Solving Arbitrary Two-Player Games using Regret Matching and Counterfactual Regret Minimization (Draft)

Michael Peters

605.724 Applied Game Theory

April 6th, 2021

Table of Contents

Introduction

Game Theory is a very popular field of study with wide applicability to most aspects of life. Whether economics, politics, science, or day-to-day interactions, game theory can offer insight and strategic advantage when applied appropriately. Game Theory compares the motivations, goals, and strategies of different players within a game in an attempt to gain insight and advantage. According to the *Stanford Encyclopedia of Philosophy* (Ross, 2019), Game Theory is defined as "the study of the ways in which interacting choices of economic agents produce outcomes with respect to the preferences (or utilities) of those agents, where the outcomes in question might have been intended by none of the agents." In this definition, an economic agent would be a player of the game, and could be an individual person, company, or even country. The outcome would be the result of the game, and the preference or utility of the agent would be the desired outcome.

There are many parameters that can shape games. According to *Applied Game Theory and Strategic Behavior* (Geckil & Anderson, 2016), games can have 2 players, or many more. Games can also be adversarial or cooperative, where players could work together towards the same goal. Additionally, games may be sequential, where each player makes a move in turn, or simultaneous, where players can make moves at any time. Another aspect of game theory is the availability of information. If both opponents know what moves their opponent has made and the related payoffs, that game is said to have "perfect information". In many games, this is not the case, and there is "incomplete information". Additionally, one opponent may have more information than the other, resulting in "asymmetric information".

Problem

As a computer scientist, it is natural to wonder how software can be utilized to help solve game theory problems. In fact, there are many different software applications in the domain of game theory.

One such area of interest in computer science is solving a given game for an optimal strategy for a given player. In this area, the game is again broken down into its key components. Once this is done, specialized algorithms can be utilized to learn the elements of the game, and to develop a strategy for each player that is most likely to yield the optimal outcome, given their utility function. These simulations would allow a human playing the game in real-life to learn the possible outcomes and potentially adopt a strategy from the system.

In this project, I have looked at two styles of games: two-person simultaneous single-turn games, and two-person zero-sum sequential games. Each of these games involve two players, but the other parameters are quite different. Solving games such as these requires specialized algorithms that can derive equilibrium strategies for each player. There are many algorithms that have been developed to accomplish these tasks. For this project, I will be implementing two game theory algorithms: Regret Matching and Counterfactual Regret Minimization, and building a web application that will allow users to define games, and solve them using the applicable algorithm.

Description of Domain

There is a whole area of research around algorithms that can accurately solve game theory games. One foundational algorithm known as Regret Matching (Hart & Mas-Colell, 2000) can be used to solve 2-player, single-turn, simultaneous games. Regret Matching works by repeating the game and measuring the "regret" of a player for each of their possible actions. After some number of iterations, this converges to a correlated equilibrium solution for both players. Regret Matching has been shown to find a correlated equilibrium for any game that meets its assumptions. Two-player, single-turn, simultaneous games are the most common games presented to students learning about game theory, so this algorithm is an important part of the field of game theory algorithms.

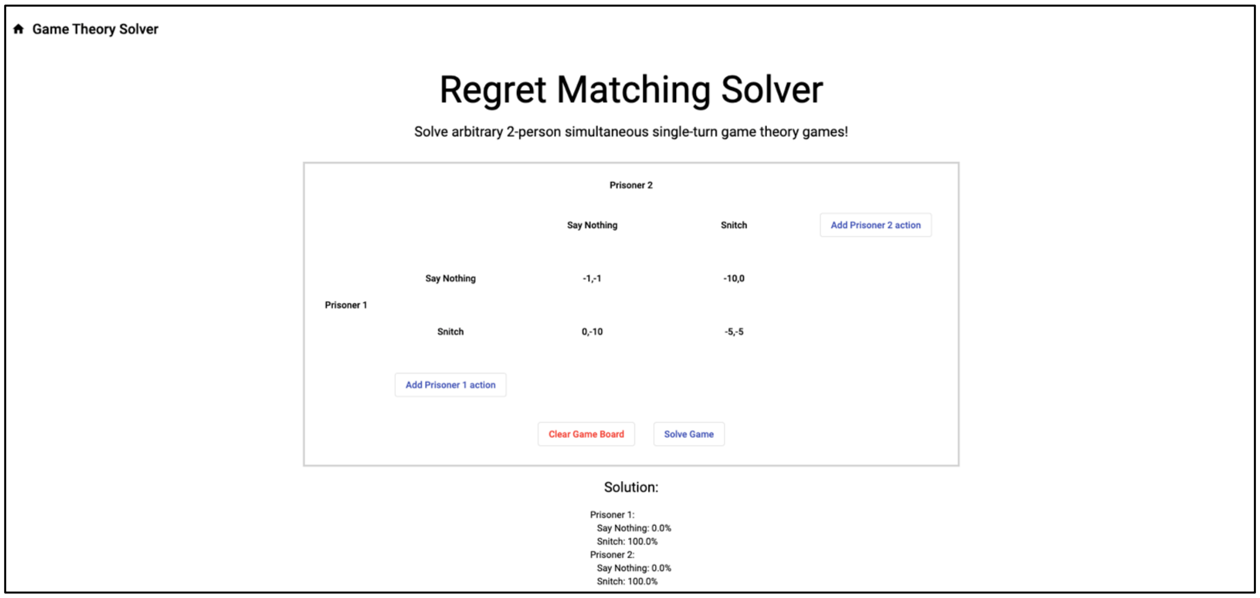
One of the more prominent algorithms is known as Counterfactual Regret Minimization, or CFR for short (Zinkevich, Johanson, & Piccione, 2007). CFR builds on the same mathematical assumptions as Regret Matching, but is extended to solve 2-player, zero-sum, sequential games, with any number of turns per player. CFR is an algorithm that gained popularity for modeling decisions in games such as poker. It works by looking at the regret of players based on the outcome of previous games. CFR works by breaking down the regret of a player in a game "into a set of additive regret terms, which can be minimized independently." The algorithm is additionally capable of solving games of incomplete information.

