Playing Tetris Using Bandit-Based Monte-Carlo Planning

Zhongjie Cai and **Dapeng Zhang** and **Bernhard Nebel**[1]

Abstract.

Tetris is a stochastic, open-ended board game. Existing artificial Tetris players often use different evaluation functions and plan for only one or two pieces in advance. In this paper, we developed an artificial player for Tetris using the bandit-based Monte-Carlo planning method (UCT).

In Tetris, game states are often revisited. However, UCT does not keep the information of the game states explored in the previous planning episodes. We created a method to store such information for our player in a specially designed database to guide its future planning process. The planner for Tetris has a high branching factor. To improve game performance, we created a method to prune the planning tree and lower the branching factor.

The experiment results show that our player can successfully play Tetris, and the performance of our player is improved as the number of the played games increases. The player can defeat a benchmark player with high probabilities.

1 INTRODUCTION

In 1985, Alexey Pajitnov et al. first invented the puzzle video game Tetris. It has now spread to almost all game consoles and desktop systems.

The standard Tetris is a stochastic, open-end board game. Players control the falling pieces in the game field and try to build fully occupied rows, which are removed in each turn. The standard Tetris field contains 10×20 cells, and the pieces are randomly chosen from 7 pre-defined shapes of blocks. In addition to the current piece, the information of the next piece is provided to the player. Removing rows in a single turn has certain rewards, which in our approach is given as 0.1 for a single row, and 0.3, 0.6, 1.0 for two to four removed rows. This encourages the players to remove multiple rows instead of only one row in one turn. The challenge of this game is that, the player needs to place the pieces in the proper positions in the game field in order to best accommodate the next pieces and remove as many rows as possible. An inappropriate placement of one piece often results in a bad situation of the game, and causes the player to spend more time to deal with. The game is over if the top of the game field is occupied by blocks and the next piece cannot be placed onto the field.

In the two or more players' competitions, if one player has removed $n(n > 1)$ rows in one turn, all other players will receive an attack of $(n - 1)$ rows of blocks, adding to the bottom of their game fields. Each attack row would contain $(n - 1)$ empty cells in random positions. Thus removing multiple rows in single turns brings even more benefits than rewards. Highly skilled human players prefer to plan and remove three or even four rows using a single falling piece, while beginners and many of the existing Tetris artificial players tend to remove rows as soon as possible in each turn to survive the game. The game is over when only one player is still alive in the competition, and of course the last player is the winner.

Researchers have created many artificial players for the Tetris game using various approaches[12]. To the best of our knowledge, most of the existing players rely on evaluation functions, and the search methods are usually given less focus. The Tetris player developed by Szita et al. in 2006[11] employed the noisy cross-entropy method, but the player had a planner for only one piece. In 2003, Fahey had developed an artificial player that declared to be able to remove millions of rows in a single-player Tetris game using a two-piece planner[6]. Later in 2005, genetic algorithms were introduced to the Tetris players by Böhm et al.[3], in which a heuristic function was evolved by using an evolutionary algorithm. In our previous work, we have developed an artificial player with a one-piece planner by using learning by imitation[14] that could successfully play against Fahey's player in the Tetris competitions.

Yet most of the existing artificial players known are based on a planner for only one- or two-piece. This paper was motivated by creating an artificial player based on the planning of a long sequence of pieces. We modeled our player in Tetris planning problem with the Monte-Carlo planning method. In order to balance the exploration-exploitation trade-offs in the planning process, we employed the bandit algorithm to guide the planning process. As for state revisiting, we created a method to store the visited game states in a specially designed database. We also created a hash function to quickly locate and operate the information of a given game state in the database. In order to reduce the branching factor of Tetris planning, we created an intuitive evaluation function and combined it with the UCT algorithm.

The highlights of this paper can be summarized as follows:

- We modeled the artificial player of Tetris using the UCT algorithm.
- Our method of the database of the visited states provided support to UCT and improved the performance of the planner.
- By pruning the planning tree, the player can defeat the artificial player developed by Fehey, which is regarded as the benchmark.

This paper is structured in the following manner. First in section 2, we present our solution on modeling the Tetris planning problem with the bandit-based Monte-Carlo planning method. Our method to design the knowledge database and store the information of the visited game states is presented in section 3. The idea of combining the evaluation function to the UCT algorithm is discussed in section 4. The experiments and the results are shown and analyzed in detail in section 5. In the final section 6, we draw the conclusion and discuss the future work.

