2023

Report – Project B, COMP30024

TEAM: RAMON VI ENTOURAGE

THE DUY NGUYEN – 1100548RAMON JAVIER L. FELIPE VI – 123328

Table of Contents

Introduction

Infexion for Project B is a two-player game, requiring an adversarial agent to compete against another player. We will discuss our approaches, its methods of evaluating the state of the board, various implemented optimizations, and performance analysis.

Our approach – Negamax

Our first practical search approach for the adversarial agent was Minimax with alpha-beta pruning. However, before becoming reasonably efficient, the complexity of Infexion has made the branching factor of the basic Minimax algorithm search tree simply too large, even with alpha-beta pruning. An important aspect to consider is alpha-beta pruning prunes branches based on what it has explored. This means that it will not consider branches that are already certainly less desirable (assuming both players play optimally).

It should also be noted that many of the possible legal moves that a player could make are not very beneficial, or even at all. Furthermore, particularly in start-game, most moves are equivalent to one another. With all of said information, we've deduced that there are 2 most important factors in Minimax approach to Infexion:

1. The order of moves - order the possible legal moves such that the moves with higher chance of being more desirable get explored first, as this would immensely enhance alpha-beta pruning.
2. The potential reductions of possible moves - with the knowledge that a lot of moves are equivalent, or not particularly beneficial, we can reduce the set of legal moves required to be explored.

To achieve these enhancements, we have employed a variety of optimization techniques, namely move ordering, SPAWN pruning, single-power SPREAD pruning and strategized endgame pruning, which we will be discussing in Optimization Functions segment. With this, we have evaluated that this was the approach that perform best.

Finally, two-player Infexion is a zero-sum game, and thus Minimax can be simplified to *Negamax*. Negamax helps make the code more elegant, and is in principle most suitable for the zero-sum nature of Infexion. Due to their equivalence, and for the sake of simplicity, we will now simply refer to these algorithms as Negamax.

Alternative approaches

NegaScout

Ultimately, Negamax does not perform better than Minimax. One of the biggest issues with Negamax is certainly performance. *NegaScout* is a Negamax-variant that more effectively utilizes move ordering's benefits via narrow window search to quickly prune nodes. To accomplish this, NegaScout uses null window search - a narrow search window to efficiently reject further expansions, and principle variation, which prioritizes searching the most promising move. Nevertheless, it has been found that, while providing some speed improvements, in specific test cases, even with varying window sizes, the agent provided less accurate plays compared to its counterpart. Even as Negamax is restricted within a smaller time constraint per move (see Desperation Play below), NegaScout would either yield more inaccurate results, or simply not show worthwhile speed improvements.

Monte Carlo Tree Search (MCTS)

The main issue of MCTS is the hefty tradeoff between resource (in our case is time) and play quality in simulating the game. Ideally, node exploration should be substantial to produce a quality result. For smaller games like *Tic-Tac-Toe*, it is very easy to achieve this since the game doesn't play out for very long. Inflexion, however, takes a lot more turns, and this number only grows larger if both agents are reasonably, and equally, competent.

For this reason, there is a decision to be made on whether MCTS should simulate a game with smarter strategies to lower depths, or to cut off in a more restrained amount of time but of lower play quality. There were several attempts made to optimize MCTS, including:

- An evaluation function for a resulting simulation after n moves
- A priority queue to prioritize checking nodes that are more relevant to winning the game.

Despite these attempts, MCTS showed unsatisfactory improvement.

Evaluation Function

The most important concept of *Infexion* is that this game is about space control. To win, it is imperative to always have more space on the board than the opponent. With this idea in mind, the evaluation function considers several of the player's and opponent's features of the current board:

1. **Quantity and Power**

   1.1. Quantity

   - Refers to the number of pieces of a player.
   - Goal: Encourage the agent to actively create/take more pieces to increase their side's population
   - Specification: This is the most obvious indicator of space control as a higher number of pieces means more spaces on the board that are yours. This on its own is faulty, and thus requires more in-depth evaluation analysis.

   1.2. Power

   - Refers to the total power of a player.
   - Goal: A factor which increases the accuracy of the evaluation score by also considering the higher power pieces, rather than only looking at the number of pieces.
   - Specification: This has implications on player's offense/defense. Higher power certainly will increase this specific score, but it won't increase the number of pieces the player has. Ultimately, high-power piece increases the offense potential of the player, but also increases their vulnerability, since letting such pieces be captured is considered more severe than if letting lower power pieces captured.

