which uses other methods is discussed.

4.1 Simple strategies

A number of strategies have been implemented to be tested and examined. The "simple" methods in this section are limited to methods that only base their decision on the current game state without performing any sort of playouts (random games) to decide what piece to buy. In our case these are the random player, rule-based placement player and a Return Zero player. All of the implemented players will be explained in the following subsections.

Random Player

A random player is implemented to be used for experiments to get a solid baseline to compare scores of other algorithms to. Additionally it is very important as it is later utilized for the Monte Carlo methods. This random player consists of a rather simple algorithm to play the game. This player will find all the placement options for all the possible tiles pieces (that can still be purchased before the end of the game) and then it will pick one of these placement options at random. If necessary the player will now skip a certain number of spaces on the "walking board" to earn enough buttons to be able to purchase the selected tile. After this it purchases the tile and places it in the chosen position.
Given this description, the first random player uses a very straightforward process consisting of the pseudo code in Algorithm 1:

Algorithm 1 The loop that makes up the random player in pseudo code.

```
1: while steps taken < total time in game (53) do
2:     Find all moves for all purchasable pieces that are able to fit
3:     Randomly select one of the (position,rotation,piece) sets
4:     if pieces that fit = 0 then
5:         Skip spaces until steps taken = total time in game
6:         break
7:     if #buttons of the player < button cost of selected piece then
8:         Take minimal number of steps required to get the required buttons
9:     Purchase the piece (spend buttons and take steps on the "walking board") and place it in
       the decided position
```

It is worth noting that the line "Take minimal number of steps required to get the required buttons" does take into account that at certain timesteps a smaller number of steps can be taken to get the required funds if the player can earn the buttons by reaching a button spot as described in

Figure 4.
One thing that can be easily noticed from Algorithm 1 is the fact that this version of the random player is actually biased towards certain pieces. Pieces with a larger number of placement options (mostly smaller pieces) have a larger chance of being randomly selected as they appear more frequently in the collection of (position,rotation,piece) sets. This combined with the fact that it is computationally very expensive to find all the possible placement options for all tiles repeatedly leads to another variation of this random player to utilize.
This variant only differs in the selection step. This altered version finds all tiles with at least 1 possible placement option and then selects one at random. Then all placement options for this piece are found and one is selected at random. In the previously given Algorithm 1 line 2 and 3 are altered to the following lines.

Algorithm 2 The altered lines 2 and 3 of the random player from Algorithm 1.

```
1: Find all tiles with at least one placement option
2: Randomly select one of the tiles in this list
3: Find all possible moves for the selected tile
4: Choose a random move from this list
```

This second variant requires less computing time as it does not need to find all possible placement options for all the tiles and it is not weighted. This is not the case for the first random player where pieces with more possible placements have a higher chance of being picked.

Value Based Return Zero (VBRZ) strategy

The Value Based Return Zero (VBRZ) player serves to highlight the performance of a strategy involving a simple evaluation of tiles, based on the current timestep in the game. This player does however not feature any mechanism for efficiently placing the purchased pieces as it uses a return zero strategy, meaning that the tile it has decided to buy (based on a value calculation described below) it will be placed in the first top most left spot where the tile fits. An example of these placements can be seen in Figure 6 where the first two moves in a Return Zero game are showcased. This method is based on a ranking at each turn that ranks all the tiles which are purchasable and are able to fit on the player board. We want our purchases to optimize the final score we achieve at the end. This final score is based on the number of buttons earned and the number of open spaces left at the end. This leads to Equation 2 which can be used to give guidelines about the number of points a certain tile will give us upon purchasing it at that point in the game:

$$\mathit{Value}(p, t(a), a) = 2 \cdot S(p) + S_B(p) \cdot B_L(t(a)) - B_C(p) - \min(T_C(p), 53 - t(a)) - \mathit{SR}(p, a, t(a)) \quad (2)$$

Here a stands for the player, $t(a)$ stands for the timestep of player a (number of steps taken on the “walking board” by player a) and p stands for piece (the tile being evaluated). Next, $B_L(t(a))$ represents the number of button spots left on the “walking board” at the current time step $t(a)$ of player a. $\mathit{SR}(p, a, t(a))$ is the number of required skips for player a at timestep $t(a)$ to be able to purchase piece p. This will be 0 if the player has enough buttons, but if the player needs 2 more buttons, they can earn these by skipping 2 spaces and thus this is accounted for in the value of the tile (for an example see Figure 4). For the other symbols see Section 3.1. The equation starts of

with a positive value for the size of the tile as open spaces left on the board negatively impact the final score. Next is the positive value that is based on the number of buttons the piece will award the player if it is purchased at this time. As tiles award a player a number of buttons equal to the number of sewn in buttons on the tile at each button spot the player passes (the amount of button spots still left for a player is based on their position on the "walking board" $t(a)$). Button cost is subtracted as each button left at the end is also worth a point. Each time step is also at least worth a single button as a turn could have also been used to move on the "walking board" to skip spaces in exchange for buttons, thus time cost is also subtracted from the value. Purchasing a piece while the player is not able to pay the full amount of time is allowed (this can only happen when purchasing a tile would send them past the end of the "walking board"). For example, if the player only has 1 time left (at timestep 52) then purchasing a piece with a time cost of 6 is allowed and this is accounted for by the $\min(T_C, 53 - t(a))$ part of the equation. Lastly, if the player does not have enough buttons they will need to skip enough spaces to gather funds and these required steps are subtracted from the value.
This formula gives a very rough overview of the presumed value of pieces at certain times, but due to the complexity of the game simply purchasing the piece with the highest return value does not give the best overall score.

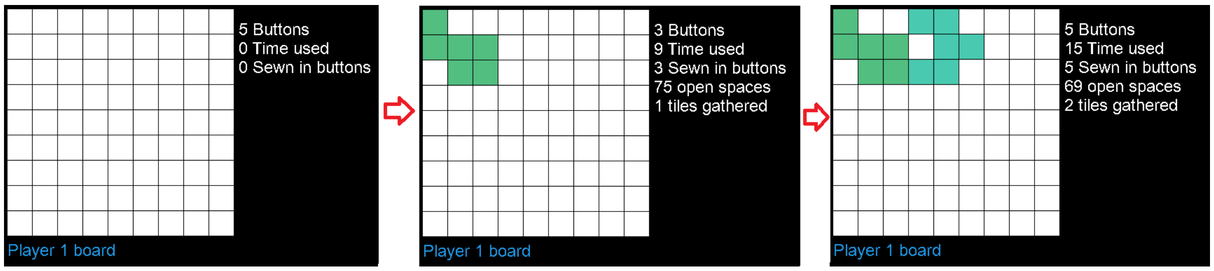


Figure 6: A diagram with three game states depicting the placement selections of the Return Zero player that simply places the piece in the most top left position.

Rule-based placement and improvements to VBRZ

The rule-based placement player is an improvement of the VBRZ player and is meant as a baseline to compare the more advanced methods against. It improves the quality of the value calculation which can also be utilized for the VBRZ player. This player uses a strategy that most resembles the thought process of a human to fill their board and choose their pieces. This player selects its pieces by assigning each piece a value at that point of time in the game and it will then choose the piece with the highest value, just like the VBRZ player. For placing the tile it will try to make as many borders of the newly bought tile connect with other pieces and borders of the board. An example of placement of tiles selected by this rule-based placement player (with $\mathit{TimeCostModifier} = 1$) can be seen in Figure 7. This figure also has the connected edges highlighted in red to visualize how the decision for its placement came to be. When comparing Figure 7 with Figure 6 one can easily notice that the placement selections are more compact, which is more like the way in which a human player approaches the game.
The value assigned to the pieces is based on a number of factors mentioned in Section 4.1 and a new parameter:

- *TimeCostModifier* (T_{CM}): A modifier that can change the weight of the time cost of a tile in the value calculation.

These attributes are then used to determine a value per piece (p) using the following formula:

$$\text{Value(p, t(a), a)} = 2 \cdot S(p) + S_B(p) \cdot B_L(t(a)) - B_C(p) - \min(T_C(p), 53 - t) \cdot T_{\text{CM}} - SR(p, a, t(a)) \cdot T_{\text{CM}} \quad (3)$$

This value calculation does however not take into account that finishing the puzzle (entirely filling the 9×9 grid) awards the player with 7 bonus points, but we can account for this by using a slightly altered final formula. This formula uses the $Value(p, t(a), a)$ calculation from Equation 3:

$$TrueValue(p, t(a), a) = \begin{cases} Value(p, t(a), a) + 7, & \text{if } OBS(a) - S(p) = 0 \\ Value(p, t(a), a), & \text{otherwise} \end{cases} \quad (4)$$

Here $OBS(a)$ is the number of open board spaces on the player board of player a. Using this formula we can get an estimation of how this tile will contribute to our final score and this can rank the tiles in our current state to decide what tile to purchase.
For example, suppose a tile is of size 6, has 3 sewn in buttons, has a button cost of 5 and a time cost of 4. The value of this tile is not static, so for this example we will say that we have enough buttons to purchase the tile, we are at timestep 25, have 5 button spots left to pass and that the tile does not fill the player board. In that case the value assigned to the tile would be:

$$2 \cdot 6 + 3 \cdot 5 - 5 - \min(4,\ 53 - 25) \cdot T_{\text{CM}} - 0 \cdot T_{\text{CM}} = 18 \ (at\ T_{\text{CM}} = 1) \quad (5)$$

These improvements made to the calculation of the value, most importantly the addition of the *TimeCostModifier*, have also been added to the VBRZ player for further experiments. Setting the modifier to 1 and disabling the full grid bonus results in the normal performance of the previously described VBRZ player.

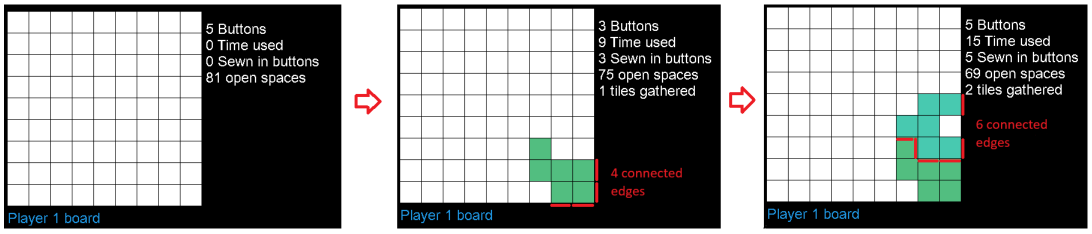


Figure 7: A diagram with three game states depicting the placement selections of the rule-based placement player that attempts to maximize connected edges.

4.2 Monte Carlo strategies

This section features all the implemented Monte Carlo strategies. All of these methods use a number of random games to get a grasp of the outcome of choosing a certain move. For all these random games the random player described in Section 4.1 will be used (the non-weighted random player to be specific). The implemented methods in this section are a pure Monte Carlo player, a greedy pure Monte Carlo player, a Monte Carlo Tree Search (MCTS) player and a greedy MCTS player.

