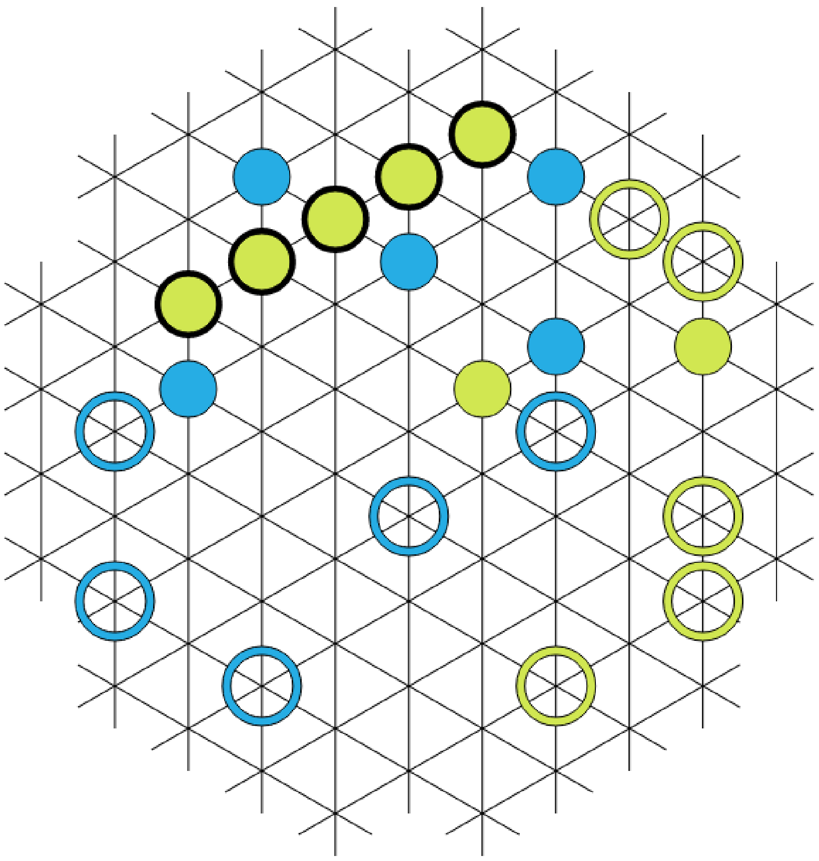


Autonomous Agents in the Game of Yinsh

AI Planning for Autonomy | COMP90054

University of Melbourne 2022 Semester 1

Oliver Ceff	542138	oceff@student.unimelb.edu.au
Oliver Bestel de Lezongard	914956	obestel@student.unimelb.edu.au
Tong Li	634325	tongl1@student.unimelb.edu.au

Youtube video:	https://youtu.be/ExCMmv0b7Ow
Github repository:	https://github.com/COMP90054-2022S1/comp90054-yinsh-project-group-53

1 Introduction

The following report investigates varying strategies for autonomous agents playing the game Yinsh. We explored blind search, heuristic search, Monte Carlo tree search, and Minimax Tree search. Each approach is discussed in the context of the intuition as to why it was chosen, parameters and specific methods implemented, strengths and weaknesses, evaluation of the approaches against baseline agents, challenges, and possible future improvements.

1.1 Yinsh

The game of Yinsh used for the process of evaluating our autonomous agents has the following restrictions:

- An initial five-second period before the game starts that can be used to set up the agent.
- A one-second timeout for each turn. Failing to adhere to this constraint three times would result in a loss.

1.1.1 General Challenges

Yinsh at a first glance presents some difficulty in terms of autonomous agents.

- It requires domain knowledge to assess counter strategies and foresee opportunities and threats.
- Its multiplayer nature requires planning moves ahead, causing difficulties in designing a good heuristic.
- When considering all possible results of multiple next turns, the state space begins to grow extremely rapidly. Specifically, starting positions in the placement phase result in an enormous state-space of size 7.9×10^{14}. If symmetries are considered then approximately 8×10^{13} starting positions are available (Heule, 2007).
- General difficult strategic choice: removing a ring also reduces mobility thus benefiting the opponent on a more open board; being offensive (building sequences) could benefit the opponent in the process.

