Spring 2023

Comparing Voting Strategies in Blood on the Clocktower

Marty Graham
Bard College

Comparing Voting Strategies in Blood on the Clocktower

Senior Project Submitted to
The Division of Science, Mathematics, and Computing
of Bard College

by

Martin Graham

Annandale-on-Hudson, New York
May 2023

Dedication

This Project is dedicated to Tina, my friend, who helped me climb a mountain.

Acknowledgements

I want to thank my advisors Sven Anderson and Keri-Ann Norton for all their support over the years and with this final project in particular. I would also like to thank my squash coach Craig Thorpe-Clarke for the unwavering support he has shown me throughout my time at Bard College.

Abstract

This project models a social deduction game called “Blood on the Clocktower.” Simulated players act according to two different algorithms, and the results are recorded across four different variables. The results show that the two algorithms, while constrained to affecting one specific mechanic within the game, produce statistically different results. This model has the potential to be used in simulating group dynamics and modeling the efficacy of certain game strategies.

Table of Contents

Project Overview

I am programming a virtual simulation of a “Blood on the Clocktower" game. Simulated games will be filled with simulated players undertaking simulated actions. This will be accomplished by programming each character and their corresponding ability. Characters are different from players. A character is a game element, while a player is a person. Players are assigned characters and will undergo actions based on their character. In this model, there is little practical difference between a player and a character. Players are assigned a character at the beginning of the game and remain unchanged for the entire game. At certain points throughout the game, players will vote to “execute” another player. My program will simulate two different voting algorithms of a Clocktower game. The first algorithm, Random, uses a random voting mechanic. This means that when players vote to execute other players, they do so randomly. My second algorithm, Grouped, creates voting groups that influence the behavior of players within those groups. Players will only vote for those that are outside of their own voting group until three players remain alive. Then they will vote randomly.

Introduction

What is Blood on the Clocktower?

Blood on the Clocktower is a social deception game created by The Pandemonium Institute. Based in Sydney, Australia, the company was founded in 2019 in order to produce their flagship game, Blood on the Clocktower. With funding from a successful Patreon campaign, Clocktower began shipping in May, 2022 with three scripts included in the box (The

Pandemonium Institute, 2019, [4]). Like many social deception games, Clocktower is built around the interactions of the players. A set of mechanics that vary in complexity provide the structure for players to deduce, lie, and engage with each other. My program takes one of the simpler scripts, Trouble Brewing, and simulates a game with a selection of those characters.

Blood on the Clocktower plays similarly to *Werewolf*, *Mafia*, or *Town of Salem*. These and other social deduction games use minimal physical elements, often relying on the social interactions of players to move the game along. In Clocktower, a group of players sit in a circle and are assigned characters at random. There is an evil team and a good team. The evil team is at a numerical disadvantage, but start the game knowing who is on their team. Each member of the good team only knows his or her own character and so needs to determine who is good and who is evil. Every player is told their own character, but must determine everybody else's character through abilities or communication. Playing as both good and evil involves pretending to be other characters and trying to determine the veracity of the other players' claims. It is a game of knowing when to withhold and when to share information. There is a logical deduction component that works alongside social reading and group dynamic manipulation.

Blood on the Clocktower is a game that is particularly hard to translate into a digital medium. It's composed of elements like, choosing, voting, talking, and randomness that all would be feasible to program and study, but in their totality, increase the complexity of modeling a game exponentially. While Clocktower has a collection of mechanics that are programmable together and separately, the human element is much more difficult to simulate. The interaction between players is what makes a real game both unique and stimulating, and these interactions suffuse nearly every other aspect of Clocktower. My program takes one concrete mechanic of the

game, voting, and examines a specific type of communication style that naturally occurs between real players.

When new players play Clocktower, they have a tendency to vote in a specific way. They are quick to trust a select few other players, and they often don't grasp certain complexities inherent in the voting mechanic. My program simulates a near zero communication situation where players' only simulated communication is the creation of voting blocs that stay static through most of the game. I use a Random voting algorithm to compare against the results of the Grouped data. This represents a first step in simulating complicated human behavior using a structured game system. The next steps for modeling human behavior would involve the exchange of information between players, and how the information ecosystem interacts with the game's core mechanisms.

Characters

Each player is assigned a character. There are four different types of character: Townsfolk, Outsiders, Minions, and Demons. Townsfolk are on the good team and have useful abilities. Outsiders are on the good team but have abilities that hinder their teammates. Minions are evil and exist to help their Demon. The Demon is the main member of the evil team responsible for winning, usually by killing a player every night. At any point in the game, if there are two players alive and one of them is the Demon, evil wins. The Demon has "decimated" the town. If the Townsfolk execute and kill the Demon, the good team wins.

Figure 1: Ratio of character types in a game

Figure 1 shows that in a seven player game, there are five townsfolk, zero outsiders, one minion, and one demon. This means that in my simulation, there are zero outsiders. The number of outsiders does have an outsized impact on games, and so the lack of outsiders in my simulation is noteworthy. However, outsider count by itself has no mechanical relation to the voting mechanics.

When players die, they remain in the game. They are still their characters, but without an ability and with reduced voting capability. They may only vote once more for the entire game. A "dead" player and an "alive" player are both still in the game.

Abilities Overview

Abilities vary. Nearly every ability interacts with information in a different way. Some receive information directly by learning that a certain character is in the game, and that it is one of two players: e.g. Chef, Librarian, and Investigator. Some abilities require specific circumstances to occur before they receive information: e.g. Undertaker and Ravenkeeper.

