Universiteit
Leiden
The Netherlands

Exploring the synergies between transfer in reinforcement learning and procedural content generation

Müller-Brockhausen, M.F.T.

Citation
Müller-Brockhausen, M. F. T. (2025, November 5). *Exploring the synergies between transfer in reinforcement learning and procedural content generation*. Retrieved from https://hdl.handle.net/1887/4282228

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/4282228

Note: To cite this publication please use the final published version (if applicable).

Chapter 2

A New Challenge: Approaching Tetris Link with AI

Decades of research have been invested in making computer programs for playing games such as Chess and Go. This paper[a] focuses on a new game, Tetris Link, a board game that is still lacking any scientific analysis. Tetris Link has a large branching factor, hampering a traditional heuristic planning approach. We explore heuristic planning and two other approaches: Deep Reinforcement Learning (DRL), MCTS. We document our approach and report on their relative performance in a tournament. Curiously, the heuristic approach is stronger than the planning/learning approaches. However, experienced human players easily win the majority of the matches against the heuristic planning AIs. We, therefore, surmise that Tetris Link is more difficult than expected. We offer our findings to the community as a challenge to improve upon.

[a]This chapter is based on the publication [175] Müller-Brockhausen, M., Preuss, M., Plaat, A.: A new challenge: Approaching tetris link with AI. In: 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, August 17-20, 2021. IEEE (2021), https://doi.org/10.1109/CoG52621.2021.9619044

2.1 Introduction

Board games are favorite among AI researchers for experiments with intelligent decision-making and planning. For example, works that analyze the game of Chess date back centuries [198, 254]. Already in 1826, papers were published on machines that supposedly played Chess automatically [38], although it was unclear whether the machine was still operated somehow by humans. Nowadays, for some games, such as Chess [120] and Go [246, 248], we know for sure that there are algorithms that can, without the help of humans, automatically decide on a move and even outplay the best human players. In this paper, we want to investigate a new game, Tetris Link, that has not yet received attention from researchers before, to the best of our knowledge (see Section 2.2). Tetris Link is a manual, multiplayer version of the well-known video game Tetris. It is played on a vertical "board", like Connect-4. The game has a large branching factor, and since it is not immediately obvious how a strong computer program should be designed, we put ourselves to this task in this paper. For that, we implement a digital version of the board game and take a brief look at the game's theoretic aspects (Section 2.3). Based on that theory, we develop heuristics for a minimax-based program that we also test against human players (Section 2.4.1). Performance is limited, and we try other common AI approaches: DRL [263] and MCTS [44]. In Subsection 2.5.1, we look at MCTS agents, and in Subsection 2.5.2, we look at RL agents and their performance in the game. In our design of the game environment for the RL agent, we assess the impact of choices such as the reward on training success. Finally, we compare the performance of these agents after letting them compete against each other in a tournament (Section 2.5.3). To our surprise, humans are stronger.

The main contribution of this paper is that we present to the community the challenge of implementing a well-playing computer program for Tetris Link. This challenge is much harder than expected, and we provide evidence (Section 2.3.2) on why this might be the case, even for the deterministic 2-player version (without dice) of the game. The real Tetris Link can be played with four players using dice, which will presumably be even harder for an AI.

We document our approach, implementing three players based on the three main AI game-playing approaches of heuristic planning, MCTS, and DRL. To our surprise and regret, all players were handily beaten by human players.[1] We respectfully offer our approach, code, and experience to the community to improve upon.

[1]Humans only played against the Heuristic, not MCTS or DRL. The Heuristic is our strongest AI as can be seen in Figure 2.6.

2.2 Related Work

Few papers on Tetris Link exist in the literature. A single paper describes an experiment using Tetris Link [188]. This work is about teaching undergraduates "business decisions" using the game Tetris Link. To provide background on the game, we analyze the game in more depth in Section 2.3. The authors are aware of a similar game called *Blokus* [55]. It is an interesting game as the branching factor is so large (≈32928 for Blokus Duo on a 14x14 board) that specialized FPGA's have been applied to build good AI for it [126, 214]. Although visually similar, the gameplay is completely different, so Blokus strategies or heuristics do not transfer to Tetris Link.

The AI approaches that we try have been successfully applied to a variety of board games [203]. Heuristic planning has been the standard approach in many games such as Hex [278], Othello [231], Checkers [205], and Chess [120, 224]. MCTS has been used in a variety of applications such as Go and GGP [44, 64, 222]. DRL has seen great success in Backgammon [267] and Go [79, 246, 263]. Multi-agent MCTS has been presented in [95].