Many variations of Counterfactual Regret Minimization have been proposed and studied. Variations such as Monte Carlo CFR (Gibson, Burch, Lanctot, & Szafron, 2012), Deep CFR (Brown, Lerer, Gross, & Sandholm, 2019), Single Deep CFR (Steinberger, 2019), and Lazy CFR (Zhou, Ren, Yan, Li, & Zhu, 2019) have been developed and shown to offer improvements on the original CFR, which is often referred to as "vanilla" CFR. Additionally, updates to the algorithm such as the work by Gibson (Gibson, 2013) allow it to solve non-zero-sum games, pending some assumptions.

Other algorithms can also be used to solve zero-sum game theory games. Gilpin and Sandholm have developed one such algorithm in their paper "Solving two-person zero-sum repeated games of incomplete information" (Gilpin & Sandholm, 2008), and Daskalakis, Deckelbaum, and Kim developed a "Near-Optimal No-Regret Algorithm" (Daskalakis, Deckelbaum, & Kim, 2015).

Approach and Implementation

For this game theory solver application, I have built a web application using modern web technologies. I have utilized an Angular and Typescript frontend with a simple node.js backend. For the types of games that Regret Matching can solve, I have developed a simple user interface that allows a user to specify the players, actions, and payoffs. The interface limits the user to 2-player games, but allows 2 or more actions to be utilized per player. Actions can be added and renamed via the interface. For each square in the game matrix, users can specify the utility of each player as any positive or negative number. Several images of this interface are shown below in figures 1-4.



Regret Matching Solver

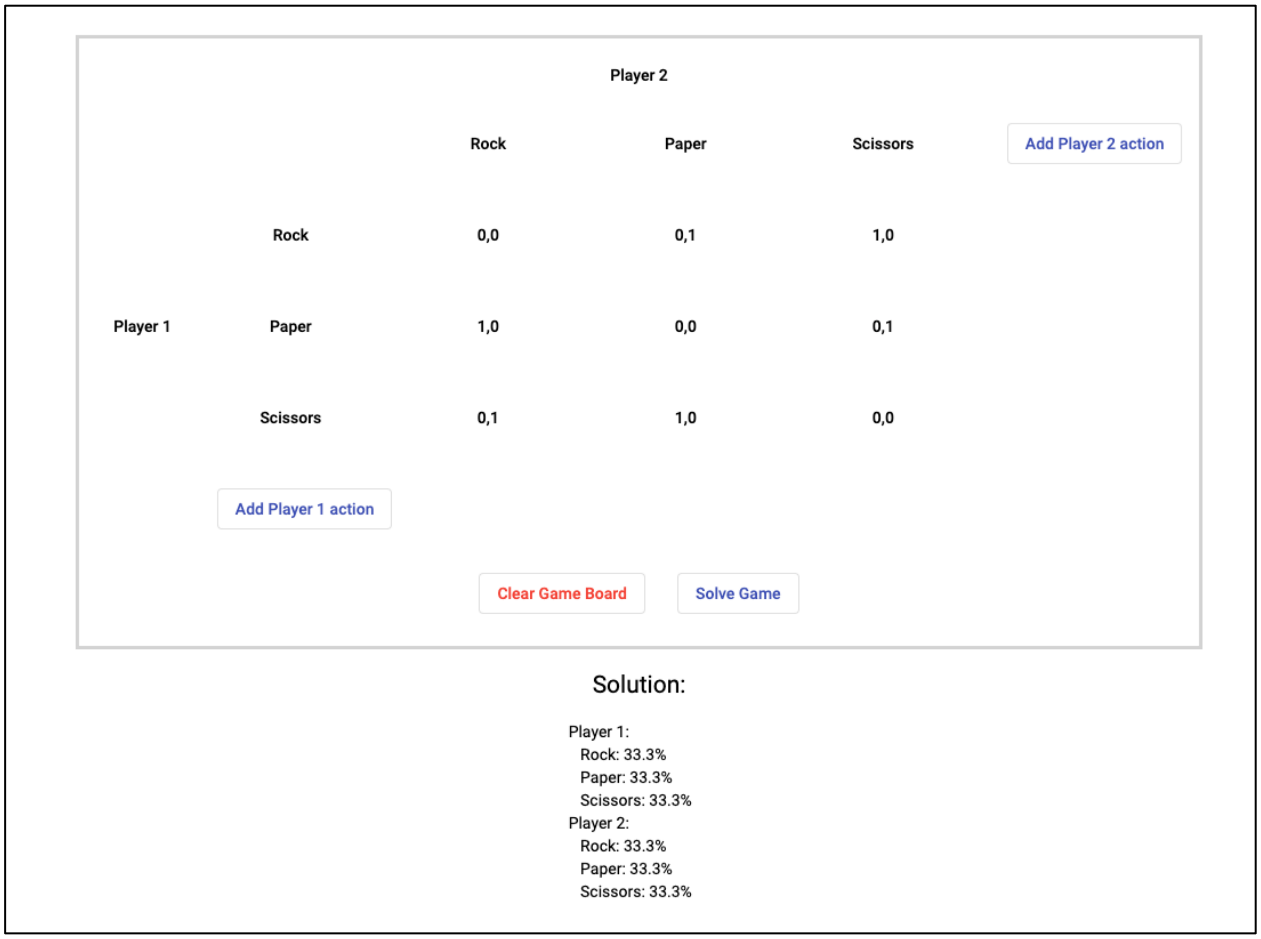
Solve arbitrary 2-person simultaneous single-turn game theory games!