2. **Cluster**

   - Refers to groups of adjacent allied pieces.
   - Goal: The quality of space taken plays an immense role in strategizing Infexion. If a player's piece arrangement is cluttered and not concentrated, then that will increase the vulnerability of all of their pieces. In that sense, it is important to form a group of allies to make sure any attack will be properly countered.
   - Specification: As stated in its goal, the space taken up by the player should be of good quality. Hence, a high number of pieces or power is not necessarily indicative of a sufficiently good board state. It is desirable if these pieces are clustered together. This introduces the idea of dominance, where a player's dominance, which coincides with the desirability of the board for said player, is determined by whether their clusters dominate the opponent's or not.

   2.1. Number of dominating clusters by quantity

   - First consideration of dominance is the clusters that dominate its opponent by size, or the number of pieces within said cluster.
   - Having a cluster that dominates many clusters is an indicator of more impenetrability.

   2.2. Number of dominating clusters by power

   - Second consideration of dominance is the clusters that dominate its opponent by power.
   - This adds another layer to domination that needs to be considered. Since it is not accurate to deem a cluster is necessarily weaker simply because it has fewer pieces. Clusters with high power also have high potential for offense. Though due to its inherent vulnerability, the factor of this score is kept notably low.

All these aspects are weighted by factors that are tuned in testing, resulting in the formula for evaluation function of the $state$ as follows:

$$Eval(state) = 2 * (N_{red} - N_{blue}) + 1.6 * (P_{red} - P_{blue}) + 1.55 * (DN_{red} - DN_{blue}) + 0.65 * (DP_{red} - DP_{blue})$$

Where:

- N_{color} is the total number of the player *color*;
- P_{color} is the total power of the player *color*;
- DN_{color} is the total number of clusters of player *color* that dominate its adjacent opponent's by number;
- DP_{color} is the total number of clusters of player *color* that dominate its adjacent opponent's by power.

Optimization Functions

Move Ordering

Move ordering significantly reduces the number of actions needed to be explored. This is based on a very simple principle that the more favorable moves getting explored first will help alpha-beta to prune more effectively. The moves are ordered (in decreasing priorities) by:

- Total power of captured pieces
- The reverse (or negation) of the total number of captured pieces
- Power of the player's piece deployed for action

There are several reasons behind this. The first two priorities indicate that we should capture as much power from the opponent as possible, but preferably pieces with higher power, or more "stacked". For example, we deem it more advantageous to capture two opponent pieces of power 3, rather than 6 single-power pieces, since the former has more stacked pieces. This is because, as we've discussed above, high-power pieces have more potential for offense and are regarded as more dangerous, which requires more immediate action.

Move Reduction

If it weren't clear before, our game-playing principle for Infexion is "cluster your allies". As we have observed, the majority of the moves, which is the primary reason for search slow-down due to its large branching factor, are not beneficial and can be effectively removed. These types of moves are:

- *SPAWN amidst nowhere*: With an emphasis on clustering, spawning at a position not adjacent to any ally is not particularly helpful. Other SPAWN cells, as long as adjacent to ally, are all considered.
- *Single-power SPREAD with no capture*: Single-power SPREAD without capturing an opponent is generally not beneficial unless it is under specific cases. Furthermore, these are moves that take up a large portion of the move set.

Nevertheless, we do not always apply move reduction. Our algorithm also has an overwhelmed detection, where if the player is already overwhelmed, their move set should not be reduced at all. And that in particular for single-power SPREAD, the algorithm will allow SPREAD action onto its own ally if the board is already highly populated (filled with a lot of power-1 pieces, which means it is perhaps desirable to build up one's attack potential not just by direct attack or spawn). The idea is to allow the agent to build up a piece's power safely for a future attack.

Endgame Play

Endgame is any state where the player overwhelmed the enemy enough to enable them to solely trade attacks until it has won. Note that we use a different 'overwhelm' detection here than that of Move Reduction. Endgame detection is comparatively a lot more conservative. Due to the nature of a true endgame, the player should be in comfortable enough position to relentlessly attack with no significant drawbacks, hence the strict rules to ensure that such is truly the case. The following are the conditions that must all be met for endgame to be valid. (The constants used in pre-conditions are derived from several play-testing to deduce the best parameters.)