[1] Universtiy of Freiburg, Germany, email: caiz, zhangd, nebel@informatik.uni-freiburg.de

1.1 Related Works

To create an artificial player for a board game, the general components are the search method and the evaluation function. The board games which are solvable by brute-force methods, such as Othello, have already been dominated by game programs using various search methods, such as LOGISTELLO [5]. Board games such as Checkers are solvable using knowledge database combined with search methods, one such example is the program named CHINOOK [10]. Many board games, e.g. Chess and Go, are currently unsolvable, thus are still challenging tasks for artificial intelligence researchers. To improve the performance of the artificial players for these board games, one of the tasks for the researchers is to balance the trade-offs between the search depths and evaluation functions [2].

The Monte-Carlo planning method (MCP) has offered a new solution to artificial players of board games. In 1993, Bernd first modeled the board game Go with the MCP algorithm [4], and his Go player had a playing strength of about 25 kyu[2] on a 9×9 board. Soon the MCP method was successfully applied in other board games, such as Backgammon[8]. In 2006, Levente Kocsis and Csaba Szepesvri developed a new search technique named UCT, which stands for *Upper Confidence Bound applied to Trees* [7], and proved that UCT to be more efficient than its alternatives in several domains. Instead of uniform sampling of the game actions, UCT uses the multi-armed bandit algorithm to guide the action selection of the planning process. Later applications using the technique, such as MoGo[3], demonstrated that this technique can be successfully applied to the game of Go.

Learning techniques have also been applied to improve the perfomance of artificial players of board games. The first such approach was the one by Samuelson in 1959 [9]. He was able to show how a program can learn to play Checkers by playing against itself. In 2010, Takuma Toyoda and Yoshiyuki Kotani suggested the idea of using previous simulated game results to improve the performance of the original Monte-Carlo Go program [13], and their work announced positive results on the larger Go board. In Tetris, Böhm et al. used genetic algorithms for the heuristic function, and our previous work had introduced learning by imitation to the artificial player of multi-player Tetris games[3].

Yet to the best of our knowledge, UCT has not been applied in artificial players for Tetris.

2 PLANNING TETRIS USING UCT

In this Section, we discuss how we model the Tetris planning problem using the UCT algorithm.

There are two possible values for every cell in the game field, e.g. occupied and unoccupied, so the standard Tetris search space consists of 2^{200} game states. The branching factor is 162 for a given game state without the piece information, which is the sum of all possible placements of actions from the 7 different pieces. The large branching factor brings us to the idea of using the Monte-Carlo planning method in our solution to the artificial Tetris player. The core feature of the Monte-Carlo planning is to sample as many future states as possible from all actions of the given state of the game for a certain period of time, and for each episode evaluate only the leaf state using a fast evaluation function. In the end, the algorithm takes the action with the best evaluated reward in the planning as the result of the algorithm.

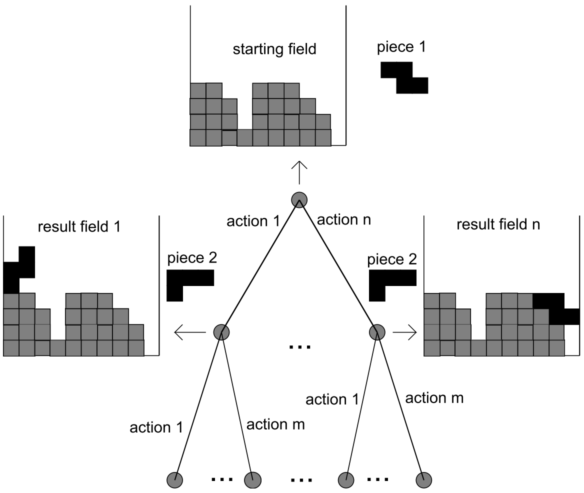

Figure 1: Node of game field state and piece in planning tree

A sample of the planning tree is shown in the Figure 1. In our model of the Tetris planning, we consider each state of the game field, together with a given piece, as a single node in the planning tree. For instance, the root node consists of a game field, which is the rectangular area with gray square blocks inside, and a piece, in this case a ”Z” shaped piece displayed by four black square blocks. The fields in the nodes are the so-called ”cleared fields”, which means that no fully occupied, removable rows are contained in such fields.