		Prisoner 2	
		Say Nothing	Snitch
Prisoner 1	Say Nothing	-1,-1	-10,0
	Snitch	0,-10	-5,-5

Solution:

```
Prisoner 1:
  Say Nothing: 0.0%
  Snitch: 100.0%
Prisoner 2:
  Say Nothing: 0.0%
  Snitch: 100.0%
```

Figure 1: The Regret Matching Solver interface, with 'The prisoner's dilemma' game loaded and solved.

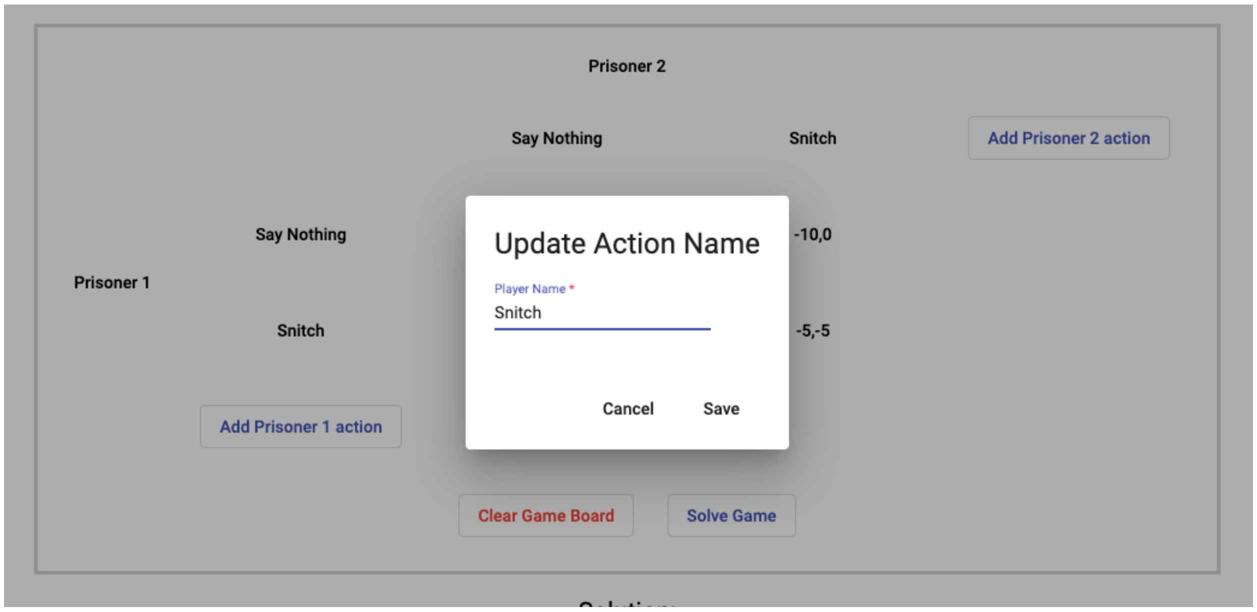


		Player 2		
		Rock	Paper	Scissors
	Rock	0,0	0,1	1,0
Player 1	Paper	1,0	0,0	0,1
	Scissors	0,1	1,0	0,0

Solution:

```
Player 1:
  Rock: 33.3%
  Paper: 33.3%
  Scissors: 33.3%
Player 2:
  Rock: 33.3%
  Paper: 33.3%
  Scissors: 33.3%
```