2.3 Tetris Link

Tetris Link, depicted in Figure 2.1, is a turn-based board game for two to four players. Just as the original Tetris video game, Tetris Link features a ten-by-twenty grid in which shapes called tetrominoes[2] are placed on a board. This paper will refer to tetrominoes as blocks for brevity. The five available block shapes are referred to as: *I*, *O*, *T*, *S*, *L*.[3] Every shape has a small white dot, also in the original physical board game variant, to make it easier to distinguish individual blocks from each other. Every player is assigned a color for distinction and gets twenty-five blocks: five of each shape. In every turn, a player must place precisely one block. A block fits if all of its parts are contained within the ten-by-twenty board. A player is skipped if they are unable to fit any of their remaining blocks. A player can never voluntarily skip if one of the available blocks fits somewhere in the board even if placing it is disadvantageous. The game ends when no block of any player fits into the board anymore.

The goal of the game is to obtain the most points. One point is awarded for every block, provided that it is connected to a group of at least three blocks. Not every

[2] A shape built from squares that touch each other edge-to-edge is called a polyomino [66]. Because they are made out of precisely four squares, these shapes are called tetromino [290].

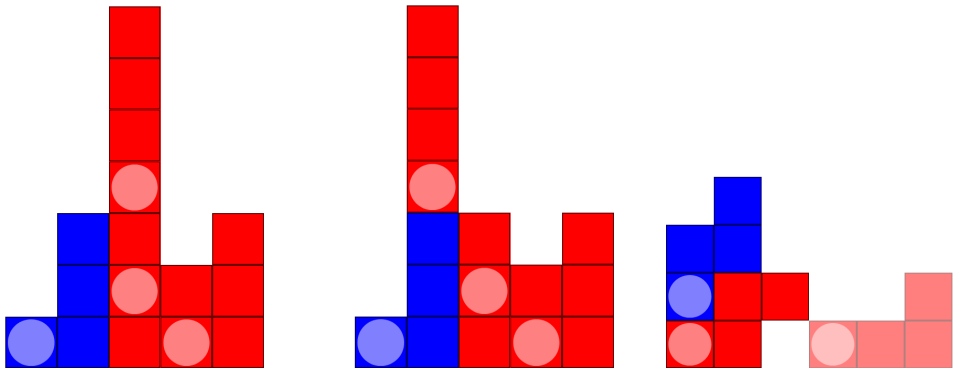
[3] The *S* and *L* blocks may also be referred to as *Z* [45] and *J* [49].

Figure 2.1: A photo of the original Tetris Link board game. The colored indicators on the side of the board help to keep track of the score.

block has to touch every other block in the group, as shown in Figure 2.2a.

The I block only touches the T but not the L on the far right. Since they together form a chained group of three, it counts as three points. Blocks have to touch each other edge-to-edge. In Figure 2.2b, the red player receives no points as the I is only connected edge-to-edge to the blue L.

A player loses one point per empty square (or hole) below block that was placed, with a maximum of two minus points per turn. Figure 2.2c shows how one minus point for red would look like. Moreover, the Figure underlines a fundamental difference to video game Tetris. In video game Tetris, blocks slowly fall, and one could nudge the transparent L under the S to fill the hole by precise timing of an action. In Tetris Link, one can only throw blocks into the top and let them fall straight to the bottom. In the original rules, a dice is rolled to determine which block is placed. If a player is out of a specific block, then the player gets skipped. Since every block could turn into one point, being skipped means potentially missing out on it. Although not a guaranteed disadvantage, as the opponent might also be skipped, the authors have abandoned the dice in their own matches as it resulted in too many games that felt unfairly lost.



(a) Three points for red **(b)** No points. **(c)** Minus point for red.

Figure 2.2: Small examples to explain the game point system.

The dice roll also makes Tetris Link a non-deterministic perfect information game. In this paper, there is no dice roll, so we analyze the deterministic version of Tetris Link. Note that we also focus on the two-player game only in this work. The three- and four-player versions are presumably even harder. In multiplayer games without teams, people might temporarily team up against the current leading player, which creates an unfair disadvantage [55]. Nevertheless, our web-based implementation for human test games[4] can handle up to four players and can provide an impression of the Tetris Link gameplay.