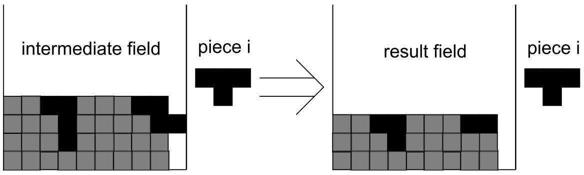

Figure 2: Procedure of removing fully occupied rows

The paths through the planning tree represent the actions associated to the given piece. By following the path of one starting node, the piece is added to the game field according to the index of the encoded action, and a target field is generated. The field will then be checked for removable rows, and if there are rows removed in the field, such rows are removed and a reward will be given according to the predefined game rule. This procedure is described in Figure 2. The fully occupied row is removed from the intermediate field, and then the resulted field and the piece i form a new node in the planning tree.

2.1 Planner Structure

The pseudo code describing our planner is displayed in Algorithm 1 and 2.

In the beginning of the planning episode, the state of the game field and the sequence of pieces are the input parameters of the planner. The planner initiates the growth of planning tree and starts planning phases. The rank of the paths in the planning tree are calculated directly by using the rewards gained from performing the action associated and removing the fully occupied rows in the field. The rewards are based on the values given in Section 1.The planning episode continues to run as many phases as possible until a certain

[2] In Go, the rank of 30 — —20 kyu refers to a *Beginner* level.
[3] Website: http://www.lri.fr/ gelly/MoGo.htm

```
Input: state, list of pieces
Output: action
initialization;
while not time out do
    search(state, first piece);
end
action ← selectBest(state, pieces);
updateTree();
return action;
```

Algorithm 1: One Planning Episode : Function **doPlanning**

time out rule is reached, and returns the action with the highest reward in the root node of the planning tree. The subtree of the node following the path of the selected action is preserved for future planning, and other nodes are deleted at the end of each planning episode to reduce memory consumption.

```
Input: state, piece
Output: reward
type ← stateType(state, piece);
switch type do
    case type == normal node
        action ← selectAction(state, piece);
        state,reward ← performAction(state, piece, action);
        updateTree(state, piece, action, reward);
        return reward;
    case type == leaf node
        return 0;
    case type == terminal node
        return -1;

endsw
```

Algorithm 2: One Planning Phase : Function **search**

In each planning phase, the planner selects and performs one of the actions for each piece in the given sequence. Each performed action results in a child node in the planning tree, together with a reward accordingly. Once all the pieces in the given sequence have been performed with an action, and the leaf node is reached, the planner sums up the total reward gained in the current search path, and updates the reward information backwards from the leaf node to the root.

In Tetris, a common game state usually does not have any information on the winning chance. To simplify the recursion on the leaf node, we consider it to be with zero reward, which means such nodes do not have any influence on their parent nodes in the search path. One exception is that, if a node contains a state of the field which is by Tetris rule a terminal state, the reward is always set to a big negative value. Such behavior makes the actions leading to terminal states less likely to be chosen by the planner.

The method for sampling the actions is the key feature of the Monte-Carlo planning. Comparing to the traditional uniformed and randomized sampling methods, the multi-armed bandit algorithm has advantages in balancing the trade-offs between explorations and exploitations during the planning process, and is proved to be more efficient than other methods in many domains[7].

2.2 Bandit Algorithm

In our method, we consider each state of the Tetris game field together with a given piece as a separate K-armed bandit machine, where K is the number of possible actions for the piece given.

Starting from the beginning of one planning episode, the number s of visits of a state in the search tree and the number t of visits of each action of the state and the piece are constantly updated according to each selection of the actions. According to the algorithm UCB1[1], the action selection is based on the upper confidence bound in track of the immediate reward and bandit score of every arm (action) of the bandit machine (game state) as:

$$I = \arg\max_{i \in 1,\ldots,K} \{R_i + c_i\} \qquad (1)$$

where R_i is the immediate reward from performing the action i, and c_i is a bias sequence chosen as:

$$c_i = \lambda \sqrt{\frac{\ln s}{t}} \qquad (2)$$

where λ is a constant factor manually chosen for balancing of the exploration-exploitation trade-offs. Higher λ values result in higher chances of randomized explorations based on the bandit scores, while smaller λ values lead to greater possibilities of selective exploitations according to the immediate rewards.

