

# Finding and Resolving Network Inconsistencies in Multiplayer Games

Newcastle University

BSc Computer Science

Author: Dovydas Simoliunas (180327267)

Supervisor: Giacomo Bergami

## Abstract

This paper contains research of network inconsistencies in multiplayer games. Here, we simulate a multiplayer game environment using a multithreaded approach. We use execution threads as players and have them play a game of Domino against each other. This is achieved by having them broadcast JSON message objects, containing the relevant game information, between themselves to simulate a P2P connection. Then, Inconsistency is injected into certain messages at the beginning and end of the game. We define two inconsistency metrics to be used to measure the impact these inconsistencies have on the game. The testing part of this project is done by running multiple Domino games and using the two metrics to calculate the effects the injected inconsistencies have on the gameplay experience. An approach to resolving the network inconsistencies using the Consensus algorithm is also provided and discussed. In the project itself, only a certain part of the inconsistency is fixed. Then, using testing, we provide proof of how important to the flow of the game and players' experience it is to have at least partial inconsistency resolution in a multiplayer game.

## Declaration

"I declare that this dissertation represents my own work, except where otherwise stated."

## Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Giacomo Bergami for his help and support throughout this project. His guidance and feedback made this project possible and helped me immensely in learning more about this topic.

Additionally, I would like to thank my family and friends for helping me keep my spirits up when writing this work. If not for their words of encouragement, I doubt I would have gotten past the initial stages of this paper, when all of this seemed too overwhelming.

## Table of Contents

Abstract.....	2
Declaration.....	3
Acknowledgements.....	4
Dissertation structure .....	7
Introduction .....	7
Aim and Objectives .....	7
Changes that occurred since the proposal .....	7
Technical Background .....	7
What was done and how .....	7
Testing approaches .....	7
Evaluation of the results .....	7
Conclusions .....	7
Future Work.....	7
Introduction .....	8
Context.....	8
Problem.....	8
Rationale .....	8
Aim .....	9
Objectives.....	9
Changes that occurred since the proposal .....	10
Challenges faced .....	11
Risks Involved.....	11
Technical Background .....	12
Literature Used .....	12
Technologies Used .....	13
What was Done, and How .....	14
Code Structure (Random message passing) .....	14
Domino Rules.....	14
Code Structure (Domino Game flow) .....	15
Domino Progression.....	16
Injecting Inconsistency.....	17

Altering the Seed .....	17
Altering the Winning Message.....	17
Alternative approach .....	18
Inconsistency Resolution .....	18
Consensus Algorithm approach .....	18
Initial Tile synchronization .....	20
Inconsistency Calculation.....	20
Total Inconsistency .....	20
Global Inconsistency .....	21
Normalizing the Metric.....	22
Testing Approaches.....	24
Control variables .....	24
Logic behind testing .....	25
Domino Test #1 .....	26
Domino Test #2 .....	30
Domino Test #3 .....	32
Domino Test #4 .....	35
Domino Test #5 .....	38
Domino Test #6 .....	41
Domino Test #7 .....	44
Domino Test #8 .....	46
Evaluation of the results .....	50
Conclusion.....	51
Project Aims and Objectives .....	51
Reflecting on the Project .....	51
Future Work.....	52
References .....	53

## Dissertation structure

### Introduction

Introduces the reader to this project by giving it context and motivation behind the tackled problem, as well as introducing the problem itself.

### Aim and Objectives

Describes the main goal this project is attempting to achieve as well as the objectives it will attempt to reach. Each objective is shortly expanded upon.

### Changes that occurred since the proposal

There have been multiple large changes added to the project since the submission of the Project Proposal assignment, this chapter describes them in detail.

### Technical Background

This chapter covers research literature and technologies used in this project.

### What was done and how

The chapter describing the code implementations, used in this project. This mostly consists of building the simulated game and explaining how it works.

### Testing approaches

The testing part of the project. This chapter contains all of the tests done in this project.

### Evaluation of the results

Chapter dedicated to evaluating the information, gained from testing, and drawing conclusions based on it.

### Conclusions

A definitive summary of work done as well as a reflective analysis of the project in regards to aim and objectives reached.

### Future Work

A short description of any future work that could be done. This includes work that was planned to do but eventually removed from the project due to various constraints.

# Introduction

## Context

Video games are a huge worldwide industry. In 2021 the whole industry was estimated to be worth \$178.73 billion. This number is forecasted to reach \$268 billion by 2025. [1] Clearly, the industry is growing fast and with it being so big – so is its player base. According to various research, there are nearly 3 billion video game players worldwide and around 56% of the most frequent players prefer to play multiplayer games. [2] Multiplayer in videogames has been around for decades. It is a genre so vast and popular that it has its own sub-genres to accommodate to different types of players. This project will focus only on the online multiplayer aspect of games.

## Problem

With the huge demand for multiplayer games there also comes a high standard of expectation for the network quality in them. The information needs to be passed through the network as quickly and efficiently as possible to maintain the real-time interaction between players. However, with the high-speed information passing comes faults. Inconsistencies like messages being dropped, altered or not handled in time. At best, this can have a slight effect on the player's immersion of the game, while at worst, it can ruin the whole experience or stop the game entirely. Initially, it might seem obvious that the developers of online games should prioritize reducing network inconsistencies as much as possible. However, focusing too much on fixing inconsistencies might eventually compromise the speed that the network messages are passed and received in. This could lead to causing an even bigger detriment to the player's experience.

## Rationale

This project will contain research on measuring and resolving inconsistencies in Multiplayer games. It will use inconsistency metrics to show the importance of catching inconsistencies early. The inconsistency measurements will be taken from a simulation of an online multiplayer Domino game that is written in C++ programming language. In this program, multiple execution threads will be playing a game of Domino using a simulation of an online environment achieved using JSON message passing between threads. Inconsistency will be injected into these communications to test how it affects the gameplay and the outcome of the game. The project will further go on to test the effect that both inconsistency detection and resolution have on the game. To measure the exact effect that an inconsistency has had on a game, this project will define and use two types of inconsistency metrics. Using these metrics, we will measure the average effect an inconsistency has on the game as well as how that affect changes when changing the basic parameters of the game – adding more players and increasing the scale of the game in other ways.



## Aim

The main aim of this project is to help reduce the number of network inconsistencies in multiplayer games. This will be achieved by researching the effects the inconsistencies and their resolution have on the gameplay and, potentially, the user experience.

## Objectives

### **1) To simulate a multiplayer game using execution threads as players.**

The game will use execution threads that pass network messages between each other to simulate network conditions. These threads will play the created game with each other to provide context for the network messages.

### **2) To inject artificial inconsistency into the simulated game's network.**

Inconsistency will be generated by randomly altering specific game messages. This will be the base of the research on inconsistency impact, detection, and resolution in this project.

### **3) To measure the inconsistency metrics of the game.**

This measurement will use two inconsistency metrics defined in this project. The effects of the injected inconsistency will be measured using these metrics to calculate the impact it had on the gameplay.

### **4) To calculate the overhead that is applied by finding and resolving network inconsistencies.**

The overhead is either the time taken to find and fix network inconsistencies or the number of messages passed between players. This calculation is important for knowing the efficiency of the inconsistency resolution implementation.

### **5) To measure how much the overhead affects player's experience when compared to the inconsistencies themselves.**

This will measure whether or not the inconsistency resolution will help with the player experience or would keeping the inconsistencies be less detrimental than taking the time to find and fix them.

## Changes that occurred since the proposal

**Change 1.** The plan for this project has changed since the proposal was submitted. The initial plan was built around turning an existing 2-Dimensional platformer game, created using the Unity game engine, into a multiplayer game by adding a working network functionality to it. The network code was supposed to be separate from the game code and used as a C++ library plug-in in the Unity project. This would have made the network plug-in reusable between multiple games.

