COMP90054 Project: YINSH

Group 42 - Uninformed Search (Video URL: https://youtu.be/oILk9woU0FE)
Tuan Khoi Nguyen (1025294), Kuoyuan Li (1072843), Liam Saliba (882039)

Introduction

This project aims to build an Artificially Intelligent (AI) Agent that plays the board game Yinsh. Our team implemented agents involving Heuristic Search, Deep Q-Network, and Monte Carlo Tree Search (MCTS), with our best perfoming agent placing in the top 10 of the course tournament using Negamax Search with alpha-beta pruning and simple domain-specific heuristics.

1 Game & Representation

Rules Yinsh is a 2-player perfect information board game, with the aim of being the first player to win 3 rings [1]. The game begins with an empty board, each player taking turns to place their 5 rings. Then, agents take turns moving rings. A ring can move across 3 axes to any empty space providing that no other rings lie in between, or no counters and other spaces lie in between. When a ring moves, a counter is left in its old position; if it moves over any counters, they will be flipped, changing their colour. When a streak of 5 same colour counters is made, the player with that counter colour wins a ring, and selects one ring to remove from the board. In the course tournament, draws happen if there is no winner after 50 turns; agents are given 5 seconds for making their first move, and 1 second for each move thereafter.

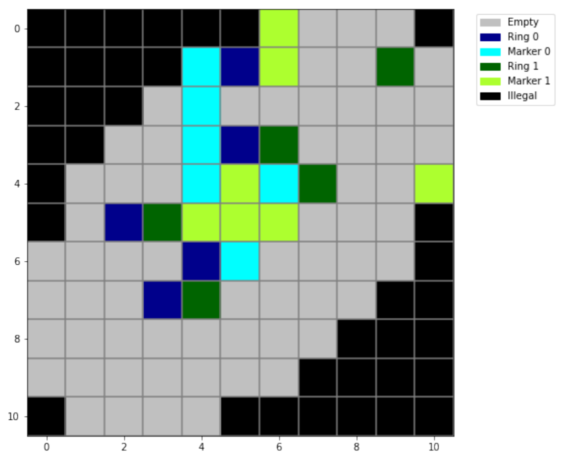

Figure 1: Square representation of a sample Yinsh game.

Board & Degrees of Freedom Yinsh can be represented as a partial 11-by-11 square grid where each ring can move vertically, horizontally, and diagonally in up-right and down-left direction, resulting in 6 degrees of freedom (Figure 1).

2 Heuristics

Given a 1-second time limit per action choice and an average branching factor of 35 actions, it is impossible to explore the entire game tree within the time-limit. Instead, we apply domain-specific heuristics to aid in search and state value estimation for agents. Yinsh is a zero-sum game, where an advantageous position for one player is an equally disadvantaged position for the other player. Hence, heuristics are generally computed by finding some score s from each player's perspective, and taking the difference between this score to determine the lead in a state between the two players.

$$h_s = s_{player1} - s_{player2}$$

Number of Rings (h_{Rings}) The aim of the game is to win 3 rings, and to get to 3 rings, each player must first win 1, then 2 rings. Significantly weighting the state of the game on who has the most rings is obvious.

Number of Counters ($h_{Counters}$) Having more counters on the board in your own colour makes it more likely that there are nearly complete sequences of counters. This gives agents a tendency to flip opponents' counters, preventing their formation of sequences.

Flippable Counters ($h_{Corners}$) One of the most common strategies of real world Yinsh players is to take control of walls and corners. This makes their counters unflippable, helping them to form sequences with less ability for the opponent to disturb. A heuristic that denotes the number of flippable counters on each player's side is implemented.

