Presidents Game Playing Agent

Nicholas Larus-Stone Juan Perdomo Matt Goldberg

December 10, 2015

1 Introduction

This paper provides an in-depth investigation of a number of algorithms applied to the multi-player card game Presidents. Presidents presents an interesting problem because it is a game involving imperfect information, making it much more difficult to choose an optimal strategy. We first applied two common multiplayer game playing algorithms to this problem: paranoid and max-n [**?**]. Neither algorithm inherently handles uncertainty, so we had to develop our own methods for dealing with imperfect information. We also investigated more complex algorithms that are better at handling uncertainty: Monte Carlo Tree Search [**?**] and reinforcement learning [**?**]. Our goal was to apply all of these algorithms to Presidents and determine which one offered us the best performance.

The algorithms we implemented were largely derived from the minimax algorithm we talked about in class. However, due to the increased complexity of Presidents, we also had to innovate in order to deal with the imperfect information–for this we drew inspiration from our unit on inference and uncertainty. We will first describe how each of the algorithms were implemented, then we will discuss their results and analyze the effectiveness of our implementations to this specific problem. Finally, we will point out some of the areas for further research that our work led us towards.

2 Background

Research into game playing has been a main focus of members of the artificial intelligence community since the mid 20th century. As mentioned in lecture by Prof. Rush, computer scientists worked on topics such as chess AI as early as 1959 [1]. Although research into game playing algorithms might seem like an inefficient investment of resources by talented researchers, breakthroughs in the algorithms and techniques designed for simple games such as chess or Go have often had repercussions in other fields that have more of a direct impact on society. At their essence, these algorithms work to find solutions to problems in which players compete to maximize their rewards under certain constraints. In the case of Presidents, however, there is the additional complication of having imperfect information as each individual agent must operate without knowing the hands of his opponents. Developments in game playing algorithms for problems like Presidents therefore can provide insights into different real life problems such as auctions in which a

[1]Rush, Adversarial Search and Games

number of agents compete to achieve certain goals using incomplete information.

2.1 Rules of Presidents

The rules of the game are simple. The deck is dealt out evenly to all players. The goal of the game is to get rid out of all your cards as quickly as possible. In order to do so, you can only play cards higher than the current card at the top of the stack. 3's are the lowest cards and 2's are the highest. Once a 2 has been played, the stack is cleared and the player who played it gets to go again. There are a few extra rules. Order of play is decided by the player who has the 3 of clubs and then others play clockwise in the order they are seated. If the stack is clear, a player can choose to play doubles or triples and the players following him must also play higher doubles or triples. A player can choose to pass at any point in the game even if he has legal actions in that state. Lastly, after the game is done, the first two players to finish become President and Vice President and the last two become Asshole and Vice Asshole. For the following round of play, the President has the right to the Assholes' 2 best cards and the VP takes the VA's best card while at the same time handing them back a card of their choice (usually their lowest). The rest of the players don't switch any of their cards.

3 Related Work

During our research into game playing methods for Presidents. We found no algorithms that dealt specifically with the game of presidents, however, we did find research in related fields such as card playing AI agents and agents for games of incomplete information, the most common of which was the popular card game Hearts. Multiplayer Games, Algorithms and Approaches by Sturtevant was was particularly helpful to learn about extensions of Minimax to multi agent games [**?**]. Koller's work on imperfect information provided insights into adaptations of traditional game playing agents for scenarios in which there is imperfect information[**?**]. Lastly, papers by Sturtevant and Browne on Max-N, Paranoid, and Monte Carlo Tree Search methods served as guides that led our initial development of the algorithms used[**?**, **?**].

4 Overview of Algorithms

In order to deal with the uncertainty of not knowing which cards

4.1 Paranoid

The first algorithm we implemented was a paranoid agent. The paranoid algorithm is a modified minimax algorithm which assumes that all the other players in the game are collaborating against it. It does this at the evaluation step, by estimating values for each of the players, then returning the max player's value minus the sum of the other players' values. This reduces the multiplayer game to a two player one in which traditional minimax can be applied.

The tricky part is that the game tree is so large that it's impossible to expand out the tree fully. So, we had a cutoff point for our algorithm to stop evaluating (usually 2 rounds of play). However, this means that we didn't have access to the results of the game, only what we thought the game looked like. This meant we had to use a heuristic to evaluate how good the state was for us. As we will talk about later, creating a heuristic for this game is quite difficult, but also incredibly important for the performance of these algorithms.

