2019

General Game Playing as a Bandit-Arms Problem: A Multiagent Monte-Carlo Solution Exploiting Nash Equilibria

Brandon Mathewe Banda
Oberlin College

General Game Playing as a Bandit-Arms Problem: A Multiagent Monte-Carlo Solution Exploiting Nash Equilibria

Matt Banda
Advisor: Bob Geitz

April 3, 2019

Abstract

This project approaches general game playing in a unique way by combining popular methods of stochastic tree searching with a Multiagent system and a unique algorithm that I call the Wise Explorer algorithm. The goal of the system is to explore the worst possible branches of the game first to rule them out, followed by an in-depth search on the most promising branches. The system constantly refers to the data it collects during its extensive search, and it outputs a strategic move for any given state of a game. In essence, if you're ever in a bind during a game of tic-tac-toe, the system will tell you exactly what your best move is.

1 Introduction

One of the main drawbacks of game playing programs in the field of A.I. is that the success of many highly renowned programs such as Alpha-Go and Alpha-Star have come at the expense of generality. Exploiting heuristics or having prior professional gameplay data is how these systems succeed, which, while impressive, requires significant amounts of human intuition about the game. This level of human intervention in game playing programs begs to ask the question: Is the program the originator of the strategies that arise, or is it simply a reflection of what humans are capable of? However, general game playing hopes to fill in this lack of generality and remove the need for human intervention.

"General Game Playing (GGP) is the art of designing programs that are capable of playing previously unknown games of a wide variety by being told nothing but the rules of the game" [6]. The input to a GGP program is the game description which specifies the goal of the game, the possible legal moves, the board state, and the conditions for termination of the game. GGP programs are also given information about what role the program will play, such as O's or X's, or black or white. These systems are also commonly under time constraints for how long they can process a move, and how long they can spend making that move.

The field of general game playing is exciting because it brings us closer to understanding how to model systems with abstract reasoning and strategic thinking, both of which are pivotal in one day creating a general A.I.. While other fields and approaches such as machine learning heavily focus on exploiting heuristics and gathering vast amounts of historical

data to solve their problems, general game playing goes further and aims to gather those heuristics autonomously. With the need for general A.I. to help us in autonomous vehicle reliability, drug and medicine development, and environmental forecasting, why not start with some board games?

Two-player games are divided into three categories: Random Games (Poker, Yahtzee, etc), Indifferent Games, and Normal Games. This paper will only focus on Normal and Indifferent games, and their characteristics will be discussed in a later section.

2 Past Work on General Game Playing

A key concept that this project drew inspiration from was the idea of Human-Inspired Algorithms for Search as a Multi-Armed Bandit Problem, a thesis by Paul B. Reverdy of Princeton University [1]. His paper "considered the problem of decision making under uncertainty in a variety of tasks", and aimed to use core concepts of human decision making from Neuroscience in congruence with modified Upper Confidence Bound algorithms from the literature to solve difficult problems in a general way. Similarly, our system takes advantage of "human-intuition" in tackling GGP, which will be discussed in detail later.

Many experts in the field have come up with successful ways to approach general game playing, such as Stephan Schiffel and Michael Thielscher from Dresden University of Technology [6]. Their program, winner of the AAAI GGP Competition in 2006, treats general game playing as a problem which requires reasoning about actions, tree-searching and pruning, evaluation functions for games that cannot be fully searched, and automatic heuristic generation. Their system implemented a search algorithm that was modified iterative-deepening depth-first search which gradually lowered the depth of the search based on how deep the majority of "best moves" were. This approach aimed to reduce the search space for larger games. They also chose to analyze the game rules to pre-process heuristics at runtime before commencing the search algorithm. Many systems follow a similar approach, with variations on how the search is conducted and which evaluation functions are used.

A key component of this project's search algorithm is similar to Uber Engineering's "Go-explore" algorithm that hoped to tackle so-called "hard exploration problems" by focusing on Montezuma's Revenge, a game that is especially difficult because of the sequential nature of its gameplay [4]. True to GGP, Go-explore did not use human-given heuristics, but managed to significantly out-perform state-of-the-art systems that were given human solutions to the game. Go-explore achieved this level of performance by exploring a vast amounts of random games, regardless of whether these games ended favorably or not. It then used this accumulated semantic-based knowledge of games that didn't go well to inform itself moving forward during its training phase.

In the following section, I will give an overview of my system and how it is implemented. My system draws a lot of inspiration from the past work explained above, but what makes this system unique is the way it combines many of the key ideas of these past works.

