COMP90054: Group Project

Zhaofeng Qiu 1101584
Aoqi Zuo 1028089
Alan Ung 834670

The University of Melbourne
June 3, 2020

Video link: https://www.youtube.com/watch?v=4tPvgutIbvI

1 Overview

When considering different techniques with which to experiment, it is crucial to consider the characteristics of the problem at hand. There is no one-size-fits-all AI technique; a given approach may excel at solving one problem yet be abysmal for solving another. Thus, we first identify the important characteristics of *Azul* below.

1. **Actions within a round are deterministic**
   In a given game state s, action a will always result in the same successive state s' if s and s' belong to the same round.
2. **The game state at the beginning of a round is stochastic**
   While actions within a round are deterministic as per (1), the state s_0^k (initial state of round k) is stochastic and is not determined by s_n^{k-1}. This is due to the randomised draw of tiles from the bag.
3. **The problem is adversarial**
   The goal of the problem is both to maximise the agent's score and win the game (i.e. end with a higher score than the opponent). Our goal is in tension with the opponent's goal.
4. **Perfect information**
   The entire game state is visible to all players. The factories, centre of the table and boards of either player are visible to both players; neither player has secret information about the game state.

This brings us to the techniques which we have implemented. They include Monte Carlo Tree Search (MCTS), Temporal Difference (TD) Learning and Minimax (with alpha-beta pruning). Our design decisions for the respective approaches are detailed below.

1.1 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning. Each search consists of a series of simulated games of self-play that traverse a tree from root state root until a leaf state is reached. MCTS consists of four phases: Selection, Expansion, Rollout/Simulation, Backpropagation.

The reason we choose MCTS is as follows:

1. *Azul* is a problem with a large state space. MCTS does not need to consider the whole state space and only spends its search time in more relevant parts of the tree. It visits more interesting nodes more often; thus, it does not waste computations on all possible branches.
2. In each turn of *Azul*, we are only given one second to make a decision. MCTS is an anytime algorithm, so it is possible to stop the algorithm at any given time to return the current best estimate.
3. MCTS is a heuristic algorithm. It can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions. It can find its own moves and learn rewards from random simulations. In this way, we can avoid the trouble of setting rewards for each step in the middle, which is time-consuming and may not be effective.

However, there are also some disadvantages for MCTS. The reliability of MCTS depends on the number of iterations. A high number of iterations is needed to be able to effectively decide the most efficient path. Especially in turn-based

games with vast combinations, when each of the nodes are not visited enough number times, MCTS can fail to find reasonable moves. In this project, one of the most severe challenges is to increase the number of iterations.

To increase the number of iterations with MCTS, we come up with the following strategies:

- Domain knowledge specific to the game is helpful. We filter out unlikely moves, such as placing all tiles to the floor. We also focus on more likely moves, such as placing many tiles to a pattern line.
- We produce rollouts similar to games that would occur between human opponents. This produces rollout results which are more realistic than random simulations. Further, nodes will require fewer iterations to yield realistic reward values. Rather than do random simulations, we utilise the strategy of `naive_player` to select moves.
- Instead of expanding any leaf node selected, we only expand a leaf node when it has already been visited more than once. In this way, our search is more broad and we can focus more on the next few steps.
- In the selection phrase, we select moves for our opponents by simultaneously learning the policy UCT for ourselves and our opponents. We choose actions for players by using the Q-function for the respective agent, based on their own reward function.

To improve the iteration times, we still come up with the idea to apply Greedy Best First Search at the first few steps of each round when the number of available moves is still too much and we only do simulations on those most interesting nodes filtered by BFS, Given additional time, we would like to see how our agent would improve by this.

Multi-armed bandits [1] is a classic reinforcement learning problem. When developing our MCTS agent, we found the main problem to be the lack of iterations. Thus, we developed the idea to only use multi-armed bandit algorithms like UCB1 and epsilon-greedy to explore and exploit all the move actions, choosing the best one based on the Q-value. In this way, we focus on analyzing the immediate choices. This concept is essentially MCTS with one layer, meaning that we only expand the root node and do selection and simulation only on the first layer of the tree.