Pure Monte Carlo

Monte Carlo methods use a large number of random playouts to decide which action is at that time deemed optimal. These random games (playouts) are used to give an estimate of how good a certain move is. A pure Monte Carlo Player first finds all of its possible actions (in the current game state) and then for all of these actions it plays a number of these playouts (on a game in which that specific action is taken). These random games are influenced by the action taken before playing the random game and thus they give an estimation of how good it is to take a certain move. Based on the performances of these random games a move will be picked, by choosing the move that had the largest average score of its random games. Given this description Algorithm 3 for the Monte Carlo player consists of the following loop:

Algorithm 3 The loop that makes up the Monte Carlo player in pseudocode.

```
1: while steps taken < total in game time (53) do
2:     Find all moves for all purchasable pieces
3:     if pieces that fit = 0 then
4:         Skip spaces until steps taken = total time in game
5:         break
6:     for all possible moves do
7:         Make a copy of the game
8:         Perform the move on this copy
9:         for the number of playouts do
10:            Make a copy of this copy
11:            Play a random game on this new copy
12:            Update average score of this move based on the outcome of the random game
13:        if average score of move > previous maximum achieved average score then
14:            New maximum achieved average score = average score of move
15:            New best move = move
16:    move To Play = New best move
17:    if #buttons of the player < button cost of selected piece then
18:        Take minimal number of steps required to get the required buttons
19:    Purchase the piece and place it in the decided position
```

This technique heavily relies on the random games and works best with a large number of playouts. A single random game does not give a very accurate idea of how good a move is, but if one takes enough of these random games and averages their results this gives a pretty good estimation of how good the move is. Due to the nature of Patchwork this method does however take quite some time due to the fact that we want to use a lot of playouts to get better results. Furthermore, for each decision on what move to pick, a very large number of moves have to be tried and ideally for all of these moves a large number of random games is played. This large number of possible moves is caused by the fact that each tile can be rotated, mirrored and placed in many different spots which means that the set of all moves ((position,rotation,piece) sets) is quite large.

Greedy Monte Carlo

A greedy Monte Carlo player is largely the same as the previously mentioned Monte Carlo player in Section 4.2. The key difference between the two is the fact that this greedy player does not choose moves that perform the best on average, it chooses the move with the highest achieved score in a single game.
For example, if a move has a very low average score but a single random game from this move leads to the largest maximum score among all possible moves this will be the move that is picked. This player is very unstable and it relies even more heavily on the random games, due to the nature of only examining the maximum while disregarding all other games of that move. This player performs very poorly early on in a game due to the fact that the random games still have many random moves to perform meaning that the games are not influenced that much by the actual move taken. Taking the average (which is done in the normal Monte Carlo player) would be more reliable at the start because of this. This player does however have the upside that during the final few moves of the game it is most likely better at finding the moves which can actually achieve the highest score. The random games started from a later stage in the game are way shorter, meaning that they give a better insight into the actual scores that can be achieved and thus taking a maximum here greedily could actually work out better than taking the average.

Monte Carlo Tree Search

Monte Carlo Tree Search is a method that is also based on random playouts like the pure Monte Carlo method, but instead of simply performing a certain number of playouts for each possible move, it will build a tree consisting of game states to determine the best move [CWvdH+08]. For each decision of what move to pick a new tree will be built. Each node in this tree consists of a game state. So we start the tree with a node that is the current game state and build from there. Once the tree is fully built by using the process that will be described later in this section, from the root node the child with the highest average score will be selected and this will be the action that is performed in the game. So the algorithm consists of repeatedly building these trees to make its decisions until the game is finished. The process of building these trees consists of performing the following four steps as many times as the number of playouts.

Selection Starting from the root node repeatedly select successive child nodes until a leaf node (a node that has not yet had a playout or a game state from which no new moves can be made) is reached. This selective process is guided by the following Upper Confidence bound for Trees (UCT) selection equation:

$$UCT(n) = \begin{cases} \frac{w_i}{n_i}(/M_S) + c\sqrt{\frac{lnN_i}{n_i}}, & \textbf{if } n_i > 0 \\ \infty, & \text{otherwise} \end{cases} \tag{6}$$

In this equation n is the node for which the UCT value is being calculated. Next, w_i normally stands for the number of wins for the node considered after action i, but Patchwork works using a scoring system, thus here it is the total accumulated score for this move. Next, n_i is equal to the number of visits to the node, that being the number of simulations started from this node or its children; M_S is a modifier that is based on a presumed Max Score (a value of 85 is used in our experiments) in the game, which is implemented to decrease the range of values derived from the

$\frac{w_i}{n_i}/M_S$ part of the equation. Normally UCT calculations are meant for a game in which one either wins or loses, so the result from $\frac{w_i}{n_i}$ is then between 0 and 1. To normalize our results to be close to a range between 0 and 1 this modifier is implemented to keep the results in a smaller range. Do note that scores can be negative meaning that not all scores modified by this value fall between 0 and 1, but the values seem to be close enough as to where some negative values do not impact the selection in a harmful way. Without this modifier exploration is greatly diminished which could be a bad thing, but the modifier is in between brackets as using it led to too much exploration which used too much memory on the real game, meaning that we could not use it during experiments on the real game. However, for a simulated version described in Section 4.3 we were able to use the equation including $/M_S$ due to the smaller search space in the simulated version. Furthermore, c is a constant which is the exploration parameter, most commonly taken to be $\sqrt{2}$, which is also the value we use. Finally, N_i is the total number of visits of the parent node of the node i being considered. This second part $c\sqrt{\frac{lnN_i}{n_i}}$ prevents the selection process from being entirely greedy and steers selection towards some options which have not been tried often, but do seem promising. In the case where a move has not yet been tried, the UCT value will be infinite to assure that it is tried first to encourage exploration.

Expansion The selected node now forms all of its potential children. In the PATCHWORK tree this means that the node forms a child for all possible actions from that node and randomly selects one of them for the next step.

Simulation From the newly formed child perform a playout. The newly formed child is in a game state that has been reached by the parent node taking the action that led to the child. This playout to be performed is a random game as described in Section 4.1.

Backpropagation Based on the score of the random game update the total accumulated score and the number of visits for each node on the path from the child towards the root.

To get a better overview of these four steps a visual overview is given in Figure 8. This figure contains made up (UCT) values as they are only there to give an example.

Greedy Monte Carlo Tree Search

Just as done with the pure Monte Carlo player we can adjust the algorithm to be a greedy variant. This is done by keeping track of the maximum score achieved in each node. This maximum score, just as the normal scores in MCTS, is based on the maximum achieved by any of its children, their children, etc. This is achieved by also updating the maximum score of the nodes that are being traversed in the backpropagation step. Once the tree is fully formed we simply select the child of the root with the highest maximum achieved score and that is the action that will be taken.

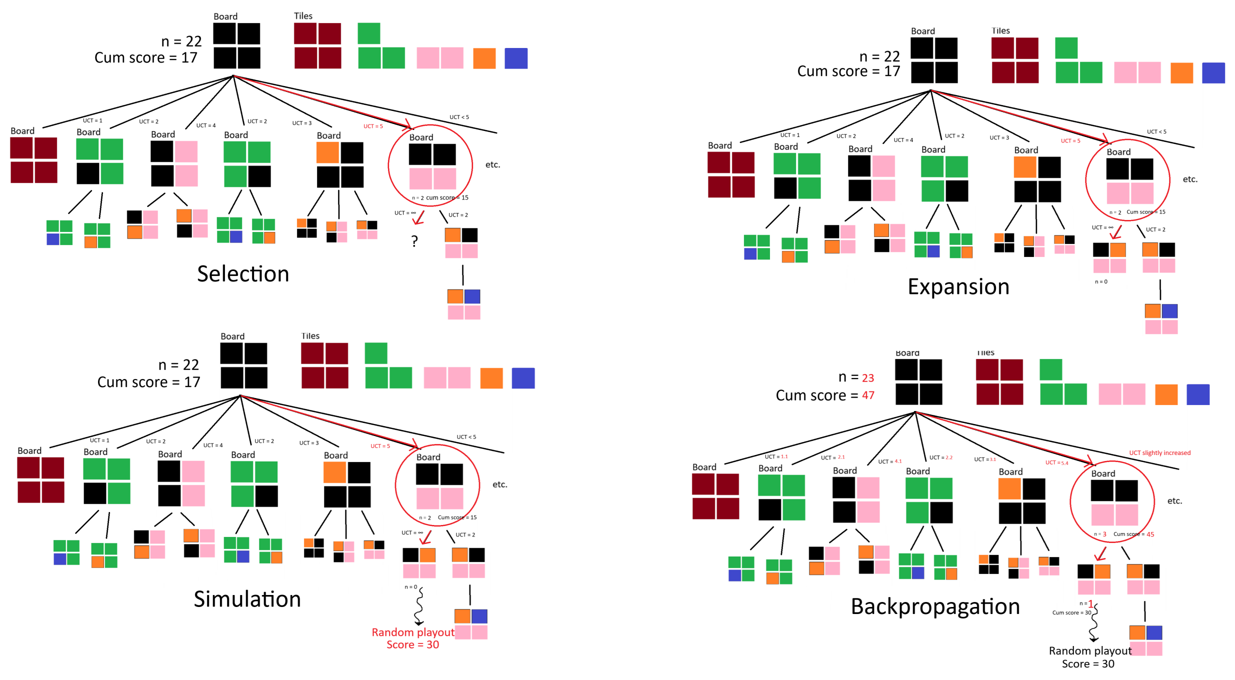


Figure 8: A diagram showcasing the four steps in the main loop of Monte Carlo Tree Search.

Optimization

The Monte Carlo methods all have in common that more playouts give a better overview of what to expect, thus it is better for its performance. Having a very large number of playouts will most likely be ideal, but we are limited by the runtime. This is why a number of techniques to reduce the runtime have been implemented of which the biggest impact will be felt on the pure Monte Carlo methods, as their main limiting factor is runtime.
Firstly a rather simple step that reduces the computing time of the first move is implemented. Each tile has many placements at the start and it can be rotated and mirrored, so for almost each spot a tile can fit in 8 different ways (4 rotations times 2 due to mirroring (except situations where tiles have some symmetry)). The ability to rotate and mirror tiles is necessary to efficiently fill the board later on, but for the first tile it is entirely irrelevant as one can imagine that all of these rotations or mirrorings can be performed by pretending the board is rotated or mirrored. As all tiles after this do have the ability to be mirrored and rotated it does not impact the solutions found in any way and cuts the runtime of the first move by a factor of (nearly) 8.
Parallelization is a technique that is based on performing multiple tasks simultaneously. For pure Monte Carlo parallelization can greatly improve performance, as all playouts can be performed in parallel and thus almost reducing the runtime by a factor equal to the number of used of threads. However, for MCTS this is a bit more difficult. Since ordinary parallelization is not possible here as the entire sequence of actions has to be sequential as the choice of what node to expand requires information from the previous move. There are, however, some techniques which can be used to parallelize some parts of its execution such as root parallelization, where multiple trees are being built at once and the final decision is based on the result of all the trees. It is also possible to simultaneously build some parts of the tree as not all parts being built depend on each other. This would in theory speed up the MCTS algorithm as it reduces runtime, but for our implementation

of Patchwork the main limiting factor is memory. The game tree is so large that the number of possible playouts is heavily limited as the branching factor is simply too large. For the first action around 1700 moves are possible (because this takes into consideration that no rotations or mirrorings are performed) and for the second action around 10,000 moves are possible, and after this the number of possible moves decreases with each layer of the tree (due to more spaces being filled). This branching factor is simply too large and there has to be something done to realistically efficiently use the technique.