3 Our System: The Multiagent Bandit Monte-Carlo System (MBM)

To note: all references to "MBM" refer to the system at a high level

Inspired by Paul B. Reverdy, this paper presents general game playing as a Bandit-Arms problem: a problem of finding the optimal strategy out of a collection of many possible strategies [1]. Each "arm" in the bandit arms problem represents a move in one of our "games", and the goal is to sample fewer and fewer arms until an optimal arm is found. We will achieve this convergence to optimality through the use of a Multiagent system.

Our system's only input for each move is a board state, where data about one board state is not used to influence how the system decides a move in a different board state. This kind of a system allows for a level of transparency about how the system works, as opposed to a "black-box" approach. The system also relies on randomly simulated games between our agents and the opposing "mock-agents". The only non-random aspect about our multi-agent system is that our agents are able to choose their initial move for the board state they are being simulated in, and the mock agents will by default will choose a winning move if they can win on the same turn.

Inspired by Schiffel and Thielscher's approach, MBM initially uses the agents to converge on "bad moves" by pruning the search space during the "pre-processing" phase. The agents will then run through simulations of our game out until a desired depth while avoiding the pruned sections of our search space. At the end of each simulation, the first move of each agent will be updated based on what happened in the current and past simulations of the given board state. This accumulated data is analyzed for a Nash Equilibrium, and the best possible move (based on simulation time and number of agents) is chosen for any given state of a game.

To note: "neutral" refers to neutral moves *and* tying moves

The main concept behind MBM is for each agent in the system to choose a move that they are most certain of in order of {win, neutral, loss} by using a collective, iteratively updated pool of simulation data. This simulation is conducted as a Monte-Carlo tree search, and the collected data contains win/neutral/loss information about all of the possible initial moves (arms) that all agents have made for the given board state. Agents then use this information to choose their initial move in the next simulation if their previous move was unfavorable. Over many simulations, the objective is for the agent population to converge on one or two strategic moves based on this iteratively updated knowledge about the board state. Moves that result in copious losses are ruled out by the population early on, while moves that consistently bring agents into favorable states for their given board state will be visited more often. The end result is a self-converging system that picks out the best arms for a given board state, and continues to explore these arms if they are more reliable than other arms. At the end of the simulations, we select the best move based purely off of the win/neutral/loss data that the agents have collected from their simulated games. The system's overall goal is not to maximize winning, but instead to choose the move that will most confidently not lose the game. This optimization happens through an Upper Confidence Bound algorithm during our Monte-Carlo tree search, and it ensures that agents pick their most confident move, even if it means choosing a low-risk tie/neutral move over a high-risk win.

Because of the built-in pre-processing phase where we run MBM to prune bad branches

of our search tree, we can rest assured that the system is only exploring branches with a high potential to be good strategic moves. Without this pre-processing phase, the system would need more agents and simulations to work effectively since the search space for good arms would be (unnecessarily) large. I call this combination of pre-processing followed by a modified UCB search the "Wise Explorer" Algorithm.

Similarly to Uber Engineering's "Go-explore" algorithm, MBM's pre-processing aims to prune out clearly sub-optimal branches, resulting in our algorithm "[returning] to promising areas for exploration" [3].

MBM is also built to play two unique types of games: Indifferent games (Nim) and Normal games (tic-tac-toe and Mini-chess). I will define these two types of games in the next section and why they must be approached differently by MBM.

4 Normal Games vs. Indifferent Games

Indifferent games are symmetric in payoffs, and are designed so that player one and player two can alter the same pieces on the board on any given turn, the only difference being that player one moves first. Nim, The 21 Game, and Dots & Boxes are examples of Indifferent games (*See Appendinx for Rules of Nim*).

Normal games are games in which player one and player two have unique pieces that they can alter/use. Tic-Tac-Toe, Minichess, and Checkers are the Normal games. In Tic-Tac-Toe, player one and player two place different pieces, and in Minichess (a smaller, 5x4 board with less pieces than chess), both players can only control their specific colored pieces.

MBM needs to approach these two types of games differently because they use the win/loss data in fundamentally different ways. Normal games require comparing the overall win/loss ratios across all branches, and picking the best branch based on the highest ratio.

Indifferent games use the intuition that the player wants to move from a state where they are less likely to win to one where they are more likely to win, since the payoffs are symmetric. In other words, given an initial move q, if I win a specific game configuration 22% of the time after 1 turn, but win that configuration 100% of the time after 5 turns, the value (1.0 - 0.22) would represent the amount that q increases your potential to win, as well as the amount that it decreases your opponent's potential to win in the future.