2.3.1 Verification that all games can fill the board

Each game of Tetris Link can be played to the end, in the sense that there are enough blocks to fill the whole board without leaving any empty squares. This can be seen in Equation 2.1: The board is ten squares wide and twenty squares high, so it can accommodate 200 individual squares. Every player has twenty-five blocks (α), each consisting of four squares (β). There are always at least two players playing the game (γ), so they are always able to fill the board.

$$\alpha \cdot \beta \cdot \gamma = 25 * 4 * 2 = 200 \tag{2.1}$$

[4] https://hizoul.github.io/contetro

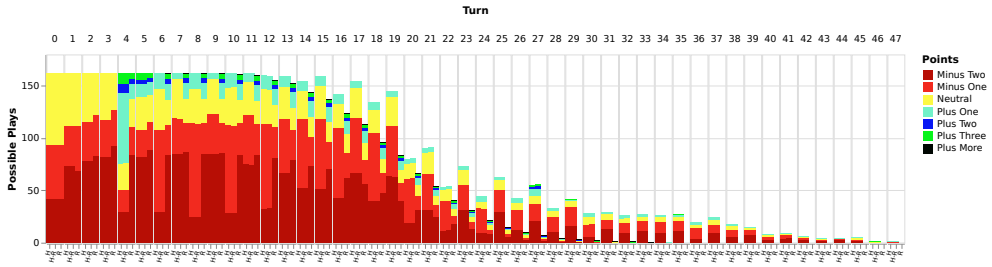


Figure 2.3: This graph underlines the game's difficulty, especially for search algorithms, by showing the average number of possible moves and their associated effect on the player's score. The agent name abbreviations stand for: R = Random, H-R = Random Heuristic, and H = User Heuristic. See Section 2.4.1 for an explanation of how the heuristic agents work.

2.3.2 Game complexity

An essential metric for search algorithms is the so-called branching factor, which specifies the average number of possible moves a player can perform in one state [81]. In order to compute this number, we look at the number of orientations for each block. The I block has two orientations as it can be used either horizontally or vertically. The O block has only one orientation because it is a square. The T block has four different orientations for every side one can turn it to. The L and S are special cases as they can be mirrored. The L has four and the S two sides one can rotate it to. Mirroring them doubles the amount of possible orientations. Hence, in total, nineteen different shapes can be placed by rotating or mirroring the available five base blocks. Since the board is ten units wide, there are also ten different drop points per shape. In total, there can be up to 190 possible moves available in one turn. However, the game rules state that all placed squares have to be within the bounds of the game board. Twenty-eight of these moves are always impossible because they would necessitate some squares to exceed the bounds either on the left or right side of the board. Therefore, the exact number of maximum possible moves in one turn for Tetris Link is 162. Since the board gets fuller throughout the game, not all moves are always possible, and the branching factor decreases towards the end. In order to show this development throughout matches, we simulate 10,000 games.

We depict the average number of moves per turn in Figure 2.3. For the first eight to ten turns, all moves are available on average. Not depicted in the Figure but based on the data, this only holds true until turn 6. After that, the number of plays may already decrease. Tetris Link is a game of skill: random moves perform badly. A game consisting of random moves ends after only thirty turns. Many holes with many minus

Agent	Random	Random-H	User-H	Tuned-H
Win Rate	48.16%	47%	71.65%	70%
Unique Games	10,000	10,000	7	50

Table 2.1: First move advantage, over 10,000 games. The first six (#1) or all turns (#2) are compared for uniqueness to see whether the same games keep repeating. The -H in the agent name stands for Heuristic (see Section 2.4.1). Draws are not counted towards wins.

points are created, and the game ends quickly with a low score. The heuristic bars show that simple rules of thumb fill the board most of the time by taking more than forty turns. Furthermore, the branching factor in the midgame (turn 13-30) declines slower and hence offers more variety to the outcomes. Another thing that can be seen in Figure 2.3 is that only very few select plays can actually lead to positive points. Most of the turns will leave the score unchanged or even decrease it. A player would only consider taking minus points if it would cost the opponent even more points than the player loses, or there are no other options. To further underline this, we did an exhaustive tree search until depth 5, which is the first turn player one is able to achieve points. Of the over 111 Billion possibilities (111577100832), only $\approx 14.36\%$ allow for an increase in points. Moreover, only $\approx 3.06\%$ allows the optimal three-point gain, which the first player aims for. $\approx 69.08\%$ of the outcomes result in a point disadvantage, and in $\approx 16.55\%$ of the cases, no points have been gained nor lost due to blockage by the opponent or lack of connectivity.

We are now ready to calculate the approximate size of the game tree complexity for the deterministic variant of Tetris Link, in order to compare it to other games. On average, across all three agents shown in Figure 2.3, a game takes 37 turns and allows for 74 moves per turn ($74^{37} \approx 1.45 * 10^{69}$). The game tree complexity is similar to Chess (10^{43} or 10^{123}) but smaller than in Go (10^{360}) [161].