In the function of the action selection, first the bandit score of each action of the given piece is calculated by using the visiting information of the node in the search tree, then the immediate reward from performing the action is returned. The sums of the bandit scores and immediate rewards are used to rank all these actions. The action with the highest sum is chosen to be the return of the function. If there are multiple actions with the same highest sum, the result is chosen randomly from the list of such actions. Notice that the Equation 2 will be invalid for the nodes of the search tree where some of the actions are never visited before. For such nodes, an action is selected randomly from all of the unvisited actions.

In the standard UCT algorithm, the search tree is updated and pruned, and only the sub-tree of the state from the selected action will be kept for the next planning episode. For board games where game states are less likely to be revisited in the future steps of a single game, such behavior would have little influence on the future planning process. But the state revisiting happens quite often in Tetris because of its game rules. Therefore, the information of the explored states in previous planning episodes would play an important role in the Tetris planning. In the next Section, we will discuss Tetris state revisiting.

3 STATE REVISITING

Before designing our database for the visited game states, we think of the information that is useful for the future planning episodes. First, we want to start each planning episode of a root node from scratch. So the information of the number of visits to nodes and actions is not to be stored, because such information is a bias to the given sequence of pieces of the previous planning episodes. The immediate rewards and targeting states of the actions associated to one node can be easily and fast computed in the planning phases, thus the information can also be ignored.

In our method of planning, a node in the planning tree is made of a state of the game field and a given piece, and the information of the node consists of the following components which is necessary to be stored:

1. The highest reward over all the actions, and
2. The highest reward of each action.

The information of the item 1 is the key to our idea of storing and reusing the information of the explored game states, because it represents the summarized results of its associated previous planning episodes, and can be easily combined with the results of any future planning episodes. The information of the item 2 can be abstracted to a list of actions that matches the highest reward, which can be combined easily with the future planning results.

Considering the planner of the artificial player plays 100 pieces in a single game, and for each planning episode, the planner explores $1,000$ phases. Then the total number of explored states of a single game is approximately $100,000$. Assume that one quarter of these explored states are revisited states, then in the end there are $75,000$ newly explored states in a single game. Rather than saving every single node in the planning tree, we store only the root node in every planning episode. This way, the size of the stored nodes is significantly reduced, whilst the most useful information of each planning episode is preserved. Since the root node would only be queried once in each episode, the time cost for storing and retrieving nodes in the database is ignorable.

Now that we have our information stored in a database, the final task is to find a fast and easy way to load and save such information. In the following Section, we will present our design of the database using hash functions.

3.1 Hashing In Database

Although not all of the game states will be explored and stored in our approach, locating a specific state in a huge amount of data is still not an easy task. One of the possible options is to use an existing database management systems. However, such systems are inappropriate for our approach, as the data we want to store are small in the single size and have little relation to each other.

In our early approach, we tried to store the data in a single file, which is easy to implement. But locating the data of a certain game state is not an easy task. One of the possible method is to use the ”sparse file” system for storage, and every state is stored to a certain position in the ”sparse file”, where positions are calculated using a hash function. Since there is an ”offset limit” in the size of a ”sparse file”, it is difficult to create a perfect hash function to generate positions for the 2^{200} states in Tetris without collisions.

Another option is to use rather a simple file system with each state hashed to a specific file. Like the ”sparse file” system, the simple file system is also dependent on the operating system to locate the entry of a certain file. The difference is that the hash function can easily be created for the simple file system. Also because game states are stored in separate files, there will be no ”offset limit” of a single file, and thus the collisions of positions are easy to be avoided.

For any file system, the key issue is to balance the number of files, the number and depth of folders, and the size of each file containing the data. Too many files or subfolders in one folder could cause more time for the operating system to locate the entry of the target in the disk. The size of each file directly affects the computation time to store and retrieve the data for the program.

In our approach, each state of the game field generates a file name directly according to the value of the field encoded by a vector of integers. In this way, every state would have a unique file name, and the collision problem mentioned above is solved. Another advantage is that, the data of the game field is hidden inside the name of the file. And from another point of view, this method reduces the size of the storage.