Pre-conditions:

1. *Player's total power is at least 12 out of maximum 49*:
   Baseline check to see if we have enough offensive power to keep attacking.
2. *Opponent cells are at least one-third of player's cells*:
   Check to see if player is overwhelming enough to keep trading captures up to an eventual victory.
3. *Opponent's cells are uniformly of power 1, except for at most one piece of any power*:
   The opponent can only have at most a single piece that has a power exceeding 1, and every other piece must be 1.
4. *Opponent's clusters cannot exceed size 2*:
   Cluster of size greater than 2 renders complexity that should not be undermined. For any larger cluster, even with very high-power player piece deployed, the player may still be successfully counter-attacked. The complicated nature of larger clusters disproves the endgame state.
5. *Player must be able to eliminate opponent's stacked piece*:
   The opponent's stacked piece is key to its offense and apparent unpredictability. As such, if the player can ensure that the piece will be captured, the endgame therefore is still validated.

Post-conditions:

After these conditions are met, the list of actions for endgame, a significantly reduced list, will be returned. But a condition that also must be met during move generation is that any player piece deployed for attack must have power of at least the cluster's size. If not, the player piece might be in danger of being counter-attacked.

As such, the final piece of information would be, whether the list of actions even has any action to begin with. With the player's power over cluster size condition, the list may be empty. If such is the case, it is simply indicative of non-endgame state, and the agent must generate move as usual.

Desperation play

In cases where time has become a clear issue hindering the agent from ever finishing the game, the agent will trigger a desperation play. In a way, our agent dynamically adapts to the situation at hand, and if dire enough in the remaining allowed time, the player will change its playstyle to something simpler.

Firstly, a time threshold was made, where if the remaining time is below it then the agent will use Negamax with depth of 2, with all optimizations, instead. This threshold is 15 seconds. Another observation we have made is that our agent does not usually take above 18 seconds to make its best move. Usually, for any move exceeding that, it often implies that most moves are not sufficiently different for the move ordering to be effective (we have observed that they are primarily SPAWN actions with minimal evaluation differences). This is most common for cases of very cluttered opponent which yields little difference in the final decision. Thus, if 18 seconds have passed for a single move, the agent should return its current best move immediately.

Performance evaluations

Approach

The adversarial search algorithms we've explored include Minimax-variant algorithms (including Negamax and NegaScout), Monte Carlo Tree search (MCTS) with different play-out strategies, and greedy search.

To compare their performances, the most obvious approach for us is to let them compete against each other, where each agent would take turns to be either RED or BLUE, under time and space constraints. We have found that greedy agent performs the worst out of all (except for random agent, and in some cases, MCTS). It however was conceived to be a good bar for other agents to compete against.

Various test cases, acting as *traps* for generally uninformed behaviors were also part of the evaluation. These are tests we gathered throughout our play testing, at which our agents were still in progress and made specific mistakes which we deemed valuable for ensuring that behaving informedly meant to not have fallen for these traps. Examples of these tests include:

- Domino play: These are plays where both sides continually deploy their single-power pieces to counter-attack at a single point of conflict which repeatedly increments the cell's power, until either the piece gets eliminated (reached power of 7), or the piece gets effectively captured by one of the players.
- Evaluation-specific play: These are tests to make sure the evaluation is mostly unbiased, assuming that certain plays might be mistakenly undervalued by the evaluation function despite being better.
- Anti-greedy play: these are not cases that will necessarily cause harm to the player, but rather it will test whether the player is informed enough to make a better play out of two seemingly similar ones.

Negamax

Negamax with alpha-beta pruning was able to immediately perform better in terms of its balance in time and quality compared to MCTS. Nevertheless, Negamax ultimately suffers from the same issue of game complexity without sufficient optimizations. While the performance is generally better, the algorithms were not capable of reaching low depth. In its unoptimized stage, it could not reach depth of 3 safely due to the unreasonable amount of time for certain plays. With depth of 2, we have found that it beats greedy agent around 8 out of 10 times.

