Youtube link: https://youtu.be/xRo-5MrllWQ

COMP90054 Azul Project Report

Azul_project_group_18

1. Techniques

Our team implements three techniques: Monte Carlo Tree Search (MCTS), A star (A*) and Breadth First Search (BFS). The final agent uses the Monte Carlo Tree Search algorithm.

1.1 Monte Carlo Tree Search

Monte Carlo Tree Search has been widely employed in gameplay. The most well-known case is that the AlphaGo defeated the Go master Lee Sedol in 2016.

The 2-player Azul is similar to Chess or Go where each player moves in turns trying to maximize the net gain. The difference is that in Azul, when a round ends, a new round will be initialized and players then compete from that point. Each round can be modeled by a game tree. MCTS searches the tree and decides the best move to act based on simulations.

In the MCTS algorithm, each node stores the current game state, the ID of the current player, the untried moves, the parent node, the last move that leads to this state, the children nodes and the number of visits and wins. The search starts from the root node. If it's not fully expanded, i.e. it has untried moves, it will be expanded by executing a move from the untried moves and a new state deriving from this becomes its child. The new node will then be rolled out to simulate the process where players take random moves in turn until the round is ended. After the simulation, the result is then back-propagated up to the parent of the rolled-out node, the parent of its parent… until the root node. If the simulation result turns out that the current player of the rolled-out node wins, the wins of its parent minus one and vice versa. Because the parent of the rolled-out node took the step leading to it thus the result of the roll-out is contrary to the value of that step.

The algorithm keeps doing the above steps until the root node is fully expanded. Then the algorithm implements UCB to choose the best child. The method stops when it reaches the simulation times. Then it returns the last move of the best child of the root node as the selected move output.

1.1.1 Challenges and improvements

a) Random roll-out: Usually the roll-out policy in MCTS is randomly selecting a move. The downside is that it might derive an inaccurate result that leads to the wrong estimation of the value of a move.

To improve this, in the roll-out stage, our algorithm sorts the available move list by the number to floor line ascending and number to pattern line descending of a move and then the next player to move chooses the first move in the sorted list, which has the highest number to pattern line while the lowest number to floor line.

b) Limited simulation times: Since the time limit of each move is 1 second, the simulation times cannot be set too high. The cost is that insufficient simulations cannot precisely reflect the value of a step. To address this, three improvements have been made. First, the number of nodes to be expanded is limited. In other words, the untried moves of a node are reduced. To achieve this, the untried moves list is sorted by number to floor line ascending and number to pattern line descending, which ensures that the front of the list is quality moves. And the size of the move list is reduced to 1/10 of the old one after experiments i.e. we only expand 1/10 of the available moves. Besides, the total moves of the roll-out stage are limited to 3 instead of moving until the end of the round. Since the simulation times are limited, to focus on exploitation, the weight of the exploration part of UBC is set to 0. Initially, the maximum simulation times were around 200. After being improved, the algorithm can simulate 300 times.

1.1.2 Strengths and weaknesses

a) Strengths: Randomness in the roll-out stage has been replaced by sorting the move list and selecting the most promising one to derive a more precise value of the step that the parent of the rolled-out node took. Untried moves in the expansion stage have been reduced and sorted and moves in the roll-out stage have also been limited to 3 steps to improve the simulation times. Intrinsically, MCTS is a heuristic search algorithm. The winning rate is the

heuristic value of a node, which derives from the simulation. As the simulation times increase, the quality of the move selected improves.

b) Weaknesses: The moves to be tried when expanding a node and the moves taken by each player during the roll-out are selected by sorting the origin available move list in an order that makes moves with the highest number to the pattern line and the lowest number to the floor line stay in front. The policy here is to try best to fill up the pattern lines without the least overflow pieces. While as human players, sometimes we'd rather overflow one or two pieces to the floor line to gain more points or avoid more loss and this algorithm cannot capture this.

1.1.3 Further improvements

a) The native deepcopy method is slow and it's the bottleneck of improving the simulation times further as it is used in both the expansion and roll-out stages. It would be better to implement a customized copy method to clone the game state input.

b) Instead of just focusing on the pattern lines, it'd be better to also check the grid. As human players, we'd like to place tiles in a way that makes them next to each other to earn more bonuses. We can start by completing the tiles in the center of the grid, which makes the connection easier. Or we can also check our opponent's pattern lines and the grid to see what type of tiles will overflow him and try to leave that type of tiles to him.