If Q is the set of all possible moves we can make given a game configuration, we would want to pick a $q \in Q$ such that the maximum-terminating-depth win percentage minus the minimum-terminating-depth win percentage is maximized. This is because in Indifferent games, you either lose or win, and so aiming to be in a high win percent position right as the game is about to end is far more beneficial than being in a high win position early on, since in the future you could be in a very bad state right before the game ends.

This distinction between Normal games and Indifferent games is important, and will be discussed at length in section 5.3.

5 MBM at a high level: Intro to implementation details

MBM has 4 major components: The Multiagent system, the Wise Explorer Algorithm, the data collection, and the Nash equilibrium.

In the following sections, I will explain how all of the components of this project are implemented, as well as how they all work together to create a general A.I. that can play Tic-Tac-Toe, Mini-chess, and Nim effectively. I will then show how this system converges to reasonably strategic moves through a Nash Equilibrium, and how this method for convergence can be useful for the field of general game playing and general A.I..

5.1 The Multiagent System

MBM uses a Multiagent system throughout its simulations, and it is integral in how MBM functions.

Initially, the user is allowed to define how many "agents" are created in the system to play a given game. Each agent has the following class variables:

- **self.move**: The move that the agent is going to make on a given turn in the simulation.
- **self.coreMove**: The initial move that an agent will make in a simulation.
- **self.change**: A boolean that indicates whether the agent should change up its coreMove in the next simulation (True), or if it should keep its coreMove for the next simulation (False).
- **self.didWin**: A boolean that indicates whether an agent won the last simulation or not.
- **self.neutral**: A boolean that indicates whether the move the agent made in the last simulation was a neutral one.
- **self.moveDepth**: An integer indicating the depth that the agent went (in turns) before the game outcome was determined, or the simulation was terminated.
- **self.player**: An integer that indicates which player the agent is (1 or 2).

Using these class variables, each agent is able to continuously add to a a collective table of data to be used for the convergence of the system. This data is a dictionary of coreMoves, with a coreMove's value being a dictionary of depths, with a depth's value pointing to how many times an agent won the simulation, lost the simulation, or was neutral in the simulation {coreMove : {depth : {status : count}}}. This dictionary is consistently populated with data from all of the agents at the end of each simulation, and is used to push the Multi-agent system towards the most promising branches.

Half of the simulations in MBM are for pre-processing, and half are for the UCB algorithm, but all of the data is retained in the same table across both of these simulation phases. A simulation in MBM is run as follows:

```
Result: Stores simulated game data into collective table
turnDepth = 0;
Agent.move = Agent.coreMove;
while turnDepth ≠ n do
    Agent.makeMove();
    turnDepth += 1;
    if Agent wins game then
        Agent.didWin = True;
        Agent.change = False;
        Agent.moveDepth = turnDepth;
        Store Data and Terminate
    end
    if The game is tied then
        Agent.neutral = True;
        Agent.change = True;
        Agent.moveDepth = turnDepth;
        Store Data and Terminate
    end
    MockAgent.makeMove();
    if Mock Agent wins game then
        Agent.didWin = False;
        Agent.change = True;
        Agent.moveDepth = turnDepth;
        Store Data and Terminate
    end
    if The game is tied then
        Agent.neutral = True;
        Agent.change = True;
        Agent.moveDepth = turnDepth;
        Store Data and Terminate
    end
end
Agent.neutral = True;
Agent.change = True;
Agent.moveDepth = turnDepth;
Store Data and Terminate;
```

Algorithm 1: A game simulation between an Agent and a Mock-Agent

This is how the Multiagent system accrues its table of data over time. The Multiagent system is able to refer to the table to decide the "best moves" after the pre-processing phase of the Wise Explorer algorithm, but to explain how the Multiagent system functions, we will need an understanding of the Wise Explorer algorithm itself.

Now that we have a good understanding of how the Multiagent system functions, I will explain how this system uses the data table to achieve strategic moves.

5.2 The Wise Explorer Algorithm

The goal of the Wise Explorer is to have a comprehensive idea of exactly what not to do when deciding how traverse the search tree, similar to how Indiana Jones knows not to step on the loose stone infront of the Golden Statue.

Now that we know how the Multiagent system works to collect data, I will explain how the Wise Explorer algorithm uses this data to influence the Multiagent system to collect the most useful information, rather than doing an exhaustive search on the entire game space.

At the beginning of the program, a user inputs a number q denoting how many simulations the Multiagent system will go through before using the accumulated data to pick the best move. Wise Explorer wants to ensure that before the end of these q simulations, we are confident about the best move.