Potential Sequences (h_{Chains}) Using regex matching on each line, we look for feasible ring-counter sequences that can lead to a ring being won on next turns, from each player's perspective. Weighting is based on how many steps away to win a ring from each sequence. Having more opportunities to win a ring is considered a better position for that player.

```
RXXXX, XRXXX, ROOOOE, XXOXX, XROOOE, XXROOXE
R:Ring / X:counter / O:Opponentcounter / E:Empty
```

Figure 2: Few examples of feasible ring sequences

Visibility for Rings ($h_{Visibility}$) Rings are best placed where they have many cells to move to, so they can flip counters as needed. A fast estimation of this is to sum the number of cells in each row, column and diagonal of each ring. A better estimation is to take blocking rings and counters into account.

Weighting & Overall Function With the target being on winning rings, substantially higher weights are set to situations where a ring is won, or when a sequence that can lead to a ring won in the next step. Then, smaller weighting can be put for the flippable counters (FM) and potential sequences. To find the most proper weighting between these two, matches between multiple agents with different FM weightings are performed. For each match, greedy is used, and weighting is set to 2 for h_{chain} and 1000 for each ring won. With the best relative weight chosen for the agents from Figure 3, the final heuristic function becomes: $h = 1000h_{Rings} + 1.2h_{Corners} + 2h_{Chains}$

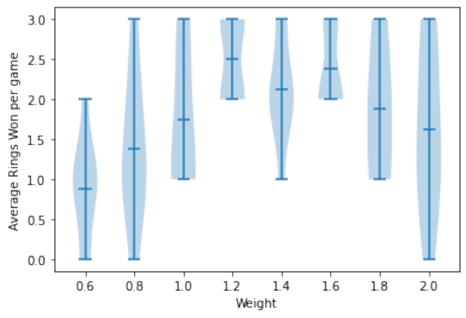

Figure 3: Match result, showing best FM weight of 1.2

Alpha-beta uses a different heuristic function that is cheaper to calculate for each state in the game tree:
$h = 10000h_{\text{Rings}} + 10h_{\text{counters}} + 1h_{\text{visibility}}$

3 Search Algorithms

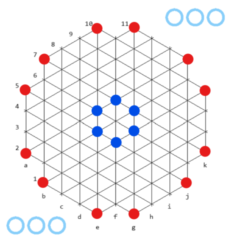

Figure 4: 18 potential placements for attacking rings (Template by sorhkrj)

Stages A game of Yinsh can be divided into 2 stages. Ring placement part happens at the start of the game, where players take turn in placing 1 ring to the board until each has 5 of them. Then the game play will start as stated in the game rules. As 2 stages have different actions, different methodologies may be needed to find the best next move.

Strategy-based Ring Placement The ring placement stage has players place 10 total rings onto the board with 85 cells leading to $\binom{85}{5}\binom{80}{5} = 7.8 \times 10^{14}$ possible states. Therefore, domain knowledge is once again used on choosing placement, based on the following criteria taken from common real-world strategies:

- Attacking rings must have access to as many corners as possible, so that they can create more unflippable counters. As shown in Figure 4, this narrows down the feasible positions to 12 corners in red and 6 central points in blue that can access multiple corners at a time.
- Defensive rings are placed either next to the most recent opponent ring, or at close line intersections between any 2 opponent rings to prevent them from creating connected counters.

Then, depending on the color, the strategy will be different. First player gets to choose the ring first, so it will set 3 attacking rings at most flexible positions initially, then place 2 defensive rings near opponent. Second player otherwise, can choose a position with less opponent disturbance at the end, hence the 3 initial rings will be near opponent's rings, and then the last 2 will be placed to a potential place that opponent have not accessed.

Greedy Search In Greedy Search, the action that generates a next state with highest heuristic value is chosen. As fewer states need to be examined, more computationally expensive heuristics can be used, aiming to prevent possible dangers and look for good sequences, which makes the agent quite informed. However, it may still suffer the horizon effect, where it can fall into traps that it cannot look ahead.