1.2 A*

A* is a heuristic search algorithm. We can use a priority queue to select the best move based on the evaluation policy here. The evaluation of a move is composed of two parts. 1) The number of empty slots left. For example, if a pattern line needs four tiles and after the move it still has 1 empty slot, the score for the move is −1. 2) The points that a move can gain given the current game state.

Given a game state, for all the available moves, we first calculate the 1) score mentioned above of a move as h*. Then we will evaluate the extra points for this move. The heuristic function calculates the difference between the extra points and h*.

1.2.1 Challenges and improvements

Overall, the A* algorithm is better than blind search algorithms, which have no search goal. It may also be possible to make guesses about the opponent's movements and then simulate them in the future. In this way, it may be possible to better determine the current move by exploring a longer path. This can beat the naive player, but it is still not optimal.

1.2.2 Strengths and weaknesses

The A* algorithm tries to execute the move that gains the most points and leaves the least spaces in a pattern line. This policy works in some cases. On the other hand, A* is more suitable to search with a clear goal. However, because both players are changing the game state in this game, therefore a player cannot statically search forward in sequence from the current position. The currently implemented algorithm only selects moves based on the current game state.

1.2.3 Further improvements

A* can be improved by simulating the opponent's move before searching. It will increase the total judging criteria in a single round, considering the time restriction. If there are too many actions in one round, A* may exceed the total time. We can find a threshold for the total action in one round. If the action is above the limitation, we can use another algorithm instead.

1.3 Breadth First Search

BFS is a commonly used queue-based blind search algorithm that can be implemented in board games. We can apply the BFS algorithm to select the best move by pushing all the possible new game states deriving from the game state input and the move leading to a new state into the queue. Then we dequeue an element from the queue and push all possible states deriving from it and the original move that leads to the state of the dequeued element into the queue again. This process repeats until we reach the time limit or there are no available moves for a state. And for each element in the queue, we calculate the end game score of the state in the element and select the one with the highest score and the move in the corresponding element becomes the best move.

1.3.1 Challenges and improvements

Due to the limited search time, it is impossible to use BFS which usually needs lots of memory and time to search all states to get the optimal solution. Therefore, our core work is how to prune the BFS searching tree in order to get the optimal solution within the specified time. We found that even if there is only one search layer, the search time in the first round will exceed the standard. It is precisely because of this situation that we ultimately decided to just search 1 layer. Unfortunately, even with only one search layer, we still occasionally have timeout errors. We have to hardcode to set the search time within 0.95s which means our program will automatically stop searching and select the best move in current results.

On the other hand, we found that the deepcopy function is very time-consuming. We sorted the current results in ascending order according to the number of tiles which move to the floor line. Obviously it is unreasonable to move a large number of tiles to the floor which will bring a lot of deductions. Finally, we only select the optimal solution from the top 10 results, which saves a lot of time.

1.3.2 Strengths and weaknesses

BFS is an optimal algorithm. When time is not the main factor, BFS can return the best results. For example, when there are only a few tiles left on the field or only a few empty places on the board, there are few possible movement results for BFS. We can search all the states in 1s. However, when there are many search results, the effect of BFS will be poor. For example, at the beginning of each round, BFS is usually interrupted due to a time limit. Even if we choose the best move among the existing results, the selection result may not be globally optimal.

1.3.3 Further improvements

There are two ways to improve the BFS algorithm. First of all, when there is more time, we can explore deeper layers in order to obtain a better solution. At the same time, we can assume that the opponent is a naive player, so that we can explore more states.In addition, we can use the BFS algorithm to explore the opponent's state. If a certain type of tiles is both what we need and what our opponent wants, then these types of tiles will be set with a relatively high reward.