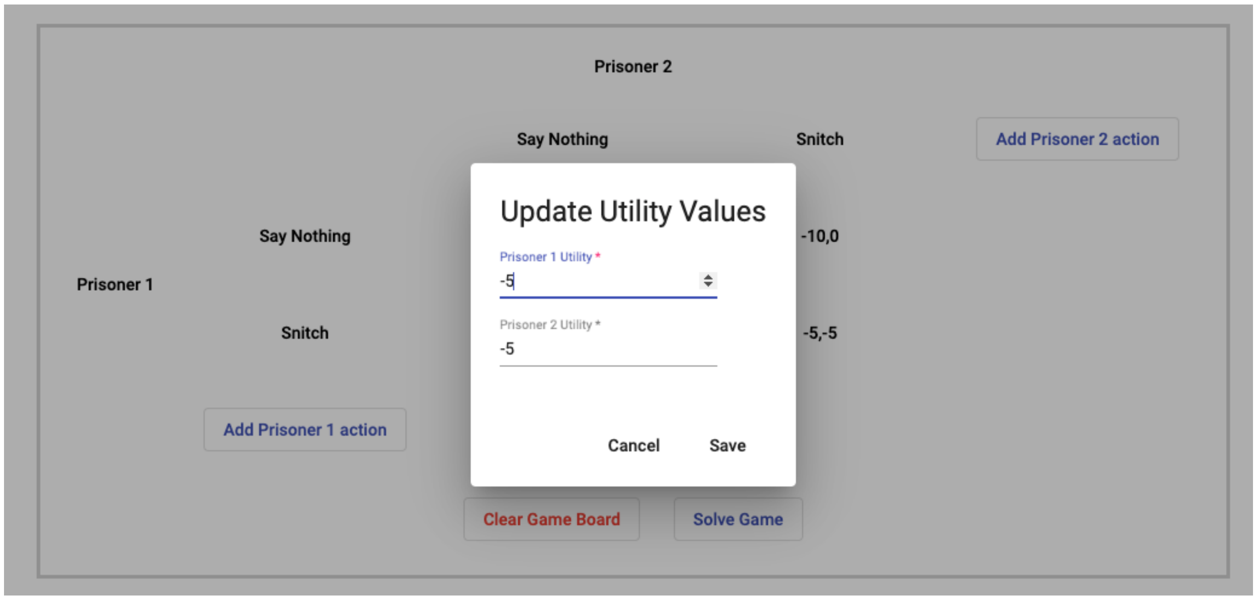
Figure 2: The Regret Matching Solver interface, with the 'Rock/Paper/Scissors' game loaded and solved.



Update Action Name

Player Name *
Snitch

Figure 3: The Regret Matching Solver interface for updating an Action name.



Update Utility Values

Prisoner 1 Utility *
-5

Prisoner 2 Utility *
-5

Figure 4: The Regret Matching Solver interface for updating the utilities for a game outcome.

In addition to implementing the UI for inputting 2-player simultaneous single-turn games, I've also successfully implemented the regret matching algorithm. I utilized a lecture from a Carnegie Mellon University game theory course (Brown & Sandholm, 2017) to learn and understand how the algorithm worked. I then completed the implementation of the algorithm in Typescript. The algorithm runs in the browser nearly instantaneously, which is promising. One tunable parameter of the algorithm is how many iterations to run before you consider the solution to be converged. I settled on 10,000, as it seemed to be a standard value and ran very quickly while still producing accurate results. Initially I was concerned that the processing resources necessary to derive solutions might prohibit a client-side implementation, but that did not prove to be an issue.

For Counterfactual Regret Minimization, the tool needed a way to allow users to define games. CFR is used to solve 2-player zero-sum sequential games, which presents an immediate problem: there is no simple way to present such games in a user interface. There can be multiple turns, strategies, payouts, and even probabilistic events in games that CFR can solve. Allowing a user to specify such a game through an easy-to-use interface would be large project on its own. Rather than spend time here, I decided to use an alternate strategy.

Instead of building a UI for the definition of 2-player zero-sum sequential games of incomplete information, I decided that allowing users to specify their own games in some standard format would be a better way to go. I spent some time researching if there were any standards for representing 2 player games of partial information, but I was unable to find any existing standards. Because of this, I decided to develop a standard method of specifying these games. I chose to use JSON-schema to build the specification, as JSON is the easiest data standard to utilize in modern web applications. The specification I've built allows users to define a game and specify game states and transitions. Game states can represent probabilistic events, such as dealing cards, or a player's turn, along with the possible actions, the partial knowledge held by that player, and the subsequent states associated with each action. This enumeration can get verbose for larger games with many possible outcomes, but is simple enough to offer the information necessary to fully understand a game. It is also built in a way that could allow automated specification of a game through a software utility that would enumerate all states for a game.

To test the schema, I've developed game specifications for two games. The first is an incredibly simple game that I've named "Even or Odd." Player 1 chooses either the number 1 or 2, and holds up that many fingers behind their back. Player 2 then exclaims "even" or "odd". If player 1 had 1 finger up, and player 2 picks odd, player 2 wins. If player 2 chooses wrong, player 1 wins. It is simple to see that the mixed strategy for each player should be 50% of each action, so this will be a good test case for my program. The second game I've encoded in the json specification is Kuhn Poker (Kuhn Poker). Kuhn Poker is a simplified version of poker that involves betting and is played between 2 players with 1 card each. This game is a classic example used for testing solver algorithms such as CFR, so it was an obvious example to implement. The game states and payouts (from the perspective of player number 1) are shown in figure 5 below.