Others interact with the Imp and so can disrupt the evil team: e.g. Slayer, Soldier, and Mayor. All of these abilities produce information for players directly and indirectly so that they can determine the characters of other players. When a player dies, they remain in the game in every aspect except that they lose their ability and certain voting power. The player continues to find the demon for the good team, or disrupt the Townsfolk for the bad team, but no longer have a mechanical impact on the game outside of a specific vote.

Structure

There is a day/night cycle. During the "day" players will talk amongst themselves privately before reconvening to talk as a group in public discussion. The players will nominate each other and vote to execute a player. This is the main method for killing the Demon. After a successful execution, "night" will fall (everyone will close their eyes) so that various character abilities may be used. Days always end with a vote. If a player receives enough votes, they are executed. Once there is an execution, the day ends immediately. Otherwise, the day ends with no deaths. During the night, abilities are used. After the night, everyone opens their eyes for the next day of talking and the cycle continues until either the Demon is executed, or the Demon is one of two alive players. Each game always starts with an initial night where the Storyteller gives starting information to certain players.

The Storyteller

There is a player called the Storyteller who runs the game for the rest of the players. A good comparison would be the dungeon-master in *Dungeons and Dragons*, but without the narrative or role-playing aspects. They are not on either team but instead try to keep the game balanced and fun. The Storyteller chooses which characters are in the game from a larger list

called a script. From the beginning of a game, the Storyteller balances the game by including some characters and not others, so that each game has a unique combination of abilities. These characters are then randomly assigned to the players. The Storyteller holds a “grimoire” filled with notes and reminders for the current game. During the night, while consulting the grimoire, the Storyteller will wake up characters so that they can use their abilities. These abilities are implemented by the Storyteller. During the day, the Storyteller will keep track of the state of the game so that they can determine which team is winning. The Storyteller will also run the voting and executing. The Storyteller is not modeled in this simulation.

Scripts

All Clocktower characters mentioned and programmed in this project are part of the Trouble Brewing script. This set of characters is the simplest. Characters and players are either good or evil, the abilities are relatively straightforward, and there is a single Demon, the Imp, that has a low impact ability. Different scripts don’t change the structure of the game; they merely change the characters that each person is assigned. My project won’t explore the drastically more complex interactions present in the more advanced scripts.

Outside Sources and Context

Researchers Burnam et al. used an extensive form of a two-person trust game to determine if people have a preconscious friend-or-foe (FOF) evaluation mechanism. They primed the FOF state by referring to the counterpart of each player negatively or positively, as an “opponent” or “partner” respectively. The researchers found that trustworthiness with a “partner” was more than twice that of an “opponent” and that trust dwindles over time regardless (Burnham et al., 2000, [1]). My Grouped algorithm simulates a version of these results. Players

are grouped together at the start of the game to simulate a friend-trust dynamic and will vote for the opposing groups. When a certain threshold is reached, specifically when there are only three remaining alive players, the voting groups dissolve and players will vote for another player randomly.

Researchers Lee & Chang applied a model to 296 online game players' data to determine the impact of trust on players' teamwork. They found that trust affects teamwork when players are more experienced through affective commitment than low experience (Lee & Chang, 2013, [2]). My model was going to implement a trust mechanic that remained between games. Certain players would trust and make decisions that would be influenced by their multi-game experience.

Researchers Sarkadi et al. programmed agents using a Theory of Mind system when interacting with other agents. Their goal was to study machine deception and used a "belief-desire-intention" model to program and quantify their machine intelligences (Sarkadi et al., 2019, [3]). My program has no actual or simulated intelligence behind its agents at the moment. However, my program aims to be the first step towards agents that "trust" and "deceive" other agents. Clocktower is a game with mechanics that change drastically depending on how players interact, so the ability of players to be able to interact with each other deceptively would bring my model closer to simulating a real scenario.

Researcher Shane Tilton used two social deception games, "Are You a Werewolf?" and "The Resistance," to teach aspects of small group communication to students taking a communications course at a university. There was a focus on the power structures that emerge in small groups. Social deduction games are useful for gathering data due to their combination of play, interaction, and digestibility (Tilton, 2019, [5]). As this study shows, games are useful in that they are designed to be teachable, while creating structured environments that are ideal for

data gathering. Studying situations built around these games, like my program aims to do, provides useful data on human psychology and game theory.

Methods

My program will be used to analyze how voting patterns affect the outcomes and length of games. I will compare the results of a Random voting algorithm with a Grouped voting algorithm. My results will be comparing two different voting methods, and will be simulating a particular type of player and their voting habits. New players and their voting habits can be varied, but my data focuses on comparing a new player who votes at random, and a new player who sticks to voting with their group. This will show how much of a difference these two voting styles have on the game across multiple variables. From their length, to who wins more often, games are significantly impacted by these voting patterns. New players may instinctively group themselves in circles of trust, but whether or not this is an effective strategy for winning games more often will be shown.