Then, all such files are separated into different folders in a folder tree of 5-depth. In each depth of the folder, there are up to 16 subfolders. This scattering method is used to avoid too many subfolders or files in a single folder. Our later experiment on random sequences of pieces showed that for the Linux operating system the computational time had less than 1% differences in runtime from the beginning to the end of the experiment, where the number of the saved states in the database increased from zero to $100,000$.

```
Input: state, pieces
Output: action
initialization;
while not time out do
    search(state, pieces, 0);
end
combineKnowledge(state, pieces);
action ← selectBest(state, pieces);
updateTreeEx();
return action;
```

Algorithm 3: One Planning Episode : Modified Function **doPlanningEx**

In each planning episode, the planner starts planning from scratch, using only the information of the game state and piece and current planning tree. Then after the planning phases are completed, the planner loads the information of the root node in the planning tree from the database. Such information is combined with the newly explored information in the planning tree. The combination rule is simple. If the highest reward of the root node in the database is bigger than in the planning tree, the information in the database will completely override the information in the planning tree, and vice-versa. If the two rewards are the same, then the list of actions matching the highest reward will be merged. Algorithm 3 shows a modification to its previous version discussed in Section 2.1.

4 PRUNING THE PLANNING TREE

The previously introduced method is based on the sampling of all possible actions of the given piece in the given game state. However, many of the actions are not worth exploring, because they often lead to useless or even bad game states. In this section, we created a method to prune the planning tree to reduce the number of actions that need to be sampled, and thus to improve the performance of the developed player.

From records of many Tetris games played by artificial and human players, we studied that one of the most important features of the proper placement of a given piece in the game field is to avoid creating holes in the field. A hole in the game field is defined as an unoccupied cell that is covered by one or more occupied cells over its top. A row in the game field containing any holes cannot be removed, and would cause the player to spend more time to deal with.

Since many actions sampled by our method would create such holes, we created a method to prune these actions from the planning tree. The pruning is based on the increment of holes from the original game state to the resulted game state by performing an action. In addition to the reward and the bandit score of each action discussed in Section 2.2, the number of holes created by the action is considered to be a negative effect on the sum of the former two parameters. The Equation 1 is then modified as:

$$I = \underset{i \in 1,\ldots,K}{\arg\max}\{R_i + c_i + P_i\} \tag{3}$$

where P_i is a negative value defined according to the number of holes created, and is defined by the following equation:

$$P_i = \gamma H_i, (-1 \leq \gamma \leq 0) \tag{4}$$

where H_i is the number of holes created by performing the action i to the game field, and γ is a negative factor according to the number of rows removed from the result game field. The reason for γ being different is that, unlike the actions that can only create holes in the field, the actions that can both remove multiple rows and create some holes may still lead to a good game state, and thus should not be totally ignored.

5 EXPERIMENTS

We have conducted three experiments for our artificial Tetris player.

The first experiment was meant to test the validity of our method. In order to measure the performances of the developed player when using the database of the visited game states to support the UCT algorithm, we started the experiment on an empty database, and let the player repeatedly play Tetris on a fixed piece sequence from the start of the game till the end. Two of the game parameters are to be evaluated: a) the final score of the game, and b) the ratio of the roll-outs in one planning episode comparing to the standard UCT algorithm. The former parameter stands directly for the performance of the developed player, and the latter indicates the effectiveness of using the database of the visited game states in the future planning process.

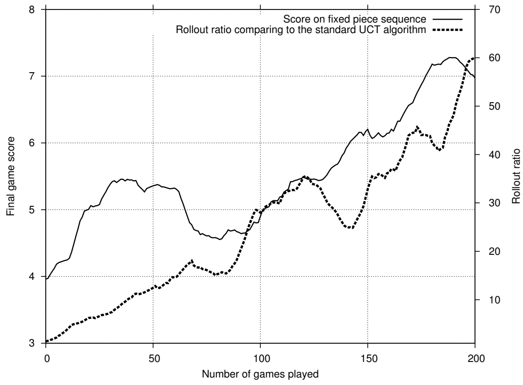

Figure 3: Experiment on a fixed piece sequence

The results of the first experiment are shown in Figure 3. The solid line in the figure displays the final score of each game. The game score is based on the sum of the number of removed rows in each turn. We use the same scoring rules for the rewards of removing rows in the standard Tetris. We can see that the final score grows as the number of played games increases. This shows that our method to store the information of the visited game states and reuse it in future planning process can successfully support the standard UCT algorithm and improve the performance.