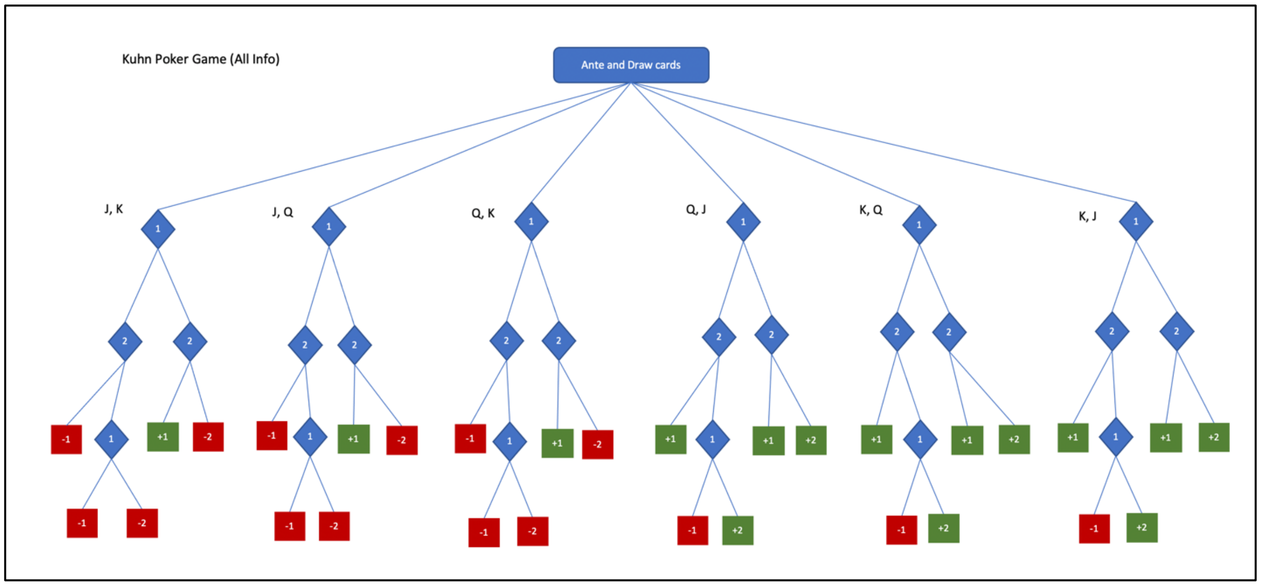
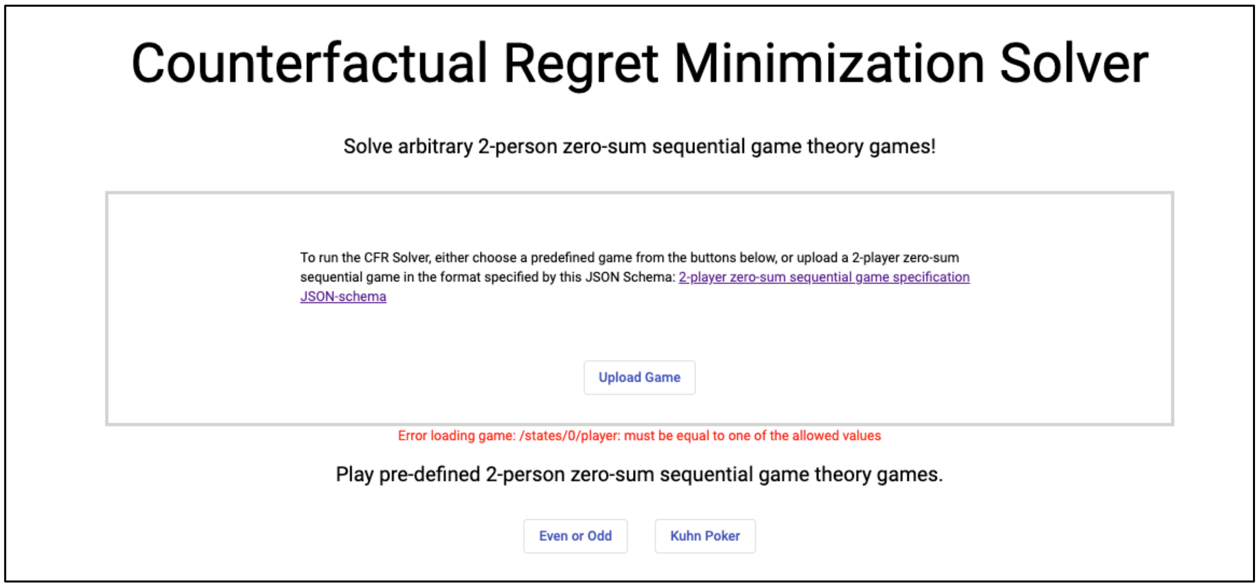


Figure 5: The game states and payouts (from the perspective of player number 1) of Kuhn Poker.

I've hosted the code for the json-schema is hosted on GitHub, along with the examples, here: https://github.com/mjp5153/2-player-zero-sum-sequential-game-spec. I'm hopeful that it can be utilized by other game theory applications moving forward.

After developing the json specification for representing 2-player games of incomplete information, I updated the web application to utilize it. The two examples are built in, as well as a method to validate JSON games against the JSON-schema. This will allow users to upload their own specified games to the system to be solved. If the uploaded game does not validate against the schema, the application will show the validation errors so that the user can fix their specification. An example of a validation error is shown in figure 5 below.


Counterfactual Regret Minimization Solver

Solve arbitrary 2-person zero-sum sequential game theory games!

