Implementation of combinatorial and graph problems as engaging mobile games

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Jyothiradithya
(112001053)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

<u>CERTIFICATE</u>

*This is to certify that the work contained in the project entitled "**Implementation of combinatorial and graph problems as engaging mobile games**" is a bonafide work of **Jyothiradithya** (**Roll No. 112001053**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr Deepak Ranjendraprasad

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

My heartfelt thanks to Dr. Deepak Rajendraprasad for their pivotal guidance and unwavering support throughout this project. His expertise and mentorship have been invaluable, shaping the direction and success of this endeavor. I am deeply grateful for his contributions, which have played a crucial role in this work.

Contents

Chapter 1

Introduction

This projects aims to introduce combinatorial and graph problems to a diverse audience via an engaging video game and collect data from their gameplay so as to arrive at conclusions about how good a media video games can be to aid in the learning of certain mathematical problems for different categories of audience.

1.1 Project Phases

In phase I, the project implements a combinatorial problem that we shall refer to as the fish-barrel problem, which shares considerable similarities to the game of Nim[1]. The game can be played against another human player or against an AI agent. The game is implemented using the Godot engine, and is a catchy game about shooting fish into barrels.

In phase II, the project aims to develop a grid based strategy game, the gameplay of which involves graph theoretic problems. We primarily tackle the Shannon Switching Game, a graph problem involving 2 players who cut or short edges on a graph. Unlike the first game which is built entirely as an implementation of the considered problem, this game is independent of the graph problem[s] considers, the graph problem instead playing

[1]https://en.wikipedia.org/wiki/Nim

a part as the strategy that the AI opponent has to use to win the game. The graph used for the problem is not inherently present in the game and instead is extracted from the grid using various methods.

Part I

The First Game

Chapter 2

The Fish-Barrel Game

2.1 The problem

Let us define the one person version of the fish-barrel game. It involves N barrels numbered from 1 to N, with the kth barrel having n_k ($n_k \in \mathbb{N}$) fishes. In one move, the player has to take out a single fish from the ith barrel for an i of their choice. They can then, in the same move, add any finite number of fishes to any of the barrels from $i+1$ to N. The game ends when there is no fish in any barrel.

On first look at the problem, one may be tempted to believe that a game of fish-barrel can last forever, since one only needs to take out one fish while they can add back any number of fishes. However, this is not true. The crux of the matter lies in the fact that if you consider any subset of barrels numbered i to N, while the other barrels may keep filling up, the ith barrel necessarily has to get exhausted in some finite number of moves.

Theorem 1 *The one player fish-barrel game is guaranteed to end in a finite number of turns.*

Proof Proof by nested induction [1]

P(1) : The game is guaranteed to end in a finite number of turns if $n_N > 0$ and $n_i = 0 \; \forall \; N > i \geq 0$ (ie there is fish in the Nth barrel and no other)

Since there is no barrel numbered greater than N, in each move the player has to take out exactly 1 fish from the Nth barrel and add to no barrels. Thus the game ends in n_N moves. Thus P(1) is true.

P(N-k) : The game is guaranteed to end in a finite number of turns if $n_k > 0$ and $n_i = 0 \; \forall \; k > i \geq 0$

P(N-k+1) : The game is guaranteed to end in a finite number of turns if $n_{(k-1)} > 0$ and $n_i = 0 \; \forall \; (k-1) > i \geq 0$

Claim 2 *P(i) is true* $\forall \; N \geq i \geq (N-k) \implies$ *P(N-k+1) is true*

Proof

Assume P(i) is true $\forall \; N \geq i \geq (N-k)$

P'(m) : P(N-k+1) is true if $n_{(k-1)}$ = m

P'(1) : P(N-k+1) is true if $n_{(k-1)}$ = 1

From assumption, all barrels numbered N-k to N will necessarily be emptied in a finite number of moves. Thus the player will have to take a fish out of the (k-1)th barrel within a finite number of moves. Once the fish is taken out of the (k-1)th barrel, the problem instance not has fishes only in barrels k and above. Thus, it has to end in a finite number of moves.

Assuming P'(m) is true

P'(m+1) : P(N-k+1) is true if $n_{(k-1)}$ = m+1

From assumption, all barrels numbered N-k to N will necessarily be emptied in a finite number of moves. Thus the player will have to take a fish out of the (k-1)th barrel within a finite number of moves, causing it to have m fishes. The current instance of the problem will end in a finite number moves as P'(m) is true. Thus P'(m+1) is true.

Thus by induction, P'(m) is true $\forall m \in \mathbb{N}^+$. Thus claim 2 is true ◇

Thus by induction, P(i) is true $\forall N \geq i \geq 1$ Thus, the one player fish-barrel game is guaranteed to end in a finite number of turns for any finite number of barrels.

$\square$

We define the 2 player fish-barrel game, where the rules per one move are the same. Additionally, each player takes turns choosing a barrel and removing and adding fishes following the rules mentioned before. The player after whose turn all barrels are empty is the winner of the game.

Corollary 3 *The 2 player fish-barrel game will have a definite winner in a finite number of moves.*

2.2 Strategy

Let us make 2 observations about the game. Firstly, it can be seen that to play the game optimally will involve paying attention the parity of each barrel. (Whether this fact is apparent simply from playing the game against an optimal opponent is in fact something I wish to explore further down the line in this project).
Secondly, it is the lowest indexed barrel that plays the biggest role in deciding the winner. All the other barrels become important only when they become the lowest indexed barrel.

Claim 4 *A player who is able to make a move such that after it all barrels have an even number of fishes is guaranteed to be able to win.*

Proof Proof by nested induction

$P(1)$: The player who is able to make a move such that there is an even number of fishes in every barrel after his move when $n_N > 0$ and $n_i = 0 \; \forall \; N > i \geq 0$ (ie there is fish in the Nth barrel and no other) is guaranteed to be able to win.
If the barrel has 1 fish before the player's turn, maintaining the even invariant would imply taking out the last fish thereby winning the game.

If there is an odd number of fishes in the barrel, the player can then take fish from the barrel leaving it with an even number of fishes. This guarantees that after the next player's turn they will leave an odd number of fishes behind. The same process can be repeated until there is only 1 fish in the barrel which the player can remove to obtain victory.
If there is an even number of fishes in the barrel for a player he will have to take one out making it odd guaranteeing the victory for the other player in the manner described above.

$P(N-k)$: The player who is able to make a move such that there is an even number of fishes in every barrel after his move when $n_k > 0$ and $n_i = 0 \ \forall\ k > i \geq 0$ is guaranteed to be able to win.

Assuming $P(N-k)$ is true,

$P(N-k+1)$: The player who is able to make a move such that there is an even number of fishes in every barrel after his move when $n_{(k-1)} > 0$ and $n_i = 0 \ \forall\ (k-1) > i \geq 0$ is guaranteed to be able to win.

To note here is that since each player can add any number of fishes to all the barrels greater in index than the one they are choosing, it is trivial to control the parity of such barrels.

Let us do induction on $n_{(k-1)}$.

$P'(1)$: Consider the case where $n_{(k-1)} = 1$. The player who makes the next move can remove the last fish from this barrel and ensure that all the remaining barrels have an even number of fishes in them. The player that follows is unable to make a move such that all barrels have an even number of fishes after their move since to do so, they have to have a at least one barrel with an odd number of fishes from so as to remove a fish from it.
Consider that After their move, barrels numbered $\{o_1, o_2 ... o_m\} = O$ has odd number of fishes in them. Now, in the first player's turn, he can choose the lowest numbered barrel among the odd barrels to remove a fish from, and adjust the others accordingly to have even number of fishes after his turn, thereby ensuring that all barrels have an even number of fishes after his turn. Now by $P(N-k)$, this player is guaranteed to be able to win.

$P'(m-1)$: Assume $P(N-k+1)$ is true for all $n_{(k-1)}$ upto some $m-1$. In addition, we also consider true the following 2 statements for all $n_{(k-1)}$ upto some $m-1$:

1. If atleast 1 barrel has an odd number of fishes at the start of a player's turn, this player is guaranteed to be able to win. (Provable similar to the statements in $P'(1)$)

2. If all barrels have an even number of fishes at the start of a player's turn, the other player is guaranteed to be able to win. (Observable from how any move this player makes will result in a situation akin to the previous statement)

$P'(m)$: Consider $n_{(k-1)} = m$.