For the first $q/2$ simulations, at the termination of a simulation, we decide each agent's coreMove for the next simulation in the following way:

Result: Agents decide coreMove for next simulation while (simNum $\leq \frac{q}{2}$)
if *Agent.change == True* **then**
    Agent.coreMove = Agent.coreMove;
    Commence next simulation;
else
    Agent.coreMove = getRandomValidMove();
    Commence next simulation;
end

Algorithm 2: The "Bad Branch" search of the Wise Explorer Algorithm

The goal of Algorithm 2 is to keep our Multiagent system focused only on moves that have sub-par outcomes in the search tree (since Agent.change indicates that the agent either won or tied). This part of the Wise Explorer algorithm ensures that during our next $q/2$ simulations where we will be depending on our table to guide us to worthwhile branches, we will more confident that we have ruled out branches that have little to no chance of winning the overall game.

For the next $q/2$ simulations, at the termination of a simulation, we decide each agent's coreMove for the next simulation in the following way:

Result: Agents decide coreMove for next simulation while (simNum $> \frac{q}{2}$)
if *Agent.change == True* **then**
    Agent.coreMove = topMove (explained in next section);
    Commence next simulation;
else
    Agent.coreMove = getRandomValidMove();
    Commence next simulation;
end

Algorithm 3: The "Promising Branch" search of the Wise Explorer Algorithm

The concept behind this second phase of the Wise Explorer algorithm is to choose the best move based on our table's equilibrium so far if the agent's simulation ended without a win.

Otherwise, if the agent did win, we would collect that data and explore another branch. This choice to explore other branches if the agent wins allows us to make sure that our top move from our table is actually reliable as a fallback, while never ruling out the possibility that there could be better branches to be searched.

This will result in a significant amount of data being collected about branches that our equilibrium thinks will not lose, making it so that our top move will change if we find that there are more reliable moves in our search.

The next section will explain how we find this equilibrium and select this top move. It will also explain why we have two different ways to determine this move depending on if we are playing an Indifferent game or a Normal game.

5.3 Finding topMove through a Nash Equilibrium using Upper Confidence Bounds

This section explains how we use the accumulated table from our Multiagent System to converge to good moves.

Informally, a strategy profile is a Nash equilibrium if no player can do better by unilaterally changing his or her strategy [2]

More formally, we can describe the strategy profile of MBM as the accumulated table, and we can describe the Nash Equilibrium through our modified Upper Confidence Bound (UCB) algorithm that selects the best strategy.

Based on an algorithm from David Silver of UCL [5], our modified UCB algorithm is described as follows: Our optimal coreMove V^* is equivalent to $Q(a^*) = \max_{a \in A} Q(a)$, where A is the set of all valid moves in our given board state, and Q is our cost assessment function at the end of the simulations.

Our goal is to estimate an upper confidence $U_t(a)$ for $a \in A$ such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$. We let t be the number of times that move a has been used as a coreMove, $\hat{U}$ be our estimated confidence, and $\hat{Q}$ be our cost function before the end of our simulations. A small t for a given a results in a large $\hat{U}$, and thus we are less certain of $Q(a)$. Inversely, a large t for a given a results in a small $\hat{U}$, and thus we are more certain of $Q(a)$.

In MBM, based on section 5.2, our most promising $a \in A$ will naturally have a low $\hat{U}$. This is because during the pre-processing phase, bad $a \in A$ have a high t, and in the second phase, promising $a \in A$ also have a high t. Since we depend on Wise Explorer's self-aligning properties, we can essentially rule out $\hat{U}$ since we will be relatively confident about the moves that matter to us. Thus, MBM can use $\max_{a \in A} Q(a)$ as its method to choose topMove, while being confident about Q.

Our cost function Q is implemented differently for Normal Games (Q_{Norm}) as opposed to Indifferent Games (Q_{Indiff}). Using the intuition from section 3, we implement Q_{Norm} as follows:

Let d represent a depth that a coreMove a reaches, and let d_{max} be the maximum depth that a coreMove a terminated at across all times it was simulated. Let a_d represent move

a's information at depth d. For a given $a \in A$, we let $w = \sum_{d=1}^{d_{max}} \text{wins}(a_d)\ \forall a_d$ that exist, and we let $l = \sum_{d=1}^{d_{max}} \text{losses}(a_d)\ \forall a_d$ that exist. We let $S = w/(w+l)$, and we return S as $Q_{\text{Norm}}(a)$. The value of $Q_{\text{Norm}}(a)$ represents the overall win/loss ratio for move a across all depths that it terminated.