To run the CFR Solver, either choose a predefined game from the buttons below, or upload a 2-player zero-sum sequential game in the format specified by this JSON Schema: 2-player zero-sum sequential game specification JSON-schema

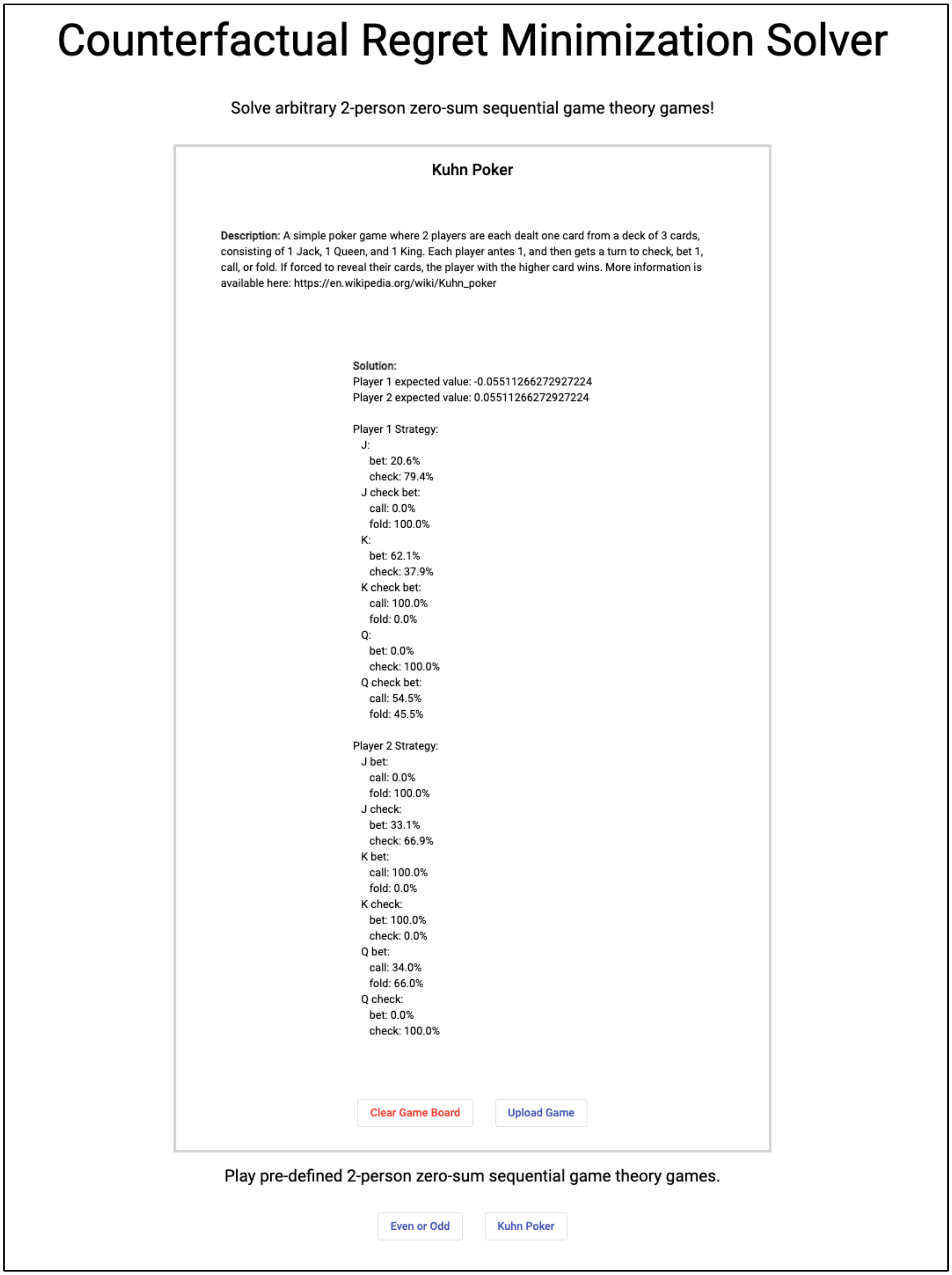
Error loading game: /states/0/player: must be equal to one of the allowed values

Play pre-defined 2-person zero-sum sequential game theory games.

Figure 5: A validation error in the game-specification JSON.

With that in place, I then had to implement the algorithm. With utilizing my available resources, such as the original CFR paper by Zinkevich, Johanson, & Piccione, the lecture notes of Brown & Sandholm, as well as a paper by Neller and Lanctot, I was able to successfully code the CFR algorithm in typescript. This task presented several challenges. Once particular sticking point was the fact that I had encoded the utility from the games as positive with respect to player 1. As the CFR algorithm recurses through steps, the perspective of the players changes. When a particular recursion step is being played from the perspective of player 2, I needed to alter the algorithm to invert the utility. This particular bug took me a long time to find and correct.

Once corrected, I was able to do some testing. Both example games that I encoded could run through the algorithm and produce the expected result: 50/50 strategies for both players in “Even or Odd” and a Kuhn Poker strategy that matches the solution detailed in the references (Kuhn Poker). The Kuhn Poker result is shown in Figure 6.


Counterfactual Regret Minimization Solver

Solve arbitrary 2-person zero-sum sequential game theory games!

Kuhn Poker

Description: A simple poker game where 2 players are each dealt one card from a deck of 3 cards, consisting of 1 Jack, 1 Queen, and 1 King. Each player antes 1, and then gets a turn to check, bet 1, call, or fold. If forced to reveal their cards, the player with the higher card wins. More information is available here: https://en.wikipedia.org/wiki/Kuhn_poker

```
Solution:
Player 1 expected value: -0.05511266272927224
Player 2 expected value: 0.05511266272927224

Player 1 Strategy:
  J:
    bet: 20.6%
    check: 79.4%
  J check bet:
    call: 0.0%
    fold: 100.0%
  K:
    bet: 62.1%
    check: 37.9%
  K check bet:
    call: 100.0%
    fold: 0.0%
  Q:
    bet: 0.0%
    check: 100.0%
  Q check bet:
    call: 54.5%
    fold: 45.5%

Player 2 Strategy:
  J bet:
    call: 0.0%
    fold: 100.0%
  J check:
    bet: 33.1%
    check: 66.9%
  K bet:
    call: 100.0%
    fold: 0.0%
  K check:
    bet: 100.0%
    check: 0.0%
  Q bet:
    call: 34.0%
    fold: 66.0%
  Q check:
    bet: 0.0%
    check: 100.0%
```

Play pre-defined 2-person zero-sum sequential game theory games.