1.2 Game Theoretic Models – Minimax

Minimax [2] is a well known two-player zero-sum game theory in artificial intelligence. The Minimax Algorithm tries to help the maximizer achieve the highest score while assisting the minimizer to achieve the lowest score. According to [3], the number of nodes in the game tree can be decreased with alpha-beta pruning. The reasons we considered this approach for our Azul agent are outlined below:

1. In the tournament, we only have one opponent. The two-player Azul game is turn-based, which satisfies the conditions for using the Minimax algorithm.
2. When we were improving our MCTS, we found the time constraint to limit the number of iterations. Since we can control the depth and reduce the number of nodes in the game tree with alpha-beta pruning, we envisaged being able to do more analysis using Minimax.
3. Our MCTS agent simultaneously learns policies to choose actions for its opponents in the selection and expansion step. This is like playing with itself. Compared with the multi-armed bandit algorithm, it may be more accurate to get the best action by trying each of them with the Minimax algorithm.
4. It is easy to generate a score for the minimax algorithm in two-player *Azul*. We use the subtraction of the scores of the two players in the current round as our evaluation value in Minimax.

In our implementation, we dynamically set the Minimax search depth because the number of actions will decrease with fewer tiles left in the same round. Further, as we only have limited computing power, before we search, we use some domain knowledge to prune the alternative moves, removing meaningless actions at the beginning.

1.3 TD Reinforcement Learning

Temporal-difference learning is described by Sutton and Barto [4] as "a combination of Monte Carlo ideas and dynamic programming (DP) ideas." TD methods, like MCTS, can learn without an MDP model—relying on *simulations* instead. However, unlike MCTS, TD methods do not wait until the final outcome is known before updating estimates [4]. This principle, known as *bootstrapping*, is the core point of difference between TD and MCTS methods.

We envisaged TD to be a likely successful approach when applied to *Azul* for similar reasons that we believed MCTS would be successful. Specifically, TD is model-free; as explained previously, developing an MDP model is a highly difficult task for this application. Moreover, like MCTS, TD learning is an anytime algorithm. This is a desirable feature due to the time constraint imposed in the tournament. We believed that TD learning may provide our agent with an edge over MCTS due to its bootstrapping characteristic borrowed from DP. Perhaps, we figured, it would learn more quickly than MCTS. The principle, elegantly explained by Sutton and Barto's analogy—driving home on the highway

and updating the live-prediction on the way (before actually arriving home)—was thought to be highly applicable to *Azul* also.

Thus, we implemented an agent using the TD(0) approach with Q-learning.

However, we quickly realised that due to the vast state space of the game, a simple tabular approach would not be successful. Even disregarding memory constraints, our agent is unlikely to visit most states. This is problematic as if we come across a state which has before been unvisited, our agent will not know what to do. To solve this issue, we implemented *linear approximation* of Q-values using a set of features. This way, similar states can be recognised by the agent to overcome the problem associated with tabular TD.

The task of selecting features proved to be quite difficult. Even after playing multiple human games of *Azul* among our team, it was difficult to pin down the most important features which encompass whether a state is desirable. Nonetheless, we devised a feature set (below) which we believed to be most sensible.

- For Each Row: The row occupancy (adding 'partial occupancy' based on the respective pattern line)
- Floor line occupancy
- For Each Colour: The degree to which a set (colour) has been completed (taking into account the pattern lines)

Feature details can be found in `players/Diamond_Three/features.py` of the `td0/modular-opponent` branch. Now, rather than update Q-values directly, we update the weight of each feature with respect to a particular action.

We designed our TD agent such that it takes another agent class to instantiate and use to simulate the opponent. We decided to make this modular and use the same interface (i.e. our agent calls the `SelectMove` method of the model opponent) in order to be highly flexible in how we simulate the opponent.

There were some significant limitations and shortcomings of our TD agent. Firstly, although an MDP model is not required, the behaviour of the opponent is still ultimately modelled indirectly by simulation. Thus, the less accurate the model opponent agent, the less effective our agent is. This is somewhat overcome by the effect of TD learning over more repeated episodes. However, this leads us to the most significant shortcoming.

The time constraint imposed in the tournament is far too restrictive for our agent to have any effectiveness without prior training. There are simply too many actions for which to learn weights within a one-second time limit. To have any effectiveness, our agent had to learn a set of weights beforehand, over many games, saving and refining the weights (i.e. carrying weights over from one game to another). This was implemented as saving our `Weights` object to a persistent file.