The C++ network plug-in was supposed to be modelled as a multi-layered stack. The stack would have had each received message run through multiple inconsistency checks before it was returned to the game. It was supposed to be based around the Chandy and Lamport's snapshot algorithm [3, 8] (see Change 2). It would also have contained a layer of inconsistency check against the game rules. Each message would go through the entire stack of inconsistency checks before the game would receive it. This, along with the entire plan for turning the Unity game into a multiplayer, made the project quite ambitious. This led to the project getting behind schedule in March. Eventually, due to time constraints, it was cut down to showcase similar information in a less complex way.

The Unity game part was dismissed because of the time it would have taken to change an entire game to adapt to multiplayer. However, the point of having an online multiplayer game to work with would have been to use it to measure the network's inconsistency metric and the time overhead taken to resolve the inconsistencies. This could all be simulated using a more straightforward approach.

In place of that, the project is now using a Domino game, quickly developed in C++. The game simulates network conditions by using a multithreaded approach. This way, the project's schedule got back on track, without sacrificing the potential for testing and research.

**Change 2.** Another change introduced since the project proposal was forgoing the Chandy and Lamport's snapshot algorithm [3, 8]. Initially, the C++ network plug-in was planned to be structured as a stack and, in the middle of the stack, the messages would be passed through the snapshot algorithm to locate any potential inconsistency appearing in the message, making using the plug-in transparent to the programmer. However, due to the simplicity and the turn-based structure of the Domino game, the snapshot algorithm itself was not needed anymore.

In order to showcase how inconsistency works, we restricted our analysis on two types of errors, error on distributing the game's random seed for distributing the card, and error on declaring the winner. The purpose of doing so is twofold: while this mimics message being affected by bursts, message changes also mimic possible effects of "man in the middle" attacks, where one malicious node might want to alter the development of the game. This is very likely as messages, as per modern Video Game industry, do not always re-send the whole state of the game, rather than incremental updates describing which is the new outcome of the turn. By doing so, it is fairly easy to disrupt the game experience, as it might be very likely that the user cannot detect errors immediately. Therefore, it becomes crucial to see whether

the game has been disrupted or not by checking the boards or players' hands for duplicate tiles as well as by searching local player states for any differences.

Therefore, despite the snapshot algorithm itself being deemed unnecessary, its idea is still being used in the project in another manner. The algorithm is mimicked by a thread barrier set at each turn of the domino game, at which, the last player will calculate an inconsistency metric for testing the amount of the overall inconsistency of the game.

For testing purposes, at the end of every turn, each player's local information is accessed and compared to calculate the inconsistency metric at that stage of the game. This simulates the snapshot algorithm but does not affect the gameplay in any way whatsoever. By using the advantage of the concurrent simulation, the local states of the player objects are accessed directly, bypassing the need for them to send it via snapshot messages. In other words, because the game simulation is multithreaded, the project produces the results of running the snapshot algorithm without actually doing so. As mentioned before, this is only used for recording the inconsistency metric throughout the game, as it is necessary for the testing part of this project (see "Testing Approaches" chapter).

## Challenges faced

While the biggest hardship of this project – time management – is already described above, another big problem stalling the development was choosing Microsoft's Visual Studio as the preferred IDE. Because of its specific linker issues and unclear error messages, the development was occasionally halted completely. At a certain point, the inconsistencies of Visual Studio caused the project to be completely reworked the C++ network code from the ground up. Luckily, this happened right after deciding to forgo the Unity approach, however, the time taken to rewrite the code nevertheless had significant negative impact on the development time.

## Risks Involved

With the code part of the project having to be restarted from the ground up late into the given development time, the main risk for it was managing to create a fully formed project in the given time. Because of this risk, many features had to be cut, as described in the sections above. Furthermore, other desired features were deemed too time consuming and were forgone to save time. These features will be described in the "Future Work" section at the end of the project.

## Technical Background

### Literature Used

#### **Distributed systems: concepts and design by George F. Coulouris. 2012 Chapters 14 & 15**

Chapter 14 introduces “concepts and algorithms related to monitoring distributed systems as their execution unfolds”. This chapter brilliantly describes Chandy and Lamport’s ‘snapshot’ algorithm – the basis of the initial approach for this project. As mentioned before, the current version of the project skips the algorithm and only simulates its results. However, the idea behind the algorithm is still used, in theory.

Chapter 15 covers coordination between distributed systems. How synchronous and asynchronous systems differ and how to handle communication failures (inconsistencies) between distributed systems. This chapter also describes the consensus algorithm, which will be used to resolve any inconsistencies appearing in the game.

This book contains a lot of the information that will be needed for the project as it covers the whole topic of distributed systems quite thoroughly.

#### **Measuring Consistency Gain and Information Loss in Stepwise Inconsistency Resolution by John Grant and Anthony Hunter 2011**

In this paper Grant and Hunter investigate “what are essential properties of inconsistency and information measures” [Grant, Hunter, 2011]. They define multiple approaches to measure and resolve inconsistencies. In §3.1, Definition 2 they define multiple inconsistency measures. In this project, the logic behind some of these approaches is used to define one of the two inconsistency metrics that will be used to measure the inconsistency of the Domino game – Total Inconsistency.

#### **Analysing Inconsistent Information using Distance-based Measures by John Grant and Anthony Hunter 2016**

This paper defines multiple distance functions used for inconsistency detection and resolution. Some of these functions are used in the other inconsistency metric used in this project – Global Inconsistency. As Global Inconsistency metric in this project is made up of three distance functions. This decision and the functions were influenced by this paper, as the Dalal measure, described in Section 3, Definition 5 of Grant and Hunter’s paper, is used in the Global Inconsistency metric calculation.

#### **Reducing the negative effects of inconsistencies in networked games by Cheryl Savery and T.C. Nicholas Graham 2014**

This paper is an outlier between all of the previously covered works. It focuses on the reaction of the players when they observe inconsistency in videogames. It analyses what type of inconsistency resolution approaches are most noticeable and annoying for the players. This information is indirectly used in the current project to estimate an average player’s reaction to inconsistency and its resolution, as well as the effect they have on the players’ experience.

## Technologies Used

Microsoft Visual Studio 2019	The IDE used to write the code for this project. It was chosen due to convenience as the author had the most experience with it, however, as previously mentioned, it did cause a lot of issues during the development.
C++ Programming Language	Programming language used for the code of this project. It was useful as it enables multithreading programming to simulate network conditions. It is also the language the author has most experience with.
GitHub	A version control tool, mostly used for preserving older versions of the code for testing later. Being able to revert to working versions of the code also helped when debugging the application.
NLohmann JSON library [9]	<a href="https://github.com/nlohmann/json">https://github.com/nlohmann/json</a> The C++ library used to create custom JSON messages. It allows the programmer to create custom JSON objects and load them with various information. This well-made library made it easy to write information to the messages before they were sent as well as read it once the messages were received.
Microsoft Office Excel	Used to store test data when calculating the average inconsistency of multiple games. It was also used to create the graphs displayed in the Testing section.

## What was Done, and How

### Code Structure (Random message passing)

This was the first layout achieved before starting work on the Domino game.

The development of the application started from creating a working message passing functionality. Initially a producer-consumer approach, where multiple threads ran two different codes, was implemented. One thread was the receiver and another - the sender. These threads used JSON messaging [9] to pass their information between themselves. The sender would send a set number of messages to the receiver and the receiver would check its own incoming messages for a set number of seconds.

Once that was working, the code for dropping and altering the messages on random occasions was added. From here the added inconsistency could be measured by counting the number of messages the sender has sent and subtracting the number of unaltered messages received from it. Later, the code was updated and the sender method was merged with receiver to make all the threads run the same code. This structure was the base of the domino game implementation described below. It allowed the threads to communicate with each other and send random information back and forth. From this point the code structure for the threads was changed to adapt to the Domino rules and a turn-based layout.