Figure 6: The Solution from the CFR-Min solver on Kuhn Poker.

The runtime of the algorithm was several seconds for Kuhn Poker, which was not so long as to necessitate a move to the server side. I did add a loading spinner to show that the algorithm was still running. Like Regret Matching, a tunable parameter of this game is how many iterations to run before considering the solution converged. I upped the default value of 10,000 to 30,000, as the additional runtime was not very noticeable and the solutions appeared better. At 10,000, the solutions suffered from rounding errors, while they seemed a bit closer to the accepted solutions at 30,000.

I added the ability to print the game results to the screen, and then implemented the ability for users to upload their own JSON schema compliant encodings of games. I tested this out with a few made up games of my own, and it seems to work well.

With the application completed, I needed to find an applicable hosting solution to launch the site on the internet. Amazon Web Services (AWS) is an immensely popular web hosting solution that happens to have a free hosting tier for static websites. Because all of the processing logic is in the front end, this project qualifies as a static site. I was able to bundle and deploy the application to the open internet using AWS this way, and the project is accessible here: http://game-theory-solver.us-east-2.elasticbeanstalk.com/cfr-minimization.

Evaluation, Results, and Reflections

This project presented a great opportunity to learn about game theory algorithms and games. I was able to utilize skills I already had to build and deploy an angular web application, while learning and employing new skills, such as the rules of different types of game theory games, and the algorithms that can be utilized to solve them. I was able to successfully implement and deploy a solver that can correctly solve both two-player simultaneous single-turn games using Regret Matching and two player sequential zero-sum games using Counterfactual Regret Minimization.

While I am happy with the result of my project, I underestimated the scope of work when making my initial project proposal. I did not foresee the need for a game representation schema to represent simultaneous games of incomplete information when I initially proposed this project. The development of that standard added a good deal of time that I had not accounted for. If I were starting over, I would account for that time, and potentially limit the scope a bit. That said, I was able to complete this task, and am quite happy with the results.

Extensions and Future Work

In the future, it would be interesting to expand on the project presented here. There are many different kinds of games that can be solved by game theory that this project does not address. It would be interesting to research and implement solvers for games that involve more than two players, aren’t zero sum games, or that involve intricate game mechanisms not covered by this project.

Additionally, it would be interesting to research and implement some of the advanced variants that have built upon CFR. Deep CFR or Monte Carlo CFR in particular would be very interesting to learn more about, and would likely enable a solver to develop optimal strategies for very complex games. Such a project would likely involve utilizing specialized hardware, and would almost certainly not run in a web application in a browser. It would also be fun to look at different algorithms that don’t utilize regret at all, to learn other methods of accomplishing this task.

One other thing that I would be very interested in pursuing would be the expansion of development of the Game Theory game representation schema. In almost all of my research, there was no common method of defining game theory games, so I believe that this is an area of need for the community. The ability to easily specify a game and to use a generic solver to develop strategies for it would make the barrier of entry to applied game theory lower than it currently is. This might allow more people to utilize these tools, and lead to growth in the field. I think the field of game theory could greatly benefit from something like, and would love to see it expanded and utilized.

Conclusion

In conclusion, I was successfully able to implement a web application that allows users to specify and solve game theory games. Specifically, users can specify and solve both two-player simultaneous single-turn games using Regret Matching and two player sequential zero-sum games using Counterfactual Regret Minimization. The latter form of game can be specified using a JSON schema that I developed specifically for this project. The web application was then deployed to the internet and is available for use here: http://game-theory-solver.us-east-2.elasticbeanstalk.com/cfr-minimization.

References

Ross, D. (2019, March 08). Game Theory. In E. N. Zalta (Ed.), The Stanford Encyclopedia of Philosophy (Winter 2019 Edition). Retrieved February 07, 2021, from https://plato.stanford.edu/archives/win2019/entries/game-theory/

Geckil, I. K., & Anderson, P. L. (2016). Applied Game Theory and Strategic Behavior. Chapman and Hall/CRC.

Bueno de Mesquita, B. (2011). 3. Applications of Game Theory in Support of Intelligence Analysis. In B. Fischhoff & C. Chauvin (Eds.), Intelligence analysis: Behavioral and social scientific foundations (pp. 57-82). Washington, D.C.: National Academies Press. doi:10.17226/13062

Zinkevich, M., Johanson, M., & Piccione, C. (2007). Regret minimization in games with incomplete information. Retrieved February 23, 2021, from http://martin.zinkevich.org/publications/regretpoker.pdf

Gibson, R., Burch, N., Lanctot, M., & Szafron, D. (2012). Efficient monte carlo counterfactual regret minimization in games with many player actions. Retrieved February 23, 2021, from https://papers.nips.cc/paper/4569-efficient-monte-carlo-counterfactual-regret-minimization-in-games-with-many-player-actions.pdf