After which, optimization strategies listed above were made for Negamax. With these, surprisingly enough, for Negamax at depth 2, not only was it remarkably faster, it also managed to never lose against greedy agent, (on average, this takes around 30 to 60 turns, under 5 seconds). We have deduced that this is due to the fact that only moves with better quality were kept, which was the reason for the playout improvement.

Moreover, the optimizations have helped Negamax agent to reach depth of 4. And since, it has never lost against greedy agent, winning on average around 20 to 50 turns. In this case, Negamax passed all of our trap test cases. Though with desperation play, for cases of very dense boards, it did not pass all. Nevertheless, the response time was a lot more desirable with minimal loss of accuracy. The random agent is much more unpredictable, however, though our Negamax agent has never lost against it either. We have found that our Negamax agent, at depth of 4, takes anywhere around under a fraction of a second to 5, and in the worst-case scenario, 20 seconds without desperation play to make a move. Though on average, it takes around 3 to 5 seconds to return an action.

NegaScout

With hopes of making full use of the benefits of move ordering, we have implemented NegaScout. We first let it compete against greedy, MCTS, and *shallow Minimax* (Minimax at depth of 2 with no domain-knowledge move reduction optimization). At depth of 4, NegaScout significantly dominates these 3 agents, similar to Negamax. However, it has slightly (and at times notably) worse play compared to Negamax's more refined plays, even with the latter using desperation play, and with only trivial speed improvements. Worse yet, NegaScout did not pass various trap tests. With varying null window sizes, it was evident that NegaScout was not substantially faster than Negamax, while being less accurate. Therefore, we deemed this accuracy tradeoff generally not worthwhile.

We also let Negamax and NegaScout compete against each other. We have found that neither dominates the other in terms of gameplay. Negamax is only marginally slower than NegaScout (on average under roughly 5 seconds slower with neither using desperation play), but is safer when it comes to our test cases. For this reason, we have decided that the performance of NegaScout was not satisfactory.

Asymptotic Analysis for Negamax Agent

1. Time Complexity

Overview

The time complexity of Negamax agent is identical to that of Minimax with alpha-beta pruning. The complexity is expressed in terms of *b*, the branching factor resulting from generating the board states from possible actions, given a parent state, and *d*, the depth at which the algorithm reaches. This results in time complexities of:

- Worst Case : $\mathbf{O}(\boldsymbol{b^d})$
- Best Case : $\mathbf{O}(\boldsymbol{b^{d/2}})$

Where *b* is the optionally optimized number of moves that the agent can make in a given turn, and *d* is the depth that the tree reaches. More detailed on this analysis will be elaborated below.

Cluster generation

The cluster generation is directly related to the complexity of the evaluation function and the endgame play. For each cluster generation, it iterates over the opponent's cells, and looks at the adjacent positions to either update its cluster, or merge its own cluster with another. Then, it will look at the player's cell and use the established information of opponent's clusters to update its data on adjacent opponent's clusters. Since cluster update and merging is optimized to O(1) time complexity, the time complexity of cluster generation is $\mathbf{O}(\boldsymbol{2n_p * 6c}) = \mathbf{O}(\boldsymbol{nc})$, where n_p is number of player cells, *n* is total number of occupied cells, and *c* is average number of clusters.

Evaluation function

The evaluation function is called at every depth limit or terminal node of the Negamax tree. It involves cluster creation and comparing adjacent opposing clusters against each other. Thus, the time complexity is $\mathbf{O}(\boldsymbol{nc}) + \mathbf{O}(\boldsymbol{c_{red}c_{blue}}) = \mathbf{O}(\boldsymbol{nc + c^2})$, where c_i is the number of clusters generated from a given board state of player color *i*, and more generically, *c* is the average number of clusters for a given board state. With the respect to the entire algorithm, since the function is called at leaf nodes, it is asymptotically proportional to the number of leaf nodes which, in its worst case, is $\mathbf{O}(\boldsymbol{b^d})$, and in its best case is $\mathbf{O}(\boldsymbol{b^{d/2}})$.