Using the intuition from section 3, we implement Q_{Indiff} as follows:

Let $S_{max} = \text{wins}(a_{d_{max}})/\text{losses}(a_{d_{max}})$, and let $S_{min} = \text{wins}(a_{d_{min}})/\text{losses}(a_{d_{min}})$ for $a \in A$. Since we want to move from a state where the player is least likely to win to a state where they are most likely to win, our return value for Q_{Indiff} is $P = S_{max}/S_{min}$. P is the ratio of how much we win in the future versus how much we lose in the beginning. We aim to maximize the strategic value of our move, and so a higher P represents moving into a better state than a lower P.

An important note on picking topMove for indifferent games in Algorithm 3: If there is an S_{max} s.t. $S_{max} = 1$, we only consider $a \in A$ where $S_{max} = 1$ during our evaluation function $\max_{a \in A} Q(a)$. This is because we want to make sure we are picking the most reliable future states over future states that don't have a 100% certainty that the player will win on that state.

For a game where the depth is set too low and there are no wins for all $a \in A$, we simply randomly select a move, since there is no information on a based on the inputted depth. If move a never loses, it will be selected as topMove move by default. Otherwise, intuitively, topMove is prioritized by $a \in A$ that win & tie but never lose, followed by $a \in A$ that tie, followed by $a \in A$ that tie & lose, finally followed by $a \in A$ that only lose (if we are unfortunate enough to be in a state where all $a \in A$ lose).

Thus, topMove in our second phase of Wise Explorer is determined by $\max_{a \in A} Q(a)$ using Q_{Norm} and Q_{Indiff}. Furthermore the move that the system chooses as the final "best move" for the inputted board state uses $\max_{a \in A} Q(a)$ at the end of all of our simulations.

The table's best move can be identified by this cost function, thus showing that a Nash Equilibrium can be derived from our system.

6 MBM Results & Performance

In sections 6.1, 6.2, and 6.3, I will be running a series test runs on MBM in order to demonstrate its strategic performance in Tic-Tac-Toe, Nim, and Minichess. The tests will involve playing MBM against a human and against itself. The tests will also have MBM make moves in pre-configured board states to demonstrate its strategic choices.

All of the tests below will be using a population of 50 agents, with 25 simulations per agent used for the pruning stage of Wise Explorer, and 25 simulations per agent used for the "Promising Branch" search of Wise Explorer (*See sections 5.1 & 5.2 for explanations of these processes*). The maximum depth that a simulation can reach is set to 200.

6.1 Tic-Tac-Toe-Tests

After running MBM against itself for a total of 100 games, MBM tied 100% of its games.

After I played against MBM for 30 games, MBM and I tied 100% of our games.

I experimented with making sub-optimal moves against MBM, and if I made a sub-optimal move, MBM would most likely win, if not tie, depending on the board configuration.

Here is an example of the most common game that MBM plays against itself, up to isomorphic states (Player 1 and Player 2 alternating going down):

*	*	*
*	O	*
*	*	*

*	*	X
*	O	*
*	*	*

*	O	X
*	O	*
*	*	*

*	O	X
*	O	*
*	X	*

*	O	X
*	O	*
*	X	O

X	O	X
*	O	*
*	X	O

X	O	X
O	O	*
*	X	O

X	O	X
O	O	X
*	X	O

X	O	X
O	O	X
O	X	O

From the game above, it is clear that MBM makes strategic moves each turn. It is also able to prevent itself from losing a game if necessary.

Another, more complex game that MBM (X's) played against me (O's) can be seen below:

*	*	*
*	O	*
*	*	*

*	*	X
*	O	*
*	*	*

*	*	X
*	O	*
O	*	*

*	*	X
*	O	*
O	*	X

*	*	X
*	O	O
O	*	X

*	*	X
X	O	O
O	*	X

*	O	X
X	O	O
O	*	X

*	O	X
X	O	O
O	X	X

O	O	X
X	O	O
O	X	X

We can see that MBM was able to avoid losing the game by the 7^{th} board state by making an optimal move in the 4^{th} board state. This demonstrates that MBM's ability to see more than 1 level of depth into the future and make strategic moves is viable, as MBM always tied against me in games like the one above.

Overall, MBM excels at tic-tac-toe, and is able to choose optimal moves 100% of the time given 50 agents and 50 simulations.

6.2 Nim Tests

For Nim, I will be using a series of sample board states that demonstrate MBM's ability to make strategic moves in Indifferent games while playing against itself.

Below we have an example of a game of Nim with 3 piles that is easily winnable by the player that moves first. MBM played against itself 50 times and reproduced the same results every time, up to isomorphic states. Given the board: | 2 | 3 | 2 |

2	0	2
1	0	1
0	0	0

1	0	2
1	0	0

This game of Nim demonstrates MBM's ability to determine the only optimal move given 7 potential moves and many potential search branches.