Algorithm 1 Pseudocode for Paranoid Algorithm

```
procedure PARANOID(state, depth, player)
    if isTerminal(state) or depth > MAXDEPTH then
        values = evaluate(state)
        return values[0] - sum(values[1:])
    end if
    if isMax(player) then
        v = -∞
        for action in LegalActions do
            nextState = getNextState(state, action)
            v = max(v, paranoid(nextState, depth + 1, nextPlayer))
            a = action if v changed
        end for
    end if
    if isMin(player) then
        v = ∞
        for action in LegalActions do
            nextState = getNextState(state, action)
            v = min(v, paranoid(nextState, depth + 1, nextPlayer))
            a = action if v changed
        end for
    end if
    return a, v
end procedure
```

4.1.1 Pruning

Something that is very nice about the paranoid algorithm is that since it is so closely related to minimax, $\alpha\beta$ pruning can be built on top of it. After the paranoid evaluation, we return an action and an associated value. This means we can keep track of what we would return from the current node and prune other sibling nodes if they would ever return a value that would not replace our value. When we implemented pruning, we saw a decrease in the number of nodes expanded by the Paranoid algorithm by about 500 nodes on average. Since the Paranoid algorithm expands about 3000 nodes when MAXDEPTH is two rounds, this was a dramatic increase in speed.

4.2 Max-n

The second algorithm we implemented was max^n. This algorithm adapts minimax to many players by having each player pick the action that maximizes the value to herself in the subsequent state. Values for states are represented by n-length tuples, where n is the number of players, and where the ith entry in the tuple is the value of that state to player i. By contrast, when expanding the game tree in minimax, player 1 assumes the player 2 is minimizing player 1's value of the subsequent state; thus, in max^n, there is no "Min" agent, and instead every agent is a "Max" agent for their own value in the tuple.

A clear specification of the algorithm(s) you used and a description of the main data structures in the implementation. Include a discussion of any details of the algorithm that were not in the published paper(s) that formed the basis of your implementation. A reader should be able to reconstruct and verify your work from reading your paper.

4.3 Monte Carlo Tree Search

The MCTS algorithm is divided into 4 key stages: selection, expansion, simulation, and backpropagation. In selection, a node is chosen starting from the root of the game tree by recursively selecting the most promising child node until a leaf in the tree is reached. After a node has been chosen, the game is played out using a default policy until a result is reached. Lastly, in backpropagation, the result of the playout is incorporated recursively into each node located in the path from the root to the selected child.

Algorithm 2 Pseudocode for Monte Carlo Tree Search

```
procedure MCTS(state)
    root ← mctsNode(state)
    while budget do
        nextNode ← selection(root)
        result ← simulation(nextNode)
        backpropagate(nextNode, result)
    end while
    action ← bestChild(root)
    return action
end procedure
```

In the context of Presidents, a separate tree data structure to the state formalism had to be developed in order to implement MCTS. Each node in the tree contained variables describing the number of times each node had been visited as well as the score of the node. Moreover, each node contained a list of hands for each player and the id of the player whose turn it was to play. Since the hands of the other players in the game are unknown, when instantiating the tree, hands had to be sampled for the other agents in the game based on the sizes of each player's hand and the set of cards yet to be played. Each child node corresponded to the actions the player whose turn it was could make based on his hand and the top card on the deck. Scores were defined as the finishing position of the agent in the game. Since higher scores are better, we took the inverse of the finishing rank to indicate the score.

In the selection stage, we used the UCT algorithm developed by Kocsis and Szepesvari discussed by Browne in his paper on MCTS methods. It takes into account exploration and exploitation of paths down the tree by weighing the score of each node and the number of times it and its parent had been visited. Once a node is selected, the tree is expanded to include the children of that node and one of the child nodes is selected at random. In simulation, a game was played out based one the hands of each player at that node. Finally, in the backpropagagration stage the normalized score (raw score divided by number of visits) is sent up the tree to the node updating the values of each node along the way. This is run until a predetermined budget is used up, which in this case, we chose to be time. After the time ran out, the agent made a move based on the child of the action with the best score.As a way of exploring different implementations, we also tested sampling hands for players a number of times and then choosing the most common action based on all the playouts.

5 Experiments

Analysis, evaluation, and critique of the algorithm and your implementation. Include a description of the testing data you used and a discussion of examples that illustrate major features of your system. Testing is a critical part of system construction, and the scope of your testing will be an important component in our evaluation. Discuss what you learned from the implementation.

Can test on a number of different axes–time to run, nodes expanded, how it does playing against dummy agents, how it does playing against other algorithms, how it does playing against itself, number of times we sample.