Voting Dynamics

During the day, every player whose character is alive will vote to execute another player. When there are only three players left, all dead players will "use up" their vote token to vote. Nominations are not implemented. Instead, players will start the game sorted into one of three voting groups. Each group will vote for the same player and will not vote for a player in their own group. This is meant to simulate networks of trust in a real Clocktower game. In a real game, players will often form groups of one to three players with whom they share information and voting power. This comes about via a confirmation chain, wherein someone's ability yields

information that indicates another player is trustworthy, or by social reading. When a player trusts another based only on their interactions, then a social trust is formed. These bonds are usually more tenuous, but have a dramatic impact on the game. The Demon's best strategy for surviving is to create social trust in combination with a mechanical confirmation (whether real or manipulated). My program's three voting groups, in a three-three-one pattern, simulate two distinct groups that trust each other, and a lone player who trusts no one else.

The existence of these voting groups is modeled off of how new players might play the game. New players often trust the first few players they encounter. They also don't adhere to any established strategy when voting. My Random and Grouped algorithms are a basic implementation of these patterns.

Implementing Certain Aspects of Clocktower

Blood on the Clocktower is a complicated game. My program has a much narrower scope. Many characters' abilities are not fully implemented. I have a day/night cycle, but I have collapsed the nomination, voting, and execution mechanics into just voting and execution. In a normal game of Clocktower, who nominates who and how everyone votes is a large part of the game. Nominating and voting normally yields a large amount of information, but my simulation simplifies the process into a single vote where every player participates.

There are seven characters in my program. I picked these seven out of the twenty-two possible characters in the Trouble Brewing script, because I wanted to represent a balanced game with every kind of character ability. Game balance is a subjective term, and there is dramatic variation in how effective each player utilizes their ability. These characters do showcase all the different ways abilities function in an average game of Clocktower.

Implemented Characters and their Abilities

The different kinds of abilities all fall into categories for when they operate. There are starting game characters who receive information at the start of the game. There are nightly characters who receive information every night. There are conditional characters who receive information or operate their ability when certain conditions are met. Each of the seven characters in my program fall into these categories.

The day/night cycle is implemented so that characters perform their actions at a specific time. During night one, the Chef and the Poisoner use their ability. During subsequent nights, the Imp, Poisoner, Empath, and Monk use their abilities. If the Ravenkeeper or Soldier are targeted by the Imp, their abilities function. The Imp's ability to kill a player is implemented. However, the Imp's special ability to target themselves and turn an alive minion into the Imp is not. The following characters are implemented:

Imp

The Imp wakes every night (except the first) and chooses a player who then dies.

Poisoner

The Poisoner is on the evil team and wakes every night. They choose a player. That player's ability malfunctions. This means that players who receive information will receive information that is false. Players with abilities will lose their abilities. Poisoning lasts one night cycle.

Chef

The Chef is on the good team and wakes on night one and learns how many pairs of evil players there are. If there are two evil players in a game sitting next to each other, the Chef would learn a one.

Empath

The Empath is on the good team and wakes every night to learn how many of their neighbors are evil. They would receive either a zero, one, or two

Soldier

The Soldier is safe from the Demon (Imp) at night. If the Imp targets the Soldier to be killed during the night, no one will die that night.

Monk

The Monk protects another player from the Demon (Imp) at night. If the Imp targets a player that the Monk has protected, that player does not die.

Ravenkeeper

When the Ravenkeeper is killed at night, they choose a player and learn their character.

Graphs and Relationships

Graphs are data structures consisting of vertices connected via edges. Edges can be weighted, changed, or directed in specific ways. Graphs are used to show relationships between entities in social networks. In my program a graph is created to represent the physical seating arrangement of the players. Players are vertices with edges connected to two other players who "sit" next to them. Both the Chef and the Empath use this graph to determine their ability information. By iterating over the graph and checking alignment of the players, the Chef's ability produces the number of evil players who are connected by an edge. The Empath checks their neighboring vertices and determines how many of those are evil.

Code review and justification

The bulk of my program is written in a class called `Game`. The `Game` class creates a single simulated game. This game is populated with players via my `Player`"" class.

Within `Game`, I use a method called `init` to initialize a Clocktower game. The method `init` creates an array list of `Players`, `Player` names, and character types. Three array lists are lined up so that the nth index of each array list corresponds to the nth index of the other array lists. A `Player` object is added to an array list, `pList`, with a corresponding name and character type. For example, a new `Player` is created with the name `Tina` and the character type `IMP`. `Tina` the `IMP` is added to `pList` and a new `Player` receives the next name and character type. There are seven players in my game which means there are seven player names and seven character types. In addition to setting various starting game variables, `init` also runs the method `createVotingGroups`.

The method `createVotingGroups` sorts every player into three categories at random. Three players go into group A, three go into group B, and the remaining player goes into group C. This is accomplished by iterating through a randomly sorted array of indices. These indices correspond to players' static positions in their starting array list.

Throughout my code, I refer to players via a player's index. This int is associated with a player and never changes.

The central looping method that runs each game is called `cycle`. A boolean called `gameOver` keeps a while loop running within `cycle` until `gameOver` is false. When the game is over, `gameOver` becomes false, and `cycle` stops running. The main methods within `cycle` are `tick`, `updateAlivePlayers`, `groupVote`, the player `act` and player `vote` methods, `nightResolve`, `dayResolve`, `gameOverCheck`, and `resetAbilites`.

The method `tick` deals with the game cycle's day/night cycle. An int called `time` can be either 0, 1, or 2. While 1 and 2 correspond to night and day respectively, 0 is a transition

value. At the end of a `cycle`, if `time` is equal to 2, `time` gets set to 0. At the beginning of a `cycle`, `time` is increased by 1 so that the cycle repeats night, day, night day.