2. Performance

2.1 Results of Against Each Other

The result of just one experiment is accidental. In order to ensure the objectivity and fairness of the experimental results, the following statistics are based on 50 experiments. First we use BFS and A* to fight and the result shows in following Figure 1:

```
Over 50 games:
Player BFS_Player earned +32.62 points in average and won 19 games, winning rate 38.00%;
Player A*_Player earned +38.84 points in average and won 29 games, winning rate 58.00%;
And 2 games tied.
```

*Figure 1: Results of BFS and A**

Then we let BFS to fight with MCTS to fight and the result shows in following Figure 2:

```
Over 50 games:
Player BFS_Player earned +32.64 points in average and won 19 games, winning rate 38.00%;
Player MCTS_Player earned +37.60 points in average and won 30 games, winning rate 60.00%;
And 1 games tied.
```

Figure 2: Results of BFS and MCTS

Finally, the result between A* and MCTS shows in following Figure 3:

```
Over 50 games:
Player A*_Player earned +32.94 points in average and won 24 games, winning rate 48.00%;
Player MCTS_Player earned +34.48 points in average and won 26 games, winning rate 52.00%;
And 0 games tied.
```

Figure 3: Results of A and MCTS*

According to our code and data only, MCTS is slightly more powerful than A*, and BFS is the worst in 3 algorithms. We believe that BFS is an extensive blind search. In this experiment, a large number of BFS were interrupted due to premature time, which may lead to poor results of BFS move selection. MCTS tries to execute the move that has number to a pattern line and low number to the floor line while A* tries to execute the move that gains the most points and leaves the least spaces in a pattern line. The policies here are similar hence the performance is close.

2.2 Results of Against baselines

Similarly, the data results are based on data from 50 independent experiments. The Figure 4 is based on MCTS:

```
Over 50 games:
Player Red_NaivePlayer earned +26.04 points in average and won 12 games, winning rate 24.00%;
Player MCTS_Player earned +32.02 points in average and won 37 games, winning rate 74.00%;
And 1 games tied.
```

Figure 4: Results of MCTS

The results of A* is in Figure 5:

```
Over 50 games:
Player Red_NaivePlayer earned +27.20 points in average and won 18 games, winning rate 36.00%;
Player A*_Player earned +30.68 points in average and won 30 games, winning rate 60.00%;
And 2 games tied.
```

*Figure 5: Results of A**

The Figure 6 shows the results of BFS:

```
Over 50 games:
Player Red_NaivePlayer earned +32.28 points in average and won 22 games, winning rate 44.00%;
Player BFS_Player earned +36.30 points in average and won 27 games, winning rate 54.00%;
And 1 games tied.
```

Figure 6: Results of BFS

In the confrontation with the naive player, MCTS achieved the best results, followed by A*, and BFS had the worst effect, which also confirms our previous conclusion. However, all three algorithms are better than the naive player which shows that even if there are very strict time constraints, the results of using the algorithm have improved. The naive player selects tiles that fill a row of the current board. After comparing different moves, BFS chooses the move that currently has the highest score. A* uses heuristic functions to preferentially explore nodes that are close to the target score. MCTS tries to execute the move that has a high number to a pattern line and low number to the floor line and this policy somehow limits the naive player.

2.3 Trial tournaments

We used the MCTS algorithm to pursue great scores. Historical results are shown in Figure 7:

58	Azul_project_group_18	20	1	215	236	2781	0	3801
52	Azul_project_group_18	23	0	189	212	2312	0	3462
40	Azul_project_group_18	31	1	136	168	1733	0	3303
32	Azul_project_group_18	29	1	114	144	1829	0	3299

Figure 7: Results in Trial tournaments

The agent used in these tournaments is not the enhanced version so we believe it will perform better in the final tournament.

Youtube link: https://youtu.be/xRo-5MrllWQ

Name: Zongcheng Du
Login: zongchengd
Student ID: 1096319

Self-reflection

In the process of completing AI projects in teamwork, I understood the importance of teamwork and team communication. In this experiment, each of our team members completed a part. After that, we compared the operation results of different codes and selected the best algorithm as the final agent. At first, we did not have a clear idea. However, after several code analysis and discussions, slowly everyone has their own views. Although each of us is responsible for the algorithm, when the team members encounter problems, we discuss in the group. In communication and cooperation, each of us has made progress.
However, we have some areas for improvement in this project. First, our time allocation is not very reasonable. After completing the basic content of the code, we did not have enough time to explore some algorithms outside the course. Secondly, when we encounter problems, we will discuss in the group, but there is no fixed meeting time. If we can regularly communicate the advantages and progress of each algorithm, our project will be smoother.