If m is odd. The current player can take a fish from the $(k-1)$th barrel and adjust the remaining barrels so that there is an even number of fishes in all barrels. By $P'(m-1)$, this player is guaranteed to be able to win.

If m is even, but there is some higher numbered barrel with an odd number of fishes, this player can take out a fish from the lowest odd barrel and adjust the other barrels so that all barrels have an even number of fishes. By Corollary 3, we can see that all barrels numbered k to N will necessarily empty out in a finite number of moves. Furthermore, by $P(N-k)$, we can see that the move after all these barrels are emptied out will necessarily be of the second player (Since the first player will be victorious in a 'sub-game' with just those barrels). Thus we can conclude that after some finite number of moves, the second player will necessarily remove a fish from the $(k-1)$th barrel. $n_{k-1} = m-1$ will be odd. Thus by statement 2.2 of $P'(m-1)$, this player is guaranteed to be able to win.

If m is even and so is every other barrel, the current player is forced to perform a move such that there is atleast one barrel with an odd number of fishes. This falls into one of the previously mentioned cases and will guarantee victory for the other player.

Thus by induction, $P'(m)$ is true for all integers m.

Thus by induction, $P(i)$ is true $\forall N \geq i \geq 1$. Thus the claim is true. □

From this claim and its proof, we can arrive at the most optimal way to play the game.

Observation 1 *The most optimal move for any player at any point in the game is to maintain the invariant that all barrels have an even number of fishes after their turn.*

The only case where an optimal move cannot be made is if at the start of this turn, all barrels have an even number of fishes.

Another interesting observation to be made is the fact that the result state of a game (provided all players play logically) can be predicted from the initial state of the game.

Observation 2 *If all barrels have an even number of fishes at the start of the game, the second player is guaranteed to be able to win. Else the first player is guaranteed to be able to win*

Chapter 3

Implementation

All code and implementation has been uploaded and is being currently maintained in my git repository [2].

3.1 CLI version

A command line version of the game is implemented using python. (At `/python-code` in the repository)

3.1.1 Design

The CLI version is played simply by entering the move as a sequence of integers, the first one being the barrel that the current player chooses to take a fish from, and each subsequent integer being the number of fishes they wish to add to each subsequent barrel.
The current state of the problem is printed after each successful move. If a move made by the player is illegal, it is mentioned and the state of the game remains the same.
The game can be played either as Player vs Player or Player vs AI. Different AI agents have been implemented.

3.1.2 AI Agents

In the implementation is 2 agents which can be used as AI opponents, namely the `EvenAgent` and `FishProblemMinMaxAgent`.

EvenAgent

The EvenAgent follows the optimal move strategy 2.2 to calculate the optimal move for a state of the problem. Implemented as follows.

```
Algorithm 1: Optimal Fish Allocation Algorithm
  Input  : List of barrels with the number of fishes in each
  Output: The chosen barrel and the list of barrels to add fish to
1  mode ← choosing;
2  chosenBarrel ← None;
3  addFishToBarrels ← Empty list;
4  for currentBarrel in list of barrels do
5      numFishes ← number of fishes in currentBarrel;
6      if numFishes is odd then
7          if mode is choosing then
8              chosenBarrel ← currentBarrel;
9              mode ← adding;
10         else if mode is adding then
11             addFishToBarrels ← Append currentBarrel to addFishToBarrels;
12 return chosenBarrel, addFishToBarrels
```

Note that the condition where all barrels are even during the start of the agent's turn is not handled here. This is because that is the only case in which the agent is unable to make the optimal move.

FishProblemMinMaxAgent

The Min Max agent tries to solve the problem by running a min max search algorithm on a state space encoding some features of the game.

A direct state space encoding would be to simply take $(n_1, n_2....n_N)$ (ordered collection of number of fishes in the barrels) as the label of a state. By capping the total number of fishes per barrel to 10, the reachable states from any initial state of the problem can be proved to be finite. The number of states in this state space is upper bound by 10^N which is a fairly large overhead for an agent that is suppose to calculate it's next move almost instantly.

There are plenty of ways to reduce this computation time. One option would be to encode the state space to have some quantification of parity instead of all possible numbers. This approach was abandoned as it brings the agent very close to combinatorial solution, offering us no new insight on problem.
Another option would be to precompute some part of the algorithm or assign weights.

No further exploration was done in this direction.

3.1.3 Code structure

The code involves 2 classes, `FishGame` and `FishProblem` which simulate the behaviour of a 2 player fish-barrel game and maintain consistency in the simulation of the problem respectively.

The class `Agent` implements the interface for an agent that can play the game like a human player. Classes deriving from it implement different AI agents, namely `EvenAgent` and `FishProblemMinMaxAgent`.

There are also support classes for defining Min-Max agents and state spaces.

3.2 Godot Version

A demonstration version of the game is created using the Godot 4 game engine[3]. (which is open source, and very light weight!). (It is present at `/Fish-in-the-barrel` in the repository). It is a 2D game which can be ported to a browser and can be played in any device (PC, Mobile etc.)

3.2.1 Design

The purpose of this version of the game is to analyse the interaction that a layman (and perhaps also BTech students) have with the combinatorial problem at the core the game. Thus it is essential that the game is engaging to a fairly wide audience, while still expressing the essence of the problem at hand.

Main Loop

The game is centered around an engaging *central game mechanic* which is the action of shooting an object at targets. In this case it is throwing fish into barrels. As the game is targeted towards mobile users, the central mechanic is implemented as a *slingshot* such as the one seen in popular mobile games such as *Angry Birds* [1].
The player has to perform only 3 major actions - Select a barrel to take a fish out of, throw fishes into the now opened barrels and confirm when their move is done. Additional constraints such as the maximum number of fishes that can be thrown per move and total number of fishes per barrel can also be enforced.

AI and difficulty settings

While playing against an AI opponent, the AI opponent will also follow the same rules, choosing a barrel and throwing fishes into the rest in a similar manner (at least visually) to the player.
Note that there are infact 2 AI agents required to create the AI opponent, one to calculate the optimal move, and the other to calculate the throw required to land a fish in the required barrel.

Additionally there will also be a difficulty settings for the AI opponent. Decreasing difficulty will cause some randomness to be introduced into the functioning of both AI

[1]Angry Birds is a mobile (and PC) game that involves shooting birds out of a slingshot to destroy structures. https://www.angrybirds.com/

agents, ie. an easier opponent may make the wrong choice of moves more often, and may also miss more often. The exact difficulty options available and the effect it has on either agent has to be carefully fine tuned so as to both feel fair for the players and also aid in data collection.

3.2.2 Implementation

FishProblem

The class representing a fish-barrel problem is created in a very similar manner as it was in the python version.

Barrels

Each barrel is also represented by a class which implements a few simple function to open and close the lid, as well as to return the number of fishes currently present in the barrel. The actual barrel is implemented in Godot as a collection of StaticBodies [2] which are arranged to form the walls and lid of the barrel, and an Area which is used to gather information about the number of fishes inside the barrel.

While not the core mechanic, the action of touching a barrel to choose it to remove a fish from is also an important part of the game. Thus it is necessary to make this action have strong visual presence. A simple change that occurs when a player chooses a barrel is that all barrels numbered greater than it opens while all barrels numbered less than or equal to it closes, signifying which barrels the player can add fishes to. To further emphasize on the visual cues, the barrel that the player chooses may shake, causing a fish to drop from it. Atleast partly justifying the removal of a fish from this barrel.

[2]StaticBodies are entities that have physics implemented in such a way that they can collide with any other body, but do not change their position, velocity etc. unless made to do so via code

Barrel Manager

The Barrel Manager is an entity which manages the updation between a collection of barrels as entities in the game scene and an object of the FishProblem class. It ensure consistency in the gameplay. For example, it throws errors if the number of fishes in a non-chosen barrel decreases during a turn. It may be interesting to note that such events are not uncommon in the initial versions of the game due to misbehaviour of the physics engine, and may be usually handled by tweaking the behaviour of each physics object, or coding in behaviour into objects that handle these edge cases.

Slingshot

The slingshot is a centre piece to the core game mechanic. It is the point from which the player can throw or ”shoot” a fish into the barrels. In appearance, it could be a literal slingshot complete with pullable straps, or could be a person throwing the fish using his hands, or any other similar entity depending on the theme of the game/level. For the demonstration purposes, it is a slingshot.

