Game Playing agents in Inflexion
COMP30024 Artificial Intelligence: Project 2 Report

Abhinav Sood and Tahmin Ahmed

1 Introduction

Inflexion is a two-player game played on a 7x7 hexagonal tiled repeated infinite grid with each player having the opportunity to either SPREAD or SPAWN tokens on their turn. The rules of the game are available in the games specification. In this report we outline a Minimax Agent that additionally utilizes $\alpha - \beta$ pruning to implement an adversarial game playing bot. Further, we outline other agents we experimented with, different strategies we used with different agents and how we evaluated our agents.

The report is structured as follows: in Section 2, we introduce the methodology we utilized to test agents. This testing method is essential to establishing why we rejected certain agents and is referred to throughout the report. In Section 3, we describe our final minimax Agent and the additional modifications made to the agent to better play the game. In Section 4, we discuss an alternative Monte Carlo Tree Search approach that we implemented, but found unsuccessful under the given computational and time constraints. Finally, Section 5 concludes the report with further discussion of related and future work that can further enhance our solution.

2 Testing Methodology

When comparing two agents, it is important to understand the agents are not completely deterministic. That is, there is often random shuffling that influences the chosen move and there are certain positions where multiple moves have the same heuristic value. In this case, an arbitrary choice can be made. Thus when comparing 2 agents, 1 game between them is never enough. For example, in an extremely rare edge case, a random agent might play optimally against a state of the art agent. Thus, to compare two agents, we use multiple trials. Since multiple trials can take hours, we limit the number of trials to a significant but plausible number like 20. We use bash scripts to automate this process. Further, in each automated test, the agent's colors are switched after each trial to enable comprehensive testing. A summary of the tests is available in the tests folder. Logs of each game in the trials are visible in the log folder. We note that in a very few cases, since the log files are overwritten if tests are re-run, very rarely are they incomplete/missing. However, the summaries aren't overwritten unless all trials finish, thus the summaries are complete.

3 The Minimax Agent (Final Chosen Agent)

The basis of our Minimax Agent is minimax search with $\alpha - \beta$ pruning. The Minimax Agent was developed to defeat a greedy agent that aimed to maximize the power difference between the player and the opponent each move. The board is stored as a simple dictionary similar to the configuration of the board in Project 1. Spreads are made efficiently using the modulus operator to loop around the board. The basic idea behind the algorithm is that you analyze possible moves in layers. In these layers, the player tries to maximise the heuristic and the opponent tries to minimise the heuristic. The returned move is the one that has the highest heuristic value after this repeated maximization minimization process. For a minimax agent to be optimal, it must repeat the min-max process until it reaches all terminal nodes. From there, we backtrack the value to the parent node and

select the child with the highest value. In this project, since it is not computationally feasible to perform such extensive search we make use of the same power difference heuristic used in our greedy agent. Our implementation of minimax is based on the implementation described in [5]. By using a depth-first like recursive approach, the algorithm uses minimal memory ($O(bd)$ where b is the branching factor and d is the depth). However, even after the application of $\alpha - \beta$ pruning, the time complexity remains exponential. We observed that a minimax agent with depth 2 was able to consistently defeat the greedy agent. By independently analyzing the games, it was clear that the minimax agent was winning not because it was playing great, but because the greedy agent was making multiple mistakes.

This necessitated the development of a stronger minimax agent described as follows.

3.1 Depth, Ordering and Pruning

The first obvious step was to increase the depth of minimax. The project was constrained to a time limit of 180 seconds for a player, for a game. The limit meant that a depth of 3 was enough to result in cases where the constraint was exceeded. Thus, we have to think of solutions that could speed up search. To test our agents, we used a depth of 3 as we had relatively significant compute time to finish the project.

At first, we considered changing the move ordering. If we explore nodes that result in α or β prunes earlier, we could potentially save lots of computation. In an agent named minmaxAgentOrder, we prioritized spreads of 6 stacks, then 5 stacks, and so on with spawns being the least prioritized. This order did not materialize in any significant speedups.We even considered that different move orders might be better for different phases of the game. For example, spawns might be more important at the start. Even after accounting for these changes, move ordering actually worsened the performance of the agent as compared to random ordering as shown by Figure 1. This was likely because our heuristic (power difference) didn't distinguish the moves enough.

So instead of move ordering, we considered more aggressive pruning. The min-max tree has

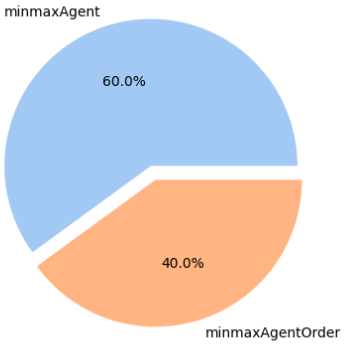

Figure 1: Proportion of games minmaxAgentOrder wins.

a very high branching factor. Each token on the board can be spread in 6 directions, and each unoccupied cell in 7x7 grid can be spawned into. This often leads to a branching factor of over 70, sometimes even over a 100 in the middle game.