### Domino Rules

Because of there being many different ways of playing Domino, it is worth clarifying the rules set for this project.

At the beginning of the game, all of the Domino tiles are shuffled into a stack, or a “deck”. Then, each player draws a set number of tiles from the deck to fill their hand. The number is a parameter decided before starting the game. A single tile is taken from the top of the deck and placed onto the board. Then, the first player makes their turn.

For this project a simple version of the board is chosen. Starting from the initial tile, the board will have two ends with a number each. For example, if the initial domino tile placed on the board is [0:1], the front end value of the board will be 0 and the back end value will be 1.

To place a tile onto the board, a number on the tile has to match with either end of the board. The tile is placed by connecting the matching numbers, making the other number on that tile that board end’s new value. Therefore if, for example, [0:2] tile were to be placed at the front of the board, the new board would look like [2:0] [0:1] and its end values would now be 2 and 1.

During their turn players can either place one tile from their hand on either side of the board or, if they do not have any matching tiles, draw one tile from the deck to their hand. If they need to draw but the deck is empty, that player skips their turn.

The goal of the game is to be the first player to empty out your hand by placing all of the tiles onto the board. The game ends after the first player wins or, after the deck is empty and no players can place tiles anymore.

### Code Structure (Domino Game flow)

As mentioned before, this project is built around testing a Domino game simulation. It uses multiple execution threads to “play” Domino against each other. A reference to a player object is passed to each thread. These objects have two incoming message queues – one that is safe, where messages are not dropped nor altered, and one that allows for inconsistencies within its messages.

The latter one is used to receive all the basic game messages, where inconsistency is allowed. This simulates the UDP network protocol with it being fast but potentially inconsistent. The “safe” message queue is used in the Domino game when resolving any inconsistencies. This allows the simulation to guarantee the inconsistency resolution messages will go through to the receiver unaltered.

Each thread generates their own local instances of Domino game objects, like the game board, the deck of all tiles, and their own hands. To keep these synchronised, one thread broadcasts a seed to all other players. Based on this seed, the players shuffle their decks and draw tiles to their respective hands from the deck. Once the hands are drawn, one tile is taken from the top of the deck and placed on the empty board. Then, player number 1 starts their turn.

Any player, during their turn, can either place a Domino tile on either side of the board or, if they have nothing that can be placed, draw a tile from the deck. A message is constructed containing information relevant to each type of action.

A “Tile placed” type of message contains the tile that is placed on the board and one of two positions of the board, where the tile was placed - the front of the board or the back. Upon getting this type of message, the receivers place the tile onto their own local board.

A “Tile drawn” type of message does not need to contain any extra information as the receivers do not need to know what tile was drawn, they only need to know to remove one tile from the top of their local deck.

Each player’s turn ends in broadcasting one of these messages with the turn’s information. The game ends when somebody wins by placing the last tile that was still in their hand or when nobody can place a tile anymore and the deck is empty.

When a player places the last tile in their hand, along with the regular “Tile placed” message information they fill out another field as well, this field is called “has won”. When a player wins in the winning “Tile placed” message they set the “has won” field to *true*. When a player receives a message that has a “has won” field set to *true*, they know to finish the game and update their local information to say that the sender of the message won the game.

Because of how the game is structured, each thread has its own local objects that are updated according to the messages that are received. Thus, the game relies heavily on synchronising these objects between the players. This is where inconsistency can be simulated.

For synchronization reasons, each player object also stores their hand and board histories.

Hand history is a set of all tiles the player has held in their hand throughout the entire game. It is used when calculating the inconsistency of the game by comparing multiple players' hand histories to see if they have held duplicate tiles.

The board history is a sequential array of actions taken in the game. Each element contains the player that performed the action, the tile that was placed on the board and the end of the board it was placed on -front or back. This can be used to recreate the board up until a certain point if an inconsistency is detected.

### Domino Progression

```
Player0 sent the seed: 2329
Player 1 received seed: 2329

P0 placed 1:3 in the back
[[1:1]] [[1:3]]

P1 placed 1:2 in the front
[[2:1]] [[1:1]] [[1:3]]

P0 placed 0:2 in the front
[[0:2]] [[2:1]] [[1:1]] [[1:3]]

P1 placed 3:3 in the back
[[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]]

P0 placed 0:4 in the front
[[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]]

P1 placed 2:4 in the front
[[2:4]] [[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]]

P0 placed 0:3 in the back
[[2:4]] [[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]] [[3:0]]

P1 placed 2:2 in the front
[[2:2]] [[2:4]] [[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]] [[3:0]]

P0 drew
[[2:2]] [[2:4]] [[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]] [[3:0]]

P1 placed 0:0 in the back
[[2:2]] [[2:4]] [[4:0]] [[0:2]] [[2:1]] [[1:1]] [[1:3]] [[3:3]] [[3:0]] [[0:0]]
```

Figure 1. Regular Domino game output (2 Players)



From the figure above the output of a simulated domino game can be observed. This is a 2-player Domino game, using tiles that only have numbers ranging from 0 to 4.

At the beginning of the game, Player0 broadcasted a randomly generated seed – 2329. This seed was used to synchronize the players' decks as they will always be shuffled in the same order when taking in the same seed parameter. Then Player0 goes on to do their turn. Each player prints their own move as they broadcast a message to the other player and at the end of their turn their local board is printed.

### Injecting Inconsistency

At certain points of the game, inconsistency is injected into the messages to corrupt the information without the receiver's knowledge.

For this game the type of inconsistency that was focused on is game messages getting altered. Messages are not being dropped in the game scenario because of the turn-based structure of the game that makes all recipients know to expect a single message from the player whose turn it is. Dropping the game messages would only cause delay but it would, in theory, not cause any game-specific inconsistencies - only time loss.

For this project, only the initial seed message and the winning player's final message can be randomly altered.

### Altering the Seed

Altering the seed creates an inconsistency between the players' game states. If a player receives an altered seed information, their deck is going to be shuffled differently than other players, thus desynchronising the entire game. Any player with the wrong seed would still assume they have the correct one and play the game normally. On their turn, they would still be placing tiles they have in their hand or drawing from their own local version of the deck. This would eventually result in the board having multiples of the same tile, which would be a clear violation of the game's logic. It could also lead to the issue of the players seeing a tile that they have in their hand being placed on the board by another player.

Altering the seed also creates another hurdle. Because of how the game is structured, each player starts the board by drawing one tile from the top of the deck and placing it into the empty board. The issue with this was seen later in the development process as the differently shuffled decks would cause an unforeseen inconsistency where a player could have a completely different looking board from the other players.

The decision to only alter the seed sent at the beginning of the game, instead of the "Tile Placed" type message, was also made in favour of reducing development time.

### Altering the Winning Message

The other inconsistency that can be injected in this game is the winning message alteration.

When a player sends out a winning message, it can get corrupted for some of the receivers and the “has won” field be set to *false*. This would lead that receiver to be unaware of the winner and continue playing the game normally.

Normally, this alteration would only cause a time loss problem, as the player would wait for other players to send their turn information for a set amount of time and then the program would force quit the game. However, if the receiver, that has gotten this altered message, is the same person whose turn is next after the winner’s, they would continue placing or drawing the tiles for their one turn. Furthermore, if all of the players would get this inconsistency, they would all continue playing the game until the turn order would come back around to the winner.

This inconsistency can completely desynchronize the game in its late stages and would definitely affect the player’s experience as it influences the main outcome of the game.

### Alternative approach

The alternative approach for creating this type of desynchronisation between the players’ game states would be randomly changing the tile information in the “Tile placed” message to another other tile. The only caveat to this would be that the new tile should also be placeable in the same position of the board. This would create a difference in player’s local board states as the player receiving the message would automatically place the fake piece. However, this project excludes this approach because this type of inconsistency would take more time to detect and resolve while the effect of it on the gameplay would, theoretically, be very similar to just altering the seed once.