Below we have an example of a game of Nim with 4 piles that is also winnable by the player that moves first, but is heavily reliant on successive optimal moves to win. MBM played against itself 50 times and reproduced the same results every time, up to isomorphic states. Given the board: | 1 | 2 | 3 | 2 |

1	2	1	2

1	1	1	1

0	0	1	1

0	0	0	0

1	1	1	2

0	1	1	1

0	0	0	1

This game shows MBM's ability to pick the optimal move across several different piles, as well as its ability to win the game in the correct sequence when the winning branch is obvious.

Finally, we have an example of a game of Nim with 3 piles, but with 11 possible initial moves to make, and only one initial move that would let player one win if player two was playing optimally. MBM has the task of finding and identifying an exact optimal "arm", so for this particular board I increased the number of agents that MBM used to 500, and the number of simulations to 500 to increase MBM's odds of finding this arm. It played this game 20 times and successfully won as player one 16 out of those 20 times. Given the board: | 1 | 5 | 5 |

0	5	5

0	1	1

0	0	0

0	1	5

0	0	1

This demonstrates MBM's ability to identify "needle-in-the-haystack" arms in large search spaces, but also demonstrates MBM's limitations in Indifferent games. Because early moves in Indifferent games can determine whether the player wins or loses, MBM usually requires a significant amount of agents and simulations to perform well in indifferent games. This is because MBM has to either get lucky and find the winning branch, or even worse, search the entire space before it finds that branch. However, there are potential ways to deal with and optimize for this problem that I will discuss in section 8.

The game above also shows that Player 2 knows that it will lose after it takes its first turn, and so removes 4 marbles from pile 3 in hopes that Player 1 makes a mistake and takes 1, 2, 3, or 5 marbles from pile 2 next turn (allowing Player 2 to win). Since these games are simulated randomly, Player 2 realizes that this is the most direct way to leverage this strategy, as it has a 4 in 5 chance of winning if Player 1 makes a bad move on its second turn.

6.3 Minichess Tests

For Minichess, I will be using two sample board states that demonstrate MBM's ability to make strategic moves in a Normal game with a large search space. Even though MBM's performance is limited in indifferent games with large search spaces, MBM performs surprisingly well in Normal games with large search spaces because of the ability to tie in normal games. Ties/neutral moves make it so that MBM doesn't have to explore every branch of the search space to make a strategic move, and since there is usually more than one way to win a normal game, MBM is much more likely to get "lucky" and find a strategic branch.

The implementation of Minichess that I use is the popular 5x4 board, with the initial board state and pieces defined below (P = Pawn, C = Castle, Q = Queen, K = King, Piece_1 = Player 1's piece, Piece_2 = Player 2's piece):

C_2	Q_2	K_2	C_2
P_2	P_2	P_2	P_2
$*$	$*$	$*$	$*$
P_1	P_1	P_1	P_1
C_1	Q_1	K_1	C_1

In the board configuration below (one that MBM played itself into), we see an example of a tie in which MBM can find no better strategy, and so remains in a loop for 20 turns before I decide to terminate the program. This indicates that MBM can make strategic moves, and prevent itself from losing, all in a large and deep search space. Given the board:

$*$	$*$	$*$	$*$
K_2	$*$	$*$	P_2
P_1	P_2	$*$	P_1
$*$	P_2	$*$	$*$
$*$	$*$	$*$	K_1

The game plays out as follows (with Player 2 going first):

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
K_2	P_2	$*$	P_1
$*$	P_2	$*$	$*$
$*$	$*$	$*$	K_1

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
K_2	P_2	$*$	P_1
$*$	P_2	$*$	K_1
$*$	$*$	$*$	$*$

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
K_2	P_2	$*$	K_1
$*$	$*$	$*$	$*$

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
K_2	P_2	$*$	$*$
$*$	$*$	$*$	K_1

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
$*$	P_2	$*$	$*$
$*$	K_2	$*$	K_1

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
$*$	P_2	$*$	K_1
$*$	K_2	$*$	$*$

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
K_2	P_2	$*$	K_1
$*$	$*$	$*$	$*$

$*$	$*$	$*$	$*$
$*$	$*$	$*$	P_2
$*$	P_2	$*$	P_1
K_2	P_2	$*$	$*$
$*$	$*$	$*$	K_1

As we can see in this board state, MBM realizes that there are no possible ways for Player 1 or 2 to win. If K_2 tries to approach K_1 from the bottom, K_2 will lose, and if K_1 tries to move from its two squares, it will be taken by a P_2. Even if K_2 were to make its way around from the top, either K_1 would take K_2 or P_1 would take K_2. This shows how deep MBM's predictions are with only 50 agents and 50 simulations, as well as how it can make strategic moves effectively in large search spaces for Normal games.