Comparing to the standard UCT algorithm as a basis for the number of roll-outs per planning episode, the roll-out ratio of each planning process with the support of the database of the visited game states is shown in the figure with the thick dashed line. The number of roll-outs is piled up when the state is revisited in the future planning episodes. The results show that the knowledge database helps the standard UCT algorithm to do more roll-outs in the planning process when the states are found revisited in the database, and the performance is hence improved.

One observation in the experiment is that, although the trend of the two results is going in a growing manner, there exist some falls of scores and ratios at some point of the experiment. After analyzing this phenomenon, we found out the reason is that at some point of the game, some newly explored states produced some immediate rewards which are higher than those of the states of the previous planning episodes. This resulted in the change of the choice of the actions for the piece at the point of the game, and lead to some future states which are brand new to the player's knowledge database. We can also see in the figure that after some more explorations of the games, the final scores soon went up again.

The second experiment was designed to analyze the total size and the revisiting state coverage percentage of the database of the visited game states. Unlike the first experiment on a fixed piece sequence, we let our player continually playing the Tetris games with randomly generated piece sequences. The main idea is to let the player meet and explore as many unvisited game states as possible, while examine the percentages of states revisited in games with completely different piece sequences.

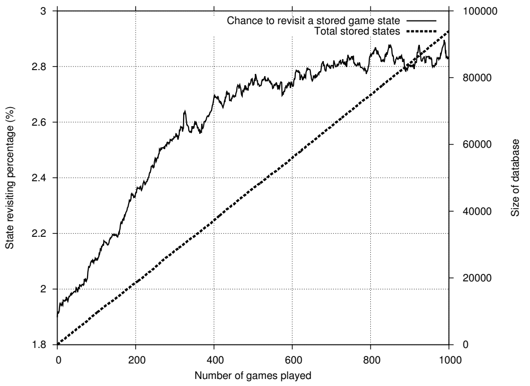

Figure 4: Experiment on random piece sequences

From the results of the second experiment displayed with the dashed line in Figure 4, we can see that the total size of the stored game states in the knowledge database is growing with a constant factor in our case. We thus can conclude that given randomly generated piece sequences, the player is able to increase its knowledge of the Tetris game.

Shown with the thick solid line in the figure, as more visited game states are stored in the database, we can see that the percentage of state revisiting is increasing. This means that is more likely to revisit a previously explored game state, even when the game has a completely different piece sequence. On the other hand, the percentage of states revisited during games with different piece sequences is still low, which means that currently the size of the database is not big enough to cover many useful Tetris game states.

Combining the results of the first two experiments provides evidence that our artificial Tetris player can successfully play the standard Tetris game using the UCT algorithm, and has the ability to learn from the played games and improve its future performance with the help of the knowledge database of the previously visited game states. By repeatedly playing the Tetris games using randomly generated piece sequences, the performance of the player will improve, as more useful game states will be covered by its knowledge database.

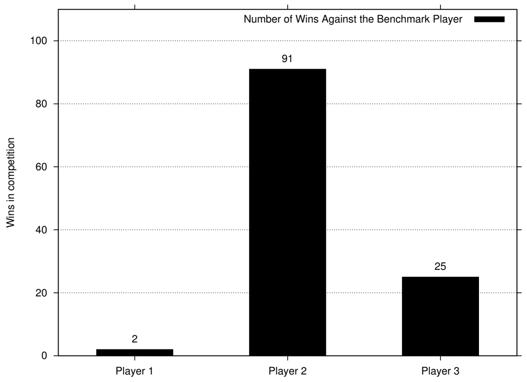

Figure 5: Competitions between players

The third experiment was the competitions against Fahey's benchmark player[6] using different approaches. The winning percentages are calculated on a basis of 100 rounds of games, and the results are shown in Figure 5. The three columns represent the different approaches used to develop the player: 1) the UCT based player with only the hashing database, 2) the UCT based player with both the hashing database and the pruning method, and 3) the SVM based pattern player in our previous work[14].

As we can see in the figure, with only the hashing database, the player's performance against the benchmark is very low. With the support of the pruning method, the performance of the player is significantly improved, as shown by the number of wins in Figure 5 increasing from 2 to 91 over 100 games, and is competitive comparing to our previous work of the SVM based pattern player.

There is a trade-off in including pruning in our approach. The complexity of the included method affects the efficiency of the planner in both the action selection and the roll-out sampling. According to the statistics of the experiments, the number of roll-outs per planning episode dropped by $1/2$ when the pruning method is included, while the number of pieces successfully played per game raised by 25 times. Conclusion can be drawn that such method is suitable in our approach to improve the performance of the developed player.