### Inconsistency Resolution

To resolve the seed inconsistency the obvious solution is to confirm the seed once it is received. This project does this by confirming it on the safe message channel. However, working under the assumption that the safe channel is unavailable, for the sake of this project, the inconsistency is not going to be resolved until it is noticed by a player. This way, the inconsistency approach where only the initial seed message is altered will affect the game in the intended way.

### Consensus Algorithm approach

The idea behind this approach is that the inconsistency would only be detected when a player makes an invalid move or place a tile that another player has in their hand. Once a player notices an inconsistency, they will start the consensus algorithm [3]. Each other player will have to send a message containing the seed they are using and their local winner information to the player that started the consensus, the coordinator. Each unique pair of players will be compared by these values. if the information of two players is equivalent, the coordinator will input their IDs into an Equivalence Class algorithm. This algorithm matches the given information into equivalence groups.

For example:

The setting is a 5-player Domino game:

Player 0: seed - 4, winner – Player 0.

Player 1: seed - 1, winner – No winner.

Player 2: seed - 1, winner – No winner.

Player 3: seed - 1, winner – No winner.

Player 4: seed - 4, winner – Player 0.

After comparing the values of each unique pair of players, the pairs (0, 4), (1, 2), (1, 3), and (2, 3) were deemed equivalent and put into the equivalence class algorithm. The algorithm would then group the equivalent players together into equivalence classes.

Output: (P1, P2, P3), (P0, P4)

In the example above the Equivalence Class algorithm grouped the equivalent players together and it can be seen that the majority of players agree that the seed is 1 and there was no winner yet.

After deciding the majority, the coordinator recreates the board history based on the correct seed. After that, the player sends out the correct seed and the correct board history to the other players. Each player then updates their own local state with the information given. The players will use the seed to reshuffle the deck and redraw their cards, while the board history will update the board up until the last move made before the players in the minority of equivalence classes would have gotten injected with inconsistency.

*Unfortunately*, due to time constraints, the implementation of the Consensus algorithm was not fully finished and will be left for the future. More about this decision is written in the “Future Work” section at the end of the project.

However, this approach is still a good example of the effects that inconsistency and its resolution can have on the player’s experience. On one hand, the game cannot continue if one or more players has a completely differently shuffled deck, as that directly defies the rules of the game. On the other - catching the seed inconsistency too late would undo a lot of the game’s progress and would require to restart the game almost from the very beginning, greatly affecting the user experience.

For example, a player that got an altered seed and shuffled the deck differently from other players but has placed no duplicate tiles, nor tiles that do not fit on other players’ boards would go unnoticed by the inconsistency detection early in the game. Then, as the game progresses more and more, that player would most likely perform an action, that other players would consider inconsistent, and the consensus algorithm would be started. In that case, the game would have to be restarted from the point of the affected player’s first move, undoing most of the progress of the game. Therefore, an argument could be made that the actual resolution of the inconsistency is more detrimental to the players’ experience than the inconsistency itself.

## Initial Tile synchronization

As mentioned before, there is also the issue of desynchronized initial board tiles. This problem allowed the project to explore partial inconsistency resolution. Resolving this issue by synchronising the first tile and not by synchronising the seeds does not resolve the inconsistencies in the game itself but it does reduce them. Partially resolving the Inconsistencies could still greatly benefit the flow of the game and the player experience. This project explores this idea further in the Testing Approaches chapter.

The synchronization of the initial tiles is achieved by having the player that broadcasts the seed at the beginning of the game, also broadcast their own initial tile to other players. This places the same initial tile into everyone's boards and allows for the game to continue without any difference in the players' boards.

## Inconsistency Calculation

### Total Inconsistency

After every game the Total Inconsistency metric for that game is calculated.

For this simulation, the Knowledgebase  $K$  of a single run of the Domino game is defined as a set of propositions  $K = \{SameSeed(i, j), SameWinner(i, j) \mid i < j \leq N\}$ , where  $i$  and  $j$  are the indices of players participating in the game and  $N$  is the total number of players. In other words, the knowledgebase states that, to maintain a consistent game, each unique pair of players needs to have the same seed and the same outcome ("who won" or a "draw").

Because whether  $K$  holds true or not is dependent on the specific game run that it is measuring, the Total Inconsistency metric is used to calculate the satisfiability of  $K$ . To do that, it needs to count how many atoms of the knowledgebase are returning *false* on the relevant Truth Assignment function  $v$ .

By using  $i$  and  $j$  to both represent the player id and its internal state representing its view of the game,  $SameSeed(i, j)$  is interpreted as  $i.seed = j.seed$  in the context of a specific game run, as well as  $SameWinner(i, j)$  is  $i.winner = j.winner$ .

Function  $v(\alpha)$  takes in a preposition as a parameter, checks whether the boolean interpretation associated with the preposition holds true in this game run or not, and returns a boolean value of the result.

The Total Inconsistency calculation consists of two main steps. Initially a Local Inconsistency is calculated between each unique pair of players. To calculate this, the program compares each player's seed and their recorded winner of the game. If both of these values are the same the inconsistency between the pair is 0. The inconsistency rises by 1 for each mismatched value.

$$I_L(i, j) = 1_{SameSeed(i, j)} + 1_{SameWinner(i, j)}$$

To calculate the Total Inconsistency, all unique pair inconsistency values are added together.

$$I_T(K) = \sum_{j < i \leq N} I_L(i, j)$$

$N$  – total number of players

The logic behind this Total Inconsistency approach is heavily based on [Grant, Hunter, 2011] [4] In this paper Grant and Hunter investigate “what are essential properties of inconsistency and information measures” [Grant, Hunter, 2011]. They define multiple approaches to measure and resolve inconsistencies. In §3.1, Definition 2 they define multiple inconsistency measures. The total Inconsistency calculation is based on the  $I_p(K) = |Problematic(K)|$  measure. Here,  $K$  is the knowledgebase. This metric calculates the size of the  $Problematic(K)$ , which is a set of all formulae in  $K$ , that are involved in at least one inconsistency. The main difference between Grant and Hunter’s approach and the one used in this project is that, instead of getting the formulae that lead to an un-satisfiable  $K$ , the goal of this the Total Inconsistency calculation is to calculate the number of atoms that are assigned *false* by the Truth Assignment function. Therefore, the  $I_p(K)$  is redefined as  $I_p^v(K)$  where the definition of  $Problematic(K)$  is changed to  $Problematic^v(K) = \{\alpha \in K \mid v(\alpha) = false\}$ ,  $v$  being the Truth Assignment function.

### Global Inconsistency

After Total Inconsistency, a more thorough calculation is done. In some cases, the altered seed can have little to no impact on the gameplay. For example, if players are drawing from differently shuffled decks but none of them end up drawing the same tiles and the game finishes quickly. In other possible cases, players having different seed can have a huge impact on the gameplay as players drawing the same tiles would be going against the predetermined rules and structure of the game.

The importance of this next calculation is that it measures the impact of the seed alteration. This measurement consists of three parts.

- 1) **Shared Histories** - Comparing the players’ hand histories, to see if the players ever held the same tiles throughout the course of the game.

In the first part of the Global calculation, the inconsistency metric is increased by the size of the intersection between each unique player pair’s hand histories. This may accurately tell the extent to which the gameplay was affected for individual players, even if some of the tiles were not placed. The formula below is used to calculate the Shared Histories.

$$SH_{i,j} = \sum_{j < i \leq N} |H_i \cap H_j|$$

- 2) **Repeated Tiles** - Checking each player’s local board for duplicate tiles.