Move reduction

Move generation involves iteration through every cell in a board to find all possible spread and spawn actions. This gives us the time complexity of $\mathbf{O}(\boldsymbol{n})$, where *n* is the number of cells on the board. It should be noted that this operation is called once for each node expansion, meaning the total time complexity of the operation within the algorithm would be proportional to the number of internal nodes of the Negamax tree, which is $\mathbf{O}\left(\mathbf{b}^{(\mathbf{d}-\mathbf{1})/\mathbf{2}}\right)$ in its best case, and $\mathbf{O}\left(\boldsymbol{b}^{\boldsymbol{d}-\mathbf{1}}\right)$ in its worst case (since we exclude the last depth, which is its leaf nodes).

Move ordering

Move ordering first looks through all the possible actions and calculates the values needed to sort the moves. The complexity of calculating these values for is $\mathbf{O}(\boldsymbol{sp})$, where s is the average number of legal actions, and p is the average power of the player's cells (required for spread). We then sort the actions by the ordering mentioned above which gives the overall time complexity of $\mathbf{O}(\boldsymbol{s}\log\boldsymbol{s}+\boldsymbol{sp})$. Overall, similar to move reduction optimization, move ordering has a total time complexity proportional to the internal nodes of Negamax tree.

Endgame play

As mentioned in Post-conditions of endgame, there are cases where the endgame is thoroughly checked before the detector realizes it isn't the endgame, which forces the move generation to do its usual routine. This causes overheads in certain cases. The endgame includes two key aspects - detection and generation:

- Detection: First, it creates the cluster, then looks through every possible direction from each opponent piece. In each direction, it will iterate over each cell to check whether there is a player's piece that can capture it. The number of operations for this process remains static at any given case, which is 6*6 = 36.
- Generation: Sort the list of actions by their endgame ordering specified above and return the list.

The time complexity would be $\mathbf{O}(\boldsymbol{nc})+\mathbf{O}(\mathbf{36}\boldsymbol{n_o}+\boldsymbol{k}\log\boldsymbol{k})=\mathbf{O}(\boldsymbol{nc}+\boldsymbol{k}\log\boldsymbol{k})=\mathbf{O}(\boldsymbol{nc}+\boldsymbol{s}\log\boldsymbol{s})-n_o$ is number of opponent pieces, k is the average number of spread actions that are generated in each endgame check, and s is average number of possible legal moves. Similar to move reduction, its time complexity is proportional to the number of internal nodes in the Negamax tree.

Total time complexity

From these, we're able to derive the total time complexity of Negamax algorithm:

- Worst Case : $\mathrm{O}(leaf)*\mathrm{O}(Eval)+\mathrm{O}(internal)*\mathrm{O}(Ordering+Endgame)$
  $=\mathrm{O}(b^d)*\mathrm{O}(nc+c^2)+\mathrm{O}(b^{d-1})*\mathrm{O}(s\log s+sp+nc+s\log s)$
  $=\mathbf{O}\left(\boldsymbol{E}\boldsymbol{b}^{\boldsymbol{d}}\right)$
- Best Case : $\mathbf{O}\left(\boldsymbol{E}\boldsymbol{b}^{\boldsymbol{d}/\mathbf{2}}\right)$

  Where $\boldsymbol{E}$ is the time complexity of the evaluation function.

2. Space Complexity

Overview

The algorithm has to store every possible move from a given board state in a specific branch. At each depth, we have to store b possible nodes, where b is the branching factor, or the number of possible moves from a given board state. Since we use only 1 board, where apply and undo actions are possible (based on COMP30024 referee code), each node is exactly a single action. This means that the space complexity is $\mathbf{O}(\boldsymbol{bd})$, where b is the possible number of moves from a given board state and d is the depth the algorithm explores.

Board

The board is a singular, full representation of the game's board. This means that the space complexity is always $\boldsymbol{\Theta}(\boldsymbol{N})$, where N is the total number of cells on the board, which is always 49, hence actually constant.

Clusters

The clusters are not stored for the entirety of the algorithm, but only each time the evaluation function is called. Thus, it won't contribute to the overall complexity of the algorithm. The space complexity for the clusters would be $\mathbf{O}(\boldsymbol{c_{red}}+\boldsymbol{c_{blue}})=\mathbf{O}(\boldsymbol{c})$, where c is the total number of clusters for a given state of the board.

Total space complexity

The total time complexity becomes $\mathbf{O}(\boldsymbol{bd})+\boldsymbol{\Theta}(\boldsymbol{N})=\mathbf{O}(\boldsymbol{bd}+\boldsymbol{N})$.