In the final board configuration below (one that MBM played itself into), we see an example of MBM demonstrating a high level of gameplay with moves that are very clearly strategic in nature, and with clear foresight taken into consideration. Given the board:

K_2	$*$	$*$	C_2
P_2	$*$	$*$	P_2
P_1	$*$	P_2	P_1
$*$	$*$	P_1	C_1
$*$	$*$	$*$	K_1

The game plays out as follows (with Player 2 going first):

K_2	C_2	$*$	$*$
P_2	$*$	$*$	P_2
P_1	$*$	P_2	P_1
$*$	$*$	P_1	C_1
$*$	$*$	$*$	K_1

K_2	C_2	$*$	$*$
P_2	$*$	$*$	P_2
P_1	$*$	P_2	P_1
$*$	$*$	P_1	C_1
$*$	$*$	K_1	$*$

K_2	$*$	$*$	$*$
P_2	$*$	$*$	P_2
P_1	C_2	P_2	P_1
$*$	$*$	P_1	C_1
$*$	$*$	K_1	$*$

K_2	$*$	$*$	$*$
P_2	$*$	$*$	P_2
P_1	P_1	P_2	P_1
$*$	$*$	$*$	C_1
$*$	$*$	K_1	$*$

K_2	$*$	$*$	$*$
$*$	$*$	$*$	P_2
P_1	P_2	P_2	P_1
$*$	$*$	$*$	C_1
$*$	$*$	K_1	$*$

K_2	$*$	$*$	$*$
$*$	$*$	$*$	P_2
P_1	P_2	P_2	P_1
$*$	C_1	$*$	$*$
$*$	$*$	K_1	$*$

K_2	$*$	$*$	$*$
$*$	$*$	$*$	P_2
P_1	P_2	$*$	P_1
$*$	P_2	$*$	$*$
$*$	$*$	K_1	$*$

This board state is packed with a lot of surprisingly good moves, which I will explain turn-by-turn:

- Player 2 moved C_2 left on Board State 1 (B_1) as an aggressive move to try and win the game fast by taking K_1 on the next turn.
- Player 1 counteracted this on B_2 by positioning K_1 so that it could take C_2 on such an attack.
- On B_3, Player 2 moved C_2 to try and take the P_1 to its left on the next turn so that it could easily take K_1 in the future. Player 2 also made this move because C_2 is not useful anywhere else on B_3.
- Player 1 counteracted this on B_4 by taking C_2 with P_1.
- On B_5, Player 2 moves P_2 to take C_2 in order to allow P_2 the potential to take K_1 on a later turn.
- On B_6, Player 1 prevents Player 2's pawns from easily taking K_1 by moving C_1 in front of the recently placed P_2 in order to force the diagonal P_2 to take C_1. This prevents a future pawn win from Player 2.
- Finally, on B_7, we see that Player 2 does respond to Player 1's move by taking C_1 with P_2 as expected.

This series of moves shows just how well MBM can perform given such a small amount of agents and simulations. Considering the possible number of moves an agent can make

on a given turn in Minichess and how deep an agent can play up to, the moves above are unlikely to be random in nature. This particular game also shows the power of the Wise Explorer algorithm in large search spaces for Normal games. With an effective runtime of $\mathcal{O}(\text{Agents}\times\text{Simulations}\times\text{Depth})$, Wise Explorer is able to eliminate and find promising branches, all while never having to do a complete search on the game space.

7 Conclusion

MBM is capable of playing games without full knowledge of the search space, or any domain-specific knowledge. It is able to identify branches of interest through its unique two-stage Wise Explorer algorithm, and it is able to play intelligently not only against itself, but against humans. Its performance in large search spaces only improves with more agents and simulations, and even still, its performance with small numbers of agents and simulations in Normal games shows just how powerful the Nash Equilibrium of MBM is.

MBM is important to the field of General Game Playing because it is able to derive strategy without automatic heuristic generation, or any biased cost functions. The only variables that are used in creating the Nash Equilibrium are win/neutral/loss statistics, and the fact that MBM plays strategically regardless of this fact will be important in figuring out ways to reduce the amount of computations and complications in General Game Solvers and General AIs.

MBM is also exciting because it is able to merge UCB Algorithms, Multiagent Systems, and Nash Equilibria to create a functional system that has the potential to become better and more efficient with future improvements. It is also a system designed to tackle the Bandit-Arms problem, which has scope far beyond General Game Playing.