Negamax $\alpha - \beta$ Pruning To explore deeper than one action into the game tree, we opt to use a type of **minimax search**. In this framework, the player chooses moves minimise the opponent's maximum payoff from making good moves [2]. This matches the Nash equilibrium solution from game theory, where players choose the best strategies that maximise their performance in response to their opponents' moves. Yinsh is a zero-sum game, where an advantage for one player is an equivalent disadvantage for the other player. For a given state with value V, our player will aim to maximise this value, while the opponent aims to minimise this value, observing the value of the state as $-V$. With this observation, we can adopt **negamax search** to simplify implementation of the minimax algorithm[1]. Here, the heuristic value of a board state can be

[1]Despite this, we still struggled to get this working perfectly!

calculated with respect to our player, and is negated when we want the opponent's valuation of the board state.

Alpha-beta pruning is employed to decrease the number of board states that need to be explored to complete a search without changing negamax's return value. For a given action a, we stop exploring actions deeper than a (pruning) if we find one that proves a is worse than a previously explored action. At a given board state, alpha is the best (largest) value for the maximising player, while beta is the best (smallest) value for the minimising player.

Alpha-beta pruning requires an action ordering that enables better moves to be explored first, so prunes can happen as early as possible. We adopt the simple ordering where 'place, move, remove' actions are ordered before 'place and move', and actions closer to the centre are chosen first (one could similarly choose actions closer to the edges).

Deep Q-Network (DQN) DQN utilises a deep neural network to approximate Q-value of state and action pairs in Q-learning framework. As an offline model-free approach, it is theoretically feasible to train a deep neural network that generates an optimal policy for Yinsh through sufficient training. Our DQN approach implemented a Convolutional Neural Network (CNN) with encoded board state input, producing 1933 values indicating Q-values for all actions. Board state is encoded using one-hot encoder, resulting in $11 \times 11 \times 4$ feature map, each channel corresponding to one token type (ring/-counter of player/opponent). We only consider placing a ring and moving a ring, ring removal is chosen randomly. This influences the performance, but reduce the output space dramatically from 157165 to 1933, hence theoretically requires less training to converge.

We proposed 2 CNN training methods: matching another agent & through replays. For the former, DQN agent matches Greedy agent and follows a standard Q-learning training procedure, where ϵ-greedy is the Multi-armed Bandit algorithm when choosing explore or exploit [3]. For replay training, we downloaded over 1000 game replays among top tournament teams to train our model, as it could encourage our model to learn their good actions and reduce training time.

To improve training, we employed experiences replay (sampling past game experience to prevent divergence), main/target networks (2 same architecture networks with different training weights for stabilization) & reward shaping using the above heuristics for a good start.

Monte Carlo Tree Search (MCTS) The search starts with repeatedly picking a feasible action from current state based on a multi-armed bandit of choice, and perform simulation on it. In each simulation, MCTS randomly plays an action until termination state. Then MCTS backtracks & add the termination reward to each state's average reward.

Using the concept of probability convergence, the training process will eventually be able to pick the action that truly results in the best reward on average. In zero-sum games, reward is usually retrieved when the game ends, but to shorten the simulation, the termination condition is set to when either a ring is won or counters running out.

4 Results & Evaluations

4.1 Result Analysis

Evaluation Method To evaluate method performances in time-limited condition, a methodology based on ladder tournament is done: starting out with random agent on the top, each agent will match against the top agent on 20 matches in total - 10 matches of one going first and 10 matches of the other agent starting. The average rings won throughout each match is compared, and if the agent can win more rings than the current top agent, it will take the top position.

New agent	Top agent	Result	Winner
Greedy	Random	3-0.2	Greedy
DQN	Greedy	1.2-3	Greedy
MCTS	Greedy	1.5-2.8	Greedy
Negamax (Depth 2)	Greedy	2.7-0.9	Negamax

Table 1: Top ladder result of agents

Analysis With results above, Negamax is the agent chosen to be submitted to the tournament. It can be seen that heuristic-based methods (Greedy & Negamax) taking turns in being the top agent, proving that under limited conditions, the heuristic have efficiently aid the agent with sufficient information to evaluate the state. 2-layer Negamax as expected, adding an extra layer prediction to the Greedy search, which alleviates its horizon effect and improve it as a result.