2.3.3 First move advantage

An important property of turn-based games is whether making the first move gives the player an advantage [289]. To put this into numbers, we let different strategies play against themselves 10,000 times to determine if the starting player has an advantage. The first six (#1) or all (#2) moves are recorded and checked for uniqueness.

As can be seen in Table 2.1, the win rate for *random heuristic* as starting player is almost 50%. Although the win rate for the first player is higher for the *tuned heuristics*, these numbers are not as representative because the heuristic repeats the same tactics resulting in only seven or twenty-nine unique game starts. If we repeat the same few

games, then we will not truly know whether the first player has a definite advantage. Especially considering that at least until turn six, all moves are always possible, there are around 10^{13} or 18 Trillion ($BranchingFactor^{Turns} = 162^6 = 18,075,490,334,784$) possible outcomes. Since the *random heuristic* has more deviation and plays properly as opposed to random moves, we believe that it is a good indicator of the actual first player advantage. Note that 47% is close to an equal opportunity. Different match history comparisons of Chess measure a difference of around two to five percent in win rate for the first player [289]. However, since neither Tetris Link nor Chess have been mathematically solved, one cannot be certain that there is a definite advantage.

2.4 AI Player Design

In this section, we describe the three different types of AI players that we implemented, based on heuristics, MCTS, and RL, respectively. For the experiments (Section 2.5), the game is coded in Rust and JavaScript (JS). The Rust version is written for faster experiments with MCTS and RL, and the JavaScript version is written to visually analyze games and also do a human play experiment. Both implementations share a common game log format using JSON in order to enable interoperability. To underline the importance of a performance-optimized version, we measured the average time it takes to simulate one single match where always the first legal move is made. The Rust implementation requires 590μs for that, whereas the JavaScript implementation needs 82ms.

2.4.1 Heuristic

We now describe the design of our heuristic player. A heuristic is a rule of thumb that works well most of the time [221]. For Tetris Link, we identify four heuristic measures: the number of connectable edges, the size of groups, the player score, and the number of blocked edges. The number of blocked edges is the number of edges belonging to opponents that are blocked by the current players' blocks. All heuristic values are positively related to the chance of winning.

Each parameter is multiplied by a weight, and the overall heuristic score is the sum of all four weighted values. For every possible move in a given turn, the heuristic value is calculated, and the one with the highest value is chosen. If multiple moves have the same maximum value, a random one of these best moves is chosen, which is why in Table 2.1, the heuristics play more than one single unique repeating game.

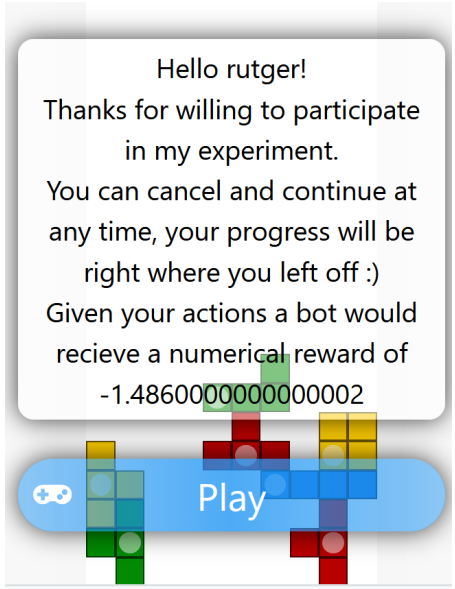


Figure 2.4: The web interface to measure the effectiveness of our Heuristic agent against actual human players familiar with the game.

The initial weights were manually set by letting the heuristic play against itself and detecting which combination would result in the most points gained for both players. We refer to this as *user heuristic*. We then use Optuna [3], a hyperparameter tuner, to tune a set of weights that reliably beat the *user heuristic*. This version is called *tuned heuristic*. To achieve a greater variety in playstyle, we also test a *random heuristic* which at every turn generates four new weights between zero to fifteen. To have an estimate on the performance of the heuristic, we let actual human players (n=7) familiar with the game play against the heuristic via the JavaScript implementation (Figure 2.4). The *random heuristic* achieved a win rate of 23.07% across 13 matches. and the *user heuristic* a win rate of 33.33% across six matches. The sample size is very small, but it still indicates that the heuristic is not particularly strong. This is supported by a qualitative analysis of the game played by the authors, based on our experience. We conclude that our heuristic variants play the game in a meaningful way but are not particularly strong.