The second part adds an extra layer of measurements as it calculates the repeating tiles on the board. This involves going through each piece placed on the board and counting the

number of duplicate tiles that were placed. Inconsistency value is increased by the number of duplicates found on the board. This makes the inconsistency metric even more accurate as it portrays how the gameplay was affected for *all* players. This function can be defined as.

$$RT_i = \sum_{t \in Board_i} (\mu(t) - 1)$$

where  $t$  represents a unique tile on the board and  $\mu: Board_i \rightarrow \mathbb{N}$  is a cardinality function, used for counting the duplicate tiles on the board.

Because of the fact that, unlike each player's hand tiles, the tiles on the board were specifically used in the game, they have direct impact on the flow of the game. This affects the inconsistency metric of the game even more than just keeping duplicates in hands. Therefore, despite nearly all duplicates being accounted for in the hand history comparisons, the duplicates on the board are counted twice in the inconsistency metric of the game.

- 3) **Board Distance** - Calculating the Levenshtein distance [10] between the boards of each unique pair of players. This takes the possible difference between the players' local board states into an account.

The final calculation in the Global Inconsistency metric is the Board Distance. This distance is measured between the boards of each unique pair of players. This achieves the goal of comparing the players' boards to account for any failures to place tiles, any dropped messages, or in any other case that a player's local board could get desynchronized from the others. The distance calculation used here is the Levenshtein distance. Usually, this formula is used to calculate distances between two strings. It counts the minimum number of character changes that need to be applied to one string to convert it to another. In this case, the same logic is applied to the board object, which is effectively just a list of Domino Tiles. For each unique pair of players, their boards are compared using the Levenshtein distance and that is added to the Global Inconsistency.

$$BD_{i,j} = L(Board_i, Board_j)$$

$$I_G = + \sum_{i < N} RT_i + \sum_{j < i < N} (BD_{i,j} + SH_{i,j})$$

### Normalizing the Metric

Initially these inconsistency metrics were supposed to be normalized. It would involve measuring the average values of the calculations mentioned above, instead of just summing those inconsistency values up.

As an example, dividing the sum of the Shared History metric by the number of unique pairs, would produce the output that is uninfluenced by the number of players in the game.

Arguments can be made that this approach would be more beneficial as it becomes easier to compare the test results if they are tied to a specific range of values or reduce the influence of outer factors.

Despite that, the metrics are not normalized for this project. This is because not normalizing it emphasizes the impact the unfixed inconsistency can have on the Domino game and helps highlight the importance of fixing the inconsistencies early better. The Testing part of this project (see below) is going to explore this impact using the approaches mentioned above.

## Testing Approaches

This chapter contains all the testing done on the code part of the project. Each testing approach will first be explained and motivated briefly. The tests will cover multiple scenarios and will be controlled using Control Variables described below. The main focus of these tests will be to showcase the impact an uncaught inconsistency can have on the gameplay.

### Control variables

The tests done for this project consist of running the same programs multiple times over. While the code for each test run may be the same, there are certain global variables, that determine key aspects of the benchmarking, that will be changed between tests. They are:

**Integer variables:**

**Number of Processes** – Number, specifying how many execution threads are going to be created and used to test this program. These represent the players in the Domino game simulation. In this project this number will range from 2 to 7.

**Number of Cycles** – Number, only used for the Domino simulation. It specifies how many games will be launched one after another. This value will either be 1 or 100 when testing the Domino game.

**Drop/Alter Chance** – Number  $x$ , used to determine the chance a message can be altered or dropped. The actual probability of it is  $1/x$  or  $100\%/x$  for percentages. This number is set to 2 most of the time throughout testing, although there can be specific situations, where increasing or decreasing the chance for inconsistency would be beneficial for testing.

**Tiles in Hand** – Number of tiles in each player's hand at the start of the game. This measurement is exclusive to the Domino simulation. Changing the number of players may require to alter the hand sizes as well to keep the games shorter or force them to be longer. It also is necessary to alter this value to prevent players from drawing more tiles at the beginning of the game than there are in the deck.

**Min. Tile Number, Max. Tile Number** – In Domino, tiles have two numbers on them. In the most popular version of Domino those numbers are in the range  $[0, 6]$ . These two variables are the minimum and the maximum values that the tiles could have. The minimum value will not be changed from 0 throughout testing. The maximum value, however, will fluctuate in the range  $[4, 8]$ . Doing this, allows expanding the deck to fit more players and measure the inconsistency of more varied scenarios. The formula for calculating the size of the deck [6] is:

$$size(min, max) = \frac{(max - min)^2 + 3(max - min) + 2}{2}$$

Therefore, when min is constantly set to 0, it becomes:

$$size(max) = \frac{max^2 + 3max + 2}{2}$$



**Boolean variables:**

**Drop Message** – boolean value, setting it to “true” enables randomly dropping the messages. It was turned off for Domino testing due to time constraints and turned on for the random message passing approach.

**Alter Seed** - when set to “true”, it enables altering the seed at the beginning of the Domino game.

**Repair Seed** – when set to “true”, the program repairs the seed alteration by confirming it with other players through a channel that is unaffected by inconsistency.

**Alter Winner** – enables the random alteration of the winning messages. Whenever a player wins the game, this has a random chance to alter their message to remove the “I have won” part from it.

**Repair Winner** – this enables the code to account for these types of errors by having the winner send a “safe” message to everyone else when they have won. The rest of the players would carry on with the game as normal but would check their safe message channel for updates after not receiving any information from a player that has ended the game. This way, the players can update their winner information to fix their Total Inconsistency.

**Synchronize the Boards** – this control variable was implemented due to an oversight that had been noticed in the message alteration approach. In the initial Domino game implementation, after the starting hand was drawn, each player would reveal the top tile from the top of their local deck and place it as the starting piece on their local board. The problem with that is that if a player gets an altered seed, their initial tile would be completely different from the rest of the players and this would affect the entire game, as tiles that are placeable for one player would not be for the other. By the end of the game, the players’ boards would look completely different. Setting this variable to “true” switches the code from this simple “local first tile” approach to one where the first player broadcasts the first tile to the rest of the players, so they would all start out with the same board.

### Logic behind testing

The testing done on this project consists of 8 tests done on the Domino game. Each test measures the effect of the injected inconsistency under different parameters. In these tests, the inconsistency will not be fully resolved as the inconsistency metrics of a resolved game are always 0 and thus do not provide any new information. Instead, a partial fix of synchronizing the initial tile will be used as a comparison of the effects of the inconsistency.

The first four tests will be run on a single game each. This is to help the reader understand each step of the game and to better demonstrate the specific reasons for inconsistency increases. Later tests will run 100 Domino games each and will display the average tendencies of those games.

## Domino Test #1

**Key points:** 2 Players, Desynchronized boards, Low board inconsistency

**Aims:** Introduction to Domino simulation testing. Explanation of the game progress with minor inconsistencies.

**Relevant Test Parameters:**

Number of Players – 2	Alter Seed – true,
Tiles in Hand – 5	Alter Winner – false,
Min/Max Tile Number – [0, 4] (28 Tiles)	Synchronize the Boards – false

**Introduction:** The first Domino game test is meant to show, in detail, how the game operates. It shows the progress of the game when an inconsistency is injected. 2 Players were selected in order to have fewer turns and an easier to read game. During each turn, a player thread will print its own action and its board, after that action. These two parameters were deemed the most important to understand the gameplay without overcrowding the board. In this specific scenario, the players will start off with their initial tiles being different but similar. This will cause a minor inconsistency.

Below, a screenshot of the test output during the game will be displayed and, after that, a short summary of what is shown there. The summary includes a short analysis of moves that caused an inconsistency to appear. Then, the relevant post-game output is presented, mainly board comparisons and the calculated Total and Global inconsistency metrics. These will be described and explained as well.