The method `updateAlivePlayers` updates the alive players array list by removing dead players from `pListAlive`. This is accomplished by clearing `pListAlive`, iterating through the player array list, and adding a player to `pListAlive` if they are alive. Any players with a false alive boolean are left out of `pListAlive`.

The method `groupVote` is used to determine which player each group will vote to execute.

Each `Player` object contains, among other things, the `act` and `vote` methods. Within `cycle`, if it is night, players will act. If it is day, players will vote. Each player runs `act` and `vote`, regardless if they are alive. If they are dead, they don't actually act and only vote once more when there are three alive players. The `act` method is a method in the `Player` class. If it is night, the `act` method will check if the given player's character name is equal to any of the seven characters. These character names are represented as strings. When a player is assigned a character name, they "become" that character, and that character's code runs when that player acts.

All seven characters have coded abilities, but the Imp's nightly ability is the most important. Every night, the Imp chooses another player to kill. When the Imp "kills" another player, that target player has a boolean called `IMP_killed` that toggles to true. If the target player is a Soldier or is Monk protected, then nothing happens; `IMP_killed` remains false. Both the Soldier and the Monk protect against Imp kills during the night.

Every player contains a set of booleans that allow players to interact with each other. If player A selects player B, then player B will switch their specific boolean to indicate a specific interaction. If a Poisoner chooses a Soldier, the Soldier player's `POISONER_posioned` boolean will switch to true. If the Imp subsequently chooses the Soldier, the `POISONER_poisoned` boolean is checked and if it's true, the Soldier is not protected from the Imp as the Soldier usually would be.

When the `vote` method is run, if it is daytime, the player votes. The voting process uses either a Random selection algorithm, or randomly selects a target player based on which group the voting player is within. These two methods of voting are referred to as Random and Grouped. All alive players vote for another player by changing the target player's `voteCount` variable. If a dead player votes, they lose the ability to do so in the future. Each player has a boolean called `voteToken` that must be true in order for that player to vote. If a player is dead, their `voteToken` changes to false after they vote. Currently, dead players will only vote when there are three alive players during the day. As it is the final day before one team wins, dead players will always use their single-use vote, rather than "waste" it and refrain from voting.

Back in the `cycle` method, the `nightResolve` and `dayResolve` methods run. If it is night, then `nightResolve` iterates through each player. If a player's `IMP_killed` boolean is true, then that player's alive boolean is set to false.

If it is day, `dayResolve` iterates over every player and runs the method `executionCheck` on each player. This method checks if a particular player's `voteCount` is more than the variable `plurality`. In the `init` method, `plurality` is set to an int equal to half of the alive players rounded up. If a player's voteCount equals or exceeds `plurality`, `plurality` is set to the `voteCount` and that player is "put on the block." This means the

player will be executed if no other player gets enough votes. I use a variable called `onTheBlock` that is set to whichever player has a plurality of votes. After iterating through each alive player, the method `execute` is run to kill whichever player has the most votes.

After the night and day resolves, `updateAlivePlayers` runs again to update the alive player array list after executions or night deaths. The method `gameOverCheck` checks if the Imp is killed or if there are only two players alive. If there are two alive players, the `determineWinner` method is run and the `gameOver` boolean is set to true. If there are more than three players alive, `determineWinner` checks if the Imp is currently on the block. This determines if the Imp was executed during that same day, and if they were, then `gameOver` is set to true and the boolean `evilWin` is set to false.

The method `determineWinner` iterates through the alive player array list. If the Imp is present, evilWin is set to true. If `evilWin` is true, then the evil team wins and if false, then the good team wins.

Finally, `cycle` ends with the method `resetAbilities`. Because no character abilities carry over until the next night, every boolean that indicates an ability for every player is reset to false. In addition, `onTheBlock` reverts to its default `Player` object (that has no interaction with any other `Player` objects), and if `time` is equal to two (day), then `time` is set to zero. When `tick` runs at the beginning of `cycle`, `time` will increase to one (night).

When a player interacts with another player, this usually occurs via a method called `randomOtherAlivePlayerExcept`. This method has a `Player` object parameter. The method generates a random int that corresponds to an index in the array list `pAliveList`, but will never generate the index of the parameter player. There are a few different versions of this

method for generating random indices for specific circumstances, but all the versions produce an int that represents an index which represents a player. This was done to avoid object pointer issues.

The method `createNeighborGraph` creates a graph representing the physical location of all the players. Each player neighbors two other players in a circle and so is represented by a vertex with two edges. This graph is used for the Empath's ability to determine how many evil players the Empath neighbors. It is also used for the Chef's ability to determine how many evil players are sitting next to each other.

In `main`, a new Game object is created, `init` is called, `createNeighborGraph` is called, and `cycle` is called. A variable within `Game` called `randomAlg` is set to true or false depending if the voting algorithm should be Random or Grouped, respectively. A total of 1,000 games are run. The final winner, night number, alive player count, and protection value are outputted. The final winner indicates which team won the game. The night number indicates which night the game ended on; if it ended during the day, the subsequent night number is recorded. The alive player count shows how many alive players were alive at the end of the game. If there was a successful protection from an Imp kill, via the Soldier or the Monk, the protection value becomes true and is false if all Imp kills succeeded.