Functionally, the core mechanic should be extremely intuitive. The player should have it figured out within minutes of playing the game, so that they can actively focus on thinking about other parts of the game. At the same time, it should be fun and engaging so that repeating it hundreds of times doesn't tire the player out. Here, the mechanism of throwing a fish using the slingshot involves pulling at the straps of the slingshot using your fingers (or mouse if playing from a PC). The path the fish is expected to follow after releasing the slingshot is highlighted on screen so as to aid the player in aiming.

Game Manager

There exists an entity which manages the current state of the game, storing information about who is currently playing, and switching control from the player if the next move is suppose to be performed by an AI. It also handles engine specific functionalities such as exiting the game to main menu once the player wins or loses.

It also calls each AI agent to do its respective work. It calls the Fish Problem Agent to calculate the move of the AI opponent, then uses the slingshot AI to shoot fishes into the right barrels.

An interesting point to note is that it sprinkles in small pauses between each of its actions so as to appear like it is being played by a human.

Slingshot AI Agent

AI agent that handles the calculations required to throw a fish into the required barrel. The calculations are done using the equations of motion for an object in the 2D plane under gravity.

Fish Problem Agent

Identical to the implementation in the python version, the agent simply outputs the optimal move at any state of the fish barrel problem. More agents with varying levels of difficulty/randomness can be derived from this agent class.

3.2.3 Difficulty implementation

Associated with each agent, namely the Slingshot AI agent and the Fish Problem Agent, are unsure versions of them which don't always make optimal moves. These have a configurable output probability of making the right move during their turn, giving us an options adjust the difficulty of the agent using a slider, such that agents which have a higher difficulty have a higher probability of making the optimal move at each turn.

Wobbly Slingshot AI

A class that extents from the Slingshot AI Agent class, the wobbly slingshot AI has a fixed probability of missing each shot it takes. This is implemented as the target of the agent being a random point in a circle of a given radius (which in 2D is just a point on a line segment) centered at the center of the opening of the barrel it is aiming at. How large the target area is compared to the lid opening area is represented by a parameter γ, which is in turn also equal to the probability that the agent will successfully land a shot.

Let γ be the probability that the agent will successfully land a given shot. Then

$$\gamma = \frac{\text{Area of barrel opening}}{\text{Target area}} \tag{3.1}$$

Let P_{shot} be the probability that all shots in the given turn is successful.

$$\begin{aligned} P_{shot} &= \gamma^n \\ \gamma &= P_{shot}^{1/n} \end{aligned} \tag{3.2}$$

Where n is the number of shots that have to be made in the given move. n varies widely from move to move and can be anywhere between 0 and N-1. Currently, an oversimplified model is implemented that simply takes n to be $N/2$, N being the total number of barrels. This is based on the observation that during the optimal move, at most 1 fish is required to be put into each barrel. A simplifying assumption made is that on average, around half the barrels may be required to be shot to. While this is not fully correct, it gives us an agent to work with.

Further down the line, this agent is not used extensively and instead γ is kept fairly high for all difficulty levels due to 3 reasons:

1. The video game engine already introduces a certain amount of uncertainty into the shots by virtue of the physics engine, which is not perfectly deterministic. Small things such as the lid alignment, the rotation of the fish when it hits a point on

the barrel, fishes colliding with each other etc. all cause the result of each shot to be uncertain to extends that cannot be easily captured as probability. A possible solution to this will be to constraint the physics engine. Such a task is too elaborate for the problem at hand and will also likely make it harder to add fun elements into the game.

2. If the target area is too big, it might accidentaly cause the fish to land in a different barrel. This is undesirable because the final move the agent makes will be different from the final move the fish problem agent makes, in not only that some shots may be missed, but some fishes may be added to other barrels.

3. The focus of the game is on finding the optimal strategy for winning. An agent that is on the easier side of the difficulty scale must be one that does not have a good understanding of the optimal strategy, not one that may know the strategy well but fails to land shots properly. Furthermore, as we explore hypotheses, we will see that we expect the player to learn more about the problem by playing against a harder agent, by virtue of the agent performing moves which are closer to the optimal strategy. The uncertainty brought about by the slingshot agent will only impede our measure of the closeness the agents strategy has to the optimal strategy.

Thus, the difficulty element is mainly characterised by the agent derived off of the Fish Problem agent instead of the Slingshot AI agent. Only at the easiest difficulty does the agent make shots that miss the barrel.

Unsure Agent

The unsure agent is a class that derives out of the Fish Problem Agent. The agent can be configured to have a certain probability of providing the correct move for a given state. The internal states of the agent include a variable α which is the probability that for each addition into any barrel in this move, the correct parity is maintained. Additionally, the

the probability of choosing the right barrel to remove a fish from is given by α^i where i is the correct barrel to choose for this move.

Let X be the random variable representing the barrel chosen to take a fish from. Let i^* be the barrel to be chosen for the optimal move. Let B' be the set of barrels b_i that have atleast 1 fish in them (and so is a candidate for removing a fish from). $B' \subset B$, where B is the set of all N barrels. Then the probability distribution for choosing a barrel at the start of the move is given by

$$P_{choose}(X = i) = \begin{cases} \alpha^i \text{ if } i = i^* \\ \frac{1-\alpha^{i^*}}{|B'|} \text{ if } b_i \in B' \\ 0 \text{ otherwise} \end{cases} \tag{3.3}$$

To note is:

$$\sum_{i=0}^{N} P_{choose}(X = i) = \alpha^{i^*} + \sum_{b_i \in B'} \frac{1-\alpha^{i^*}}{|B'|} = 1$$

Once a barrel is chosen, for each barrel to add a fish to, the agent makes the right move (ie. maintains the right parity) for this barrel with a probability of α. Note that the agent may add 0, 1, 2 or 3 fishes, as per required parity. The actual number of fishes is irrelevant and only the parity is analyzed here. Thus the probability that the move is correct for barrel b_i is given by $P_i = \alpha$. Probability of all such throws being current is given as below

$$P_{add,i^*} = \prod_{i=i^*+1}^{N} P_i = \prod_{i=i^*+1}^{N} \alpha = \alpha^{N-i^*} \tag{3.4}$$

Thus, the success probability, ie. probability that this agent will choose the right barrel to remove a fish from and correctly preserve the parity of every subsequent barrel is as below

$$P_{correct} = P_{choose}(X = i^*) \times P_{add,i^*} = \alpha^{i^*} \times \alpha^{N-i^*} = \alpha^N$$

Thus, given the final probability, we can configure the agent to have α as follows:

$$\alpha = P_{correct}^{1/N} \tag{3.5}$$

Following is the top level pseudo-code of the algorithm used by the unsure agent, assuming α is already preconfigured according to required difficulty.

1. Find the barrel to be chosen to take a fish from. Let it be i^*. Also find all barrels that have at least 1 fish. Let the total number of such barrels be $n' + 1$.

2. Construct a probability distribution such that the barrel i^* gets a probability of α^{i^*}. All other barrels that have at least 1 fish gets a probability $\frac{1-\alpha^{i^*}}{n'}$. Sample from this distribution to get the chosen barrel i'.

3. For each barrel from $i' + 1$ to N, find the parity and the correct move to maintain the parity invariant. Then sample from a distribution of probabilities (α, $1 - \alpha$) and assigned values (true, false) (ie. toss a coin with α probability of head). If the sample value is true, the parity invariant is maintained for the barrel (by throwing 0 or 2 fishes into the barrel if it is even, or 1 or 3 fishes if it is odd). Else it is not (in a similar fashion as before).

4. return the move obtained from this process.

This method has a few corner cases where the algorithm or probability conditions fail. For example, if there is only 1 barrel with any fish in it, the algorithm fails as the probability distribution for choosing fishes is not normalised. The consistency of probability is also not maintained as for this case the probability of making the correct move is either 1 (if the barrel has an odd number of fishes before this turn) or 0 (the barrel has an even number of fishes before this turn).

In the algorithm, this is handled by normalizing the probability distribution. The inconsistency in probability is for now not handled as it does not cause a very large functional difference to gameplay.