6 CONCLUSIONS

In this paper, we have developed an artificial Tetris player using the bandit-based Monte-Carlo planning method. Different from many existing artificial Tetris players, our player is built on the ten-piece planner. Our idea is to use the Monte-Carlo planning method to sample the possible actions of the given pieces in the game field, and use the bandit algorithm to balance the exploration-exploitation trade-offs and guide the planning process.

One of the key challenges of our work is to find a good solution to make use of the information of the visited states during the planning process, as such information is not kept and reused in the standard UCT algorithm. We created a method to store the information of the visited game states in a specially created database file system. The information can be loaded and reused in the future planning episodes when the states are revisited, and the scheme provides our artificial player with the learning ability.

The high branching factor causes the planner to spend much of its time exploring possible actions, while many of such actions are useless and often lead to unwanted game states. We created a method to prune the planning tree during the planning process to reduce the number of actions to be explored, and eventually improve the game performance of our player.

The experiment results show that our player can successfully play the Tetris game. By using the stored information of the visited game states as a support to the UCT algorithm, the results of the experiments show that the performance of our player improves as the number of games played increases. The player could explore the unvisited Tetris game states using randomly generated piece sequences and improves its game performance. With the pruning method, the developed player has significantly higher chance to win a multi-player Tetris game in competition against the benchmark of the Tetris players.

6.1 Future Work

The results of our second experiment on randomly generated piece sequences showed that our database has not yet covered a high percentage of the useful game states. In the next step, we will continue the experiment on exploring unvisited game states for the database, and analyze the use of larger database in the Tetris games.

Currently we use an intuitive method to prune the planning tree. Although the overall performance of the developed player is improved, careful studies are needed to analyze the trade-offs between more complex pruning methods and the changes in the player's performance. This is another interesting topic be in our future plans.

REFERENCES

[1] Peter Auer and Jyrki Kivinen, 'Finite-time analysis of the multiarmed bandit problem', in *Machine Learning*, pp. 235–256, (2002).

[2] Hans J. Berliner, Gordon Goetsch, Murray Campbell, and Carl Ebeling, 'Measuring the performance potential of Chess programs', *Artif. Intell.*, **43**(1), 7–20, (1990).

[3] Niko Böhm, Gabriella Kókai, and Stefan Mandl, 'An evolutionary approach to Tetris', (2005). In Proceedings of the sixth metaheuristics international conference (MIC2005).

[4] Bernd Brügmann. Monte-Carlo go, 1993. Unpublished technical report.

[5] Michael Buro, 'From simple features to sophisticated evaluation functions', in *Computers and Games, Proceedings of CG98, LNCS 1558*, pp. 126–145. Springer-Verlag, (1999).

[6] Colin Fehey. Tetris AI. `http://www.colinfahey.com/tetris/tetris_en.html`, 2003. www accessed on 02-August-2010.

[7] Levente Kocsis and Csaba Szepesvri, 'Bandit based Monte-Carlo planning', in *In: ECML-06. Number 4212 in LNCS*, pp. 282–293. Springer, (2006).

[8] François Van Lishout, Guillaume Chaslot, and Jos W.H.M. Uiterwijk. Monte-Carlo tree search in Backgammon, 2007.

[9] Arthur L. Samuel, 'Some studies in machine learning using the game of Checkers', *IBM Journal of Research and Development*, **3**(3), 210–229, (1959).

[10] Jonathan Schaeffer and Robert Lake, 'Solving the game of Checkers', in *Games of No Chance*, pp. 119–136. Cambridge University Press, (1996).

[11] István Szita and András Lőrincz 2, 'Learning Tetris using the noisy cross-entropy method', *Neural Computation*, **18**, 2936–2941, (2006).

[12] Christophe Thiery and Bruno Scherrer, 'Building controllers for Tetris', *International Computer Games Association Journal*, **32**, 3–11, (2009).

[13] Takuma Toyoda and Yoshiyuki Kotani, 'Monte Carlo Go using previous simulation results', *Technologies and Applications of Artificial Intelligence, International Conference on*, **0**, 182–186, (2010).

[14] Dapeng Zhang, Zhongjie Cai, and Bernhard Nebel, 'Playing Tetris using learning by imitation', in *GAMEON*, pp. 23–27. EUROSIS, (2010).