Results

Overview

My program ran 1,000 times, simulating 1,000 games. I tracked four variables for two different algorithms. The first algorithm implements random voting each day. The second

implements grouped voting each day. For each of these algorithms, I tracked the following variables:

Winner

The winning team of each game, ie. Good or Evil.

Number of Nights

The night number when a team won. If a team won during the day, the subsequent night is recorded as that is similar to how a real game works.

Alive Players

The number of alive players remaining when a team wins.

Protection

Whether the Imp tried to kill someone during the night. True means that the Imp failed to kill during at least one night, andfalse means there were no successful protections

Charts

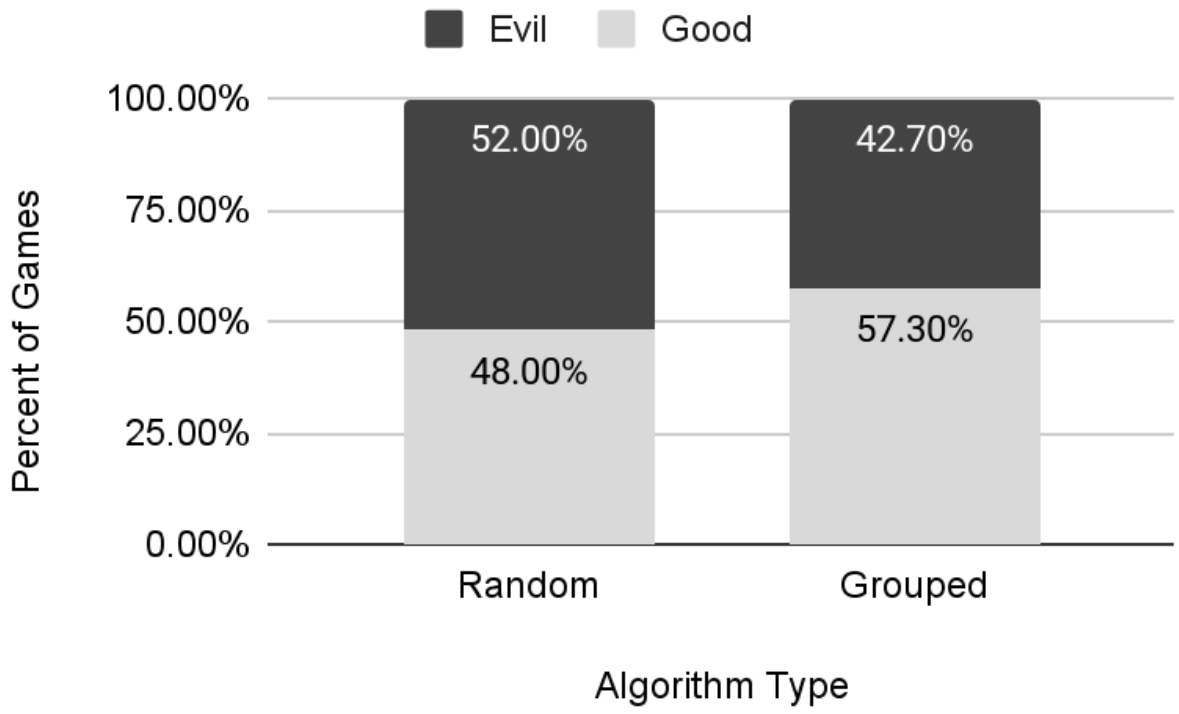

Figure 2: The number of games that the evil team and good team won using both the Random and Grouped algorithms.

The proportion of games in which evil won was compared for the Random and Grouped conditions (Figure 2). A t-test for proportions rejects the null hypothesis that the two are equal (t=4.181; p=3.027e-1)).

As shown in Figure 2 the Random and Grouped algorithm produced a different ratio of good and evil team wins. This could be because the good team's only way to kill the Imp is to execute them. Random voting results in fewer executions in the early game, because it's less likely that a player would get four or more votes. In the Grouped data, there are more consistent executions, and therefore there are more chances for the Imp to be executed. The difference in

execution frequency and consistency might be a cause for the different percentage of good and evil team wins between the two algorithms.