Feel of the difficulty

While the probability of an agent making the right move can be configured into the agent using functionality such as the ones implemented in the wobble agent and unsure agent, the actual difficulty that the players may face is dependant on many more factors. For example, if an agent is implemented which will choose the correct move with a probability of $P_{correct}$. But this is implemented as only the very last choice the agent makes having a probability of being correct equal to $P_{correct}$, all other moves being correct with 100% probability. In this case, the probability itself would not have an extremely large bearing on how much difficulty the player faces trying to win against this agent, since all moves this agent makes are extremely close to the optimal move.

The unsure agent (3.2.3) appears to have a fairly good scaling of difficulty as an agent with lower probability of success has both a lower probability of choosing the correct barrel and also has a lower possibility of choosing how many fishes to add to each barrel. One interesting observation is that if the max barrel is one with a lower index, the agent has a higher chance of choosing it than if it is of higher index. This can be interpreted as the agent *leaving more options open* in the sense that a lower indexed barrel allows the agent to modify more higher indexed barrels in its turn.
However to actually test the feel of the difficulty brought about by such a system, a pilot or testing with a few individuals is required.

Chapter 4

Data Collection and Hypothesis testing

4.1 Data Collection Framework

A bare framework for data collection is currently implemented using HTTP requests in Godot. A server to catch and maintain this data in JSON format is also created. (code can be found at `/server` in the repository). Currently, the framework logs event occuring in game, such as each move, the state of the game before and after the move and if a player or an AI wins a game.

The framework can be made to capture particular data from the game, such as summaries of all runs made by a particular user, or number of times the player maintained an invariant in a run of the game. What data is to be collected and in what format depends largely on the hypothesis we are trying to test.

4.2 Hypotheses

4.2.1 Learning Rate against varying difficulty

Hypothesis

A better understanding of the optimal strategy is developed by a playing against a harder opponent than an easier one, given the same number of games.

Formalization

Formally, let us define a quantity we shall call as *Understanding score* U which represents a measure of the understanding a player has of the optimal strategy. We define the understanding score a game G for player P as follows: $U(G, P) =$ number of times the invariant was maintained at the end of player P's turn / total number of turns of player P in game G. Furthermore, we define an *understanding threshold* U_0 which represents a sufficient level of understanding for comparison.

Thus the hypothesis is formalised as follows:
For N games, If $\frac{1}{N}\sum_{i=1}^{N} U(G_i, P_h) \geq U_0$ and $\frac{1}{N}\sum_{i=1}^{N} U(G_i, P_e) < U_0$, where G_i represents individual games, P_h represents the player playing against the harder opponent, and P_e represents the player playing against the easier opponent, then it can be concluded that a fewer number of games against a harder opponent is sufficient to develop an understanding of the optimal strategy compared to playing more games against an easier opponent.

Data collection

For testing this hypothesis, per user per game understanding scores are required. these can be obtained simply by calculating them from the data about the runs of the game.
A manual survey enquiring about the players opinion of the optimal strategy can also be used to support evidence for or against the hypothesis.

4.2.2 Hosting

The game, as well as the data collection server are hosted on an AWS EC2 instance, using Apache2.

The game is exported as a Web game via Godot Export Manager, so that it can be run on a browser. It is to be noted that to run a Godot game in a browser, certain resources are required, which are requested by the headers from the server.

```
Header set Cross-Origin-Opener-Policy "same-origin"
Header set Cross-Origin-Embedder-Policy "require-corp"
Header set Access-Control-Allow-Origin "*"
```

However, these resources will only be granted by the browser if the website is trusted, by virtue of it having an SSL certificate and being an HTTPS page. To circumvent this matter, we use a self generated SSL certificate. The user will simply have to confirm to the browser that the certificate is trustable.

The data collection server is hosted in the same server, in the HTTP port.

Part II

The Second Game

Chapter 5

The Shannon Switching game

Unlike the approach taken with the previous game in which a combinatorial problem was directly implemented as a video game (heavily limiting the game design), this part of the project aims to implement a video game such that one or more graph problems are intrinsically linked to the functioning of the game, but doesn't define the design of the game.

To achieve this, we implement a strategy game that takes place on a square grid. Taking inspiration from popular strategy games such as Age Of Empires [1], Sid Meier's Civilization series [2], Clash of Clans [3] etc, the game consists involve 2 (or more players) who either take turns or play simultaneously to construct different structures and control troops to destroy major structures of the other players.

[1] A series of real time strategy games where all players simultaneously build civilizations and commands troops to war on an open grid

[2] A series of turn based strategy games where players take turns making moves against each other on a hexagonal grid simulating civilisations and war strategies

[3] A very popular tower defense game, where one player sets up their 'base' for defense and another sends troops to attack it

5.1 The problem

We start off by considering the Shannon Switching game. It is a turn based 2 player game involving an undirected, unweighted multigraph and 2 marked vertices A and B. One of the players, called **Cut** can, in their turn, remove an edge from the graph. The other player, called **Short** can short an edge on the graph, which effectively means that it can now not be cut by Cut. Short wins if they can guarantee a path from A to B that cannot be cut by Cut. Cut wins if they manage to prevent that, ie. disconnect A and B.

5.1.1 Optimal Strategy

Referring to Mansfield's paper on strategies for Shannon Switching game[4] and 2 of my seniors' interpretation of it [5], we come up with a procedure to find the optimal strategy for short to win the game.

Let us consider the 2 player Shannon switching game on an undirected, unweighted multigraph G with 2 marked vertices A and B, called the source and target respectively. Furthermore, we consider the act of shorting an edge between vertices u and v as removing all edges between u and v, and merging vertices u and v (ie. all edges that have an endpoint in u or v will now have that endpoint in the merged vertex instead)

Theorem 5 *Short has a winning strategy (regardless of whether they play first or not) if G has a subgraph containing A and B that has 2 edge disjoint spanning trees.*

Proof If the next move is of the Cut player, and they cut an edge that is in one of these spanning trees, this will cause this spanning tree to be separated into 2 disjoint components. The Short player finds an edge in the other spanning tree which joins these 2 components (this edge is guaranteed to exists as spanning trees have to be connected) and shorts it. In the graph obtained after shorting, we once again obtain a subgraph with 2 edge disjoint spanning trees.

If the next move is of the Short player, they can short any edge in either of the spanning trees.

It can be seen that the Short player maintains an invariant that there exists a subgraph to the main graph that has 2 edge disjoint spanning trees.

The game will continue until A and B themselves are shorted, guaranteeing an un-cuttable path from A to B. □

We state without proof that the opposite is also true, ie. G has a subgraph containing A and B that has 2 edge disjoint spanning trees if Short has a winning strategy (regardless of whether they play first or not).

Dual

Let us constrain our Shannon Switching game to be played on a planar grid. This is in line with our problem as the kind of graphs we will deal with can all be drawn on a grid and therefore must be planar (we will look into the details of this in the next chapter). Furthermore, we also add an extra constraint that it should be possible to draw an edge between A and B without making the graph non-planar. In grid terms, this is like constraining A and B to opposite ends of the grid.

Now we construct a dual for the graph as follows:

1. Add a transient edge e_t between A and B, while preserving the planarity of the graph, to get G_t
2. Construct the dual of G_t^d (by converting faces of G_t to vertices and adding edge e^d to pair of vertices that represent the pair of faces that are separated by edge e in G) [6]
3. Obtain the final dual graph G^d by removing the equivalent of the transient edge e_t, e_t^d in G_t^d

Let us also label the vertices on either ends of e_t^d in G_t^d as A^d and B^d (order does not matter). A^d and B^d also exist as is in G^d.

Theorem 6 *Cut has a winning strategy (regardless of whether they play first or not) in a Shannon switching game over graph G if the dual G^d of G (constructed using the procedure described in 5.1.1) has a subgraph containing A^d and B^d that has 2 edge disjoint spanning trees.*

Proof The set of edges of G^d has an equivalence relation with the set of edges of G (Simply matching e^d to e). Thus, a game of Shannon switching game on G has an equivalent game on G^d, where if edge e is cut in G, we short e^d in G^d, and if an edge e is short in G, we cut e^d in G^d. A^d and B^d are source and target nodes in the Shannon switching game over G^d. The Cut player in the original game plays as the Short player in the dual game and vice versa. Let us call such a game the dual game of the original Shannon Switching game on graph G.