Brown, N., Lerer, A., Gross, S., & Sandholm, T. (2019, May 22). Deep counterfactual regret minimization. Retrieved February 23, 2021, from https://arxiv.org/abs/1811.00164

Steinberger, E. (2019, October 04). Single deep counterfactual regret minimization. Retrieved February 23, 2021, from https://arxiv.org/abs/1901.07621

Zhou, Y., Ren, T., Yan, D., Li, J., & Zhu, J. (2019, September 25). Lazy-cfr: Fast and near-optimal regret minimization for extensive games with imperfect information. Retrieved February 23, 2021, from https://arxiv.org/pdf/1810.04433.pdf

Gibson, R. (2013, April 30). Regret minimization in non-zero-sum games with applications to building champion multiplayer computer poker agents. Retrieved February 23, 2021, from https://arxiv.org/abs/1305.0034

Gilpin, A., & Sandholm, T. (2008). Solving two-person Zero-sum repeated games of incomplete information. Retrieved February 23, 2021, from https://www.cs.cmu.edu/~sandholm/repeated.AAMAS08.pdf

Daskalakis, C., Deckelbaum, A., & Kim, A. (2015). Near-optimal no-regret algorithms for zero-sum games. Retrieved February 23, 2021, from http://people.csail.mit.edu/costis/optimalNR.pdf

Hart, S., & Mas-Colell, A. (2000). A simple adaptive procedure leading to correlated equilibrium. Retrieved March 8, 2021, from http://wwwf.imperial.ac.uk/~dturaev/Hart0.pdf

Kuhn Poker. (n.d.). Retrieved March 08, 2021, from http://gambiter.com/poker/Kuhn_poker.html

Brown, N., & Sandholm, T. (2017). Extensive-form games and how to solve them: Part 2. Lecture presented at CMU 15-381 and 15-681 Fall 2017. Retrieved February 27, 2021, from http://www.cs.cmu.edu/~15381-f17/Lecture20_games.pptx

Neller, T. W., & Lanctot, M. (2013, July 9). An introduction to counterfactual regret minimization. Retrieved April 13, 2021, from http://modelai.gettysburg.edu/2013/cfr/cfr.pdf

JSON schema. (n.d.). Retrieved March 28, 2021, from https://json-schema.org/

Angular. (n.d.). Retrieved April 07, 2021, from https://angular.io/

Typed JavaScript at any scale. (n.d.). Retrieved April 07, 2021, from https://www.typescriptlang.org/

Node.js. (n.d.). Retrieved April 07, 2021, from https://nodejs.org/

Amazon Web Services (n.d.). Retrieved April 19, 2021, from https://aws.amazon.com/

Appendix A: Relevant Links

1. 2-player zero-sum sequential game specification JSON-schema: https://github.com/mjp5153/2-player-zero-sum-sequential-game-spec

2. Source code for the Game Theory Solver application: https://github.com/mjp5153/CFR-minimization-solver

3. Link to the Game Theory Solver, hosted on AWS: http://game-theory-solver.us-east-2.elasticbeanstalk.com/

Appendix B: Game Specification example

The following json is a schema-compliant variant of the "Even or Odd" game, where player 1 can hold up 1-5 fingers (instead of 1-2), and player 2 still just picks even or odd. This can be used to test the "upload game" feature of the CFR Min solver.

```
{
  "name": "Even or Odd",
  "description": "A simple game, where player 1 puts up either
1, 2, 3, 4, or 5 fingers behind their back, and player two
guesses even or odd.  If player 2 guesses correct, player 2
wins.  If player 2 guesses incorrectly, player 1 wins.",
  "start": "pick",
  "states": [
    {
      "name": "pick",
      "player": 1,
      "knowledge": "",
      "actions": [
        {"1": "1"},
        {"2": "2"},
        {"3": "3"},
        {"4": "4"},
        {"5": "5"}
      ]
    },
    {
      "name": "1",
      "player": 2,
      "knowledge": "",
      "actions": [
        {"even": "1 even"},
        {"odd": "1 odd"}
      ]
    },
    {
      "name": "1 even",
      "result": 1
    },
    {
      "name": "1 odd",
      "result": -1
    },
    {
      "name": "2",
      "player": 2,
      "knowledge": "",
      "actions": [
        {"even": "2 even"},
        {"odd": "2 odd"}
      ]
    },
    {
      "name": "2 even",
      "result": -1
    },
    {
      "name": "2 odd",
      "result": 1
    },
    {
      "name": "3",
      "player": 2,
      "knowledge": "",
      "actions": [
        {"even": "3 even"},
        {"odd": "3 odd"}
      ]
    },
    {
      "name": "3 even",
      "result": 1
    },
    {
      "name": "3 odd",
      "result": -1
    },
    {
      "name": "4",
      "player": 2,
      "knowledge": "",
      "actions": [
        {"even": "4 even"},
        {"odd": "4 odd"}
      ]
    },
    {
      "name": "4 even",
      "result": -1
    },
    {
      "name": "4 odd",
      "result": 1
    },
    {
      "name": "5",
      "player": 2,
      "knowledge": "",
      "actions": [
        {"even": "5 even"},
        {"odd": "5 odd"}
      ]
    },
    {
      "name": "5 even",
      "result": 1
    },
    {
      "name": "5 odd",
      "result": -1
    }
  ]
}
```