Project B Report: Leo Robinson, Mitch O’Grady

Describe your Approach

We implemented multiple agents to arrive at our final agent, with each providing some kind of scaffolding for the next.

To begin with, we implemented a simple greedy agent (**OneMoveStrategy2**) that looks for the Spread move that gives the biggest possible decrease in opponent power, i.e. by taking over tiles with the greatest cumulative blue power. If no Spread action decrements the opponent’s power by more than 1, it instead checks for a “better” Spawn move by looking for the highest blue power tile it could spawn next to, in order to play a rewarding Spread action next turn.

Note that we use net gain/loss of opponent tiles and not the player’s tiles. We used this strategy in **OneMoveStrategy**, but found that it was not very efficient in finding genuine winning moves. By using opponent power as a motive, this ensures the strategy’s greatest priority is winning.

In addition to the OneMoveStrategy, we also implemented a basic agent, the **RandomStrategy**. This agent chooses its moves randomly without any look-ahead or decision-making based on the game state. In the first round of the game, it randomly places a tile on any available location.

From the second round onwards, as long as the total board power is less than or equal to 48, the agent randomly decides between two actions: placing a new tile, or taking a Spread action. It picks a random location for placing a new tile or a random Spread move according to its generated decision.

The purpose of this agent is to serve as a baseline to compare the performance of our more advanced strategies, examining how much the added complexity of decision-making and look-ahead contribute to improved gameplay.

Building upon the OneMoveStrategy, we implemented a more advanced agent called **TwoMoveStrategy** that performs one level of look-ahead to aid in decision-making. This strategy creates a OneMoveStrategy object to predict the opponent's move and another OneMoveStrategy object to determine the best move for the agent after the opponent's predicted move.

The agent first iterates through all possible Spread moves, calculating the net gain after the opponent's predicted move and the agent's predicted next move. If a Spread move results in an immediate win for the agent, it takes that move. If a move results in an immediate loss for the agent, it is only considered as a last resort when no other moves are possible. This strategy also factors in the possibility of spawning new cells on the board if the total board power is less than 48.

TwoMoveStrategy iterates through possible Spawn moves to find the move with the highest net gain after the opponent's predicted move and the agent's subsequent move. As with Spread moves, it avoids making Spawn moves that result in an immediate loss to the player.

This is an adaptation of the minimax algorithm introduced in lectures.

TwoMoveStrategy is intended to make more informed decisions by anticipating the opponent's moves and planning its actions accordingly. As a result, it can strategically make Spread actions and place tiles in a more informed way compared to OneMoveStrategy.

We also trialed a **ThreeMoveStrategy**. As it sounds, it performs two levels of look-ahead. Similarly to TwoMoveStrategy, it iterates through possible moves and predicts their performance. Utilising TwoMoveStrategy to predict the resulting opponent move, and then another TwoMoveStrategy to predict the player’s following move, followed by an opponent move predicted using TwoMoveStrategy again, and then finally a OneMoveStrategy to find the best available move for the player. Here is the sequence:

1. Iterate through possible moves for the *n*th move of the game
2. Predict opponent’s next move (*n*+1)using TwoMoveStrategy
3. Predict player’s next move (*n*+2) using TwoMoveStrategy
4. Predict opponent’s next move (*n*+3) using TwoMoveStrategy
5. Predict player’s next move (*n*+4) using OneMoveStrategy

However, we found this agent to be rather ineffective. In a time and space context, it was extremely slow and wasteful. It also proved to not be as successful as TwoMoveStrategy (TwoMoveStrategy always wins when playing ThreeMoveStrategy).

Performance Evaluation

Here is a table detailing the performance of the mentioned look-ahead strategies.