2.4.2 MCTS

For applications in which no efficient heuristic can be found, MCTS is often used, as it constructs a value function by averaging random roll-outs [44]. Our MCTS implementation uses the standard UCT selection rule [143]. As further enhancements, we also use MCTS-RAVE [44] and MCTS-PoolRAVE [219] to see whether the modifications help in improving the quality of results. Furthermore, we experimented with improving the default (random) policy by replacing it with the heuristic. However, the heuristic calculation is so slow that it only manages to visit ten nodes per second. We want to stress that this is only the MCTS heuristic that is slow as it was not optimized for speed at all. The game simulation itself is fast processing a full match of random actions in on average 590μs, so up to around 1694 matches per second.

MCTS is well-suited for parallelization, leading to more simulations per second and hence better play [44]. We implemented tree parallelization, a frequently used parallelization [85]. In tree parallel MCTS, many threads expand the same game tree simultaneously. Using 12 threads, we visit 16258 nodes per second on average with a random default policy. To put this into perspective, this is $1.63e^{-9}$% of all 10^{13} possibilities in the first six turns. Thus, only a small part of the game tree is explored by MCTS, even with parallel MCTS.

2.4.3 Reinforcement Learning Environment and Agent

A RL environment requires an observation, actions, a reward [136], and an RL agent with an algorithm as well as a network structure. To prevent reinventing the wheel, we use existing code for RL, namely OpenAI gym [39] and the stable-baselines [117], which are written in Python. To connect Python to our Rust implementation, we compile a native shared library file and interact with it using Pythons *ctypes*. As RL Algorithm, we exclusively use the DRL algorithm PPO2 [234]. For the network structure, we increase the number of hidden layers from two layers of size 64 to three layers of size 128 because increasing the network size decreases the chances of getting stuck in local optima [148]. We do not use a two-headed output, so the network only returns the action probabilities but not the certainty of winning as in AlphaZero [248].

The observation portrays the current state of the game field. Inspired by AlphaGo, which includes as much information as possible (even the "komi"[5]), we add additional information such as the number of blocks left per player, the players' current score, and which moves are currently legal. For the action space, we use a probability distribution

[5] *Komi* refers to the first turn advantage points [246].

Reward Type	Steps	Episode Reward	Score
Guided	3183.49	-0.17	-5.6
Simple	10000.0	-0.0	-12.25
Score	6214.45	-0.09	-6.88

Table 2.2: Results of self-play with different reward types until either a local optimum or 10,000 steps have been reached. Step, Reward and Score show the average of all seeds.

over all moves. The probabilities of illegal moves are set to 0, so only valid moves are considered. For the reward, we have three different options. They are calculated using the current players score (δ), the size of a connected group (ϵ) and a scolding parameter (ζ)

1. Guided: $\frac{\delta+\epsilon}{100} - \zeta$
2. Score: $\frac{\delta}{100}$
3. Simple: ± 1 depending on win / loss

The *Guided* reward stands out because it is the only one that reduces the number of points via *scolding (ζ)*. If the move with the highest probability is illegal, then the reward will be reduced by -0.004, and the first legal move with the highest probability is chosen. This should teach the agent to only make valid moves. This technique is called reward shaping, and its results may vary [106].

In order to detect which one of the three options is the most effective, we conduct an experiment. Per reward function, we collect the averages for the number of steps it took, the average reward achieved, and what the average score of the players was in the results. Our results, shown in Table 2.2, indicate that the *Guided* reward function works best. It only takes around 3183 steps on average to reach a local optimum, and the average scores achieved in the matches are the highest. We define a local optimum as the same match repeating three times in a row. The *Score* reward function also lets the agent reach a local optimum, but it takes twice as long as the *Guided* function, and the score is slightly lower as well. The *simple* reward function seems unfit for training. It never reached a local optimum in the 10,000 steps we allowed it to run, and it got the lowest score in its games.

2.5 Agent Training and Comparison

For our experimental analysis, we first look at the performance of the MCTS agent (Section 2.5.1) and the training process of the RL agents (Section 2.5.2). Finally, we

compare all previously introduced agents in a tournament to analyze their play quality and determine the currently best playing approach.

2.5.1 MCTS Effectiveness

Setup

Initial test matches of MCTS against the *user heuristic* resulted in a zero percent win rate, and a look at the game boards suggested near-random play. We use a basic version of MCTS with random playouts because using our heuristic as guidance was too slow. AlphaZero has shown that even games with high branching factors such as Go can be played well by MCTS when guided by a neural network [248]. However, without decision support from a learned model or a heuristic, we rely on simulations. In order to see if this guidance is the reason for bad MCTS performance, we abuse the fact that the *user heuristic* plays very predictably (Section 2.3.3). We use the RAVE-MCTS variant (without the POOL addition), pre-fill the RAVE values with 100 games of the *user heuristic* playing against itself, and then let the MCTS play 100 matches against the *user heuristic*. We repeat this three times and use the average value across all three runs. We run this experiment with different RAVE-β parameter values. This parameter is responsible for the exploration/exploitation balancing and replaces the usual UCT C_p parameter. The closer the RAVE visits of a node reach RAVE-β, the smaller the exploration component becomes. Furthermore, we employ the slow heuristic default policy at every node in this experiment. We simulate one match per step because otherwise, the one-second thought time is not enough for the slow heuristic policy to finish the simulation step.