How do we effectively discard useless moves? We came up with a unique token differential metric. If a move results in no change in the number of tokens on the board (like a spread of 1 into nothing), we discard it. This resulted in massive speedups allowing us to now utilize 3 depth minimax. To ensure that this did not result in pruning out useful moves, we compared this newly made agent (minmaxAgentPruned) with the original (minmaxAgent). The results are shown in Figure 2. Additionally, It is interesting to note that power difference on the other hand did result in useful moves being discarded in some niche cases.

3.2 A failed attempt at a Better Heuristic.

To address the issues we encountered during ordering, we experimented with multiple heuristic combinations. We implemented a heuristic that measured what proportion of the power of the opponent the player could reach ($power\_within\_reach\_heuristic$). We also tested a heuristic that tried to measure how well you surrounded your opponent's ($surround\_heuristic$) tokens. Unfortunately we

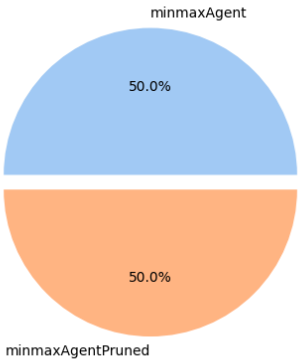

Figure 2: Proportion of games minmaxAgent-Pruned wins.

weren't able to test all viable combinations extensively. We varied the weights of the heuristics by the turn number as well, but none of the combinations performed better than just power difference alone. We considered implementing grid search, and even tuning algorithms like Simultaneous perturbation stochastic approximation [2] (this technique has been previously tested in chess engines like Stockfish to combine and tune different evaluation functions), but the implementation of these would require running the program possibly for weeks. Thus we ended up sticking with the simple power difference heuristic. We performed a very comprehensive test on one combination of $power\_difference + 0.5(token\_difference)$. The intuition being to encourage more aggressive spreading behaviour. However, the standard power difference agent defeated an agent with that modified heuristic (12 to 8 wins, refer test with minmaxAgent-Pruned and minmaxAgentPrunedAdv in logs, tests).

3.3 Smarter Spawn behaviour and Further increasing Depth

All spawns over the short term (1-2 moves), influence power difference similarly (except in certain board patterns). Thus often the spawns that are chosen may not lead to good long term play. We modified the order of spawns considered by carefully evaluating the neighbouring tokens of an empty space on the board. From our insight into the game, you do not want to spawn next to an enemy, and ideally you have some grouping but not in a single column/diagonal. More importantly, when you have the opportunity to overwhelm an enemies tokens by surrounding them at the start safely, you should evaluate that possibility. This becomes less important once a good amount of tokens are present on the board. The process of fine-tuning the Smarter Spawn behaviour is displayed by Figure 3, where our agent named minmaxAgentPrunedUpdatedSpawn, when developed over multiple iterations, achieves better performance than our standard minmaxAgent-Pruned.

Further, the early game is the most important part of the game to establish an advantageous position. Mistakes in the early game, result in catastrophe later. Since the Smarter Spawn behaviour pruned certain moves, we were able to increase the depth to 4 for the first 20 moves of the game when Smart Spawn behaviour is most active.

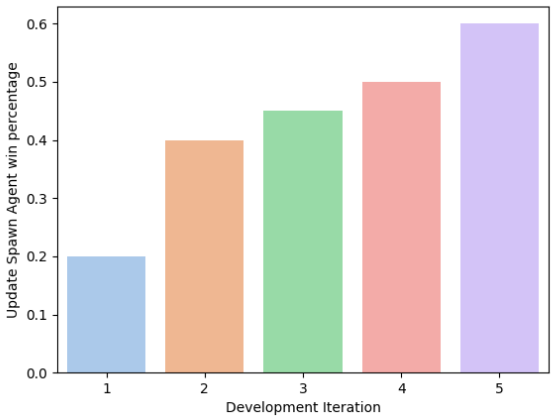

Figure 3: How we iteratively imporved the updated spawn agent. Refer the logs of the different minmaxAgentPrune vs minmaxAgentPruneUpdatedSpawn trials and the agent vs minmaxAgentPrune trial.

3.4 Guaranteed end of games

These changes allowed us to have a relatively strong adversarial minmax agent. Finally there

were a few positions where no moves, except useless moves exist. In this case the agent returns greedy output. When time is running low, (less than 30 seconds), we decrease our depth to 2 quickly finish the game.

4 Monte Carlo Tree Search Agent

Alongside our minimax agent, we also developed a Monte Carlo Tree Search (MCTS) agent. Monte Carlo Tree Search (MCTS) is a popular algorithm for finding optimal decisions in complex and uncertain domains. It has been successfully applied to various games, such as Go and Chess with good results. Although we haven't chosen the MCTS agent as the final agent, we include our findings here for completeness. We describe the main components of our MCTS algorithm, such as the board representation, the playout simulation, and the node selection strategy. The performance of the algorithms was evaluated against a random player and a greedy player.