Ultimately, we found our MCTS player to be more effective. However, given additional time and resources, we would have liked to see how our TD agent would improve after having the chance to train extensively over a **broad range** of opponents in order to truly refine the weights. Perhaps, that is, after training extensively against other teams' players.

2 Assessment of the Final Tournament Agent

2.1 UCB1 and Epsilon Greedy

Our best agent uses UCB1 with $C = 0.5$ in our Selection process. The Formula of it is shown below. We also try 0.1, 2, 5 as well. But their result is not as good as 0.5. It is better not to encourage our agents to explore that much since the number of iterations we can do in one second is small. Epsilon Greedy also is worse than UCB1. Exploration actions in Epsilon Greedy are entirely blind to promising arms, and the initialization trick only helps with cold start. These limitations are severe in practice.

$$\mathrm{UCB1}(S_i) = \bar{V} + C\sqrt{\frac{\ln N}{n_i}}$$

3 Experiments and Evaluation of different Techniques

3.1 Experiments

For each of the techniques, we let them fight with our best agent. In our best agent, the exploration constant of UCB is 0.5, the discount factor is 1.

the result is shown below:

Technique	Special Setting	Result of it	Result of the best agent

MCTS	epilonPlayer01	wining rate=15%, average points=55.10	wining rate=75%, average points=65.75
	epilonPlayer005	wining rate=40%, average points=61.30	wining rate=55%, average points=64.20
	future Discount $C = 0.95$	wining rate=65%, average points=65.80	wining rate=30%, average points=62.75
	Mab_Only	wining rate=35%, average points=56.30	wining rate=65%, average points=63.90
	Minimax	wining rate=25%, average points=48.65	wining rate=75%, average points=59.65
	myPlayer	wining rate=55%, average points=66.20	wining rate=45%, average points=54.35
	UCB1	wining rate=55%, average points=63.75	wining rate=45%, average points=65.70
	UCB01	wining rate=65%, average points=66.30	wining rate=35%, average points=60.45
	UCB5	wining rate=55%, average points=62.55	wining rate=40%, average points=58.70

Table 1: Experiments

3.2 Game Theoretic Models – Minimax

The result of using Minimax is not good. One reason is that it is slow. We cannot do enough analysis in the first half of the round. We can only access about 300 nodes in the game tree on one second, which we found out based on the log generated by the test matches. We can make a rough estimate of the number of moves we can handle corresponding to the depth. For a two-depth game tree, we can handle at most $\sqrt[2]{300} \approx 17$ moves in one second. For a tree depth game tree, we can only handle at most $\sqrt[3]{300} \approx 7$ moves in one second. This means that most of the time, in a round, we can only search at depth 1. Even though we have used our domain knowledge to prune the moves when expanding our tree nodes, and have implemented Alpha-beta pruning, the number of moves at most of the time is still too large to do a deep search.

Another reason is that we cannot make full use of the one second. The number of search nodes increases exponentially related to the search depth. However, the number of moves goes down linearly in the same round. Not like MCTS, which allows you to check whether you still have time or not before you do the next iteration, the number of searches is defined while the depth is set in the Minimax. You cannot do more search even though you still have time.

3.3 Multi-armed Bandit Algorithm Only

The result is terrible actually. The reason is that when there are many moves, the end of the round is too far from the current state. The analysis based on an inaccurate deep simulation is pointless without a useful heuristic for the moves. We cannot get enough information without getting deeper. Also, we think that both the UCB1 and Epslion greedy we used cannot get enough iteration when the number of moves is large at the start of a round. When there are little moves left, it would be better to go deeper since the number of iteration is enough for a small set of actions.

References

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[2] M. Willem, *Minimax theorems*. Springer Science & Business Media, 1997, vol. 24.

[3] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

4 Self Reflection

4.1 Zhaofeng Qiu

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

- Cooperation is important. Aoqi and I each implemented our own MCTS. Although we did repeated work, we checked each other's implementation details and improved our understanding of the algorithm through discussion. With a deep understanding of the algorithm, we were able to improve the algorithm for the game.
- Communicate more when encountering problems. When we were implementing TD, there were a lot of bugs. We discussed how to solve them together and solved them efficiently. Also, through communication, we can learn from others.

ii. What did I learn about artificial intelligence?