Since G^d has a subgraph containing the source and target which has 2 edge disjoint spanning trees, by 5, the game will continue until A^d and B^d are shorted, at which point, there will be a path between A^d and B^d, all edges in which has been shorted.

Now we claim that this state is a winning state for the cut player in the original graph. To prove this claim, consider the shorted path, which we shall call P^d, in the transient dual graph G_t^d. P^d along with e_t^d forms a cycle in G_t^d. A cycle in a dual graph is a cut in the primal graph. Thus, P, the set of corresponding edges to P^d in G_t, along with e_t, forms a cut in G_t. This cut is between a component having A and one having B (since A and B are on either sides of e_t). P is also a cut separating A and B in G since e_t is an edge between A and B. Also note that all edges shorted in G^d in the dual game is an edge cut in G in the primal game. Thus, we can see that the cut player has cut enough edges to separate A and B into different components thereby preventing the Short player from finding a path between them and thus winning the game. □

From 5 and 6, we can find the optimal strategies for Short and Cut players.

Observation 3 *The optimal strategy for the Short player is to try to find 2 edge disjoint spanning trees in G, and to maintain their existence as an invariant if they exist, by following any move of the Cut player that separates any of these spanning trees into 2 components by shorting a corresponding edge in the other spanning tree that joins the components back together.*

The optimal strategy for the Cut player is to do the same with the dual of this game.

Finding the spanning trees

We shall use an algorithm that is a modified version of the one described by Roskind and Tarjan [7], the modification is a simplification to accommodate that we only need to construct 2 edge disjoint spanning trees (instead of some k). The algorithm is similar to Kruskal's algorithm [8] where 2 forests are maintained instead of 1. A forest is maintained as a graph and as a DSU, the DSU lets us check easily whether adding an edge causes a cycle to form.

We consider each edge in the main graph (greedily if weighted), and follows a policy to find which forest to add this edge to. Thus, we have 2 cases depending on the result of adding this edge to either forest.

Case 1: There is at least one forest, adding this edge to which does not form a cycle. In this case we can simply add this edge to this forest.

Case 2: Adding this edge to either forest causes a cycle to form in the corresponding forest. In this case we adopt an augmentation procedure described below.

We first construct an auxiliary directed graph over the edges of the original graph. Consider 2 edges u and v in the original graph. They have an edge in the auxiliary graph, directed from u to v if they satisfy the following properties:

1. u and v are in different forests, or u has not been added to any forest while v has.

2. If u belongs to forest A and v belongs to forest B (A and B are different as per the

previous point), removing u from A and adding it to B causes a cycle to form in B, and v is an edge in this cycle.

3. If u has not been added to a forest and v is in forest A, then adding u to forest A causes a cycle to form and v is in this cycle.

Run a search on the auxiliary graph, starting at the newly considered edge, and stopping successfully if it finds a vertex in the auxiliary graph (which is an edge in the original graph) that has no outgoing edge (ie, in the original graph, removing this edge from its current forest and adding it to the other forest does not cause a cycle to form). It may be noted that this vertex will never be the starting vertex, since such a case is already handled in case 1.

If this vertex is found, a path can be found in the auxiliary graph from the starting vertex to this vertex, which in the original graph is an ordered list of edges, the first element of which is the newly considered edge, and each of the following edges fall alternatively into either forest. We shall call such a set an augmenting set. We can augment such a set by adding the first edge to the forest that the second edge belongs to, and then swapping the forest in which each of the remaining edges belong to. We leave out the proof of how this maintains that both forests stay consistent.

The algorithm returns 2 spanning trees successfully if all edges have been successfully added to either of the forests, and all nodes are present in the same component in either spanning tree.

However, since we require to find the spanning trees only for a subgraph of the given graph, which contains the source and target nodes, we shall take the 2 forests obtained by the above algorithm and consider only the spanning trees in these forests, which contain both A and B. we further purge these spanning trees by deleting vertices which are not present in both spanning trees. If we obtain a pair of spanning trees in this manner, they are guaranteed to be a pair of edge disjoint spanning trees on the subgraph of the main graph. For now, we do not prove if such a method is guaranteed to find a solution if it exists.

Suboptimality

While 3 details a optimal strategy, it is constrained by the fact for the Short player, the optimal strategy only exists if the graph, in its current state, has a subgraph containing A and B that has 2 edge disjoint spanning trees. A similar condition holds for the Cut player, with the condition being imposed on the dual graph instead. However, since we are considering a video game, it is necessary for either player to make a move even if the optimal solution does not exist, and this move should cause an optimal state to emerge if the opponent player does not play optimally in the next move.

To analyze this further, let us consider a graph where it is not possible to obtain 2 edge disjoint spanning trees, but it is possible to obtain a pair of spanning trees that is edge disjoint for all pair of edges except one. Let us call this edge which is common to both spanning trees a *knot*. It can be seen that if the next move is the Short players, infact the optimal move it to short the knot, since if they do, starting from the next move, there will be a pair of edge disjoint spanning trees, allowing them to have a winning strategy.

Thus we devise the following strategy to find the pair of spanning trees that have the minimum number of knots:

1. Double each edge in the original graph, and assign to these newly added edges a weight that is double of the weight of the edge it was duplicated from. Since the original graph was unweighted, assume each edge in the original had a weight of 1. Thus, all the newly added edges have a weight of 2.

2. Run the weighted version of Tarjan's algorithm[7] on this new graph. (The weighted version is only different from the unweighted one in that the edges considered for adding to the forests are considered in the increasing order of weights)

If it exists, this algorithm is guaranteed to return a pair of spanning trees that span the entire graph and has the minimum possible number of knots.

From this, we have 3 cases for the best move a Short player:

1. No such pair exists, in which case the Short player does not have a good strategy.
2. A pair of edge disjoint spanning trees exists, in which case they can employ the strategy described in 3
3. A pair of spanning trees that share one or more knots, in which case the best move is to short a knot.

The Cut player can apply the same strategy on the dual game.

We observe empirically that if the subgraph with the spanning trees of the primal and dual game share the same set of edges, the total number of knots in both games combined is even, and if either player in their corresponding game shorts a knot, a knot is added to the spanning trees of the other player.

Chapter 6

Implementation

6.1 Game Design

The video game is implemented as follows: The game takes place on a square grid, with different parts of the grid obscured depending on the level/map. There exists 2 factions - humans and goblins. Humans have built a settlement which the goblins are trying to destroy. However, before the goblins begin their assault, both side are allowed to strategically place towers. The human team can create defense towers which can block off a grid cell and stop the goblin troops from moving over it. The goblin team can create support towers which prevent the human team from building towers near it.

At a single glance, this game is considerably more similar to the game of Hex[9], a turn based 2 player game which is played on a hexagonal grid which is bounded by a rhombus. Each player is assigned a color and a pair of opposite sides of the rhombus. The players take turns filling cells of the grid with their own color until a path is formed between any pair of opposite sides of the rhombus, at which point the player to whom these sides are assigned to wins.

Generalization of Hex on any graph will provide a problem which can effectively be described as the Shannon switching game on vertices. Determining the optimal strategy for this problem is a PSPACE complete problem.[10]

However, we approach the problem from a different angle, constructing the levels/maps in such a way that there are there are large open spaces (representing nodes) connected by narrow passageways (representing edges), such that the game is softly constrained to play out as if it were the Shannon switching game. In this game, the defence towers placed by the human team represents a move by Cut, which blocks a passageway. The support tower placed by the goblin team represents a move by Short, making a passageway unblockable. Refer figures 6.1 and 6.2.

It may be noted that it is not necessary for a player to play the game in this manner. For example, the human team can decide to construct a line of towers to try and block an entire section of an open space that represent a vertex. However such moves are very wasteful and quite trivial to counter against as long the map has a clear distinction between the two kind of areas mentioned earlier.