1.1.2 State-Space Reduction Strategies

Given the enormous state-space of Yinsh a natural improvement is to reduce this size through a few strategies. Yinsh is a hexagonal board with only four types of pieces. Thus, in the early stages of the game, symmetry can be utilised to reduce the state-space. Regardless, this strategy diminishes in usefulness as the likelihood of symmetric states decreases in the late game. Other methods for state-space reduction include sudden death and threats (Heule, 2007). These methods involve recording state positions where the opponent is forced into a vulnerable position, in which you can assume they will not make a move that would result in the loss of a piece. In these cases, the strategy is to forgo iterating more states beyond these positions as it is a waste of memory. In Yinsh there is more potential for a sudden death threat, such as in the case where the state contains an opponent's ring, that if moved, would result in our agent scoring a sequence immediately. However, designing an agent to force the opponent into these situations as well as recognising these states is not a trivial task. It is shown that knowledge-based methods are more appropriate for solving games with a low decision complexity, while brute-force methods are more appropriate for solving games with low state-space complexity (Jaap van den Herik et al., 2002). Thus, it seems logical that for Yinsh we should focus on an informed heuristic, some programming-based approaches as well as a refined state-space.

2 Implemented Techniques

This section outlines the variety of approaches we investigated and implemented for the autonomous agents, as well as a reflection on their performance.

One of the core restrictions of implementing autonomous agents in Yinsh is that the entire state space is too large for simulation. This rules out approaches that derive policies or action values for each possible state. Thus our approaches all attempt to work on a subset of the state space. For the early stages of the game, we developed a number of programming-based approaches that would result in an advantageous placement phase. Following that we prioritise developing a useful heuristic for the game, as the standard goal-counting measure (forming a sequence of five) was often out of reach in the limited computational time.

2.1 Programming-Based Optimizations

The placement phase of Yinsh contains a lot of flexibility and freedom for where the agent places the rings which ultimately sets the agent up for the remainder of the game. The state space is enormous, and we found that developing a heuristic for this phase of the game is infeasible, so we developed a number of programming-based approaches.

2.1.1 Placement Phase - Prioritise Centre

In the ring-placement phase we elected to have the agent prioritise the centre of the board, which is the most common strategy as there is more potential for forming rows from all directions. This was done by sorting the actions by their Euclidean distance from the board centre. Upon placing subsequent rows the agent would ignore actions that would place a ring adjacent to existing rings. This prevents the agent from blocking itself whilst also having a relatively even distribution of rings around the centre.

We found that this strategy is much better than random placement. However, it lacks any complexity in considering the opponent's ring placement and does not create any offensive advantage by potentially blocking the opponent's rings.

2.1.2 Placement Phase - Prioritising Minimal Shared Lines

We implemented a strategy for the placement phase whereby the agent would minimise the number of shared lines between the five rings. At each location on the board, there is an intersection of 3 lines. By minimising the number of lines shared by each ring we in turn maximise the amount of movement capable of each ring. In doing so the agent is able to have more freedom earlier in the game. However, ultimately we found that the benefit of prioritising the centre of the board outweighed the benefits of this approach so that was the final chosen strategy for all agents.

2.1.3 Neighbouring Markers

We had our agent filter actions by only placing a marker next to already-placed markers, when possible. The incentive behind this behaviour was to prevent placing markers far away from each other by sequentially placing them next to each other. Although a rather naive strategy, it provided a useful advantage over placing markers randomly. Ultimately we were to develop a heuristic that behaved in this manner naturally, without having to programmatically code it in.

2.2 Baselines

Two baselines were implemented for the purpose of evaluation (Both provided by COMP90054 staff). These two baselines can be described as follows:

- An agent that picks a random action from the list of all available legal actions.
- An agent that uses a breadth-first search with a timer to prevent a timeout warning. The terminal state of this breadth-first search is an increase in score, meaning that it returns the first state it finds that leads to an increase in score. If the timer expires it returns a random action from the list of available legal actions.

2.3 Blind Search Approach: Combining Breadth-First Search (BrFS) and Depth-First-Search (DFS)

The general intuition behind this approach was that by combining BrFS and DFS and specifying at which depths each of them is used that the following could be achieved:

- At a depth of one from the root state a BrFS is used to find whether there are any rewards that can be achieved within that specific turn. If there are, the action that leads to the reward is chosen. If not we perform a DFS.
- The intuition behind this was that as we move past a depth of one, BrFS becomes ineffective as the state space grows too quickly to evaluate if there are rewards in one second. The DFS would then pick a path that places a marker at the beginning of a path that could lead to a point. The depth of the DFS was limited so that this path to a point was in a reasonable number of turns.