MCTS tree size reduction & heuristic

The tree being too large limits us to a UCT selection process with limited exploration which is possible by removing the $/M_S$ part from Equation 6. This will steer it more towards exploitation, thus reducing the size of the tree.
Another technique that reduces the tree size is the use of a heuristic. A heuristic can limit what tiles are able to be selected at a certain time by not allowing all moves to be picked. If we were to only allow a selection of the tiles to be picked at certain times the width of the tree would decrease drastically. This would allow for more playouts and overall better performance, as long as this selection of tiles is picked well. The value based VBRZ and rule-based placement player feature a calculation for the value of a tile at a certain time so we can re-use part of Equation 3. It does need some minor adjustments, namely the removal of the required skips (SR) attribute as we want the heuristic ranking to be static to keep runtime low. Including the number of required skips would result in a calculation for each use of the heuristic which we try to avoid. Thus the equation for the values here is:

$$Value(p, t(a), a) = 2 \cdot S(p) + S_B(p) \cdot B_L(t(a)) - B_C(p) - \min(T_C(p), 53 - t) \cdot T_{\text{CM}} \tag{7}$$

To keep us from having to re-calculate many of these values we use a database containing the heuristic values (obtained using Equation 7) to not cause a negative impact on runtime as it now only requires one lookup of an array value. This database is of size $Numberoftiles \times Numberofpossibletimesteps$, where only the timesteps where one could purchase a tile (0–52) are relevant and only the purchasable tiles are considered(0–32). This database contains the ranking of each tile at each specific timestep and thus allows us to limit what tiles are able to be picked based on their ranking.
For our implementation we have decided that at each timestep the better half of the purchasable tiles will be accessible to use. It is worth noting that later on in the game most of these tiles will have been used so the exact number of tiles we allow to be used is: $Totalnumberoftiles/2 + numberofownedtiles$. So only tiles with a ranking $\leq Totalnumberoftiles/2 + numberofownedtiles$ will be allowed to be picked at any time. This will still limit selection later on as the top ranking tiles have already been purchased at this point, so the extra allowed tiles will still limit the total selection of tiles to around $Totalnumberoftiles/2$. This selection process later on includes a bit more than half of the remaining tiles in the game as the $numberofownedtiles$ is used which also counts the given 1×1 tiles. This is done so that tile 0 (see Figure 2) is still allowed to be picked at the end of games as it is crucial for filling the entire board in most games and it would be excluded at certain points, where it would be a good pick without this modification.

4.3 Simulations

In order to get an idea of an upper limit for the possible scores within the game an altered version of the game is also implemented. This version does not feature a player board and thus reduces the amount of calculation time by an exponentially large factor. The amount of space is still limited by a maximum of 9×9 available spots to fill, but the shape and placement of all tiles is disregarded (their size is still highly relevant). Where there were previously around 1700 moves possible for the first action this number is now reduced to 33 (the number of purchasable tiles in the game). After the first action there are then 32 possible moves left (instead of around 10,000 moves in the real game) and this number keeps decreasing by 1 with each action until the board is nearly full where less moves will be possible as we can still not fill more than 9×9 spaces (tile size needs to be $\leq$ open board spaces). This means that all games within this simulation are much quicker and the number of possible playouts is much larger leading to the opportunity to do more extensive testing. To increase the efficiency even further the 1×1 tiles do not have to be placed here and the number of open final spaces will thus be decided by $\max(OBS(a) - 5,\ 0)$. This simulated version has all methods implemented, that being: the random player, VBRZ and rule-based placement player (which are equal here as they only differ in placement), pure Monte Carlo and MCTS. These simulations can be used to get an upper limit, but it does not guarantee that the selected tiles can all fit together to actually properly fill the player board. With the five 1×1 tiles it is very likely that almost all configurations are possible, but we do not have proof for this. We did, however, examine a number of games from the simulated version and found that, for the examined games all 30 of them had a tile selection that was able to fit on the 9×9 board. These 30 games were selected based on the tiles they pick to include a wide range of used tiles (and all of them filled every space on the board). It is possible to check whether the collection of tiles is able to fill a board by using a polyomino placement problem solver [Mea].

4.4 Reinforcement Learning

As mentioned in previous sections, a good approach to improve overall scores and to reduce runtime is the reduction of the number of moves that are playable at a time. We have already done this statically by applying a heuristic (as explained in Section 4.2) that limits the selectable tiles. This method is static, but another approach that can be used to reduce the allowed moves is a dynamic approach, which is why we will be using a Reinforcement Learning technique.
Reinfocement Learning is an area within machine learning that does not rely on large datasets created beforehand. Instead, it learns all of its behavioural patterns dynamically during runtime. For our case we will dynamically learn what tiles to exclude from the games we play. This reduces overall runtime as there are fewer tiles present in the game, meaning that each lookup of all the possible moves will be faster. For the Monte Carlo methods the smaller selection of tiles also means that there are fewer possible actions to try, meaning that fewer random games have to be performed. So we want a way to dynamically choose what tiles to exclude based on results gathered during runtime.
We start off by playing a number of games with all tiles present in the game. We play these games and for each tile we keep track of the number of uses it got and the cumulative score of all games using that tile. After a number of games we then perform what we will call a "learning update". A learning update uses the data derived from the previously performed games to make an

educated decision of what tiles to exclude going forward. During this learning update we will rank all tiles and based on this ranking we will then remove a number of tiles equal to the parameter *TilesToRemoveperlearningupdate*.

This ranking process goes as follows: firstly we find all tiles with zero uses (if any), meaning that they have not been used in a single game. For these we unfortunately do not have data about their performance in real games so we have to rank these based on a heuristic value. For this we use a modified version of Equation 7. As we are not at a certain timestep we have to remove parts of the equation requiring the information of the current timestep. This leads to the following equation:

$$Value(p) = 2 \cdot S(p) + S_B(p) \cdot 4.5 - B_C(p) - T_C(p) \cdot T_{\text{CM}} \tag{8}$$

The main changes in this new equation are the fact that the $B_L(t)$ variable from Equation 7 is replaced with a static value (4.5) which is close to the average number of button spots left on the "walking board" during the span of the game. The other change made to the heuristic equation is the removal of the reduced time cost which was relevant if the player bought a tile with a time cost that would make them pass the final space (53) on the time board. This does not happen often, thus no changes were made other than the removal of this factor.

Now that all the unused tiles are ranked we remove the lowest ranked tiles until we are at *TilesToRemoveperlearningupdate* removed tiles. If there are enough unused tiles to do this, then this learning update is done and we continue playing games until the next learning update is reached after playing a number of games. If there are fewer unused tiles than *TilesToRemoveperlearningupdate* then we will remove the used tiles that seemed to perform the worst in actual games. We rank all used tiles based on their $\frac{Cumulativescore}{uses}(+uses)$ and then eliminate the worst ranking tiles until *TilesToRemoveperlearningupdate* tiles have been removed. The part of the equation "$+uses$" is in between brackets as we will experiment with it enabled and disabled. It is there to prevent tiles that are in nearly every game from being removed in favour of a tile that happened to be in a single game which went well.

This process of playing a number of games and then performing a learning update repeats until we reach the limit of *TilesToRemove* at which point no learning updates will be performed anymore. We do not want to end up removing too many tiles as there have been games where 17 tiles are purchased. We want to be on the safe side, so a limit of 13 tiles to remove is set (the *TilesToRemove* parameter). This still leaves these games with 20 tiles left to pick from, meaning that this should not lead to scenarios where all available tiles are purchased in a game.

4.5 Perfect finish

We have implemented an option to brute force the puzzle from a certain point. This means that all combinations of possible options from that point onward will be tried guaranteeing that the optimal solution from that point on will be found. The width of the game tree is, however, very large, so this can only be properly utilized for the last few moves.

4.6 Combined player

To attempt to reach a score as high as possible we use a mix of multiple techniques mentioned in the previous Section 4.1 and Section 4.2. To reach a high score we have to strike a good balance between exploration and exploitation as we can not simply exhaustively go trough the huge search

space of the game. This means that the start should focus mainly on setting up a strong consistent start for the player by purchasing tiles that perform well over a number of games and this is why the standard Monte Carlo player is used at first. The Monte Carlo player should in this case base its decision on the average returns of the games because the greedy Monte Carlo player gives a very poor idea of what actual good moves are at the start.
After a certain number of actions we are far enough into the game so that we can start using MCTS at a proper number of playouts. This means that the number of playouts is actually larger than the number of possible moves at that point. Once the number of playouts is around 2 to 3 times the size of the number of possible moves MCTS is a better fit than pure Monte Carlo. This is the case, because MCTS more efficiently uses its playouts on games with more potential and wastes less time on moves that are sub-optimal (instead of pure MC which spreads all of its playouts equally among all possible moves).
After the search space becomes small enough we use the perfect finish, so the entire search space will be analyzed and this ensures that the last few moves in the game will always be played perfectly.

5 Experiments

To assess all the implemented players/strategies a number of experiments have been conducted. For each experiment the experimental details will be reported in their relevant sections. The game has been implemented in C++ and experiments have been performed on the following hardware: an AMD Ryzen 9 5900X processor, with a clockspeed of 3.7GHz, 12 processor cores and 24 threads. The memory consists of 32 Gigabytes of DDR4 RAM with a speed of 3600MHz. All experiments have been performed on Windows 11 (version 23H2) using WSL2 with Ubuntu (version 22.04.1 LTS).

5.1 Different methods

We compare all the implemented methods mentioned in Section 4. This includes all the simple strategies in Section 4.1 as well as the Monte Carlo methods in Section 4.2. In some experiments the simulated version of the game from Section 4.3 will also be utilized.

Random players

The random player has two distinct implementations as mentioned in Section 4.1, the weighted and non-weighted player. We have also included the random player playing the simulated version of the game (the version without the board, see Section 4.3). We test these with 10,000 repetitions each. The gathered data can be seen in the following table:

Method	Av S	Sd S	Mn S	Mx S	Av O	Sd O	Mn O	Mx O	Av T	Sd T	Mn T	Mx T	Av B	Sd B	Mn B	Mx B	Av R (s)
weighted	-6.62	11.65	-54	29	18.29	3.93	6	36	18.53	0.93	14	21	13.09	2.33	4	21	0.0257
non-weighted	1.90	11.81	-45	41	16.33	3.96	3	37	18.21	1.06	13	21	13.37	2.62	4	23	0.0088
simulated	12.88	16.02	-50	56	13.90	7.33	0	42	18.35	1.41	13	23	15.48	2.31	13	23	0.0003

Table 1: Table containing the results of 10,000 random games performed with the weighted, the non-weighted and the simulated random player. Here S stands for Score, O stands for Open spaces, T is the total number of Tiles acquired during the game, B is the number of sewn in Buttons on the player board and R is the runtime. Av is average, Sd is standard deviation, Mn is minimal, Mx is maximum.