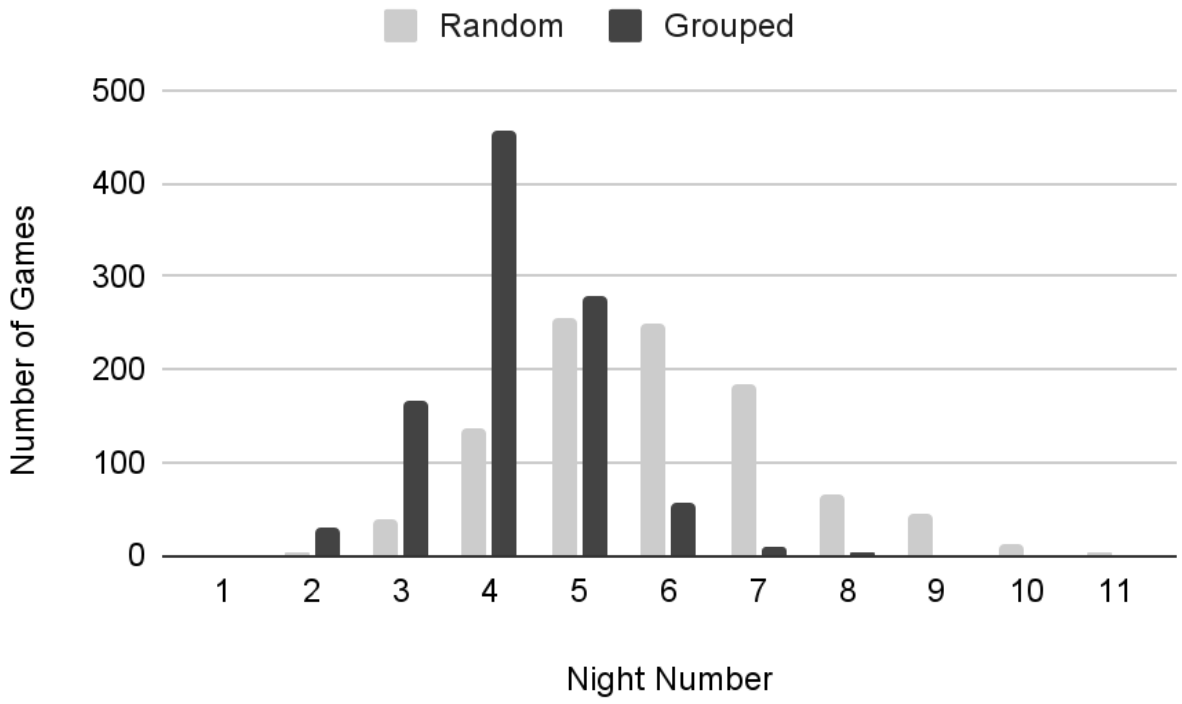

Figure 3: The number of games that ended on a particular night between the Random algorithm and the Grouped algorithm

The number of games in which a certain night was reached by the end of the game was compared for the Random and Grouped conditions (Figure 3). A t-test for proportions rejects the null hypothesis that the two are equal ($t=4.181$; $p=3.027e\text{-}5$). This means that there is a statistically significant difference between the two data sets (Random and Grouped). The Grouped algorithm also has a larger kurtosis than the Random algorithm.

The game ended on the fourth night more often with Grouped voting both compared to the Random games and the other Grouped games. This is probably because of the amount of starting players interacting with the amount of players that die every day and every night. If there

are seven starting players, one is executed during the day and one is killed at night. By day two, there are five players alive. Repeating the cycle yields three players alive on day three. On this day, either the Imp is executed and the night number is equal to four, or night four is reached and the Imp kills someone, ending the game with an evil win. The difference between the two data sets is substantial and indicates that the game ends earlier on average with the Grouped algorithm. This consistent one execution per day and one kill per night formula would result in such a skew for the Grouped data versus the Random. With the Random data, each player is voting randomly. This results in fewer players in the beginning of the game gaining enough votes to reach a plurality. As the plurality needed to execute gets lower, due to the amount of alive players shrinking, the number of successful executions would increase on average. Just missing one or two executions in the early game would cause the difference in distribution that we see in Figure 3.

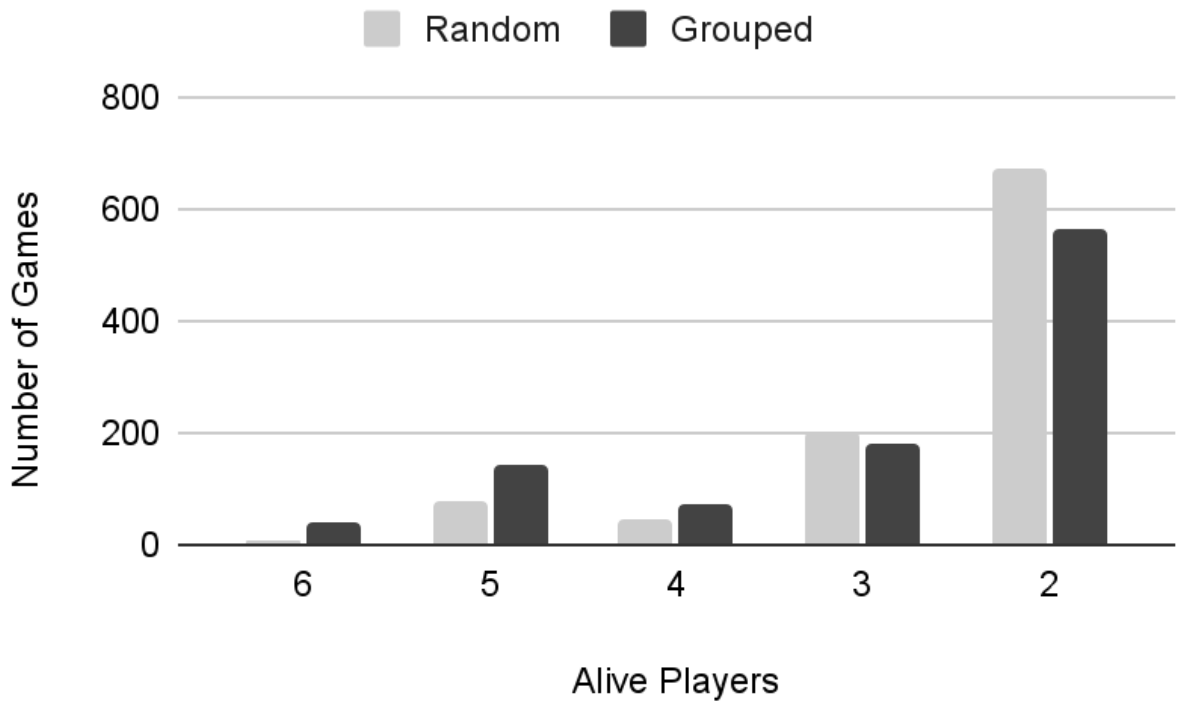

Figure 4: Number of Alive Players at the end of a game using the Random and Grouped algorithms, respectively.