- When implementing AI algorithm, not only should we consider whether the algorithm can provide a good result, but also we should consider the hardware it runs on. Some algorithms have better results than others, but may require more computation cost. For example, many deep neural network algorithms can provide very high accuracy, but they cannot run in embedded devices with poor performance. Comprehensive consideration is required when developing algorithms, and the amount of calculation required is an important consideration.
- Some model-free reinforcement learning algorithms can be built on the premise that there is no domain knowledge, but proper introduction of domain knowledge can effectively improve the algorithm.
- When the performance of the algorithm is limited by the computing power, we can try pruning, so as to apply the limited computing power to the right place.

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

- The AI we used to play the tournament is my implementation. I tried lots of different approaches to improving our AI's performance. I helped our group improve from 40th place to 7th place in the tournament ranking. In addition to learning a lot about artificial intelligence, I also played lots of Azul games online to learn the domain knowledge and apply them to our AI.

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

- Improve my efficiency, improve my leadership, and increase communication with teammates to avoid repeated implementation of some of the algorithms.

4.2 Aoqi Zuo

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

- Actually, I am really enjoying this team working experience. Cooperation and competition can further promote our learning enthusiasm and increase our research motivation. Zhaofeng and I both implemented our own MCTS algorithm at the beginning, but we all encountered some problems. After frequent discussions, we all solved the problem about the algorithm. Afterwards, in order to improve algorithm performance for *AZUL*, we developed our strategies and discussed with each other meanwhile. We played with each other's agent, and tried to go beyond each other. In this way, our motivation has been inspired and our agents' performance have greatly improved. Besides, I think my coding style improves a lot due to the guidance of teammates.

ii. What did I learn about artificial intelligence?

- There are many algorithms in the field of artificial intelligence. But for a particular problem, only some of them are suitable. So it's very important to analyse the problem detailedly beforehand, then choose the right algorithm. In this way, we can avoid doing a lot of useless work.
- Besides, applying different algorithms in different phrases of a problem is also very helpful sometimes, like for MCTS, in order to simulate more interesting nodes, we may try to apply Greedy Best First Search at the beginning.
- Moreover, domain knowledge is also very important to solve the problem effectively. And artificial intelligence is really an interesting area. I'm very lucky to have this subject and know more about AI.

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

- During the implementation of MCTS, I came up with some strategies to improve the performance of the algorithm, then I implemented these strategies and discussed the effectiveness with Zhaofeng. During the discussion, we were both inspired and tried to improve our agents in many ways.

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

- I often fall into a small problem and overlook the whole view. In fact, as long as we change our perspective , the problem may be solved directly. Moreover, compared to experiments, I always pay more attention to theoretical derivation. But in fact, experiment is the only way to test the truth. For example, when implementing MCTS, due to the low number of iterations, I thought about whether the implementation of the algorithm has gone wrong for a full two days. Later I felt that there was no need to always entangle the algorithm. In fact, there are many other methods based on domain knowledge that can greatly improve the number of iterations. And after several experiments based on domain knowledge, the results confirmed my thoughts.

4.3 Alan Ung

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

- Working in a team means more than splitting responsibilities. The team excels when everyone shares ideas and members share responsibilities. Collaboration, rather than divide-and-conquer, is the best approach. Friendly competition within the team is also certainly helpful but remembering that we work as one is key.
- Fixing a bug in code or understanding the mechanics of an algorithm is often best achieved by explaining it to the team. This is often the best way to uncover misunderstandings and also get everyone on the same page.
- A good team is all about communication. Frequent updates, sharing ideas and asking for advice are all important aspects.

ii. What did I learn about artificial intelligence?

- When deciding on what approaches to consider, the problem at hand must be carefully considered; there is no one-size-fits-all solution.
- Different approaches can overlap substantially. For instance, dynamic programming, temporal-difference and Monte Carlo methods can be thought of as lying along the same spectrum. By the same token, it pays to think outside the box and take particular aspects from different techniques.
- Thinking about the problem as a human, in addition to thinking about the algorithm, is important. This helps to gain domain knowledge to solve the problem.

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

- I encouraged our team to think about different approaches and explore temporal-difference learning. Even if ultimately, the agent I created was not used to compete in the tournament, we still learnt a lot about AI from exploring this technique.
- Further, I encouraged our team to discuss the problems that our agents were facing in-depth and communicate to solve the issues.

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

- It is very easy to become stuck down a rabbit hole when a problem is encountered. However, it should be remembered that any problem can be broken down and tackled, one step at a time. Taking a break and taking a holistic view can solve the problem.