Results

Our MCTS implementation can play well with a decent win rate against the user-heuristic, as shown in Figure 2.5. This result underlines that in games with high branching factors, MCTS needs good guidance through the tree in order to perform well. Figure 2.3 also supports this as there are very few paths that will actually lead to good plays. The declining win rate with a higher RAVE-β value suggests that exploration on an already partially explored game tree worsens the result because the opponent does not deviate from its paths. The rise in win rate for a β value of 5000 after the large drop in 2500 underlines the effect of randomness involved in the search processes.

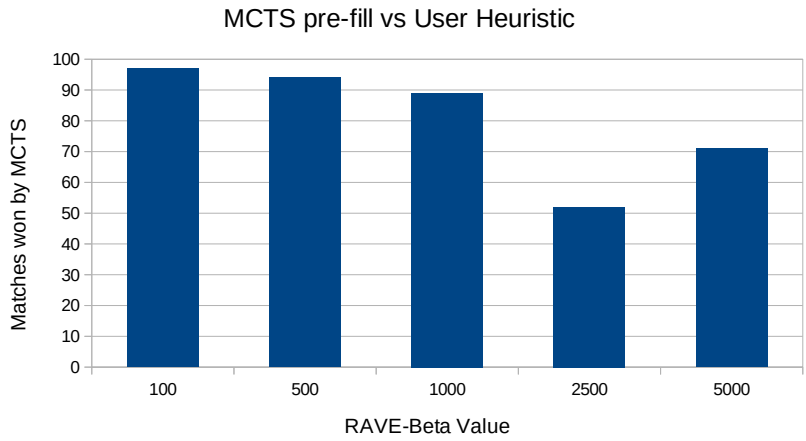


Figure 2.5: Win rates of pre-filled MCTS playing against the *User-Heuristic* compared by the RAVE-β parameter.

Even though the heuristic-supported playout policy works well, we will still use a random playout policy for the tournament (Section 2.5.3). Pre-filling the tree is very costly and would provide an unfair advantage to the MCTS method.

To underline that we believe the bad performance stems from the large branching factor and few paths that lead to positive points, we test our implementation on the game of Hex. In different board sizes (2x2 to 11x11) of Hex, our MCTS plays against a shortest path heuristic. The result is striking: as long as the branching factor stays below 49 (7x7), MCTS wins up to 90% of the matches. For larger branching factors, the win rate drops to 0% quickly.

2.5.2 RL Agents Training

Agents

We define an RL agent as the combination of reward type, algorithm, and training opponent. We use the guided reward function because it worked best in our experiment and call this agent *RL-Selfplay*. (This is a neural network only RL, without MCTS to improve training samples, that plays against itself [Selfplay].)

In addition to this rather simple agent, we introduce the *RL-Selfplay-Heuristic* agent. It builds on a trained *RL-Selfplay* agent where we continue training by playing against the heuristic. Observation and reward are the same as for *RL-Selfplay*.

From the first turn advantage experiment, we know that the heuristic plays well even with random weights. That is why we also introduce an agent called *RL-Heuristic*. This agent outputs four numbers that represent the heuristics weights (Section 2.4.1). We use a modified version of the guided reward function using the players own score (δ), the opponents score (η) and the size of a connected group (ϵ):

$$\frac{(\delta - \eta) + \epsilon}{100} \tag{2.2}$$

Group size stands for the total number of blocks that are connected with at least one other block. This is added because we want the algorithm to draw a connection between the number of points gained and the number of connected blocks. However, mainly the difference in points between itself and the opponent is used as a learning signal, so it aims to gain more points than the opponent. Scolding is not necessary anymore as we do not have to filter the output in any way.

Setup

In this section, we detail the training process of the RL agents. Each training is done four times, and only the best run is shown. Agents are trained with the default PPO2 hyperparameters, except for *RL-Heuristic*, which uses hand-tuned parameters.

When playing only against themselves, the networks still quickly reached a local optimum even with increased layer size. This optimum manifested in the same game being played on repeat and the reward per episode staying the same. This repetition is a known problem in self-play and can be called "chasing cycles" [279]. To prevent these local optima, we train five different agents against each other in random order. To be able to train against other agents, we modified the *stable-baselines* code.