The proportion of games in which evil won was compared for the Random and Grouped conditions (Figure 4). A t-test for proportions rejects the null hypothesis that the two are equal (t=7.264; p=5.353e-13).

The vast majority of games for both Random and Grouped datasets end with two players alive. This mirrors many real games of Clocktower and lines up with the data on who wins each game. Figure 2 shows that each team is within a 20% win rate of the other. In particular, Figure 2 shows that in the Random voting dataset, the evil team wins more than the good team. For the evil team to win, there must be two players alive at the end of the game, and so Figure 4 correlates with Figure 4's percentage of good and evil wins.

Additionally, Figure 4 shows that the Grouped dataset results in more games ending with more than two players than the Random dataset. This correlates with Figure 2 where the Grouped dataset had more games ending at earlier nights than the Random dataset. For a game to end with six players alive, there must be a successful execution on day one, and the execution must succeed on the Imp. The Grouped algorithm results in more consistent day one and two executions which would explain the slightly higher percent of games ending with so many alive players.

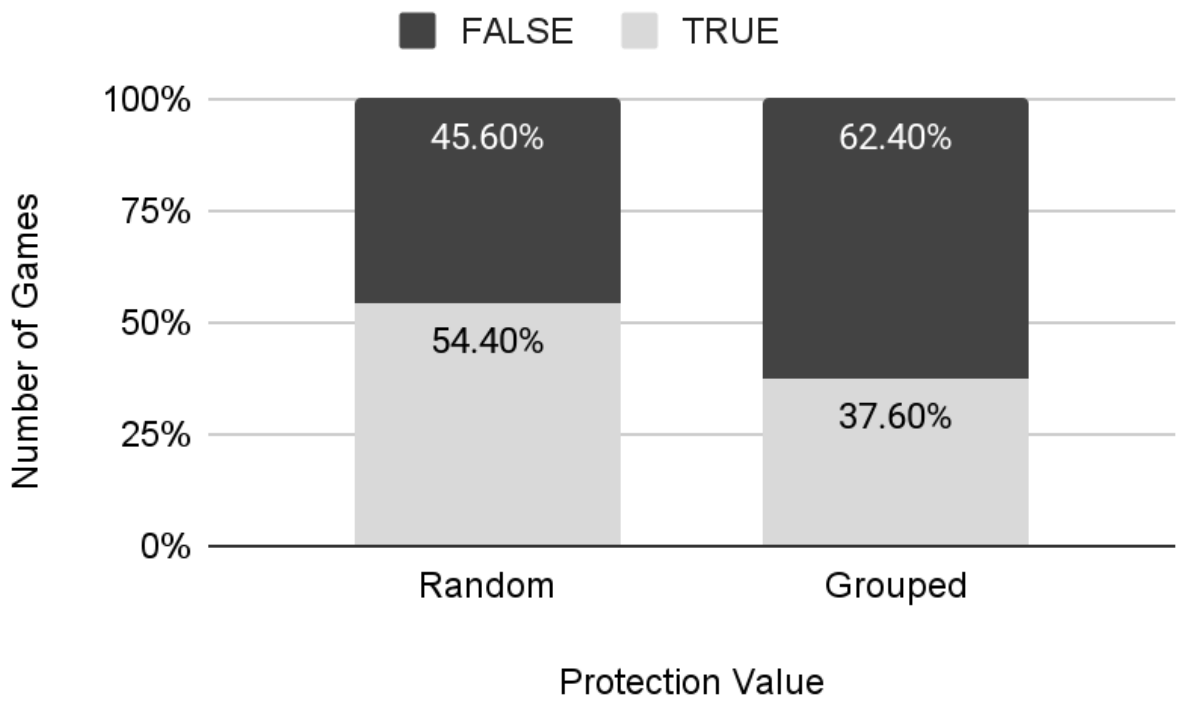

Figure 5: Shows if there was protection that occurred in a game

In Figure 5, the t-test number is *7.642* and the p-value is *3.274e-14*. This means the two data sets were statistically distinct.

The Random and Grouped datasets change how the voting algorithm works during the day. Protection happens at night. In theory, voting and protection have no relationship, however the data heavily suggests the opposite. Changing the voting patterns of the players somehow influences the amount of protection in a game. In the Grouped games, there was significantly less protection. This means that the Imp tried to kill the Soldier less, or that the Monk protected an Imp killed player and so that player didn't die. One reason why this relationship might exist is the higher number of executions in the Grouped dataset. More executions means more opportunities for the good team to kill the Soldier or the Monk, which would hinder protections.

Final Discussion

Statistically, there is a significant difference between the Random and Grouped voting algorithms. This is unsurprising. In a game of only seven players, each death represents significant progress towards the Imp killing most of the town, or the townsfolk executing the Imp. By grouping the players into triplets, it ensures that by day two, the number of votes needed to execute a player is often equal to a single voting group's voting power.

This data is useful, because it demonstrates the significant impact a simple algorithm can have on a social simulation. By changing the voting structure slightly, every recorded aspect of the game was altered. The winningest team was swapped, and the time it took for games to be completed noticeably changed. My program is very limited in scope, but it demonstrates the data that can be obtained from a basic set of variables.