4.1 Board Representation

We use a Board class to store and manipulate the board state. The Board class has two main class variables:

- adam: A hashmap that maps each board state to its corresponding metadata. The metadata includes the previous board states that lead to the current state, the number of wins and playouts for the current state, the children states generated from the current state, and the color of the current player.
- actions: A hashmap that maps each pair of board states to the Action that connects them.

Additionally, corresponding metadata is represented by a class called MCTS_Values, containing mainly 5 attributes: parents, wins, playouts, children, color.

4.2 Playout Simulation

Our MCTS algorithm consists of two stages: training and decision.

In the training stage, we perform the following steps until either time runs out or a set number of iterations are completed:

1. We access the current board state from the adam attribute of the Board class. If it does not exist already, we create a new MCTS_Values object for it and add it to adam. This is considered as the original board for any current board state.
2. We choose a candidate next-board state by using a heuristic function that selects the best action based on the UCB formula:

   $$UCB1(n) = \frac{U(n)}{N(n)} + C * \sqrt{\frac{logN(Parent(n))}{N(n)}}$$

   Where $U(n)$ is the is the number of wins after taking the action leading to the MCTS_Values node, $N(n)$ is the total number of playouts, $Parent(n)$ is the total number of actions taken from the current state, C is a constant parameter that controls the exploration-exploitation trade-off.
3. We simulate a playout of the game from the candidate next-board state to a terminal state (win, lose, or draw) by using a node selection strategy.
4. We encode the result of the playout as 1 for win, 0.5 for draw, and 0 for lose.
5. We increment the wins and playouts attributes of the candidate next-board state by the result and 1 respectively.
6. We use the parents attribute of the candidate next-board state to access all its predecessor states from adam and update their wins and playouts attributes accordingly. Depending on the color attribute, this may be different. For example, if the candidate next-board state has color red and result 1, then all its red predecessors will increment their wins by 1, while all its blue predecessors will increment their wins by 0.
7. We add the candidate next-board state to adam and store its description in the children attribute of the original board state.
8. We create an Action object that describes how to move from the original board state to the candidate next-board state and add

it to actions. During the decision stage, we get the child with the highest playouts attribute, and return the associated action.

4.3 Node Selection Strategy and further Optimization

MCTS is a computationally intensive algorithm that requires a large number of iterations to converge to the optimal action. However, this may not be feasible in real-time applications where the response time is limited. To overcome this challenge, we exploit a node selection heuristic on top of the fact that MCTS stores precomputed data in the search tree. We propose the following strategy to reduce the computational time: We leverage previous knowledge and avoid redundant computations for board states that have been already explored. We expect that this strategy will produce well-chosen actions in a shorter time.

4.4 So why did we not choose MCTS as our final agent?

Our strategy also has some limitations and drawbacks. First, we impose a restriction on the number of children that each board state can have in the search tree. We set this limit to 5 to reduce the memory consumption and the branching factor. However, this may sacrifice the quality of the search and might miss some potentially optimal actions that are not among the 5 selected children. For these reasons, our MCTS algorithm does not perform well against our min-max based algorithm. We had specialized heuristics that didn't pan out against the greedy agent when used in play-outs and selections, nonetheless, when using the greedy agent for play-outs and selection, we were able to defeat the greedy agent.

5 Future Work and Conclusion

We believe with fewer restrictions on computing resources, a min-max-mcts combination could outperform our min max agent like the greedy-mcts combination did. There have been experiments into combining mcts with min max to provide more comprehensive game agents in certain game domains where mcts performs poorly [1]. More recent developments in Deep neural network have also resulted strategies that learn without knowing the rules of game [3]. Machine learning evaluation functions are certainly another area of research worth exploring for our game agents, especially if storage requirements are relaxed. We note that this may require a significant investment of resources, for example, the success of Alpha-go [4] as an MCTS agent required a multi million dollar evaluation neural net and excessive parallel compute resources during play. In conclusion, we developed a successful minimax agent that utilizes specialised knowledge of the game and several optimisations to play effectively. We further explored other algorithms like MCTS and determined their viability for the project. We comprehensively tested the agents for our decisions and ended with insights for playing the game better ourselves.

References

[1] Hendrik Baier and Mark HM Winands. "MCTS-minimax hybrids with state evaluations". In: *Journal of Artificial Intelligence Research* 62 (2018), pp. 193–231.

[2] Levente Kocsis and Csaba Szepesvári. "Universal parameter optimisation in games based on SPSA". In: *Machine learning* 63.3 (2006), p. 249.

[3] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[4] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[5] M. Vandegar. *Implementation of the Minimax algorithm with alpha-beta pruning.* https://github.com/TheCodingAcademy/Minimax-algorithm. 2022.