Results

The training process for *RL-Selfplay* peaked around 1.5 million steps. For *RL-Selfplay-Heuristic* we use the two best candidates from *RL-Selfplay*, namely #3 after one million steps with a reward of 0.04 and #1 after 1.5 million steps with a reward of 0.034. The training of *RL-Selfplay#1-Heuristic* reaches its peak after 3.44 and *RL-Selfplay#3-Heuristic* after 3.64 Million steps with a reward of 0.032 and 0.024. These are our first RL agents that can achieve a positive reward while playing against the heuristic.

The *RL-Heuristic* training worked well, achieving mostly a positive reward. However, looking at the output values, we realize the reward function design was unfortu-

nate. It sets all weights to zero, except for the enemy block value to fifteen and the number of open edges between four and seven. So by negating the player's score with the opponent's score, we have unwillingly forced the heuristic to focus on blocking the opponent over everything else. Needless to say, with these weights, the *RL-Heuristic* rarely wins. Although it manages to keep the opponent's score low, it does not focus on gaining points which leaves it with a point disadvantage.

2.5.3 Tournament

Setup

In the tournament, all presented AI approaches play against each other. Every bot will face every other bot in 100 matches. We have five different RL bots, three MCTS bots, and three heuristic bots. Each agent gets at most one second of time to decide on their move, and they are not allowed to think during the opponent's turn. Every bot will play half of its matches as first and the other half as second player. The bot's skill will be compared via a Bayesian Bradley Terry (BBT) skill rating [287]. The original BBT [287] ratings range from 0 to 50. By adjusting the BBT-β parameter, we change this to represent the ELO range (0 to 3000) [103].

Results

The final skill rating is portrayed in Figure 2.6. The three heuristic agents take the top 3, followed by RL and MCTS. Remarkably, the *tuned heuristic* performed best, even though it is only optimized to play well against the *user heuristic*, but yet it performs best across all agents.

Seeing *RL-Heuristic* as the best *RL* approach shows that the other RL agents are far from playing well. Yet all RL agents consistently beating MCTS with random playouts proves that the agents definitely learned to play reasonably.

It is interesting to see that the MCTS-UCB (14% win rate) variant performed best because the other two variants [RAVE (0.02%), PoolRAVE (0.04%)] were conceived in order to improve the performance of UCB via slight modifications [219]. Please note that this poor performance in Figures 2.6 and 2.7 is caused by the tree not being pre-filled as it was in Figure 2.5 (Section 2.5.1). This shows the importance of pre-filling MCTS in Tetris Link due to the few paths that result in positive points (see Figure 2.5).

The skill rating omits information about the quality of the individual moves. To gain further insight into that, we provide Figure 2.7. Here, we can see that every

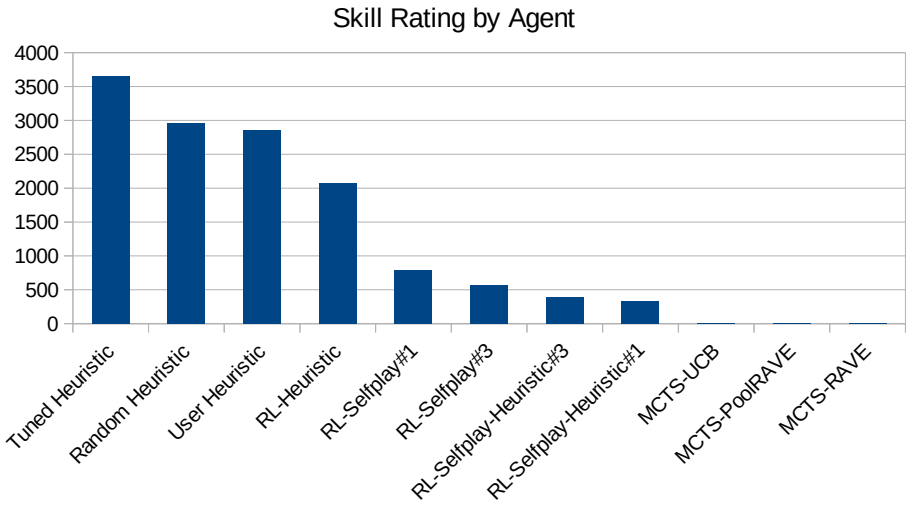


Figure 2.6: The skill rating of the agents that participated in the tournament. MCTS uses a non pre-filled tree, resulting in bad performance (Section 2.5.1).