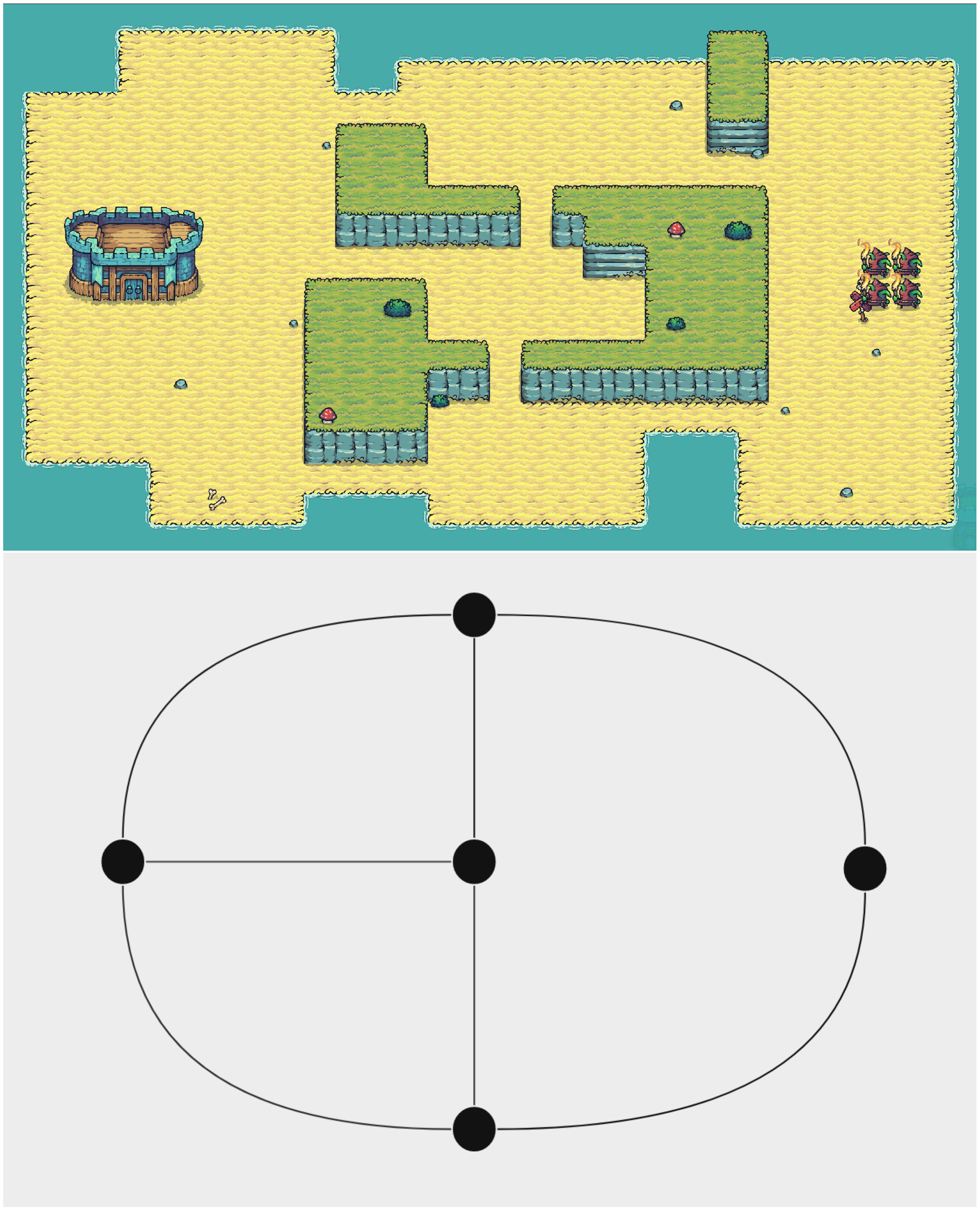
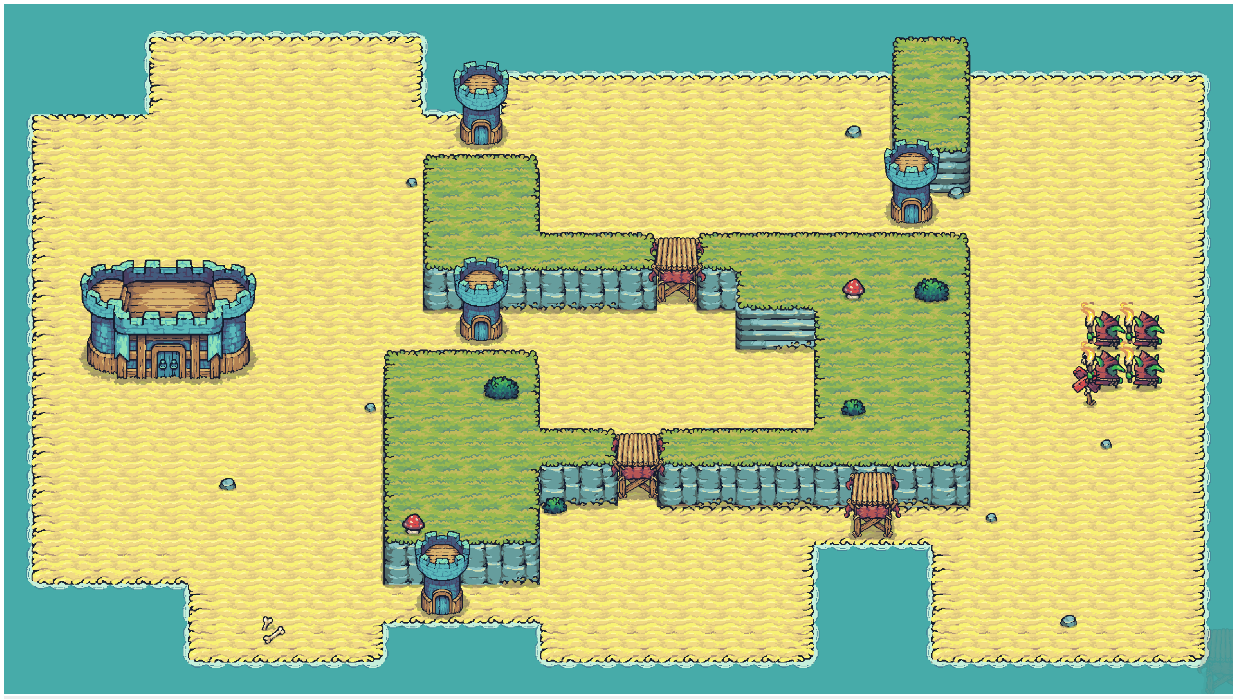


Fig. 6.1 First picture shows a screen shot of the game, the human settlement being on the left and the goblin troops on the right. The in between area is the grid, with the cliffs being parts of the grid that are inaccessible. Second picture is the corresponding graph for the Shannon Switching Game, the leftmost and rightmost vertices being A and B.