This intuition was incorrect and led to poor performance. Once you move past the BrFS and into the DFS, there are so many paths that could result in a point in multiple turns that it is practically the same as picking a random action. Furthermore, this strategy did not account for opponent turns.

2.4 Heuristics

We developed two heuristics to evaluate the agent's performance in any given game state which are detailed below.

2.4.1 Longest Line Heuristic

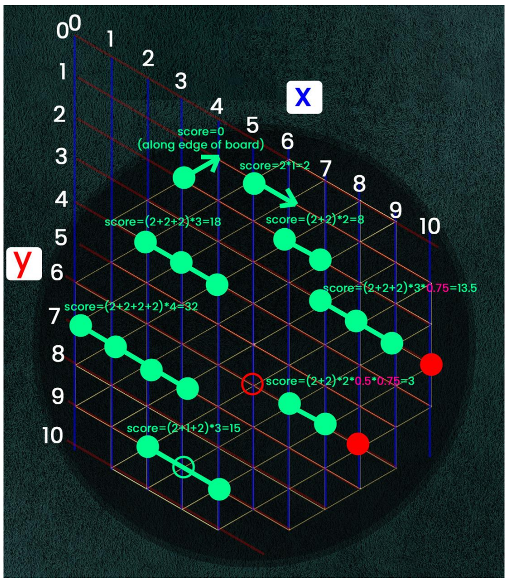
The line heuristic looks at the current board and counts the maximum number of the agent's markers that are in a straight sequence. The heuristic would then be five (the number required to get a sequence) minus this number. This equates to the minimum number of moves the agent needs to make to form a sequence of five. Unfortunately, this heuristic was found to be too simplistic, as there are cases when the agent is misrepresentative of how close the agent is to form a sequence. For example, When there are four markers in a row on the edge of the board there is no possible way for the agent to place a fifth marker down, so in this scenario, we had to omit the six edges of the board from being counted. In addition, the heuristic does not take into account the opponent markers and the possibility of flipping them. By taking into account markers that can be flipped, as well as the placement of rings on the board (and thus potential future markers) this heuristic could be improved.

2.4.2 Static Evaluation: Scoring System Heuristic

We implemented an arbitrary scoring system to evaluate the agent's performance on the board. With the weaknesses of the lines heuristic described above taken into account, the aim was to evaluate the performance of the agent with a number of things in mind: the number of sequences across the entire board (not just the longest), the advantage in having rings and markers in a sequence (as rings can be a marker in a future move), and whether the sequences are blocked by the opponent.

The below figure shows some examples of this scoring system. In general, a marker is worth two points and a ring is worth one point. Longer sequences are worth significantly more than shorter sequences, and a sequence is penalised by a factor if it is blocked by an opponent's marker or ring (the factor is higher if it is an opponent's ring - as these cannot be flipped).

The heuristic algorithm iterates over all marker positions in the board and checks for sequences along the three intersection lines with the marker positions. We empirically tweaked a number of hyperparameters such as the marker and ring values, and the block penalisation factors.



Examples of scores using static evaluation

This heuristic provided a significant boost in performance over the line heuristic as it allows the agent to build multiple sequences across the board at once. If one sequence is blocked, then the agent will prioritise building another sequence. We further strengthened this heuristic by evaluating the opponent as well. Then by taking the difference between the agent and the opponent, we find an objective evaluation that portrays how well the agent is doing at any given time. We continued to improve this heuristic and used it in a variety of search and simulation algorithms described below.

2.5 Heuristic Search

Using the static evaluation heuristic described above we implemented two search algorithms.

2.5.1 Greedy Best-First Search (GBFS)

Using GBFS we had our agent explore the next set of available actions using the heuristics described above. If an action was found that had a better heuristic than the current state then the agent would record this action as the best action to take. After the final iteration, the agent chooses the one with the best heuristic. The limitation of this approach is that the agent only looks at the next immediate action, rather than simulating more actions in the state-space. Overall the GBFS agent achieved a higher win rate against a random agent: over 100 games earning 1.75 on average and 81% win rate vs. the random agent with 1.01 on average and 43% win rate. However, this approach performed poorly against a breadth-first-search algorithm scoring 1.68 on average with a win rate of 38%; vs. BFS scoring 2.42 on average with a win rate of 80%. The heuristic could be improved by considering only up to *n* longest lines (with suggested values for *n* being 3-4) as this would encourage the agent to not create lines all over the board. Additionally, factoring in threats and sudden death positions as a large boost to the heuristic value would make the agent more aggressive in forcing the opponent into vulnerable positions.