8 Future Directions

MBM isn't a perfect system. It has downfalls in efficiency compared to other GGP systems, but makes up for these downfalls in its simplicity compared to other systems. With some additions and modifications to MBM in the future, MBM could work around some these inefficiencies, and work with some of its downfalls.

As mentioned in Seciton 6.2, MBM has a difficult time finding specific branches in large search spaces for Indifferent games since it relies on random simulations. One way to increase the possibilities of MBM finding these important branches is through multiprocessing. By allocating groups of agents to different CPU cores, or even different systems, we would be able to parallelize multiple groups of agents and combine their data tables either at the end of each simulation, or while finding a final Nash Equilibrium.

If we combined the tables at the end of each simulation during multiprocessing, the computation time of MBM would be reduced by a factor of the number of cores. In essence, it would reduce 8-minute computations to 1-minute computations, and using supercomputing, the time reduction would be huge.

If we combined the tables when finding a Nash Equilibrium at the end of the simulations, we would be able to compare multiple equilibria. This would hopefully result in at least one of those equilibria having the specific, optimal branch. This method would also have the benefit of reducing the possibility of the population getting "stuck" on a branch for too many simulations, as topMove would likely be different for each individual agent population. Overall, this method would likely be the best approach, as it has the benefit of multiple cores and less computation time, along with the ability for different agent populations with unique tables to increase the certainty that topMove is optimal.

An addition to MBM that could result in better moves at the cost of efficiency could be to calculate a best possible next move for coreMove. Essentially, if an agent had a coreMove a, another simulation would be run to determine its best nextMove b. This would increase the time complexity of MBM drastically since an entire simulation along with Nash Equilibrium analysis would have to be done to find b, but it could potentially converge on optimal branches even more effectively. These simulations for b could be done on separate cores to reduce the computation time, and how b is used in determining topMove would need to be taken into consideration.

Running a small simulation that reaches 2 or 3 levels deep for coreMove might also be a viable way to check for any obvious downfalls and corrections in the future, while not having to simulate an entire game. This concept of being "extra-certain" about an agent's coreMove could even be applied to all of the moves made in a simulation, but the effectiveness of this method would need to be tested first.

There are many other way to potentially improve on MBM, including implementing Automatic Heuristic Generation functions, or trying to gather data on all moves at all depths during simulations to assess the Nash Equilibrium in a more comprehensive way.

9 Acknowledgements

I have had so much fun with this project, and I cannot end this paper without thanking the people that have helped me along the way!

I want to thank Bob Geitz, my thesis advisor, for being a source of certainty that what I was doing was worthwhile, and helping me brainstorm and think through the many ideas in my head.

I want to thank Adam Eck, my A.I. and Machine Learning Professor, for showing me where this project was going through a technical lens, and how it related to the field of A.I.

Finally, I want to thank my friends and family for being a constant source of motivation and affirmation.

A final thanks to Oberlin College and the people in it for teaching me how to love Computer Science, and to love learning. Starting this paper was daunting, but I have learnt so much from this experience, and can't wait to explore the field of General A.I. more in the future.

Shoutout to the CS150 critter lab for being the spark to my love of Multiagent systems.

A Appendix

Rules of Nim: The rules of Nim are as follows: There are P piles of marbles, and each pile has m marbles. Each turn, a player can choose a pile p_i for $i \in P$, and they can choose how many marbles p_i that they want to remove from p_i (players must remove at least one marble per turn). The goal is to be the player to remove all marbles from a pile such that all piles at the end of the player's turn have 0 marbles left. Player one and two alternate turns.

References

[1] Paul Benjamin Reverdy. "Human-inspired algorithms for search A framework for human-machine multi-armed bandit problems". PhD thesis. Princeton University, 2014.

[2] Martin J. Osborne and Ariel Rubinstein. *A course in game theory*. The MIT Press, Boston, 2016.

[3] Tony Peng. *Uber AI Beats Montezuma's Revenge (Video Game)*. 2018. URL: `https://medium.com/syncedreview/uber-ai-beats-montezumas-revenge-video-game-dee33417a56e`.

[4] *Montezuma's Revenge Solved by Go-Explore, a New Algorithm for Hard-Exploration Problems (Sets Records on Pitfall, Too)*. 2019. URL: `https://eng.uber.com/go-explore/`.

[5] David Silver. *Exploration and Exploitation*. 2019. URL: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/XX.pdf`.

[6] Stephan Schiffel and Michael Thielscher. *Fluxplayer: A Successful General Game Player*. URL: `http://ggp.stanford.edu/readings/fluxplayer.pdf`.