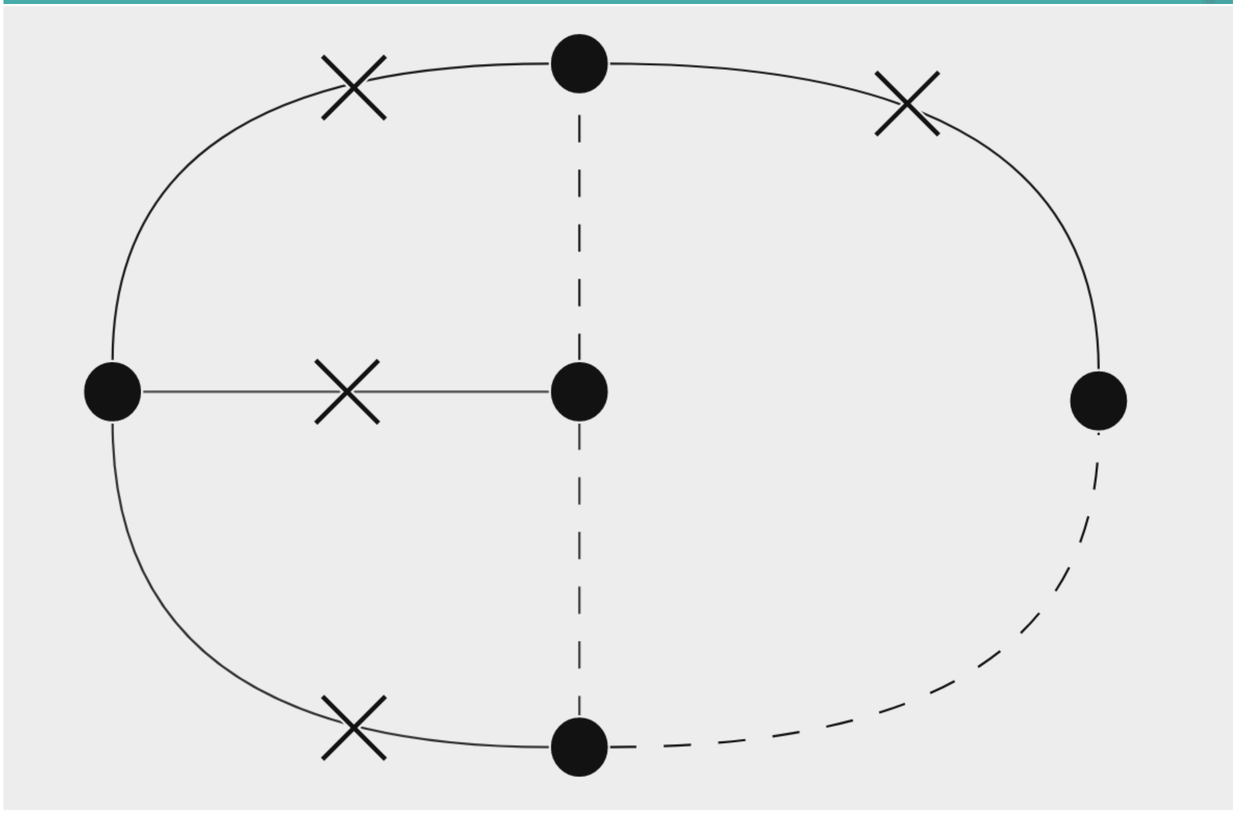


Fig. 6.2 First picture shows the completed game with the blue towers being places by the human team and the red towers placed by the goblin team. In this case the human team won as the goblins are unable to find a path to their settlement. Second picture shows the corresponding state of the Shannon Switching Game graph. In this case Cut has won as there is no path from A to B. (Dotted edges represent short, crossed edges represent cut).

6.2 Agent Design

Before implementing the algorithm of an AI agent playing the Shannon Switching game, it is required to implement an algorithm that extracts the graph from the grid. Here, grid can trivially be represented as a graph directly be converting each cell into a vertex and creating an edge between each each cell and its 4 neighbouring cells. Let us call this graph the *grid-graph*. We convert the grid-graph into the graph required for the Shannon Switching game by running a search algorithm and classifying each node based on whether it is has a degree less than or equal to 2. For this purpose, we define the a data structure called an *islet* (refer code block 2), which is a Disjoint Set Union of nodes of the grid-graph, and a boolean representing whether it is an *island*, which are nodes in the final graph, or a *bridge*, which are edges in the final graph. Refer algorithm 3.

Furthermore, a modified version of this algorithm can be used to construct the dual graph. The modifications are:

1. The grid is complemented, ie. grid cell which were previously occupied are now considered part of the graph and vice versa. Further, all grid cells which were bridges in the original grid graph are transferred over to this dual grid graph

2. The condition to be added as a bridge is changed, from the node having a degree 2, to the node already being a bridge in the primal graph.

3. A line is drawn between the position of A and B of the primal graph on the grid. During the post processing stage of the IslandBridge Algorithm, if an edge is between two islands, and one of those islands are above this line while the other is below, the union is not done and they are kept as separate islands. This is equivalent to finding to adding the transient edge that goes through infinity from A and wind back to B before the formation of the dual, and removing this edge after forming the dual.

Now let us implement and analyse different solutions to the Shannon switching game.

Algorithm 2: Islet Data Structure

```
1: struct Islet {
2:     node representative {Canonical node representing the islet}
3:     boolean isIsland {Boolean indicating if the islet represents an island}
4: }
```

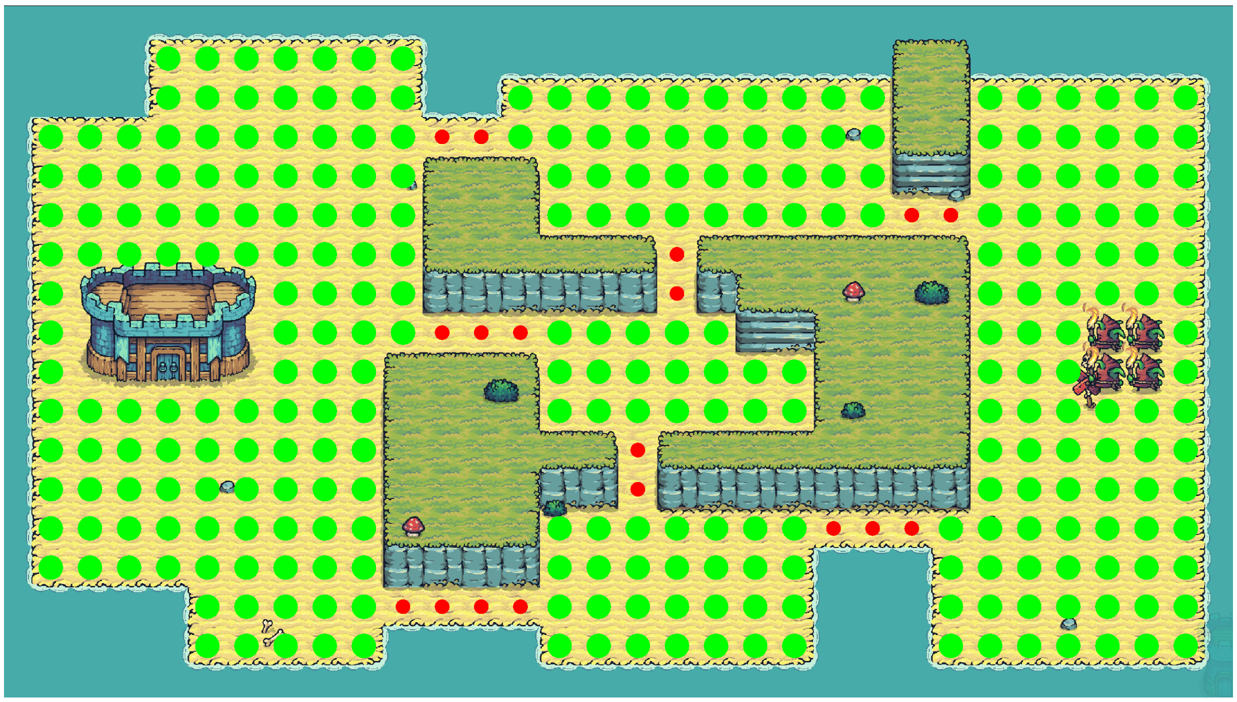


Fig. 6.3 The result of running Algorithm 3 on the map from figure 6.1. The green grid cells are part of a node and the red grid cells are part of an edge.

Algorithm 3: IslandBridgeAlgorithm

Input : gridGraph : grid-graph for a grid

Output: The corresponding Shannon Switching game graph

```
1  isletGraph ← Empty graph; currentIslet ← None;
2  for node in DFS(gridGraph) do
       // Note: Assuming DFS runs a depth-first search on the input
          graph and yeilds each node in order of visiting
3      if currentIslet is None then
4          currentIslet ← Empty Islet;
5          currentIslet.isIsland ← Degree(node) > 2;
6          currentIslet.AddElement(node);
7          continue;
8      if Degree(node) ≤ 2 and currentIslet.isIsland then
9          newIslet ← Empty Islet;
10         newIslet.isIsland ← False;
11         isletGraph.addEdge(currentIslet, newIslet);
12         currentIslet ← newIslet;
13     else if Degree(node) > 2 and not currentIslet.isIsland then
14         newIslet ← Empty Islet;
15         newIslet.isIsland ← True;
16         isletGraph.addEdge(currentIslet, newIslet);
17         currentIslet ← newIslet;
18     for neighbor in Neighbors(node) do
19         if Visited(neighbor) and find(neighbor).isIsland then
20             isletGraph.AddEdge(currentIslet, find(neighbor));
21     currentIslet.AddElement(node);
   // Post processing stage
22 for edge in Edges(isletGraph) do
23     if edge.from.isIsland and edge.to.isIsland then
24         union(edge.from, edge.to);
25 for node in Nodes(isletGraph) do
26     if not node.isIsland then
27         if Degree(node) = 1 then
28             union(Neighbors(node)[0], node);
29         else
30             isletGraph.RemoveNode(node);
31             isletGraph.AddEdge(Neighbors(node)[0], Neighbors(node)[1]);
32 return isletGraph
```

6.2.1 Two Player Shannon Strategy

In this section we implement the strategy described in 3 on the Island-Bridge graph we obtained in the previous section. We only implement the strategy for the Short player. The cut player can follow the same strategy on the dual problem.