Methods that requires training (MCTS & DQN) on the other hand, have worse performance, showing that while they may be efficient in the long run, it may be expensive to train or store the data. Therefore, under limited time and resources for both development and processing, more informative search is the preferred way to achieve a reasonable performance.

Informative vs Depth Trade-off The Greedy agent uses well-informed heuristics (2), but these are too expensive to calculate for each explored state in Negamax. Negamax instead uses a sum of the difference between players of the number of rings won, the number of counters on the board, and the visibility of rings on the board (how many cells each ring 'sees', estimating the movement of a players' rings). With these simpler heuristics, the agent can explore to a depth of

2 within the time limit, allowing the heuristic value of the opponents' actions in response to our actions to be evaluated and back-propagated. This complete search (with pruning) enables the Negamax agent to test each of its moves against an opponents' reply, so it does not trap itself on a single turn. Instead, Negamax is blind to traps that are set by its opponent beyond its horizon (at a depth of 4). Negamax can only concede rings to its opponent when all actions lead to conceding that ring. The tournament environment seems to run an order of magnitude slower than that of our development machines, where it is possible to evaluate at a depth of 3, but we noticed that this could misinform the agent, as opponent replies to those next moves could not be explored within the time limit, leading to the agent trapping itself. We observed that a complete search at depth 2 performed better than an incomplete iterative deepening search at higher depth (where search halted immediately at the time limit) as good moves could be missed, and blunders could be made.

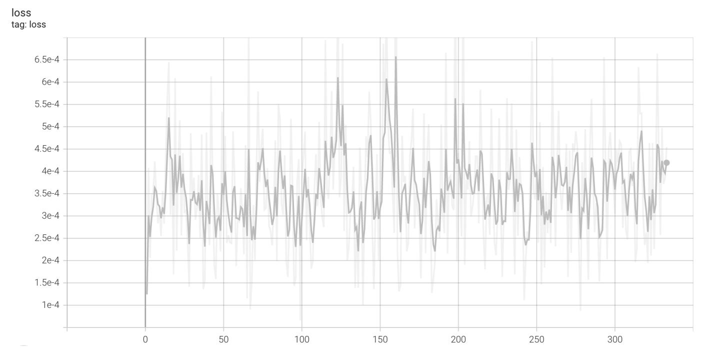

Figure 5: Last 300 games training loss of DQN model using method 1

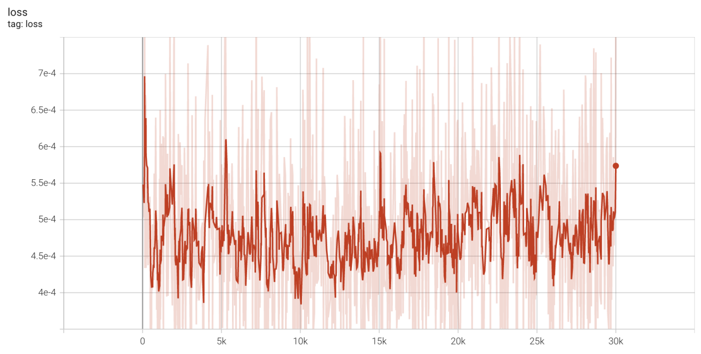

Figure 6: Training loss of DQN model using method 2

DQN Training Performance We trained our model for approximately 800 games using agent matches and found the training very slow, with the loss failing to converge (Figure 5). In addition, we found the agent makes few desirable actions, causing difficulty learning good actions.

For replay training, we trained our model for 1 week with 50000 epochs (1500 games). Despite this, training loss continues oscillating (Figure 6). The model plays reasonable actions occasionally but fails to defeat the Greedy agent. 3 potential reasons contribute to the failure of our DQN model. Firstly, the network architecture has large improvement space and needs further tuning. Second, training time is still not enough for the model to learn sufficient information. Third, for top teams with less stochastic strategy, less states will be observed, causing the model to overfit & fail to generalise on other unseen states.