**Output:**

```
Player0 sent the seed: 3560
Player1 received seed: 4191
```

*Figure 2. Domino Test 1. Seed difference*

The goal of this test is to showcase a simple game's progression when an inconsistency is injected. It can be seen that Player1 received an incorrect seed for this game.

```

P0 placed 0:3 in the front
[[3:0]] [[0:4]]

P1 placed 1:3 in the back
[[3:0]] [[0:3]] [[3:1]]

P0 placed 1:3 in the front
[[1:3]] [[3:0]] [[0:4]]

P1 drew
[[1:3]] [[3:0]] [[0:3]] [[3:1]]

P0 placed 1:1 in the front
[[1:1]] [[1:3]] [[3:0]] [[0:4]]

P1 drew
[[1:1]] [[1:3]] [[3:0]] [[0:3]] [[3:1]]

P0 placed 0:1 in the front
[[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:4]]

P1 placed 0:4 in the front
[[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:3]] [[3:1]]

P0 drew
[[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:4]]

P1 placed 2:4 in the front
[[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:3]] [[3:1]]

P0 placed 2:2 in the front
[[2:2]] [[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:4]]

P1 placed 0:2 in the front
[[0:2]] [[2:2]] [[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:3]] [[3:1]]

P0 placed 3:4 in the back
[[0:2]] [[2:2]] [[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:4]] [[4:3]]

```

Figure 3. Domino Test output. 2 Players with inconsistency

After 2 turns have passed, it is evident that the seed alteration affected the initial tile of the board. Player0 got the Tile `[[0:4]]` as the starting tile, while Player1 got `[[0:3]]`. Coincidentally, in the first turn, Player0 placed Tile `[[0:3]]`, which duplicates that tile in P1's board. On the next turn, Player1 places `[[3:1]]` at the back of the board (right) but it does not fit on P0's back of the board, therefore, Player0 drops that tile. This creates an even larger distance between players' boards. Later in the game, Player1 places Tile `[[0:4]]`. This increases the inconsistency metric, since `[[0:4]]` was the initial tile for P0.

In the end, P0 won this game by running out of tiles in hand. Because most of the tiles got placed in the front of the board, which remained synchronized for both players, the distance between boards remained mostly the same.

In the post-game output, a clearer comparison of the two players' boards is printed:

```
Boards:
P0: [[0:2]] [[2:2]] [[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:4]] [[4:3]]
P1: [[0:2]] [[2:2]] [[2:4]] [[4:0]] [[0:1]] [[1:1]] [[1:3]] [[3:0]] [[0:3]] [[3:1]]
```

Figure 4. Domino Test output 1. Board comparison

```
Hand Histories:
P0: [[0:1]] [[0:3]] [[1:1]] [[1:3]] [[2:2]] [[3:4]]
P1: [[0:0]] [[0:1]] [[0:2]] [[0:4]] [[1:3]] [[2:4]] [[3:3]]
```

Figure 5. Domino Test output 1. Hand Histories

From Figure 4, it can be seen that the left side of the board was completely synchronized. The difference in the boards is detected only when the initial tiles `[[0:4]]` and `[[0:3]]` are reached. Furthermore, it is worth noting that the winning tile `[[4:3]]`, placed at the back of the board by Player0, was not placed on Player1's board. However, the boards are still equal in length because P1 attempted to place tile `[[3:1]]` near the beginning of the game but it didn't fit P0's board and the tile was dopped as well. In this case the boards seem to be mostly consistent with each other and only have 1 repeating tile each.

After the game, the inconsistency metrics are calculated:

```
Total Inconsistency: 1
Global Inconsistency: 7
```

Figure 6. Domino Test output 1. Inconsistency metrics

Total inconsistency is simple to explain, as one player got the seed altered and no players had the winner information changed (that control variable is turned off for this test). Therefore, the Total inconsistency of this game was 1.

In other terms, the Knowledgebase for this specific run of the game was:

$$K = \{SameSeed(0,1), SameWinner(0,1)\}$$

Given that  $v$  is the current game's Truth Assignment function, the next step is to calculate its inconsistency by getting the number of atoms that return *false* when passed to a Truth Assignment function  $v$ .

$v(SameSeed(0,1)) = false$ , as  $0.seed = 3560 \neq 4191 = 1.seed$ , and

$v(SameWinner(0,1)) = true$ , as  $0.winner = P0 = 1.winner$ .

Since only one atom returned *false*, the Total inconsistency is:  $I_T(K) = 1$ .

The calculation for Global Inconsistency is more convoluted. It adds up all of the duplicate tiles from both players boards, hand histories and the Levenshtein distance. The board distance of 2 can be easily seen in Figure 4, as only the two rightmost tiles in the player's boards are different. The figure also shows one pair of repeated tiles on each board, therefore that metric will add 2 to the Global Inconsistency. Figure 5 shows the hand histories

overlapping by 2 tiles –  $[[0:1]]$  and  $[[1:3]]$ . All these values add up to the Global Inconsistency value of 7, which is correctly calculated and shown in Figure 6. The Global Inconsistency values throughout the game are displayed below in Figure 7. When pairing these values to the game turns, the increases can be seen relating to the game turns shown in Figure 3.

```
Global Inconsistency throughout the game:  
Turn 1: 3  
Turn 2: 4  
Turn 3: 5  
Turn 4: 5  
Turn 5: 5  
Turn 6: 6  
Turn 7: 6  
Turn 8: 7  
Turn 9: 7  
Turn 10: 7  
Turn 11: 7  
Turn 12: 7  
Turn 13: 7
```

*Figure 7. Domino Test Output 1. Global Inconsistencies in Turns*

## Domino Test #2

**Key points:** 4 Players, Desynchronized boards, High board inconsistency

**Aims:** Showcasing increasing effects the inconsistency can have in a Domino game when compared to previous test.

### Test Parameters:

Number of Players – 4	Alter Seed – true,
Tiles in Hand – 5	Alter Winner – false,
Min/Max Tile Number - [0, 6] (28 Tiles)	Synchronize the Boards – false,

**Introduction:** This test increases the number of threads, or players, participating in the game. Similar to Domino Test #1, this test is meant to showcase the game flow and the amount of network inconsistency when altering the seed on a game with unsynchronized boards. Unlike Domino Test #1, this test will not follow the game turn-by-turn and will not contain the same level of detailed analysis, as the previous test covered all of that thoroughly. Instead, this test will focus on showcasing the post-game output and metrics.

### Output:

```
Player0 sent the seed: 5172
Player3 received seed: 899
Player2 received seed: 5172
Player1 received seed: 9765
```

Figure 8. Domino Test Output 2. Player Seeds

Again, inconsistency is injected into the initial seed message that Player0 sends out. Figure 8 shows two players in this game receiving an incorrect seed.

```
Boards:
P0: [[1:4]] [[4:5]] [[5:2]] [[2:1]] [[1:5]] [[5:3]] [[3:1]] [[1:0]] [[0:6]] [[6:6]]
P1: [[0:0]] [[0:1]] [[1:5]] [[5:4]]
P2: [[1:4]] [[4:5]] [[5:2]] [[2:1]] [[1:5]] [[5:3]] [[3:1]] [[1:0]] [[0:6]] [[6:6]]
P3: [[0:0]] [[0:2]] [[2:2]] [[2:5]] [[5:5]]

Hand Histories:
P0: [[0:0]] [[0:1]] [[1:3]] [[2:5]] [[3:5]] [[6:6]]
P1: [[0:0]] [[1:2]] [[1:5]] [[2:3]] [[2:6]] [[3:3]] [[4:5]] [[5:6]]
P2: [[0:5]] [[0:6]] [[1:2]] [[1:4]] [[3:3]] [[4:5]] [[5:5]]
P3: [[1:1]] [[1:4]] [[2:2]] [[2:5]] [[3:4]] [[5:5]]
```

Figure 9. Domino Test 2. Boards and Hand Histories

Post-game output (Figure 9) shows the four players' boards and hand histories. The difference in the boards of the players, who had inconsistency injected, is clear. In the beginning of the game, players 1 and 3 started their boards with different tiles than P0 and P2. This meant they

could not place every tile the other players have sent them in a message, leading to their boards ending up being much smaller. This greatly affects the Global Inconsistency metric.

```
Total Inconsistency: 5
Global Inconsistency: 49
```

Figure 10. Domino Test output 2. Inconsistency metrics

Figure 10 shows the Total Inconsistency of this test being 5. This is due to the fact that 2 out of 4 players got an inconsistency injected at the start of the game. As mentioned before, Total Inconsistency is calculated by comparing the seed and the winner of each unique pair of players. In this case, with 4 players, there were 6 unique pairs – {P0, P1}, {P0, P2}, {P0, P3}, {P1, P2}, {P1, P3}, and {P2, P3}. Because only {P0, P2} pair of players had identical seeds and the winner information was not altered in this game run,  $I_T(K) = 6 - 1 = 5$ .