We follow the algorithm described in subsection 5.1.1 to find a pair of edge disjoint spanning trees in a subgraph of the graph (or the dual of the graph if it is the Cut player). If such a pair does not exist, this strategy is not able to perform a move anymore. It may return a random edge.

If such a pair exists, the Short player will preserve it and keep track of their optimal move. As follows:

1. If the Short player is going first, we short a random edge from the subgraph obtained.

2. On a move of the Cut player, if they cuts an edge that is not in the subgraph obtained, in the next move we short a random edge in the subgraph.

3. On a move of the Cut player, if they cut an edge that is in the subgraph obtained, we remove this edge from the corresponding spanning tree it is in (let us call this spanning tree s_1). We then take each edge in the other spanning tree and check if they fall between the two components obtained due to this cut. Such an edge will be necessarily found, upon which we add an identical edge s_1 using the next turn, which is of the Short player. Even though technically this is a knot, it can no longer be cut. Thus this procedure is identical to shorting this edge.

We can see that this is a direct implementation of the optimal strategy in question. Thus, if either Cut or Short is able to follow this strategy, they are guaranteed to win.

This strategy suffers from 2 major drawbacks. Firstly, it only calculates the spanning trees at the beginning of the game. If the spanning trees do not exist then, the agent assumes it is impossible to win the game, while in practice, it should also be able to look out for the opponent performing a suboptimal move after which it can re-establish an optimal strategy. Secondly, the agent should be able to continue playing even if there is no optimal strategy, instead opting to follow the best move policy described in 5.1.1.

6.2.2 Graph Update Shannon Strategy

This version of the strategy aims to fix the first drawback of the previous implementation. To achieve this, we first try to find the pair of edge disjoint spanning trees in the original graph. If they exist, we can roll back to the previous strategy and maintain the winning strategy. If it does not exist, we perform our moves randomly, and recalculate the spanning trees after each turn.

We cannot use the exact implementation for cutting and shorting we used in the previous strategy to achieve this, since we need a modified graph each time a move is played when there is not optimal strategy. During each cut turn, we actually cut the edge off the graph, and during each short turn, we short the corresponding edge in the graph. Each time the graph is updated, we have to recalculate the spanning trees.

6.2.3 Self Weighted Shannon Strategy

This version of the strategy aims to implement the suboptimal strategy described in 5.1.1. We do this by copying the graph, assigning a weight of 1 to each edge, duplicating each edge and assigning a weight of 2 to each duplicate edge. We then run the weighted version of the algorithm for finding the pair of edge disjoint spanning trees. Depending on the output of the algorithm, we adopt one of 3 alternatives:

1. If no pair of spanning trees is found, short a random edge

2. If a pair of spanning trees is found with no knots, short a random edge from either spanning tree

3. if a pair of spanning trees is found with one or more knots, short any knot

Further, we recalculate the graph after every cut move. We follow shorting and cutting in the way the previous version of the algorithm did.

Chapter 7

Future works

7.1 Theoretical framework

1. At the end of subsection 5.1.1 about finding the pair of edge disjoint spanning trees, we mention a procedure we are using to find a subgraph of the input graph in which a pair of edge disjoint spanning trees are present. While this method works empirically, we have yet to look into the proof of correctness of it.
   It is of interest to note that the obvious and surefire alternative to this is to brute force the spanning tree algorithm over each subgraph of the input graph. However this method is inviable as we expect to have to deal with large graphs.

2. I wish to write formal proof for the approach of doubling edges to find the pair of spanning trees with minimum number of knots. Furthermore, this algorithm also finds a pair of spanning trees that span the entire graph, even if there is a subgraph where there exists a pair of edge disjoint spanning trees. This fact need proving, and also requires an analysis on which is the better strategy, or if they are equivalent.

3. The idea of knots can be analysed in more depth. We observe empirically that if either player "unties" a knot by shorting it, a knot gets added to the graph of the other player. Infact the entire optimal strategy described in 3 can be visualised as

a game of trying to pass the knot on to the other player. Building a theoretical framework for this idea might be useful.

4. We have yet to touch upon any idea of Heuristics for any non optimal case. Particularly when there is no good move possible or when there are multiple knots and any of them could be a good move. For example, there are cases where a player has 2 knots, playing one of which could immediately end the game while the other will let this player play for a few more turns, then they should always go for the one which gives them more moves, since there is a higher chance of the opponent making a blunder in those moves.

5. A time complexity analysis of all the algorithms proposed in the report, as well as optimization wherever required. We know for a fact that algorithm 5.1.1 for finding the pair of edge disjoint spanning trees is not optimal in terms of time complexity, since Tarjan's paper[7] details a more efficient implementation. The Island-Bridge algorithm (3) also requires a time complexity analysis, and perhaps an overhaul on the side of optimization.
   The graph update strategy of the agent, which recalculates the spanning trees each turn, can be optimized to use some local feature of the graph and spanning trees instead of recalculating the entire graph again. To help with this, an interesting data structure called the link-cut tree[11] can be used. This data structure is similar to the DSU, except that it also has a disunion operation which can be used to separate 2 trees. Interestingly, this data structure was also published by Tarjan.

7.2 Implementation

1. **Optimization** : While lag has yet to be noticed during gameplay testing so far, optimization on the level of implementation has a lot of scope. The data structures used to maintain a graph, tree, search result etc. could be changed to more optimized

alternatives. A server thread could be maintained that calculates and keeps the edge disjoint spanning trees saved and can be queried by AI processes in the game when required.

2. **Edge weighting** : A good policy for weighting edges may improve the performance of the AI agent in non optimal cases. For example, if the island graph edges are weighted by their distance (on the grid-graph) to A or B (or both), the Short player will favour trying to find the solution in the least amount of moves and only move to larger paths if forced by cut. Similarly, cut will favour trying to cut off the shorter paths letting the game go on for longer.

3. **Difficulty** : Like the previous game, an implementation of difficulty based on a probabilistic approach is possible for this game as well. However, we can see that we have already come up with multiple strategies which get progressively better at playing the game. The implementation of weighted graph and heuristic is also in this line. Thus, which algorithm, weighting strategy, heuristic etc to use could in themselves be a considered a process choosing the difficulty level of the game.

4. **Gameplay changes** : Currently, the game is implemented as if it is in itself the shannon switching game. However, we wish to implement it like a real time strategy game, where the graph problem and strategy will only form one major component of the gameplay. Other components will include base building, resource gathering, satisfying side objectives on the map etc. Furthermore, we would like to remove the turn-based constraint on the game and instead enforce a set of rules which emulate a turn based game. For example, each player has to collect resources to build towers. The process of collecting resources take time, forcing each player to wait an amount of time before making their move. This amount of time will be just enough for the other player to make their move.

References

[1] J. Hamkins, *Proof and the Art of Mathematics.* The MIT Press, 2020, sections 4.6 and 7.2.

[2] unniisme. (2023) fish-in-the-barrel. GitHub. [Online]. Available: https://github.com/unniisme/fish-in-the-barrel

[3] Godot Engine Contributors. (2023) Godot engine documentation. Godot Engine. [Online]. Available: https://docs.godotengine.org/en/stable/

[4] R. Mansfield, "Strategies for the shannon switching game," *The American Mathematical Monthly*, vol. 103, no. 3, 2001.

[5] N. Nandan and N. Loganathan, "Graph theory / combinatorics based mobile games," 2023, bTech Project.

[6] Wikipedia contributors. Dual graph. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Dual_graph

[7] R. E. T. James Roskind, "A note on finding minimum-cost edge-disjoint spanning trees," *Mathematics of Operations Research, Vol. 10, No. 4*, 1985. [Online]. Available: https://www.jstor.org/stable/3689437

[8] Wikipedia contributors. Kruskal's algorithm. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[9] ——. Hex. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Hex_(board_game)

[10] S. Even and R. E. Tarjan, “A combinatorial problem which is complete in polynomial space,” 1976.

[11] D. D. Sleator and R. E. Tarjan, “A data structure for dynamic trees,” *Journal of Computer and System Sciences*, vol. 26, no. 3, pp. 362–391, 1983.