Table 1 shows that the weighted random player performed quite poorly, which is explained by a number of factors. The number of open spaces is on average quite large with an average of 18.5, resulting in an average of 37 score lost (see Equation 1). This is explained by the fact that the weighted random player is unable to balance its economy during a game as randomly picking tiles leads to a small number of Sewn in buttons, which means that the button spots are not utilized that well. This leads to fewer tiles being purchased as more time has to be spent skipping spaces on the "walking board" in order to even purchase tiles.
The non-weighted random player performs significantly better than the weighted player while taking much less runtime. The runtime reduction is due to the fact that the non-weighted variant does not need to find all possible moves for all tiles. It has an equal probability to pick any tile that fits at any point which helps with overall scoring, as it does not have an increased probability to pick the smaller tiles which the weighted version does have. In general larger tiles are more expensive in both time and button cost, but they offer many more sewn in buttons on average as can be seen

in Figure 2. The larger tiles also cover more spots and Table 1 reflects this by having a smaller number of open spaces on average for the non-weighted player. The number of tiles purchased is a bit smaller than that for the weighted player, because larger tiles are generally more expensive. This, however, is not a problem for its score as these tiles on average fill more spaces. Furthermore, the larger number of average sewn in buttons compensates for the more expensive tiles since less time has to be skipped (exchanged for buttons), which reduces the impact of the tiles being more expensive. All of these factors lead to the non-weighted player scoring higher on average.
Lastly the simulated player performs significantly better than both real players. The simulated random games are not weighted as no placement options have to be considered. The number of spots to fill is still restricted to the total number of spaces on the player board, that being 9×9 (the board size). This simulated player performs significantly better purely because the shapes of tiles no longer impact the game. This leads to larger tiles being able to be picked more frequently, which was not the case for the real game players, as these also had to deal with placement restrictions. This increase in larger tiles being able to be picked later on in the game led to a significantly larger number of spaces filled and sewn in buttons. This leads to a better economy during the game, meaning that more tiles could be bought, leading to a larger score.

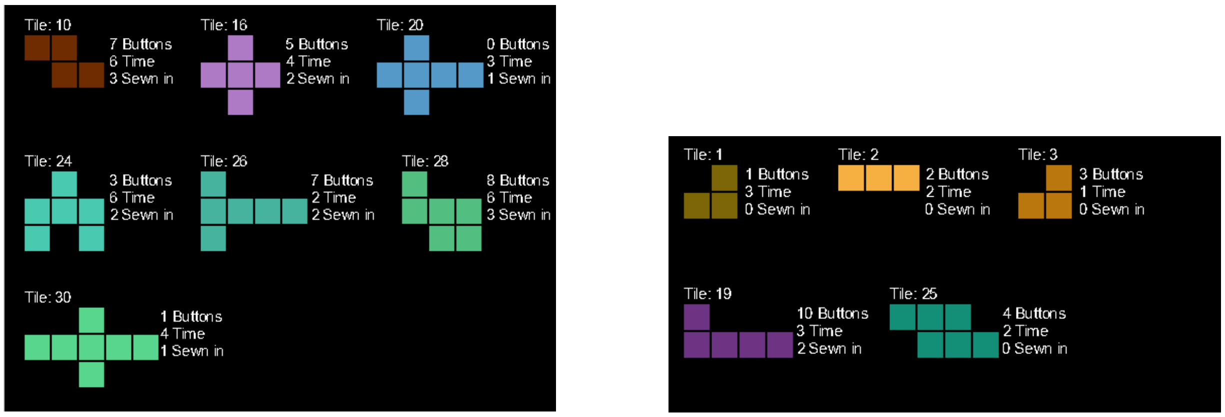


Figure 9: Two figures showing the best and worst performing starting tiles in the various random games.
On the left: the tiles that were in the top 5 of one of the three types of random games. On the right: the tiles that were in the bottom 5 of one of the three types of random games.

Table 2 shows the impact of the first tile on the average score achieved in those random games. The table shows the 5 best and 5 worst performing starting tiles per random player. The best tiles are the same for the most part, with the total selection only containing 7 different tiles with their precise characteristics shown in Figure 9,left. These tiles are all strong starting tiles and have a few things in common. They are almost all quite large and have a large number of sewn in buttons, with the major exception to the large number of sewn in buttons being tile 10 and 20. These tiles may only have a single sewn in button, but their cost is very low (even zero for tile 20) and they are relatively big especially for this low cost, meaning that it is a great starting pick. Overall these starting tiles show that it is very important to start off with tiles that are large, cheap and feature a lot of sewn in buttons to help with the players' economy during the game.
On the other hand, the worst starting tiles of Table 2 as seen in Figure 9,right seem to have less in common when looking at them. Tiles 1, 2 and 3 all have zero sewn in buttons, which explains why

they are a bad pick at the start, along with the fact that they are small, meaning that they would have been better utilized later on in the game to fill spots where other tiles can no longer fit. Tile 25 is big, but it has no sewn in buttons and leaves the player in an unfortunate position, as picking the tile at the start places them at timestep 2 with only a single button left. As for tile 19, it has 2 sewn in buttons, which is quite good, but picking it as ones starting tile simply costs too much time as one needs to skip 5 spaces first and purchasing it lands the player at timestep 8 with only a single tile.

weighted:

tile	avg Score
20	4.30
28	3.34
10	3.32
26	1.49
24	1.40
...	...
1	-13.23
19	-13.54
2	-14.28
3	-14.95
25	-17.45

non-weighted:

tile	avg Score
20	12.47
28	10.60
10	9.77
24	9.18
30	9.11
...	...
2	-4.68
1	-5.91
19	-6.09
3	-6.83
25	-8.40

simulated:

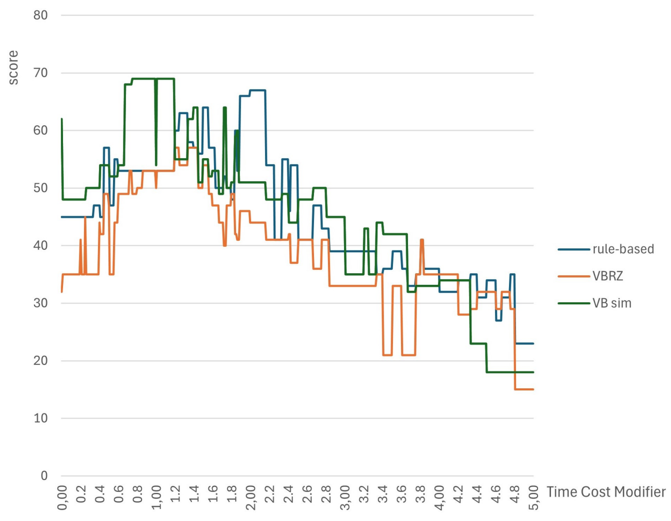
tile	avg Score
20	25.83
24	22.60
30	21.95
10	18.60
16	18.08
...	...
19	5.48
1	5.27
2	5.24
3	4.68
25	1.66

Table 2: From left to right the weighted, non-weighted and the simulated random players with the average achieved scores when purchasing the tile as the first purchase. The table contains the 5 best performing first picked tiles and the 5 worst performing first picked tiles. Results are gathered from 10,000 random games per method

Value Based players

The two value based methods, VBRZ and the rules-based placement player, both choose their tile based on the value calculated from Equation 3. Both of these players are deterministic in contrast to all other utilized methods, so there is no need for a large number of repetitions. The only change in behaviour is caused by altering the *TimeCostModifier* which will be tested in this experiment. Alongside the two real players a simulated variant of the Value Based player is tested. This player does not need to place tiles in any way so the tile selection solely relies on the value of tiles, while still being limited by the restriction preventing the player from filling more than 9×9 squares. The results have been given in two graphs where the first depicts the acquired scores at different values for the *TimeCostModifier* and the second depicts the number of open spaces at the end of the games.
The graphs in Figure 10 show that the *TimeCostModifier* (TCM) has a very large impact on the achieved score and spaces filled. The score graph shows very large spikes in score which is explained by the fact that a small change to the TCM can lead to a single different action being taken which changes the entire game drastically. For all three methods the optimal values seem to fall in the [1,2] range. This is mostly explained by the fact that taking a value of 1.0 precisely mimics the exact score achieved by purchasing the tile at that time, meaning that it is a pretty good estimation, which is why the optimal values are close to it. For both the rule-based placement and normal VBRZ players it appears that values a bit larger than 1 proved to be more effective, which is explained by the fact that this decreases the value of tiles that cost much time, and this results

in the player using their time more efficiently.



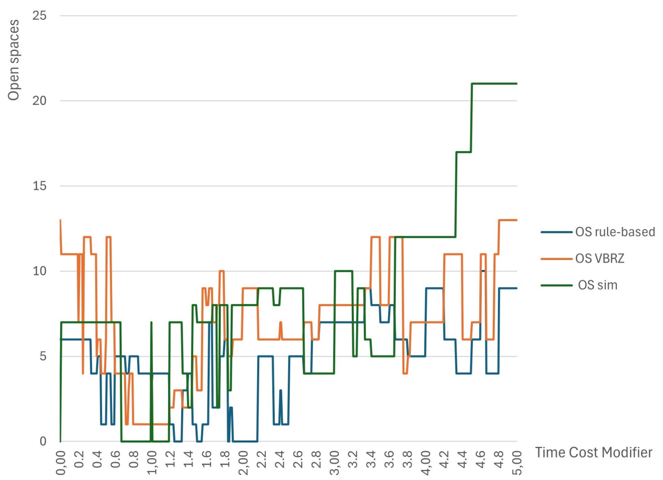


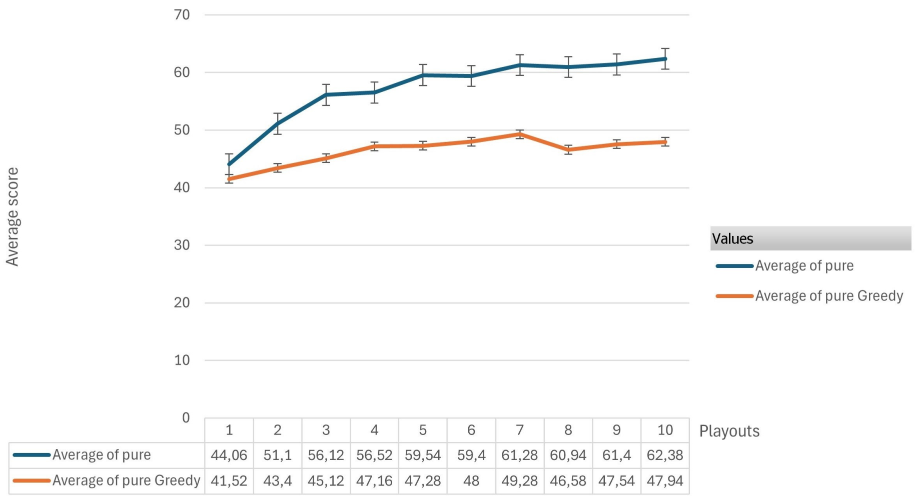
Figure 10: For both figures the methods used are the Value Based Return Zero player, the rule-based placement player and the Value Based simulation. All increments of the $TimeCostModifier$ are in steps of 0.01. On the left: a graph depicting the score obtained for various values of the $TimeCostModifier$ on the horizontal axis and the score on the vertical axis. On the right: a graph depicting the number of open board spaces for various values of the $TimeCostModifier$ on the horizontal axis and the number of open board spaces on the vertical axis.