2.5.2 Breadth-First Search (BrFS) with Heuristic Fallback

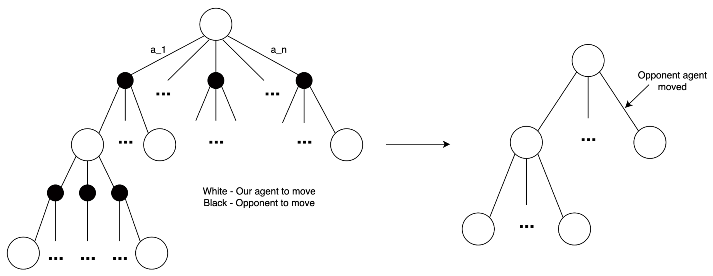
Given the pitfalls of the greedy approach, we utilised a combination of BrFS with the scoring system heuristic described above. We ran a standard BrFS, upon reaching a time-out the agent chooses an action with the highest value for the scoring system heuristic. We found that this agent performed the best overall. Against traditional BrFS the agent scored 2.7 on average with a win rate of 80% over 50 games. This strategy found a good balance between refining the state-space and pure search.

2.6 Monte-Carlo Tree Search

2.6.1 Design

Due to the large state space, it is difficult for breadth-first approaches to reach greater depth within the one-second think time. We tried to mitigate this problem by designing a multi-agent Monte-Carlo tree. Firstly, the UCB1 bandit guides the agent to explore more promising nodes in the game tree. The tree is able to terminate training upon hitting the time limit and still return a somewhat informed best action.

To model the opponent's choice, the ExpectiMax tree was not suitable because the transition is not probabilistic. Also, if every potential counter move of the opponent is considered, it could result in unnecessary simulations and significantly increase training time. Instead, we opted to use a second agent for the opponent's move to make the transition deterministic. As shown below, it simplifies the problem and makes training easier.



As transitions are deterministic, the reward function only depends on the board information. Our candidates are the heuristics in Section 2.4. Scoring Heuristic is used for our agent. A quick but less sophisticated reward function that simply counts the number of counters on the board is implemented for the opponent agent.

2.6.2 Performance and Challenges

The performance is poor, barely beating the random baseline. This is because the number of simulations in each round is highly limited by time constraints. Also, no inheritance of the sub-tree in the previous round is implemented, since the actual opponent might not be in the expanded MCTS, and matching nodes would take too much valuable time.

2.7 Minimax with Alpha-Beta Pruning

The Minimax with alpha-beta pruning is a well-known strategy for autonomous agents in multiplayer games. It is a recursive algorithm that generates a heuristic value for leaf nodes, and depending on whether we are simulating our turn or an opponent's turn, the maximum or minimum values of a node's children are assigned to the parent node. The alpha and beta parameters are used to prune branches and subtrees to decrease the size of the search. Due to Yinsh's rapid state size expansion, our agent in Yinsh was unable to successfully run without timeout warnings at a depth of two using the heuristic in 2.4.2. It can be run at a depth of one, however, this defies the purpose of the minimax algorithm.

3 Final Tournament Agent

The final tournament agent utilised BrFS with the heuristic fallback option if no reward was found in the BrFS search. The strengths of the agent reside in the static evaluation heuristic being calculated initially followed by a BrFS. The heuristic allows the agent to quickly evaluate a state, both in terms of its own performance and the opponents, allowing it to make an informed action choice when it cannot reach a goal state.

The approach contained weaknesses, however. The hyperparameters could be tuned with respect to the value of a marker, a ring, or the factor to diminish a score whether it is blocked by an opponent. Additional empirical testing could refine these hyperparameters to get a better heuristic. Furthermore, the heuristic could be adapted to reward the agent for putting the opponent into vulnerable positions, such as placing a ring on the edge of an opponent's sequence.

4 Internal and External Evaluation

In the preliminary stages of the project the two agents given to us, namely random and BrFS, were used for baseline evaluation for our agents, 50 games each.