5.1 Methods and Models

Wrote dummy agents to play lowest legal card–naive algorithm. Baseline to test against that for how good our agents were.

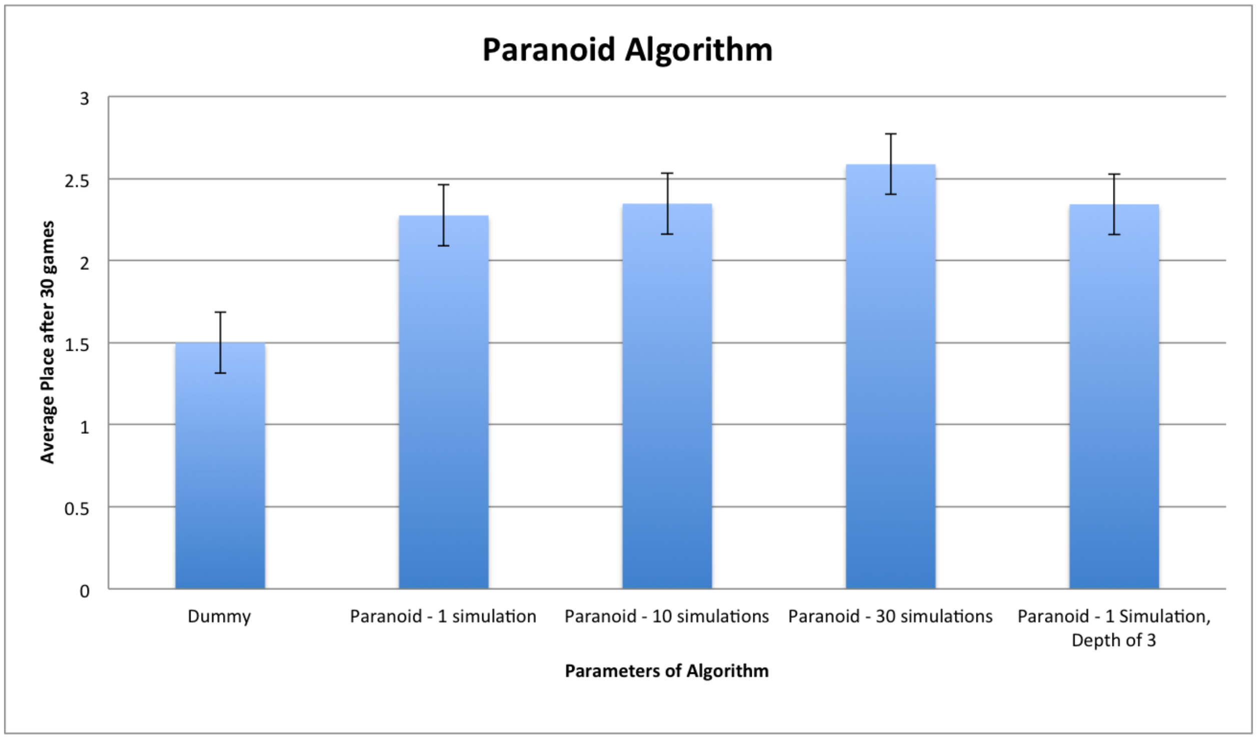
5.2 Results

For algorithm-comparison projects: a section reporting empirical comparison results preferably presented graphically.

Table 1: showing time to run on 50 games, average score
Graph 1: Paranoid (5 trials, include error bars)
Bar graph showing:
Average score for Dummy vs Dummies
Paranoid vs Dummies (sample once, 500 nodes)
Paranoid vs Dummies (sample 5 times, 500 nodes)
Paranoid vs Dummies (sample 25 times, 500 nodes)
Paranoid vs Dummies (sample 25 times, 1500 nodes)
Graph 2: Max-n
Graph 3: MCTS

5.2.1 Paranoid

The paranoid algorithm we implemented did worse than even a basic rule based agent:



We hypothesize that there are two main reasons that this happened.

5.2.2 Monte Carlo Tree Search

In the end, Monte Carlo Tree Search performed significantly worse than the baseline as seen in the following table:

MCTS Benchmark Results after 30 Simulations:

	Number of Hands Sampled	Exploration Constant (c)	Budget (in s)	Score (0-3)
1	1	10	1	2.8
2	1	10	.5	2.4
3	1	1	.5	2.5
4	10	10	1	2.66
5	10	1	1	2.63

Even after altering the parameters for budget and exploration, the performance of the algorithm remained largely unchanged. In order to examine its shortcomings, we used the verbose option on the play game function to analyze its choice of actions in different scenarios. As it turns out, the MCTS agent consistently plays all its high cards first and is later forced to pass in the later stages of the game. The reason for this is that there is a dichotomy between its own actions and the actions it simulates in the tree.