For the VBRZ player the optimal TCM was 1.4 with a score of 57. This maximum score is quite a bit lower than the maximum of the other two methods due to the fact that the placement strategy of the VBRZ player never managed to fill all spaces, thus missing out on the 7 bonus points. This also explains why its optimal TCM is not at its lowest point on the open spaces graph as the lowest point was 1 here, while at the optimal TCM there are 2 open spaces. This single extra open space reduces the score by two, but the tile selection at $TCM = 1.4$ is better even though it has an extra open space.
For the rule-based placement player an optimal TCM of 2.0 achieved an impressive score of 67. The rule-based placement player did manage to fill the board for several TCM values due to the tile placement strategy having a more compact configuration that proved to be more effective than the VBRZ placement strategy. This more compact strategy is less restrictive on what tiles are able to be picked later on in the game, meaning that the tiles with the best value can be picked longer without being hindered by the board being too full.
For the simulated player a TCM around 1.0 proved to be the most effective with a score of 69, however at exactly 1.00 a large dip is made with only a score of 54 and 7 open spaces. This dip is caused by only a 0.01 change in the TCM as the scores for 1.01 and 0.99 are both 69 with the same action sequence. When looking at the sequence at 1.00 we found that the third tile selection changed which caused a slew of different decisions that led to a much worse score. A large TCM proved to be worse here while the other two players worked better at a larger TCM. It is most likely that the action sequence for the other two players at TCM 1.0 led to a certain tile placement

that prevented a good tile from being picked afterwards which is not a problem here as there is no board restricting our tile selection.
For the original VBRZ player without the improvements made to the value calculation the score and behaviour are exactly the same as in Figure 10 at $TCM = 1.00$ and it achieves a score of 50, showing that the *TimeCostModifier* makes a big impact on the achieved score.

Pure Monte Carlo players

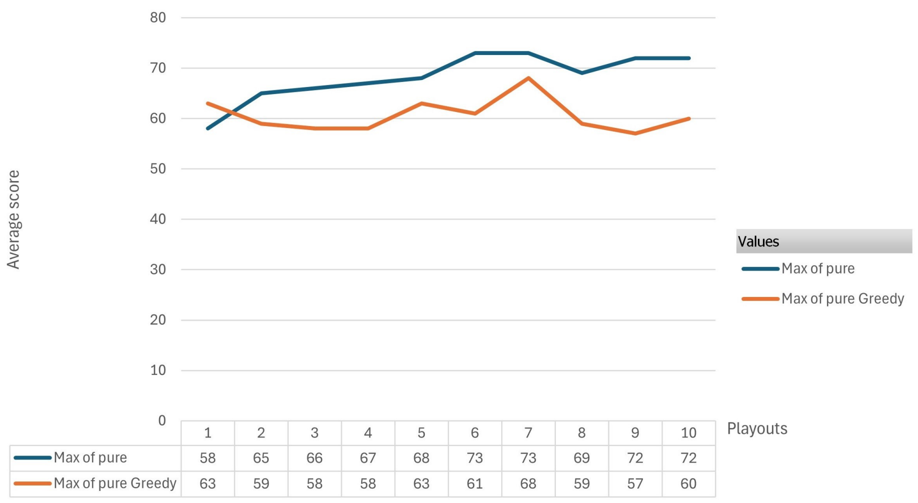
For the pure Monte Carlo (MC) players the main factor impacting their performance is the number of playouts performed. A larger number of playouts means that more random games have to be played leading to longer runtimes per game, but the approximations of what tiles are good will be better, because the results are based on more random games. In this experiment we compare the performance of the greedy and the normal pure MC player by looking at the performance of 50 repetitions for each number of playouts from 1 to 10. We also looked as the normal pure MC player with larger playouts ranging from 20 to 100 playouts in steps of 20.



	1	2	3	4	5	6	7	8	9	10
Average of pure	44,06	51,1	56,12	56,52	59,54	59,4	61,28	60,94	61,4	62,38
Average of pure Greedy	41,52	43,4	45,12	47,16	47,28	48	49,28	46,58	47,54	47,94

Figure 11: The greedy and non-greedy pure Monte Carlo players with their average achieved scores at different numbers of playouts. Results were gathered from 50 repetitions for each number of playouts. Bars along the lines depict the standard deviation at that point. Right below the graphs are tables depicting the exact values in the graph.

When looking at the graphs in Figure 11 and Figure 13 one can clearly see that overall a larger number of playouts leads to substantially better average performance. This is explained by the fact that a smaller number of random games leads to an overall worse and more noisy prediction of the impact caused by a tile being purchased. The greedy variant seems on average to perform quite a bit worse than the player that uses the average rather than the maximum. For playout count one the two are functionally exactly the same and the score difference is still about 2.0, which shows that the data is a bit noisy, but when looking at both graphs in Figure 11 and Figure 12 it is clear that the difference between the two is large enough to confidently say that the greedy player is worse. The greedy player also improves less with a larger number of playouts, which is explained by the fact that its approximation of what a good move is only gets slightly better at higher playout counts. Taking the maximum of a number of random games gives an improved estimation, but it is still a pretty poor estimation due to the nature of the random games.



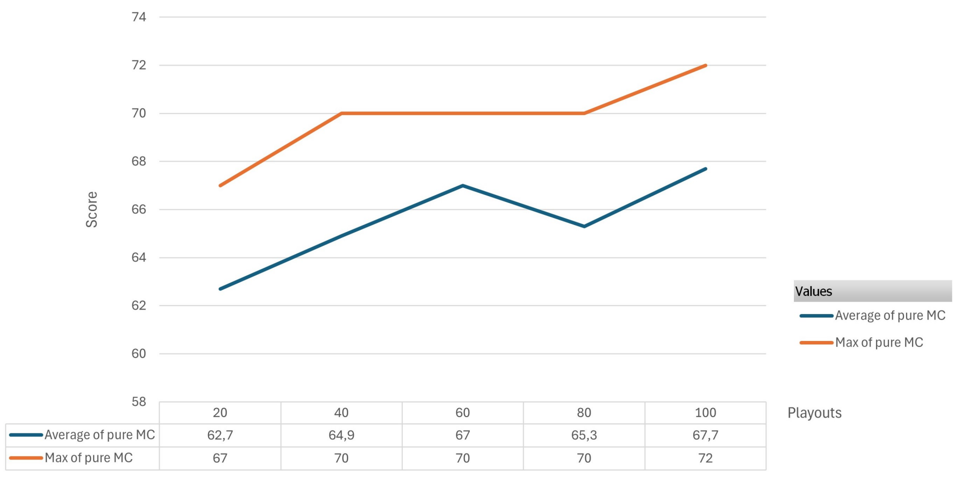
	1	2	3	4	5	6	7	8	9	10
Max of pure	58	65	66	67	68	73	73	69	72	72
Max of pure Greedy	63	59	58	58	63	61	68	59	57	60

Figure 12: The greedy and non-greedy pure Monte Carlo players with their maximum achieved scores at different numbers of playouts. Results were gathered from 50 repetitions for each number of playouts. Right below the graphs are tables depicting the exact values in the graph.

When looking at the maximum scores achieved in Figure 12 and Figure 13 it can be seen that in general a larger number of playouts helps, but due to the random nature of Monte Carlo methods sometimes a larger score is achieved at smaller playouts.
Ideally a larger number of repetitions would have been used, but as can be seen in Table 3 the games took very long, meaning that creating these two graphs in Figure 11 and Figure 12 already took close to 40 hours. This limited the number of repetitions we have tested. The greedy runtimes are not included, since these are nearly equivalent as their calculations are exactly the same only differing in selection. The same goes for the games at large playout counts as the runtime increase from more playouts is linear.

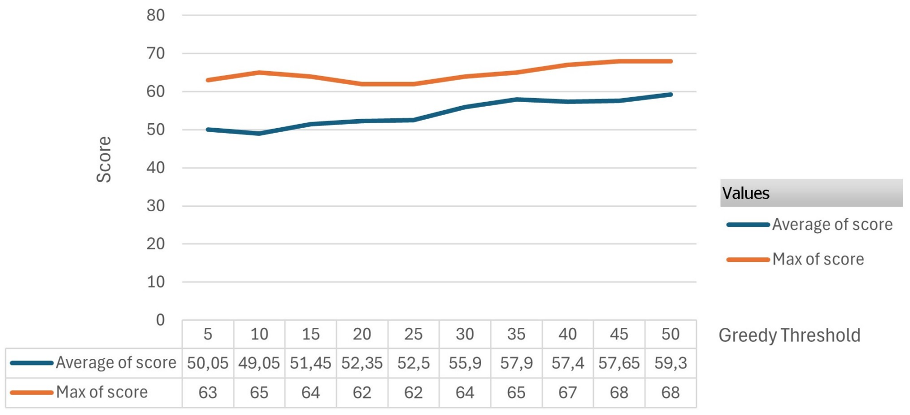
Playouts	1	2	3	4	5	6	7	8	9	10
pure MC runtimes (s)	24.43	50.37	73.38	96.77	125.31	146.47	170.87	195.27	222.47	244.07
pMC non parallel runtime (s)	462.58	965.88	1403.36	1854.46	2318.34	2801.73	3253.11	3700.71	4174.42	4615.89

Table 3: A table depicting the average time of a pure Monte Carlo game at different numbers of playouts. Along with the real used time the time it would take without parallelization is given to show the impact of using multi threading.



	20	40	60	80	100
Average of pure MC	62,7	64,9	67	65,3	67,7
Max of pure MC	67	70	70	70	72

Figure 13: Large playout counts for the pure Monte Carlo player with 10 repetitions each.



	5	10	15	20	25	30	35	40	45	50
Average of score	50,05	49,05	51,45	52,35	52,5	55,9	57,9	57,4	57,65	59,3
Max of score	63	65	64	62	62	64	65	67	68	68

Figure 14: A mixed player that utilizes both greedy and pure Monte Carlo moves (at 5 playouts) with 20 repetitions per greedy threshold.