4.2 Potential Improvements

As mentioned, due to time and resource constraints, many ideas were either unfeasible or is not implemented in time. They will be possible for implementation when further development resources is provided:

Fast action generation Our game rule implementation has optimisations to improve the speed of generating actions for all agents. This included caching the output of pure functions, hard-coding a deepcopy routine, and improving sequence detection. Overall, a reasonable 5x speedup is observed (0.5s to 0.1s to explore 2300 states in Negamax), but search is still bottle-necked in sequence detection. It is feasible to turn sequence detection into a vectorised dictionary lookup with a theorised 100x speedup, enabling deeper search.

Database Training Storing data and retrieve them when needed is a common and efficient method for training agents. Having this would help solving the resource problem, and continuously improve models after each run. Examples include average reward for MCTS or heuristics for search trees.

4.3 Conclusion

Overall, Yinsh is a game with large branching factor, which requires either long training or well-informed agent. Under limited time and resources, the latter has proven to be efficient, with our heuristic aiding both Greedy and Negamax to reach a very well-performed position on the tournament. Reinforcement learning methods, while not performing as good, may still be potential for future training when further development allows. Results from this project may become meaningful for further applications in real life, such as strategy evaluation or implementing a training game bot.

Bibliography

[1] M.J.H. Heule and L.J.M. Rothkrantz. Solving games. *Science of Computer Programming*, 67(1):105–124, 2007.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

A Appendix: Self Reflection

A.1 Kuoyuan Li's self reflection

What did I learn about working in a team?

Through this teamwork experience, I have learned a lot about both software development in a team and communication. I focused on implementing the Reinforcement Learning approach. During deciding which specific technique to attempt, I asked other team members, heard their advice and finally decided to implement DQN. During the implementation process, I also sought assistance from other players to improve the training. They suggested me to shorten each game time by stopping when a ring is removed, which accelerates the training process. After I implemented DQN and let it train, I assisted in debugging other's approaches such as negamax. These bring me experiences of effectively working with other people in a team on developing AI agents, and how to assist each other to create synergy.

During the coding, report writing and video recording phases, we actively communicated with each other face-to-face or via messages to discuss our thoughts and try to improve the quality of our agents, report and video. When I tried to debug others' approaches, I communicated with them to understand their code. These practices improves my communication skill.

What did I learn about artificial intelligence?

In this project, I implemented DQN and assisted in implementing negamax and Monte-Carlo Tree Search. The experiences of implementing various agents strengthened my understanding of materials from class. In lectures, we discussed Q-learning, value approximation and game theory, but not discussed their implementation in depth. By doing this project, I learned how to code these approaches from scratch, what are their limitations of them and what are potential solutions. Additional, I learned a lot of new techniques about reinforcement learning and game theory. In lectures, we only had a brief discussion on doing value approximation with the deep neural network. I further explored its details online and looked at some example codes. I also self-learned other techniques that are not covered in the lectures such as experiences replay and main/target network.

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

For AI agent development, I made substantial efforts and attempted a wide range of mechanisms to improve the DQN agent. I also assisted in other approaches such as MCTS and negamax. After I implemented DQN and waited for the training to finish, I helped other team members to complete or debug the Monte-Carlo tree search and negamax approaches. I successfully spotted some issues in the negamax algorithm and fixed them. I believe my efforts saved their time from completing and debugging. For other parts, I double-checked our submission to make sure no files are left. I finished my allocation on time with satisfactory quality.

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

For agent development, despite the suggestion from Tim and my tutor that only a few teams implemented DQN successfully in past years, I was still over-confident that I could build a DQN agent successfully and spent too much time on it. It took a very long time that I didn't expect. I didn't start this project very early due to another commitment. I should start early to leave enough time to train. For teamwork, I am very satisfied with the overall process. One of my issues is that I didn't update my teammate frequently on what I was doing and how it went. Due to the lack of communication, sometimes my other teammate did the part that I have already done.