When it expands a node and simulates a playout, it simulates itself playing as a dummy agent. Therefore, if a node corresponds to playing a 2 (meaning everyone else is forced to pass), it simulates from a state in which it has one card less than everyone else. In the simulation, it plays as a dummy agent meaning it gets rids of its lowest possible card all the time. In this scenario it makes sense for the MCTS agent it to play a high card to get 'ahead' and the play low the rest of the game as a dummy agent. However, the next time it is forced to play a card in the game it again picks a high card which runs contrary to what it simulated in the algorithm. In order to fix this, the simulation would have to play as itself and not as a dummy agent. However, this risks running into a circular definition of the algorithm and would involve huge computational costs as it would need to simulate itself repeatedly in order to choose a single action.

Given this analysis and the results of the simulations, we see that the poor performance of the algorithm is more due to inherent shortcomings of MCTS in its application to Presidents than from suboptimal choice of parameters. In order to improve the overall performance, a new approach is required.

5.3 Discussion

Overall, there are optimizations to be made in the approach to solving the constraint of imperfect information. As of now, the sampling of hands is done uniformly for each player. However, in the game of presidents, since the president and the asshole switch hands it is more likely that the asshole have a worse hand than the secretary, who didn't switch any cards, and that the secretary in turn have a worse hand than the president. This issue could be approached in several ways.

The first way would be to estimate the average hand strength of each respective player rank at each point in the game and then sample hands for that player based on that average. After playing a large number of games, we could infer the distribution for the hands of each depending on how many cards they have and then sample from that distribution. One major issue with this approach is that it is blind to past behavior of each agent. An agent could have been playing high cards all through the game and then be left with just low cards. However, the distribution for the player's hand at that point in the game could still be very high, giving the player an unrealistic set of cards.

The second way to approach the problem would be to create a filtering algorithm that estimates the current hand strength based on the history of play of the player. Each card played would be considered an emission and the hidden states would be the player's hands after each time it plays. The trouble in this approach would be estimating the transition probabilities between hidden states as well as the emission probabilities for the played cards. Emissions are not independent of the top card on the deck. Also, it is not clear whether playing a high card means your next state is weaker or stronger. If for example, an agent plays a 2 and gets to go again (this is a rule of the game), he might: 1) have another 2 and a low card which menas he has the chance to win, or 2) he might be left with a 3 and be forced to pass for the next rounds.

In terms of improving MCTS, there is a lot of work that can be done refining the different parameter values. The performance of the algorithm is dependent on how well it selects nodes in the tree and then expands them. This exploration is regulated by means of a constant that can be altered. Moreover, the time budged can be changed to that it expands more of the tree. By optimizing the values for these two parameters, one can increase the efficiency of the algorithm and the quality of the actions taken. More importantly however, there are improvements to be made in the simulation of a playout from a node in order to remove the differences between actual and simulated games.

6 References

References

[1] Sturtevant, Nathan. *Comparison of Algorithms for MultiAgent Games.* Computers and Games. 2003

[2] Browne, Cameron. *A Surver of Monte Carlo Tree Search Methods.* IIEEE Transactions on Computational Intelligence and AI in Games. 2012

[3] Fujita, Hajime. *A reinforcement learning scheme for a multi-agent card game.* Systems, Man and Cybernetics. 2003

[4] Kjeldsen, Tinne Hoff. *John von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts* Arch. Hist. Exact Sci. 2001

[5] Sturtevant, Nathan. *Multi-Player Games: Algorithms and Approaches.* 2003

[6] Koller, Daphne, and Pfeffer, Avi. *Generating and solving imperfect information games.* IJCAI. 1995

A Program Trace

Appendix 1 – A trace of the program showing how it handles key examples or some other demonstration of the program in action.

B System Description

Appendix 2 – A clear description of how to use your system and how to generate the output you discussed in the write-up and the example transcript in Appendix 1. N.B.: The teaching staff must be able to run your system.

C Group Makeup

- Matt Goldberg - Wrote the state, agent, and game modules that provide the necessary formalisms and rules to encode the game of Presidents. Also implemented the Max-N algorithm.
- Nicholas Larus-Stone - Implemented the repeated games function that allows a list of agents to play multiple games. Implemented the Paranoid algorithm and the RL algorithm.
- Juan Perdomo- Implemented the formalisms necessary to implement the MCTS algorithm as well as implemented the algorithm itself.

  All contributed equally to this writeup.