Mixed greedy and pure Monte Carlo

While the greedy player seemed to perform much worse than the non greedy MC player there is a chance that it could still be useful to help make decisions later on in the game. This is why we have ran an experiment where a greedy threshold is implemented. If this threshold is set at 20, for example, this means that moves made before timestep 20 will be normal pure MC moves and after timestep 20 they will be greedy pure MC moves. This could offer a good hybrid where the start can be taken using the averages of the normal pure MC player and later on transitioning to the greedy choices that could potentially lead to better endings, as the moves that seem to have the largest potential will be picked rather than the on average best performing moves. For the experiment the tests were performed at 5 playouts and 20 repetitions were performed per *Greedy Threshold* value. The graph in Figure 14 shows improvement with each increase in the *Greedy Threshold* parameter, but it never surpasses the performance of the normal pure Monte Carlo player in Figure 11 and Figure 12. This means that this increase in performance is most likely only caused by fewer moves being taken greedily, which is better as the greedy moves are simply worse than the normal MC moves.

Large playouts on simulated pure MC

We have experimented with a larger number of playouts for the simulated version of the game as this would have taken far too long on the real game. The results can be seen in Table 4. When comparing performances with the real game depicted in Figure 13, Figure 12 and Figure 11 it is clear that the simulated version manages to achieve higher scores while at the same playout count. This is explained by the fact that it does not have to place the 1×1 tiles, so these can not be placed sub-optimally and the player is not restricted by the tile shapes other than being limited to 9×9 spots that can be filled. The table seems to support the idea that using more playouts leads to a better score when looking at the averages. The maximum scores do however tell another story (we have seen this previously in Figure 11 and Figure 12), but here something else is the cause. Here it was not entirely caused by pure chance due to the random games. At playouts 1000 and 10,000 the variety of moves picked decreased drastically, which led to very safe games with little variance which got a higher average score, but the maximum score suffered due to this. This got especially bad at 10,000 playouts where 67 out of 100 tests all had the exact same action sequence, which meant that it has become too stable for its own good. This set of moves led to a score of 74, which is good, but higher scores have been achieved at lower playout counts. In this case using an average with enough playouts led to overlearning.

Playouts	1	10	100	1000	10,000
Average score	43.19	66.32	70.63	71.07	72.41
Max score	65	75	78	76	76
Standard deviation	31.35	9.83	5.63	6.10	4.73
Average runtime (s)	0.0048	0.050	0.47	4.32	42.99

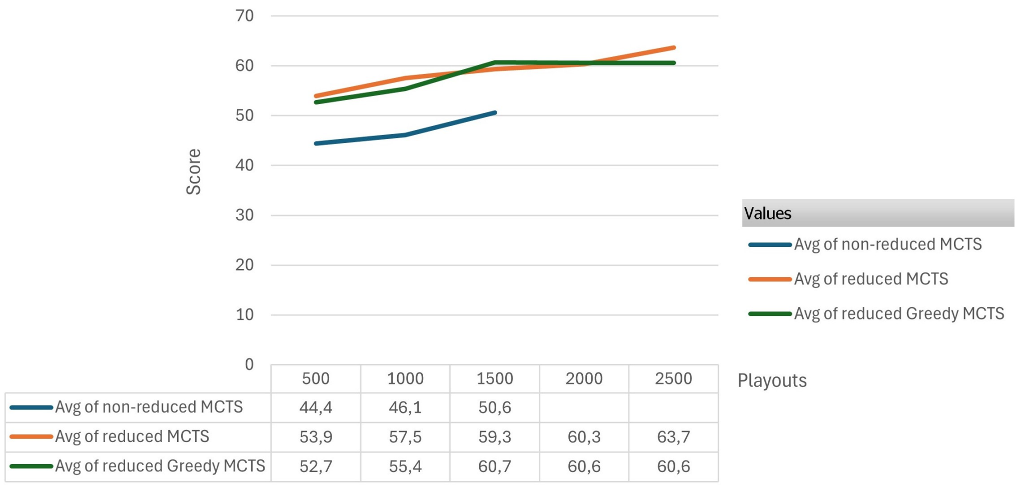
Table 4: A table containing the data for various playouts performed on the simulated version of the game. All results were gathered from 100 repetitions.

Tiles purchased:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Normal game	1693	11228	7688	5579	4119	2960	1394	650	465	164	123	76	53	17	9	2
Heuristic used	822	4298	3357	2559	1938	758	599	350	326	287	120	125	97	42	8	4

Table 5: Table depicting the number of possible moves of two games from the MCTS experiment. One of the games has the heuristic tile selection applied while the other does not.

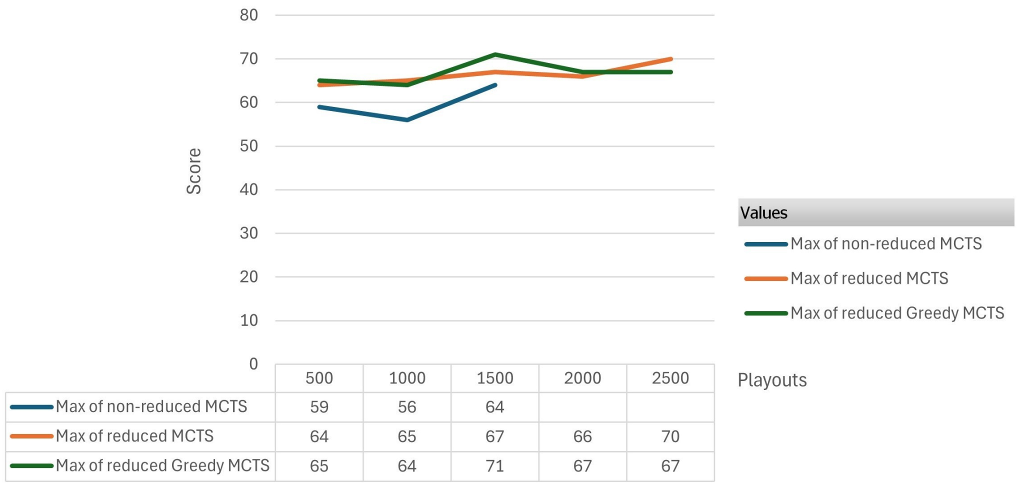
Monte Carlo Tree Search players

For the Monte Carlo Tree Search (MCTS) player the runtime is an important factor, but for our implementation the main limiting factor was memory. The game tree being built simply gets very large quickly which limits our testing capabilities of the method here. To combat this as mentioned in Section 4.2 we have implemented a heuristic to slightly relieve this issue, but we were still only able to test with a limited number of playouts on the real game. To get an idea of why memory is the issue one can see from Table 5 that the branching factor for an example game is very big. For the first move there are 1693 moves and afterwards there are 11,228 moves, meaning that for this example the first two moves already have a branching factor of 1693×11228, which is just above 19 million. Seeing how many moves are still present after this the size of the full tree is in order $O(n!)$. We do manage to drastically reduce this branching factor using our heuristic from Section 4.2, but the branching factor is still quite large and in order $O(n!)$. Lastly, the simulated game will be utilized as these have a way smaller search space, which allows us to test performance at very large playouts.



	500	1000	1500	2000	2500
Avg of non-reduced MCTS	44,4	46,1	50,6		
Avg of reduced MCTS	53,9	57,5	59,3	60,3	63,7
Avg of reduced Greedy MCTS	52,7	55,4	60,7	60,6	60,6

Figure 15: A graph containing the average scores of the MCTS players with either the heuristic applied to tile selection (reduced) or not (non-reduced). Results were gathered from 20 repetitions.



	500	1000	1500	2000	2500
Max of non-reduced MCTS	59	56	64		
Max of reduced MCTS	64	65	67	66	70
Max of reduced Greedy MCTS	65	64	71	67	67

Figure 16: A graph containing the maximum scores of the MCTS players with either the heuristic applied to tile selection (reduced) or not (non-reduced). Results were gathered from 20 repetitions.

Monte Carlo Tree Search

The graphs in Figure 15 and Figure 16 show the results of games with playout counts as high as 2500 (larger playout counts kill the process due to memory restrictions), but for the non-reduced MCTS player the maximum playout count that could be used was 1500 due to memory restrictions killing the process if we were to go any higher. For this experiment we have also had to use a sub-optimal UCT calculation without the M_S from Equation 6, meaning that exploitation is heavily prioritized with nearly no exploration. The graphs show that a larger number of playouts increases the average performance with some minor deviations due to the small number (20) of repetitions performed. Overall the non-reduced player performed far worse which is explained by looking at Table 5. At the playouts used for the non-reduced player most of the time the number of playouts does not exceed the number of moves in that state. When this is the case MCTS is basically a worse performing version of pure MC as this means that for a selection of the possible moves a single playout is performed. MCTS only properly works when more playouts than the number of possible moves from the root is possible as this allows the algorithm to reach deeper branches. This is also the reason that the heuristic is so effective as it gets faster to the point where playouts exceed the total number of moves possible in the state. However, as we can see in Table 5 there are still some moves for the reduced player where it is basically a worse pure MC with a single playout. This is the reason why it does not perform as good as hoped, but despite this restriction some very impressive scores have still been reached as can be seen in Figure 16. So while the method shows great potential, it is simply not viable for the real version of the game with a branching factor this large without drastic adjustments. Only 20 repetitions have been performed per playout due to the long runtime of the methods which can be seen in Table 6. The pure MC methods also had long runtimes, but we could reduce these by nearly a factor 20 which allowed for more extensive testing, but here no parallelization is performed due to the memory constraint.

Playouts	500	1000	1500	2000	2500
Avg runtime non-reduced MCTS (s)	41.11	178.22	267.35	N/A	N/A
Avg runtime reduced MCTS (s)	65.27	224.26	383.68	498.19	646.13

Table 6: Table depicting the runtimes of Monte Carlo Tree Search performed on the real game at various playout counts.

Simulated Monte Carlo Tree Search

The branching factor of the game with placement of the tiles might be too large to test it extensively, but this does not mean that the same holds for the simulated version of the game. In the simulated version at the start there are as many moves as there are purchasable pieces and with each purchase this goes down by one (or sometimes a bit more at the end of the game where the board is completely filled). This means that the game tree is very tiny compared to the real version of the game, thus we are able to test at playouts up to 120,000, at which point the time per game was very long, but the results were the best so far. In this version we can also use the proper UCT Equation 6 with the "$/M_S$ part" of the equation enabled. For this experiment we have tested at varying playouts with either this "$/M_S$ adjustment" to the equation either enabled or disabled.

Playout count	10	100	1000	10,000	20,000	40,000	80,000	120,000
Average score $/M_S$	48.53	58.49	66.27	71.44	73.09	74.65	76.95	77.31
Max score $/M_S$	65	73	77	80	81	82	82	81
Average runtime $/M_S$ (s)	0.098	2.480	15.365	83.772	227.325	589.752	1002.636	1290.100
Average score NO $/M_S$	47.84	68.56	71.98	72.07	72.36	72.70	71.97	72.19
Max score NO $/M_S$	66	77	78	78	79	78	78	77
Average runtime NO $/M_S$ (s)	0.088	1.185	8.255	53.129	127.588	294.092	467.834	597.747

Table 7: A table containing the results from simulated Monte Carlo Tree Search games. Results were gathered from 100 repetitions for counts 10–20,000, 50 repetitions for 40,000–80,000 and 25 repetitions for 120,000.