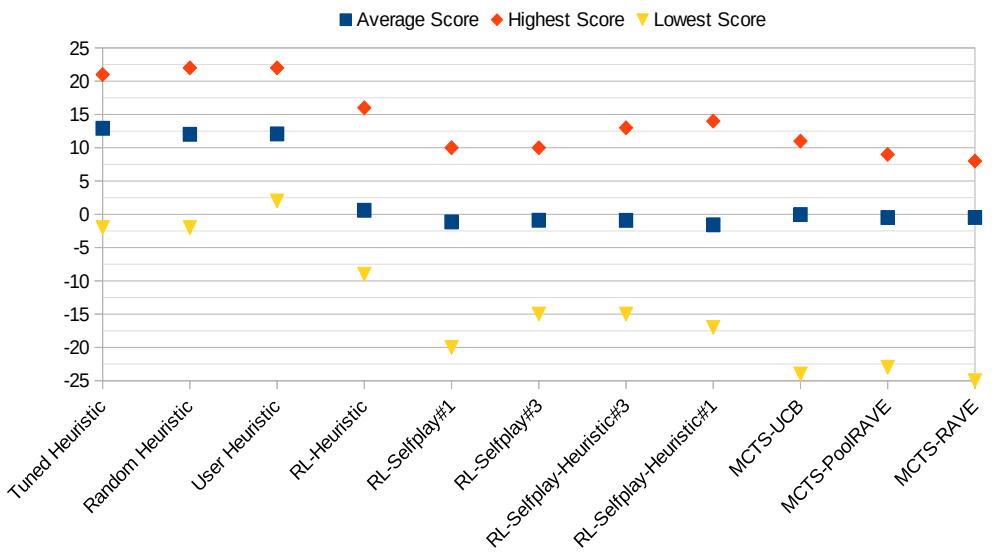


Figure 2.7: Visualisation of the scores that agents achieved in the tournament. Agents are sorted by the skill rating in Fig. 2.6.

agent manages at least once to gain 8 points or more. This means that every agent had at least one match it played well. Looking at the lowest achieved scores and average scores, we find that every agent, except for the pure heuristic ones, plays badly, considering that they only make ±3 points on average.

2.6 Conclusion and Future Work

Board game strategy analysis has been done for decades, and especially games like Chess and Go have seen countless papers analyzing the game, patterns, and more to find the best play strategies [248]. We contributed to that field by taking a close look at the board game Tetris Link. While the strategy is key to winning, some games, such as Hex, give the first player a definite advantage. We have experimentally shown that there is no clear advantage for the starting player in Tetris Link (Section 2.3.3).

We have implemented three game-playing programs based on common approaches in AI: heuristic search, MCTS, and RL. Despite some effort, none of our rule-based agents was able to beat human players.

In doing so, we have obtained an understanding of why it may be hard to design a good AI for Tetris Link:

- Especially at the beginning, the branching factor is large, staying at 162 for at least the first six turns.
- The majority of possible moves results in minus points making the number of good moves diminishingly small (see Figure 2.3 and Section 2.3.2).
- Mistakes / minus points can hardly be recovered from. The unforgivingness for these moves may make it harder to come up with a decent strategy, as generally postulated by [63].
- Many rewards in the game stack — they come delayed after multiple appropriate moves because groups of blocks count and not single blocks. This makes it especially hard for MCTS and RL to learn the correct sequences.

All this holds true for the simplified version we treat here: no dice, only two players. Adding up to two more players and dice will also make the game harder.

With a solid understanding of the game itself, we investigated different approaches for AI agents to play the game, namely heuristic, RL and MCTS. We have shown that all tested approaches can perform well against certain opponents. The best currently

known algorithmic approach is the tuned heuristic, although it can not consistently beat human players.

Training an RL agent (Section 2.5.2) for Tetris Link has proven to be complicated. Just getting the network to produce positive rewards required much trial and error, and in the end, the agent did not perform well even when consistently achieving a positive reward. We believe the learning difficulty in Tetris Link comes from the many opportunities to make minus points in the game. One turn offers usually offers one plus to three points, or six at most if two groups are connected, but that means that multiple previous turns that were well planned and gave zero points if not even more minus points had to be made. Hence recovering from minus points is difficult, meaning small mistakes have graver consequences.

Although MCTS performed poorly in our tournament, we have shown that with proper guidance through the tree, MCTS can perform nicely in Tetris Link and Hex (Section 2.5.1). That is why a combination where RL guides an MCTS through the tree might work well, e.g. AlphaZero [248] or MoHex v3 [98], and is something to try in future work.

As computer game researchers, we found ourselves challenged to create a good agent to play the game Tetris Link. We tried a large variety of classic approaches and were not able to achieve the results we hoped for. We invite the research community to use our code and improve upon our approaches.[6]

[6]https://github.com/Hizoul/tetris-link-research