The Global Inconsistency metric for this test is much higher than in the previous tests. Introducing more players and more inconsistency into the game can highly increase this number. It is clear from the boards and hand histories shown above that the players were completely out of sync from each other. Below is a graph, marking the increase in the Global Inconsistency throughout the game. On the other hand, if the error is detected in early stage of the game (as soon as the wrong seed is detected), then later stage of the games will not be affected by the inconsistency increase. This suggests that the earlier the errors are caught, the better, as the players will experience less diverged game experiences, and also suggests that an increase number of players will considerably increase the overall amount of errors within the game, as many possible faults might have happened.

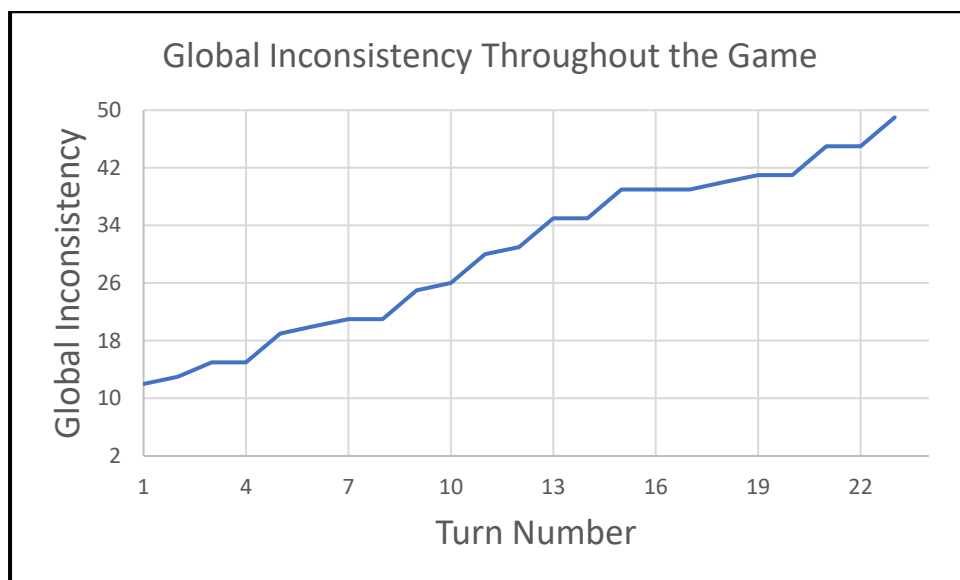


Figure 11. Domino Test output 2. All Global Inconsistency values

### Domino Test #3

**Key Points:** 4 Players, Synchronized Boards, Low Inconsistency

**Aim:** Showcasing the effects of synchronizing the boards with injected seed inconsistency.

**Test Parameters:**

Number of Players – 4

Alter Seed – true,

Tiles in Hand – 5

Alter Winner – false,

Min/Max Tile Number - [0, 6] (28 Tiles)

Synchronize the Boards – true

**Introduction:** Due to the board desynchronization, the game in the previous test case was borderline unplayable. Players getting a different board setup before the game even starts ruins the experience from the beginning and detach players from an experience that is meant to be shared. Because of this, a partial solution to resolving inconsistencies was added into the project. Synchronising the initial tile on the board, by having a chosen player broadcast it to everyone, instead of each player choosing it themselves

This approach shows how partial fixes to inconsistency resolution can mitigate the gameplay errors. As per previous discussions, this is going to be quantified by our global inconsistency metric.

To showcase that, this test will produce the output of a game that synchronizes the first tile placed on the board, despite players having a seed inconsistency. By all players placing the same tile at the start of the game and under this project's rule that the tiles placed by players will not be altered when sent in network messages, this approach completely removes the difference between the player's boards. It is safe to assume that this would reduce the inconsistency metric and could vastly improve the players' experience, despite their decks still being shuffled completely differently.

**Output:**

```
Player0 sent the seed: 606  
Player3 received seed: 977  
Player2 received seed: 8210  
Player1 received seed: 4397
```

*Figure 12. Domino Test output 3. Seed inconsistency*



```

Boards:
P0: [[4:2]] [[2:2]] [[2:3]] [[3:0]] [[0:5]] [[5:5]] [[5:5]] [[5:4]] [[4:3]] [[3:3]] [[3:1]] [[1:4]] [[4:6]] [[6:6]] [[6:6]] [[6:4]] [[4:2]]
P1: [[4:2]] [[2:2]] [[2:3]] [[3:0]] [[0:5]] [[5:5]] [[5:5]] [[5:4]] [[4:3]] [[3:3]] [[3:1]] [[1:4]] [[4:6]] [[6:6]] [[6:6]] [[6:4]] [[4:2]]
P2: [[4:2]] [[2:2]] [[2:3]] [[3:0]] [[0:5]] [[5:5]] [[5:5]] [[5:4]] [[4:3]] [[3:3]] [[3:1]] [[1:4]] [[4:6]] [[6:6]] [[6:6]] [[6:4]] [[4:2]]
P3: [[4:2]] [[2:2]] [[2:3]] [[3:0]] [[0:5]] [[5:5]] [[5:5]] [[5:4]] [[4:3]] [[3:3]] [[3:1]] [[1:4]] [[4:6]] [[6:6]] [[6:6]] [[6:4]] [[4:2]]

Hand Histories:
P0: [[0:1]] [[1:4]] [[2:4]] [[2:6]] [[4:5]] [[4:6]]
P1: [[0:1]] [[1:3]] [[2:3]] [[5:5]] [[5:6]] [[6:6]]
P2: [[0:5]] [[2:4]] [[3:4]] [[4:6]] [[6:6]]
P3: [[0:0]] [[0:3]] [[1:6]] [[2:2]] [[3:3]] [[5:5]]

```

Figure 13. Domino Test output 3. Boards and Hand Histories

After the game has finished, it can be clearly seen that, despite all players getting different seeds, their boards look completely the same. While they still have repeating tiles on the boards and their hand histories still overlap by some tiles, all players have experienced a more consistent game. Unlike in previous tests, the game allowed each player to see all of the tiles that an opponent placed on the board and guaranteed that all those tiles fit on every other player's board. In other words, when all boards are synchronized, the game becomes more playable by consistently following through on actions that the users take. This is reflected by the inconsistency metrics as well (Figure 14).

```

Total Inconsistency: 6
Global Inconsistency: 21

```

Figure 14. Domino Test output 3. Inconsistency metrics

The post-game output shows a Total Inconsistency value of 6 and a Global Inconsistency value of 21. As mentioned in Domino Test #2, a 4-player game has 6 unique pairs of players. Because all 4 players had a different seed from each other, there was no pair of players whose seeds were identical and, therefore, the Total Inconsistency metric is correctly calculated at a 6.