Table 7 shows that for this simulated version of the game the MCTS method performs well with great averages and the highest achieved maximum score up until this point. The average score without "$/M_S$" is higher for all tested playout counts below 20,000, due to the fact that the increase of exploration with "$/M_S$" included appears to come at the cost of consistency at smaller playouts (10,000 and smaller). This is because at small playout counts there is not enough exploitation within the playouts due to the increased exploration. However, at 20,000 playouts and higher this increase in exploration is no longer a detriment, but a large upside, as can be seen by the large increase in average and maximum score gained at higher playout counts for the player with "$/M_S$". This is explained by the fact that it has reached a good balance between exploration and exploitation at these large playout counts. The same increase in average and maximum score at higher playout counts can not be seen for the MCTS without "$/M_S$" included, as this player spends the extra playouts exploiting the same moves repeatedly without gaining any significant extra information from the extra playouts. When looking at the maximum score, we found that larger playout counts did, however, not keep increasing the maximum past 40,000 playouts as we have most likely nearly reached the limit of what this MCTS simulated player can find.

Usage of heuristics

The heuristic proved to help with the tree size and allowed more playouts which led to better performance for the MCTS player so we decided to test using this heuristic on some of the other players to reduce their runtime and potentially increase their achieved scores.

	Average score	Max score	runtime (s)
weighted random player	11.74	43	0.020
non-weighted random player	17.43	45	0.010
pure MC at 5 playouts	58.71	70	30.54
pure MC at 10 playouts	61.88	73	60.55

Table 8: A table showing the achieved scores from games with the implemented heuristic applied to limit tile selection. The results for the random players are gathered from 10,000 repetitions and for the MC players 100 repetitions were used.

The results can be seen in Table 8 with the average and maximum score displayed for the various players. The heuristic improves the scores achieved by the random players (for scores without the heuristic applied see Table 1) drastically which can be easily explained by the fact that the tiles that are not that good at that time have been excluded which benefits the random players' performance a lot. For the Monte Carlo player it is however a bit more complicated. On average the Monte Carlo games using the heuristic have a smaller average score (for scores without the heuristic applied see Figure 11 and Figure 12) which is caused by a smaller percentage of boards that are entirely filled when using the heuristic. For 5 playouts without the heuristic the percentage of boards that were completely filled was 44 and at 10 playouts this was 50. With the heuristic implemented this dropped to 35 at playout count 5 and it dropped to 45 at playout count 10. While excluding the tiles based on their value, their shape is entirely disregarded, meaning that sometimes a tile that could help with entirely filling the board is excluded at a timestep where it was required to fill the board. This experiment shows that there is potential in the use of a heuristic, but this implementation is just a bit too limited for Patchwork.

5.2 Reinforcement Learning

To test our implemented Reinforcement Learning framework we have tested it on non-weighted random games and pure MC games with 5 playouts. The experiments consists of a number of rounds and after each round a learning update is performed, which removes a certain number of tiles.

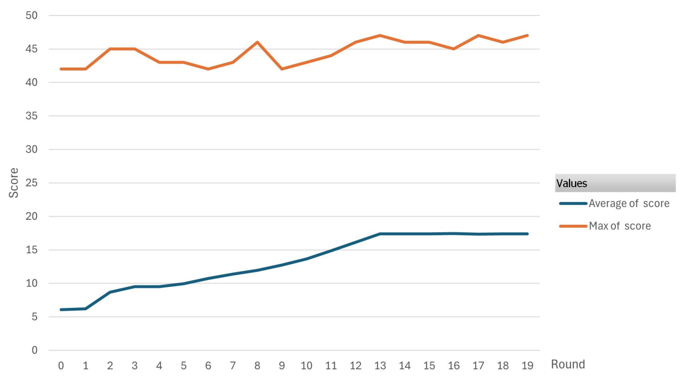


Figure 17: A graph depicting the average and max score of the non-weighted random player with restricted tile selection by the reinforcement framework. Rounds lasted 1000 games and the results were gathered from 50 repetitions.

The results from the random games can be seen in Figure 17. For this experiment we have used 20 rounds, where after each round a single tile was removed until there were 20 left. Rounds consisted of 1000 random games, meaning that many random games were played before even a single tile was removed from the selection. When looking at the average of the random games it shows a steady increase with each removed tile and this reaches its limit at the round where no more tiles are being removed due to minimum number of tiles in the game being capped at 20. The maximum does improve slightly over time, but due to the nature of the random games this increase is minimal.

For the experiment using the pure MC player we have used 5 playouts, rounds consisted of 5 games and in each learning update 3 tiles were removed. The scores from 10 repetitions can be seen in Table 9 and the percentage of full boards can be seen in Table 10. For this experiment at 5

Round	0	1	2	3	4	5
Average with boost	58.80	58.46	58.20	56.64	56.76	54.76
Max with boost	69	69	69	72	72	69
Average without boost	57.84	59.76	58.48	59.01	56.20	56.04
Max without boost	64	70	70	68	67	65

Table 9: A table depicting the average and max scores gathered from the Monte Carlo Reinforcement Learning experiment.

Round	0	1	2	3	4	5
With boost percentage full boards	24	44	24	20	16	12
Without boost percentage full boards	32	22	24	12	6	4

Table 10: A table depicting the percentage of filled boards at each round in the MC Reinforcement Learning experiment.

playouts we have to compare the scores achieved by the pure MC player at 5 playouts in Figure 12 and Figure 11 with an average score of 59.54 and a max score of 68. The results have been gathered at quite a small sample size so the data is a bit noisy, but overall a steady decrease in average and max score is very apparent. Round 1 for the MC player without the boost seemed to perform a bit better which is caused by the removal of 3 tiles which were simply quite bad due to their high cost. Here in nearly all of the 10 games tiles 3, 9, 19 and 25 (see Figure 2) have been removed as they had not been used and had the worst value from the unused tiles. The decrease in overall score at later rounds is explained by looking at Table 10 which shows that with the removal of these tiles the percentage of games that reach a full board decreased drastically. This table also shows that with the boosted score for tiles that are used often, the percentage of full boards is quite a bit higher. This is explained by the fact that without this boost tiles such as 0, 1 and 2 (see Figure 2), which are great for filling the board, had been removed by the MC player without the boost.
Overall the removal of tiles is very tricky due to the fact that each tile that may appear bad on paper perhaps is the best fit in a situation. At very large playouts with very stable players it could potentially help with runtimes but at small playouts it simply hinders the games too much.

5.3 Combined player

For the combined player we will not run a large structured experiment as the runtimes are very long. We have used the pure Monte Carlo player at 500 playouts for the first 6 purchases and after this we were able to use the MCTS player at 3000 playouts until move 12 at which point we used the perfect finish to end perfectly. We have ran this combined player 3 times with their runtime being around 10 hours each. The highest score on the real game we have been able to achieve by doing this was at 74 with the other two games scoring 72 and 70. This marks the highest score we were able to find on the real game. This is attributed to the fact that the MC player set up a good solid start and the MCTS player had a playout count high enough to build large enough trees to get a good grasp of what good moves were and the perfect finished ensured that the ending was player perfectly.

5.4 Highest achieved score(s)

To find a score as close as possible to the optimal score we use the simulated version of the game with the perfect finish option enabled. The MCTS player showed the best performance, which is why we will be using the MCTS player at large playouts to find a score as high as possible. We run a number of repetitions with 80000 playouts for MCTS with a perfect finish.
The best game we have found ended with a score of 84, 14 purchased tiles and 18 sewn in buttons in total. The sequence is shown in Figure 18 alongside the highest scoring game without the perfect finish. Proof of it being able to fit on the board is shown in Figure 19. To confirm that it would fit we have used an online polyomino placement problem solver [Mea] that uses the Dancing Links algorithm [Knu00].
This highest scoring game shows that during a good game you want to skip as few spaces as possible as the time to buttons exchange is not a good trade for the most part. The game was also able to very efficiently space out its purchases where it bought many tiles just before a button spot. Button spots in the game are at timestep 5, 11, 17, 23, 29, 35, 41, 47 and 53 and it managed to use these very well. For the ending sequence the only major improvement that the perfect finish brought was the option to find a sequence that kept a tile with a high time cost for its last purchase at timestep 52 meaning that a tile with a timecost of 5 was essentially reduced to a timecost of 1 here.

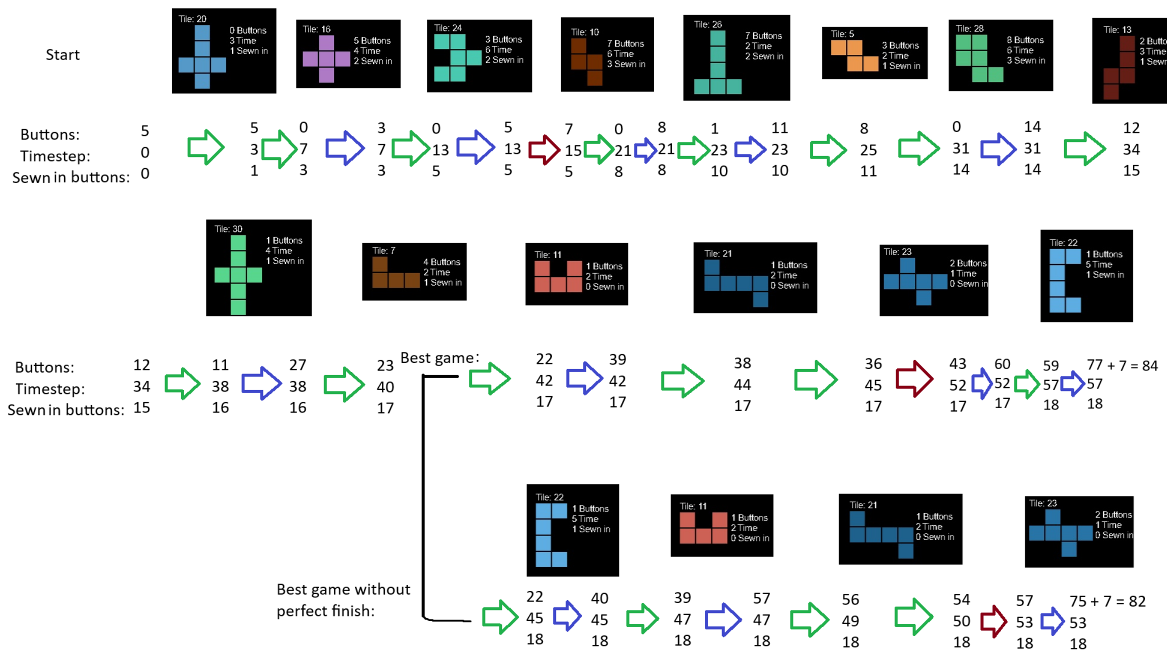

Start
Buttons:
Timestep:
Sewn in buttons:
Tile: 20 0 Buttons 3 Time 1 Sewn in
Tile: 16 5 Buttons 4 Time 2 Sewn in
Tile: 24 3 Buttons 6 Time 2 Sewn in
Tile: 10 7 Buttons 6 Time 3 Sewn in
Tile: 26 7 Buttons 2 Time 2 Sewn in
Tile: 5 3 Buttons 2 Time 1 Sewn in
Tile: 28 8 Buttons 6 Time 3 Sewn in
Tile: 13 2 Buttons 3 Time 1 Sewn in
Tile: 30 1 Buttons 4 Time 1 Sewn in
Tile: 7 4 Buttons 2 Time 1 Sewn in
Tile: 11 1 Buttons 2 Time 0 Sewn in
Tile: 21 1 Buttons 2 Time 0 Sewn in
Tile: 23 2 Buttons 1 Time 0 Sewn in
Tile: 22 1 Buttons 5 Time 1 Sewn in
Best game:
Best game without perfect finish:
77 + 7 = 84
75 + 7 = 82