A.2 Tuan Khoi Nguyen's Reflection

What did I learn about working in a team?

It has been a good experience working with this team for the project, and I have learned a lot for myself through it. First, same as every group work, I get to know more on a person's style of working, which from that, I can know if my own way of working is sufficient enough for everyone, and adjust it accordingly (like how those Reinforcement Learning models adjust their functions). I also learned how can each individual's strength can be combined for quality work. Our work seemed to flow nicely, with each member contributing incrementally to the overall success. Furthermore, as all of us - the team members - are also working at the same time, I realized that it is very important to keep track of everyone's work and being able to form a reasonable time management for the team is critical.

An additional thing is that during the project, all 3 of us had COVID, and I definitely learned on how to ask for help when being sick, as well as how to readjust the team workflow for sudden cases like this.

What did I learn about artificial intelligence?

In the aspect of artificial intelligence, I have learned through 2 ways: the subject content, and the online contents I found through researching for this project. In the subject content, I have learned about how to categorize a type of problem or method, how to represent a possible game state, and a brief introduction into Reinforcement Learning, where I know about Q-function and methods to learn this function as the game progress, such as Monte Carlo Tree Search and model-free Reinforcement Learning methods. On the internet, I learned methods that extend my knowledge from the course content. I learned how to form a better Yinsh heuristic based on real-world player and know how to make good strategy as a result, learned about common algorithm of zero-sum games such as Quiesence Search, Negamax and Minimax, Zobrist Hashing, etc. I also know briefly about distributed Reinforcement Learning, but to be able to apply it, I probably need more time and resources.

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

On agent development, I am proud of my ideation part, where my effort of shaping game strategies for the agent is successful. The heuristic that I invented worked out so well that with only Greedy Search, it is already around the Top 10. From this, my teammates have successfully extended and modified it into an operational Negamax tree, taking us even higher on the leaderboard. I also came up with a lot of other ideas that could not be done in time, but I definitely will be trying on those when the condition allows.

For group work, I am confident with my effort trying to keep everyone up to date with their progress. I was able to take initiative, without taking too much work into myself, and I believe that contributes to everyone's workflow.

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

For the agent development, I think my main problem is that I wanted to try too many ideas within a short amount of time. I felt that I was a bit greedy when we stuck at Greedy (as the best agent - pun intended), as we were so close to beating Advanced staff. I should probably try to think with more focus, and slowly incrementing the success.

For group work, I am fully satisfied with the team dynamics. If there is anything, maybe I will need to practice my communication ability. As I tried my best so that all members can keep track of everyone's progress, I ended up asking for progress or give out ideas quite frequently, and I believe it made my teammates thinking I was quite a bit rushed. While I do hope to help everyone keeping track of each other's progress and ideas, I would prefer to do that without letting my teammates getting nervous themselves.

A.3 Liam Saliba's Reflection

What did I learn about working in a team?

Tough time deadlines are tough. Working on multiple branches was very challenging, and divying up tasks was similarly challenging. I clashed a fair bit with what Khoi was working on, and had to keep up to date with what the group was working on, so time management was critical. Communicating who does what is very important.

What did I learn about artificial intelligence?

Through this project, I have learned a lot about AI, not only through the course content, but also the knowledge that I obtained off the internet while researching for model improvement. I have learned about how formulate my search problem, how to identify the game that I am trying to implement as a problem. Furthermore, I have been able to learn more about common zero-sum game algorithms through research, which will definitely help me on my future projects. I did not get to apply as much of the course content that I would have liked, which makes me want to look carefully over the course in my own break!

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

I was lucky to finally get a model working that defeated the advanced agent, with the help of my team.

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

Time management – and communicating what exactly I'm working on, planning what needs to be done.