As the team progressed and implemented multiple agents, we designed an internal system to evaluate agents against each other. An element of randomness had to be introduced into the agents without it. For example, two of our initial heuristic search agents would play out identically in every game, due to the nature of our deterministic programming-based placement phase. During an internal evaluation, the team randomises a fixed number of moves for both agents at the start of the game and simulates 50 such games.

The following table summarises our internal evaluation, from which ranking can be derived.

		Winner			
		BrFS + DFS	MCTS	h-BrFS	GBFS
Loser	BrFS+DFS	-	37%	80%	75%
	MCTS	63%	-	88%	86%
	h-BrFS	20%	12%	-	10%
	GBFS	25%	14%	90%	-
	Random BL	88%	58%	100%	94%
	BrFS BL	6%	2%	92%	22%

Comparison of agents against each other and baselines

In addition to this, we found all our evaluations against a baseline, and against internal agents were consistent with their relative performance on the competition leaderboard. As such, the leaderboard was only used as a sanity check, and was not the first choice for evaluating our agents in most cases.

5 Further Improvements

We had to fall back to either working with a subset of the successor space or limit our depth in search to a small number. Improvements can be made from both angles. Searching through the entire successor space of a parent, as well as simulating more turns reduce the chances of overlooking promising moves.

Furthermore, a more sophisticated programming based placement phase can be implemented such as state-space reduction by symmetry, or by accounting for additional turns from the above improvements.

Dedicated countering strategies, such as forcing the opponent into one end of our four-counter sequence would also be a significant improvement.

We can improve the MCTS by increasing the number of rounds of training and simulations. Also, the replacement of the rollout function with the Neural Networks' decision process implemented by AlphaGo can be simplified and added to the agent to improve performance and accuracy.

6 Conclusion

We designed and implemented various AI agents playing the game Yinsh, where their advantages and disadvantages, potential improvements, and evaluation were discussed. BrFS with a heuristic fallback had the best performance overall due to extensive research on strategies of the game while implementing the heuristic. Overall, we found that for the game of Yinsh, an informed heuristic based on extensive domain knowledge provides the greatest improvement, combined with strategies that reduce the dimensionality of the state space to allow for more efficient searches.

7 References

Heule. (2007). Solving games Dependence of applicable solving procedures. *Science of Computer Programming*, *67*(1), 105-124. 1

Jaap van den Herik, Artificial Intelligence, & Rijswijck. (2002). Games solved: Now and in the future. *Artificial Intelligence*, *134*(1), 277-311. 1

Reflection

Oliver Ceff

What did I learn about working in a team?

Our team was very effective in distributing the workload in an efficient manner. I learned some useful ways in which to collaborate on a coding project, for example, for myself I spent a majority of time developing a useful heuristic whilst my teammates implemented other algorithms into the Yinsh API. From there, my teammates were able to utilize the heuristic I developed in their approaches.
We also spent a significant amount of time aiding each other in understanding the concepts of the algorithms we were trying to implement. We discussed a variety of approaches and collaborated in coming to conclusions on where it was best to focus our attention. For example, after agreeing that the state-space of Yinsh was too large for any naive blind search algorithms, or value iteration methods, we settled on focusing on a heuristic, MCTS and other more suitable approaches to the domain.
Overall, I learned that a proactive team that is decisive and consistent with deadlines is always more efficient in the long run. With these aspects in mind, our team performed very strongly and was a pleasure to work with.

What did I learn about artificial intelligence?

Firstly, from a few research papers on games and solvability, I learned a variety of tricks and techniques that can be used to minimize the state space. Whilst we ultimately found that implementing these techniques was not trivial, there is much room for improvement in the techniques we did use.
I learned much about many specific AI algorithms and the situations when they are appropriate. We spent a disproportionate amount of time working on developing code for an agent only for it to perform poorly, given the domain. From this, my takeaway is that it is important to understand the pros and cons of an agent before utilizing it in a domain, given it may already be apparent that it is not appropriate.
Lastly, I learned that often simpler is better. Model-free and simulation algorithms, as well as reinforcement learning, require substantially more development and testing time. Whereas it is often the case that spending the time developing a heuristic search algorithm with a simple search can outperform a poorly designed algorithm that is more complex.