Figure 18: A figure depicting the game with the highest score we have found with or without the perfect finish option enabled. Green arrows indicate a purchase of the tile depicted above, blue arrows indicate the passing of a button spot (awarding the player buttons equal to their sewn in amount) and red arrows indicate skipping spaces (to exchange time for buttons)

Based on this highest scoring game we presume that a potential maximum score of the single-player game to fall somewhere in the 84–90 range. The game with a score of 84 used its time very well, but there was still an action that skips 2 spaces early on which could potentially be used more

efficient. Other than this the game seems to be played nearly perfectly, which is why we presume that the maximum is in between 84 and 90.
This highest scoring game we found is, however, not possible in the 2-player game as it assumes that it is able to acquire all five 1×1 tiles. With this action sequence the player wants to go from timestep 25 to 31, while there are 1×1 tiles at timestep 26 and 32, meaning that it misses out on at least one of them in the 2-player game. This can be remedied by purchasing tile 0 (see Figure 2) which fills 2 spaces. We have verified that there exists an action sequence in the 2-player game, where the first player gets 3 1×1 tiles and perform the action sequence in Figure 18 along with purchasing tile 0. This extra purchase required means that it has to spend 2 more buttons and 1 extra time, thus bringing the highest found score for the 2-player game to 81. This collection of tiles with tile 0 being included was verified to fit on a board using the polyomino placement problem solver [Mea] and can be seen in Figure 20.

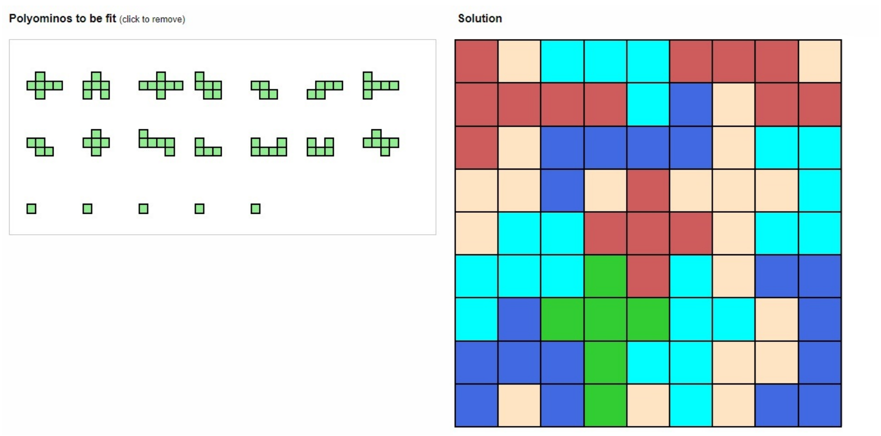


Figure 19: A figure depicting a filled board with the tiles used in the best game we have found [Mea].

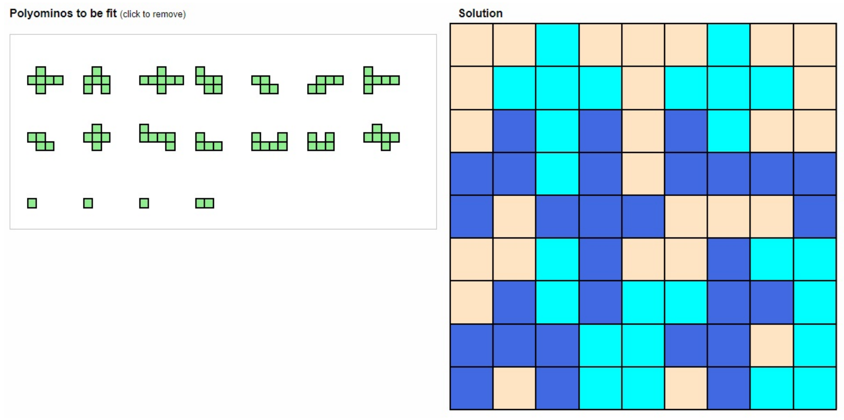


Figure 20: A figure depicting a filled board with the tiles used in the best real game we have found [Mea].

Recap of highest achieved score per method

We close this section off with Table 11, which features all the maximum and average score achieved for the most important used methods to give an overview of all the used methods' performance. For full details and notes on performance we refer the reader to the respective sections within Section 5.

Method	Average score	Maximum score	runtime
Random weighted	-6.62	29	0.0257 s
Random non-weighted	1.90	41	0.0088 s
Random simulated	12.88	56	0.0003 s
Value Based Return Zero (best scoring TCM)	N/A	57	N/A
Value Based rule-based placement (best scoring TCM)	N/A	67	N/A
pure Monte Carlo (10 playouts)	62.38	72	4m 4.07 s
pure Monte Carlo (100 playouts)	67.7	72	40m 40.70 s
pure Monte Carlo greedy (10 playouts)	47.94	60	4m 4.07 s
simulated Monte Carlo (100 playouts)	70.63	78	0.47 s
simulated Monte Carlo (10,000 playouts)	72.41	76	42.99 s
Monte Carlo Tree Search (with heuristic)	63.7	70	10m 46.13 s
Monte Carlo Tree Search (with heuristic) greedy	60.6	67	10m 46.13 s
simulated Monte Carlo Tree Search (40,000 playouts)	74.65	82	9m 49.752 s
combined player	72	74	10h
simulated Monte Carlo Tree Search (80,000 playouts) w/ perfect finish	78.32	84	28m 5.326 s

Table 11: Table containing the scores and runtimes of all the most important methods used.

6 Conclusions and Future Work

We have implemented and tested a number of algorithms on a single-player version of the board game Patchwork. We have done this using several methods, but mainly Monte Carlo methods have been utilized to find a score as high as possible. We have done testing on a real version of the game where the player has to place tiles on a board, as well as a simulated version of the game where the player does not have to place tiles on the grid (this simulated version is still limited by the number of spaces on the 9×9 board, so tile shape is irrelevant but size still matters).
From our experiments we have learned much about the behaviour of Monte Carlo methods on this version of the game, such as impact of playout counts and the performance of different Monte Carlo methods. Overall we have found that the Monte Carlo Tree Search algorithm performs the best, but this method is flawed by the fact that its memory usage for a game with such a large search space is simply too high. This is why we have implemented a number of techniques to reduce the size of the search space. Our attempts include the use of a heuristic limiting tile selection as well as a Reinforcement Learning framework that learns what tiles to exclude based on a number of games, but unfortunately both of our attempts negatively impacted the score. For the MCTS algorithm the heuristic did however improve performance as the smaller tree size caused by the heuristic allowed for more playouts, which increased achieved scores.
For the real game this limited number of playouts for Monte Carlo Tree Search meant that the pure Monte Carlo player was able to perform better as we could play many more random games without being limited by memory usage. To find the highest score for the real game we have combined the pure MC player and the MCTS player with brute force to find a score of 74. For the simulated game however, this memory usage was not a problem as the search space was far smaller, meaning that we have been able to use MCTS at very large playout counts combined with brute force at the end to find a score of 84 for this altered version of Patchwork. This action sequence was then verified to be possible in the real (non-simulated) game. This action sequence that achieved a score of 84 can be slightly altered to be achievable in the real 2-player game for a score of 81.

6.1 Future work

In the future it would be interesting to further analyze the methods we have used at higher playout counts on the real (non-simulated) single-player game. This could not be done as currently the runtime is too long to experiment on the real game extensively. We believe that the largest gain here could be made by finding a heuristic that is able to limit placement options for a tile, but it is very hard to make a good heuristic for this due to the complex nature of the game.
We believe that the most interesting research building on this work (on the single-player game) would come from more time and effort spent on including Reinforcement Learning techniques. While we have incorporated a mechanism that removes poorly performing tiles, we believe that this only scratches the surface of what can be done. Developing a method that limits tile selection only at certain time steps, would be the best step to take next. This could greatly reduce runtime as a smaller set of tiles has to be analyzed at any one time, but this would be difficult to fine tune, mostly due to the large impact any small purchase can have on the flow of the game.
It would also be interesting to run experiments on versions of the game with altered rule sets such as a more limited number of 1×1 tiles or more or fewer button spots present in the game.
Lastly, while quite some work could still be done on the single-player variant of the game, it would be interesting to test these methods on the normal 2-player game with both players trying to win.

References

[CRM+13] Roman Chirikov, Paolo Rocca, Luca Manica, S. Santarelli, Robert Mailloux, and Andrea Massa. Innovative GA-based strategy for polyomino tiling in phased array design. In *Proceedings of the 7th European Conference on Antennas and Propagation, EuCAP 2013*, pages 2216–2219, 2013.

[CWvdH+08] Guillaume Chaslot, Mark Winands, H. Jaap van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4:343–357, 2008.

[DI22] Justin Dallant and John Iacono. How fast can we play Tetris greedily with rectangular pieces? In *Proceedings of the 11th International Conference on Fun with Algorithms, FUN 2022*, volume 226 of *LIPIcs*, pages 13:1–13:19, 2022.

[Gee] Board Game Geek. Patchwork. `https://boardgamegeek.com/boardgame/163412/patchwork`. Accessed on June 25th 2024.

[Knu00] Donald E. Knuth. Dancing links. `https://arxiv.org/pdf/cs/0011047`, 2000.

[Kot12] Róbert Kotrík. Searching for a strategy of monopoly game using cognitive and artificial intelligence approach. Master's thesis, 2012.

[Lag20] Mikael Zayenz Lagerkvist. State representation and polyomino placement for the game patchwork. `https://arxiv.org/abs/2001.04233`, 2020.

[Mea] C. Meadors. polyomino-solver. `https://github.com/cemulate/polyomino-solver`. Accessed on June 28th 2024.

[Ros14a] U. Rosenberg. Patchwork, the board game. 2014.

[Ros14b] Uwe Rosenberg. Patchwork rulebook. `https://cdn.1j1ju.com/medias/74/af/f2-patchwork-rulebook.pdf`, 2014. Accessed on May 13th 2024.

[SCS09] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo tree search in Settlers of Catan. In *Ethical Theory and Moral Practice*, pages 21–32, 2009.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[TYA22] Kazuki Takabatake, Keisuke Yanagisawa, and Yutaka Akiyama. Solving generalized polyomino puzzles using the Ising model. *Entropy*, 24:354, 2022.

[ZZN11] Cai Zhongjie, Dapeng Zhang, and Bernhard Nebel. Playing Tetris using bandit-based Monte-Carlo planning. In *Proceedings of AISB 2011 Symposium: AI and Games*, 2011.