Strategy VS	RandomStrategy	OneMoveStrategy2	TwoMoveStrategy	ThreeMoveStrategy
RandomStrategy (random)	Random	OneMoveStrategy is most effective against RandomStrategy. Greedy algorithm will always favour random plays.	TwoMoveStrategy always wins. Not as quick or effective as OneMoveStrategy, TwoMoveStrategy plays more defensive, as it expects the opponent to play greedily.	ThreeMoveStrategy timed out
OneMoveStrategy2 (greedy)		Consistent: first player always wins (very quickly)	TwoMoveStrategy wins quite easily. This makes sense as this agent	ThreeMoveStrategy wins relatively quickly

			expects the opponent to play as OneMoveStrategy.	
TwoMoveStrategy (one look-ahead)			Consistent: first player always wins (takes longer)	ThreeMoveStrategy times out, however TwoMoveStrategy clearly dominates.
ThreeMoveStrategy (two look-ahead)				Timed out.

Hence, out of these purely look-ahead agents, we chose TwoMoveStrategy.

An overview of TwoMoveStrategy's expected performance:

- Should always win against random players, albeit slowly
- Will always win and perform best against greedy players
- Most likely will be outdone by more complex agents

The performance of each agent was assessed by manually running multiple games in a terminal and observing the play speed in real time. This method is appropriate for the current stage of development, as significant differences in runtime frequently occur due to updates to our code. TwoMoveAgent's run speed visibly decreased towards the end of longer matches as the board filled and the branching factor increased, but we are satisfied with the agent's overall speed and its ability to quickly win games against our random agent with a 100% success rate. Our primary objective was to significantly outperform random chance, and the TwoMoveAgent achieves this goal reliably, as well as beating greedy algorithms.

Our attempts at implementing Monte Carlo Tree Search were not successful in time, with success rates approximating random chance. We would expect additional refinements to our implementation of this method to yield significant win-rate and runtime performance improvements. A working version would also allow us to tweak and fine-tune the method to improve performance against a variety of opponents, and give us a scaffolding to implement more sophisticated machine learning elements.

As the main difficulty associated with ThreeMoveStrategy was its slow computation rate, this could likely be enhanced with the addition of alpha-beta pruning, which we were unable to implement in time for submission.

We had hoped that our Monte Carlo Tree Search algorithm would be working successfully, however we could not get it completed. Ideally this agent would have been able to outplay all opponents; whether random, greedy or more complex. Our next best agent is the TwoMoveStrategy, which we have chosen to submit. We know that TwoMoveStrategy outplays random and greedy opponents, but will most likely lose to more intelligent agents.

Other Aspects

As our previous code was based strongly around the dictionary structure for representing boardstates, we migrated that code and adapted it to work with the new Hex data structure classes. We had hoped to be able to leverage more of the functions we previously implemented, extending our code to work with heuristics where appropriate, but this was not capitalized on in the time we had.

We widely used classes to implement our various agents:

- A **BoardState** class to store the information of the board, as well as play history, and many methods used in our strategies. This class was designed to be all-encompassing and a base for all board-related functions, queries and attributes. We reused much of the code from the BoardState class in part A.
- **Agent** classes and subclasses based on the strategies used, to allow for efficient reuse of code and transferral/alteration of agents. One notable implementation was that of the **OneMoveAgent** as a sub-tool used in the **TwoMoveAgent** to predict the opponent's next turn.
- **Strategy** classes, utilized by agent classes, to further streamline transferring and altering our agents.
- **MonteCarloTree** and **Node** classes used within the tree. Used to perform the actions involved in a Monte Carlo Tree search, with inbuilt methods for the likes of running, selection, expansion, simulation, backpropagation, and candidate selection.

Supporting Work

The referee program was unmodified for our usage, though by including print statements in our Monte Carlo Tree Search code and canceling matches half-way through, we were able to comb between referee statements and examine these diagnostic statements to broadly assess how this agent was performing, and the number of visits it was managing to probe by certain stages of each match.

References

GitHub Copilot was used to assist in completing some helper functions.
ChatGPT and GPT-4 were used as consulting tools, but we did not end up generating any code with them that we considered appropriately helpful to include.