In the process of learning AI, especially in this assignment, I used AI algorithms to solve practical problems. This method has benefited me a lot. First, I try to consider how to apply the fixed algorithm to a specific problem. Secondly, I have a deeper understanding of the advantages and disadvantages of various algorithms. Finally, I found that using AI algorithms to solve game problems seems to be applicable to other games as well. The whole process is very interesting. I decided to try to use AI algorithms after the final to realize the intelligent players of Japanese Mahjong.

In our group, I am a participant in the entire project. I participated in the implementation of the BFS search model, tried to use the A* algorithm, and was responsible for writing many parts of the entire report. Finally, I modified and edited the video of the last demo. Through the research and discussion of the algorithms with team members, I understand the key ideas of various algorithms. In addition, I am the person who puts forward suggestions in the group. After carefully analyzing the code of the team members, I put forward my own opinions and corrections. In the report, I made recommendations to the content of the team member's report according to the task requirements. Every time my team members can discuss with me with an open mind and decide the best solution.

I think I have some improvements in this project. First of all, when I receive the task, I should have a clear schedule. In this way, after completing the basic code, I can have more time to think about how to improve move choices. Secondly, I have been thinking about how to choose the appropriate algorithm to solve this allocation problem. But one possible result is to use two or more algorithms to solve this problem. For example, MSCT is used when there are many possible movements, and BFS is used in a few moves.

Youtube link: https://youtu.be/xRo-5MrllWQ

Name: Ruofan Zhang
Login: ruofzhang
Student ID: 1029050

Self-reflection

Through this subject, I've learned multiple AI techniques and implemented some of them. This subject gives me a more clear idea of what AI is and how to approach it. In the Azul project, one thing haunted me was how to make the agent "think" as naturally as humans, since my agent frequently made bad moves. It selected the move in a somewhat rigid way while human players elastically apply different strategies. The reason is that I didn't tell it how to act in some specific scenarios and that's due to my limited competence but at least I tried and that's what I need to improve the most.

Based on the teamwork experience, I've learned that finding solid teammates and enabling an active team vibe where everyone is willing to share and help is important. Otherwise, you can only count on your own. It's also important to maintain harmony in the team after all this task can be hardly completed without collaboration.

Regarding the best aspect of my performance in the team, I coded the MCTS algorithm, finished my part of the project on time and ensured quality, motivated my teammates to complete this task, and helped debug when someone encountered problems in coding.

Youtube link: https://youtu.be/xRo-5MrllWQ

Name: HongCheng Xu
Login: xhc2081
Student ID: 1001893

Self-reflection

1. What did I learn about working in a team?

The most important thing in working with a group is brainstorming at the beginning of the project. Compared to the individual project, most of the time, we can only focus on one entry point to a project. Working in a group can combine unique perspectives from our group mate and produce the most effective solution. Furthermore, dealing with a group project creates enthusiasm for studying new knowledge rather than studying alone. There is another benefit that cannot be neglected, which is the skill advantages complementary. Everyone has their own preferred area in a group project. We can learn from other's energetic aspects and share the parts we are good at. Before we start our project, we need to build trust for each other. You cannot do teamwork with the one you do not trust. We did pretty good at communicating with others.

2. What did I learn about artificial intelligence?

As the project requirement, everyone needs to choose at least one technique in the techniques lists. That is the full content in this subject, and we should be familiar with them all. Maybe not at the beginning, I study them deeply through the project. The one thing that I consider most is how to use these algorithms. We know those concepts in the class, but how to implement it into a real program? I need to see the rules clearly and draw a diagram that helps me make the code realize. Focus on the algorithms themselves, blind search algorithms, heuristic search algorithms, and Monte Carlo Tree Search are the most familiar parts that I studied.

3. What was the best aspect of my performance in my team?

A* algorithm code
PPT + presentation first half
Some part of the Report

4. What is the area that I need to improve the most?

Debugging. This is a fantastic debugging experience.
Understand the logic of the code deeply.