Next Steps

Further Graph Implementation

My entire program could be implemented as a graph. Players would be vertices and edge weights would determine the order of abilities and/or type of relationships. A few graphs with their own unique set of edges would easily represent all the information that is conveyed between players and all the character abilities. Graphs, if created properly, could function with a more modular program. New characters, player counts, and research variables could be added to the program without hard-coded sections needing to be manually changed to accommodate them.

Bayesian Probability

Actions in my program are nearly completely random. The biggest and simplest change to make my model more realistic would be to add Bayesian probability. Players would have probabilities associated with certain actions like voting, targeting, trusting, etc., and these probabilities would change throughout the game. Certain actions would result in changes in these probabilities so that if one player does something that another character finds “untrustworthy,” then that player could update their trust variable accordingly. With this updated trust variable, the probability that that player targets the untrustworthy player might increase, for example. Players could form “beliefs” based on these numbers, and these beliefs would then influence their actions. Bayesian probability would make my model more realistic at simulating more human-like decisions.

A Complete Character Sheet (more characters)

In the Trouble Brewing script, there are twenty-two characters. While I implemented the main and only Demon, the Imp, there are still three other minions, many more townsfolk, and four outsiders that do not exist in my program. Not only would I need to add these to a future version of my program, but I would also need to standardize the character code. I would need to write cleaner code that works in a vacuum. The characters would be implemented as modules so that new characters from other scripts could be added, and they could interact with each other without hard coding their interactions. In addition, each character's ability could be broken down into a set of preexisting rules. By mixing and matching a basic set of these rules, every existing character could be created from scratch, and new or custom characters could be created that have immediate functionality with the rest of the characters.

A Comprehensive Selection Algorithm for Imp and Other Characters

Currently, every character is limited in their selection to a random player that is (usually) not themselves. In a real game of Clocktower, selecting players with their abilities is a huge part of how player's interact with the game and each other. While I'll touch on "reasoning" for the selections separately, at the very least, I could analyze common strategies and simplify and translate them into an algorithm. This would be on a character-by-character basis and so would not be very efficient. Another possibility would be to use strategy archetypes, eg. characters always target players who they don't trust or who they haven't talked to that day. Basic strategy algorithms would greatly increase the simulation's complexity.

Player Personalities: Playstyles and Memory

Players and characters are currently interchangeable in my code. I assign players a certain character, and the combination is locked in for the entire game. In a real game, players and characters are almost wholly separate. Players often play a certain way based on their characters, but pretending to be another character is one of the biggest elements of the deception of this social deception game. Players can be given certain core playstyles. These core behaviors are then mapped onto whichever character they happen to be playing as. This would create a huge number of combinations of player play style and character play style.

Adding memory would also increase the complexity. In a real game, players have conscious and unconscious biases that they build up over time. Some players naturally trust other players for reasons that have nothing to do with the current game. If certain players succeed as evil, other players might be slower to trust them in future games. A memory mechanic could be implemented as a weight that affects certain probability stats. The probability of various actions could be set depending on character type, player play style, and/or other factors like seating arrangement, and memory could be a final weight to influence certain relationships between players.

A Trust Mechanic

While the inclusion of memory between games would be a useful thing to model, trust would be a foundational aspect of the Clocktower model. Trust would be the main statistic behind many of the “decisions” players make. Groups that form via trust will invariably alter the outcome of each game, and so a simulated trust mechanic would be one of the most important inclusions to my model.

Nominating and Voting

The simplest way I can implement better voting would be to have the evil team not vote for each other. In my semi-random game state, this one change would increase the amount of evil wins drastically. The immediate next steps would be to create repercussions for when evil players vote against their groups. When they vote against the grain to save each other, there would be a higher probability that they would be executed next. This would simulate real players taking notice of aberrant voting patterns.

Nominations would be low priority. It would require a substantial coding investment for little practical change. The most important aspect of executions is the player being executed and not the players putting them on the block. However, simulated players who “nominate” (in whatever form) other players could be used in algorithms as targets for future nominations, or to build trust with certain other players.

Bibliography

[1] Burnham, T., McCabe, K., & Smith, V. L. (2000). Friend-or-foe intentionality priming in an extensive form trust game. *Journal of Economic Behavior & Organization*, *43*(1), 57–73. https://doi.org/10.1016/S0167-2681(00)00108-6

[2] Lee, C.-C., & Chang, J.-W. (2013). Does Trust Promote More Teamwork? Modeling Online Game Players' Teamwork Using Team Experience as a Moderator. *Cyberpsychology, Behavior, and Social Networking*, *16*(11), 813–819. https://doi.org/10.1089/cyber.2012.0461

[3] Sarkadi, S., Alison, Rafael, Peter, Simon, & Martin. (2019, October 11). *Modeling deception using theory of mind in multi-agent systems—IOS Press*. https://content.iospress.com/articles/ai-communications/aic190615

[4] The Pandemonium Institute. (2019). *About Us | Blood on the Clocktower—A captivating new social bluffing game*. Retrieved May 3, 2023, from https://bloodontheclocktower.com/about-us

[5] Tilton, S. (2019). Winning Through Deception: A Pedagogical Case Study on Using Social Deception Games to Teach Small Group Communication Theory. *SAGE Open*, *9*(1), 2158244019834370. https://doi.org/10.1177/2158244019834370