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

I spent the majority of my time, at first, developing the programming-based approaches in the placement phase of the game. These strategies allowed our agents to start the game from fairly decent positions from the beginning.
Following that, I prioritized developing a useful heuristic as it was important for the agent to have a metric for how well they are performing in the game. This heuristic provided a major milestone in the agent's performance and was useable as reward shaping, and state-space reduction in simulation methods. After identifying the agent's increase in performance with the heuristic I merged this with the BrFS code to create our highest performing agent.

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

I need to improve my understanding of more advanced AI concepts beyond just heuristic search and classical planning. Whilst the concepts of simulation, reinforcement learning and model-free learning I understand, implementing them into the game of Yinsh was very challenging.
Integrating code with another API was also very challenging. An important aspect of code development is the ability to read other code and be able to use it. In this instance, it was challenging to interact with the code, and debug the code that I had written.

Oliver Bestel de Lezongard
What did I learn about working in a team?
Over the course of the project, I enjoyed the support of the team as I realised the solutions that we dedicated significant time to were not performing well. It was comforting knowing that we were all having a tough time. This shows that it is often good to have a team when confronting challenging projects. As a team, we worked together well. We efficiently prioritised different features in order to make the most effective use of our time. The range of skills that come with a team allows you to perform better across a wider range of areas, and learn about new skill sets at the same time. Having a team to bounce ideas off is one of the best aspects. Sometimes you may have a misinterpretation of a concept, and bouncing ideas off team members allows you to more easily detect these ideas and not go down the rabbit hole of implementing an incorrect interpretation with little success. I generally enjoyed team discussion of concepts and felt it helped me solidify the foundations of concepts I was not entirely sure about.
What did I learn about artificial intelligence?
I learnt that although artificial intelligence concepts can intuitively make sense, it can be difficult and tedious to implement them in systems that have many moving parts. Once our more technical implementations had failed to produce the results we desired, I remembered how Nir mentioned that often a good heuristic with a heuristic search is often the best choice. I furthered my knowledge on all the topics we implemented and learned that AI is not necessarily a topic in which working towards goals always results in improved performance. We dedicated significant time, and did not see the results we wanted. Implementing autonomous agents in games like Yinsh is difficult.
What was the best aspect of my performance in my team?
I think the best aspect of my performance in the team was my determination to try and improve our tournament standing despite the failure of concepts like MCTS, for which we had a lot of hope in. With a background in maths, I also believe I was a good person to bounce ideas off and can pick up concepts relatively quickly.
What is the area that I need to improve the most?
I need to find a better and more methodical approach to working through understanding big systems. Working on agents like MCTS (which Tong eventually got right) in this system was very overwhelming at first. I need to fine-tune my implementation of recursive approaches in algorithms.I am a lot less comfortable with recursion than iteration specifically when it comes to putting it into code rather than the concept making sense in my head.

Tong Li
What did I learn about working in a team?
Communication is a very important aspect of teamwork. Even though the team was only able to meet virtually, we responded to each other fairly quickly and bounced ideas off each other, allowing free flow of ideas whilst maintaining a steady project progress rate. Trusting each other is also vital for team success. At times implementations were not going the way we wanted, but we did not hesitate to allow room for experimenting and help each other out when needed. When someone was on a lead, we were able to quickly react to it and build on each other's success. A sense of purpose is what keeps a team going. The team had expectations of each other, as such there was a sense of calling and we not only tried not to let each other down but also pushed ourselves to our limit to progress the project.
What did I learn about artificial intelligence?
It is domain-dependent. While it is interesting, the technical implementation largely depends on the task at hand. That is, unlike supervised/unsupervised machine learning where popular packages like sklearn exist, classical AI planning, and reinforcement learning would require some domain knowledge and understanding of the framework before any meaningful implementation can be done. It is also very powerful. When done right, its impact is huge and far-reaching.
What was the best aspect of my performance in my team?
Personally, I would say I performed well in terms of going along with what the team needed. When teammates were in need I would try my best to help; when difficult questions were asked I was able to conduct research and report back to the team with well-thought-out ideas and evaluations. While I did contribute to the implementation of agents, I do think I am lacking in general programming ability. However, I tried my best not to let it affect my own and the team's progress and was able to make meaningful changes.
What is the area that I need to improve the most?
One area I need to improve is efficiency in programming. It was a great opportunity to learn from my teammates and see them in action. They were quick in terms of rationalising a novel idea and realising implementations in a timely manner. I realised I am lacking in programming ability and was motivated to constantly learn and improve.