Despite the Total Inconsistency of this game being the highest it could be in a 4-player game, where only the seed was altered, the Global Inconsistency metric is relatively low when compared to the previous test. Due to the fact that the boards were synchronized, there occurred no unfit tile dropping and so, the Global Inconsistency was only affected by the duplicate tiles on the boards or drawn by different players.

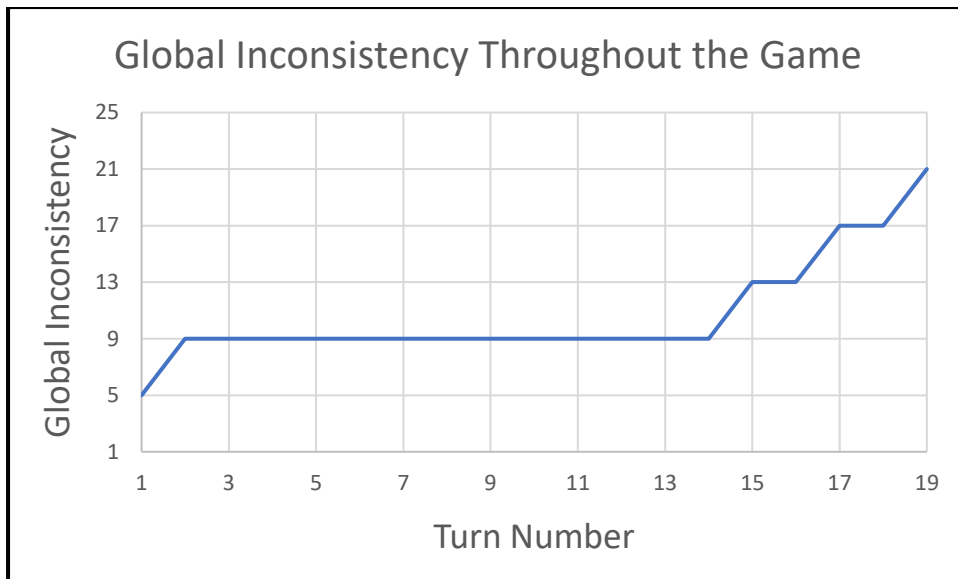


Figure 15. Domino Test output 3. All Global Inconsistency values

Figure 15 shows the gradual increase in the Global Inconsistency throughout the game. Interestingly, it can be seen that after the first turn the inconsistency only increases in increments of 4. This can only be caused by players placing a tile that already exists on the board, as the inconsistency calculation adds up the duplicate tiles in **all** players' boards, therefore the effect that one duplicate tile placed on the board has on the Global Inconsistency metric is equal to the number of players, receiving that tile as a duplicate. In this case, because the boards have no difference between themselves, the effect of a duplicate tile on the board is 4.

Figure 15 shows that after the first turn, the inconsistency value was 5. Because of that, it is safe to assume that all five of the duplicate tiles, seen in the players' hand history (Figure 13), were drawn in the initial draw before the start of the game, when all players drew their starting hands.

## Domino Test #4

**Key Points:** 4 Players, Winning message altered, High Impact, Multiple Winners

**Aims:** Showcasing the possible effects of the game-winning message getting altered.

**Test Parameters:**

Number of Players – 4

Alter Seed – false,

Drop/Alter Chance – 2 (50%)

Alter Winner – true

Tiles in Hand – 5

Min/Max Tile Number - [0, 6] (28 Tiles)

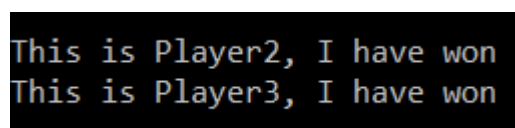
**Introduction:** Until now, all of the tests were focused on analysing what happens when there is an inconsistency at the very start of the game – shuffling the deck or placing the initial tile. The tests measured the long-term impact an inconsistency, that was not caught early, had on the entire game. However, there is another type of inconsistency that is used in this project – altering the winning message. This inconsistency is injected at the very end of the game but can still have a large effect on the game itself.

As stated previously, when a player places their final tile left in the hand, along with the action information another field is sent in a message. The most straightforward way to describe this field would be that it is the “I have won” field. Before sending their winning turn information, the player sets this message field to *true*. Inconsistency is injected by changing this field back to *false* so that a player receiving it would not know to stop playing the game.

This test describes a scenario where altering the winning message can have a relatively high impact to the game. In this test, the seed will not be altered for any player, therefore it is expected to not see any inconsistency at the start of the game. However, when a player wins a game there will be a 50% chance for each player to not receive the message information, stating that the player has won and has already quit the game. In the specific scenario described below, no players will receive any winning messages and there will be two back-to-back “winners”.

**Output:**

The game is started and, after some time, these two lines (Figure 16) are printed. A few seconds later, the game ends.



```
This is Player2, I have won
This is Player3, I have won
```

Figure 16. Domino Test output 4. Winning output

Normally, each player that has received the “I have won” message should print out a confirmation of the winner. In this case, not only did no player print out a message confirming

Player2 as a winner, but Player3 announced their own victory as well – with no one else confirming this one either.

```
Boards:
P0: [[0:2]] [[2:2]] [[2:3]] [[3:1]] [[1:5]] [[5:2]] [[2:6]] [[6:4]] [[4:2]] [[2:1]] [[1:6]] [[6:3]]
P1: [[0:2]] [[2:2]] [[2:3]] [[3:1]] [[1:5]] [[5:2]] [[2:6]] [[6:4]] [[4:2]] [[2:1]] [[1:6]] [[6:3]]
P2: [[0:2]] [[2:2]] [[2:3]] [[3:1]] [[1:5]] [[5:2]] [[2:6]] [[6:4]] [[4:2]] [[2:1]] [[1:6]] [[6:3]]
P3: [[0:2]] [[2:2]] [[2:3]] [[3:1]] [[1:5]] [[5:2]] [[2:6]] [[6:4]] [[4:2]] [[2:1]] [[1:6]] [[6:3]]
```

Figure 17. Domino Test Output 4. Boards [continued]

```
[[6:3]] [[3:3]] [[3:0]] [[0:4]] [[4:1]] [[1:1]] [[1:0]] [[0:5]] [[5:5]] [[5:6]] [[6:6]] [[6:0]] [[0:0]]
[[6:3]] [[3:3]] [[3:0]] [[0:4]] [[4:1]] [[1:1]] [[1:0]] [[0:5]] [[5:5]] [[5:6]] [[6:6]] [[6:0]] [[0:0]]
[[6:3]] [[3:3]] [[3:0]] [[0:4]] [[4:1]] [[1:1]] [[1:0]] [[0:5]] [[5:5]] [[5:6]] [[6:6]]
[[6:3]] [[3:3]] [[3:0]] [[0:4]] [[4:1]] [[1:1]] [[1:0]] [[0:5]] [[5:5]] [[5:6]] [[6:6]] [[6:0]]
```

Figure 18. Domino Test output 4. Boards [continuation]

```
Hand Histories:
P0: [[1:6]] [[2:4]] [[3:3]] [[3:4]] [[4:4]] [[4:6]] [[5:6]]
P1: [[0:0]] [[0:2]] [[0:3]] [[0:5]] [[1:3]] [[3:5]] [[3:6]] [[4:5]]
P2: [[0:4]] [[1:2]] [[1:4]] [[2:5]] [[5:5]] [[6:6]]
P3: [[0:1]] [[0:6]] [[1:1]] [[1:5]] [[2:2]] [[2:3]]
```

Figure 19. Domino Test output 4. Hand Histories

Figures 17 and 18 show each player’s boards. As the seed was not altered for this test, there are no tile duplications neither in the players’ boards (Tile [[6:3]] repeating in Figures 17 and 18 only marks continuation of the board between two figures rather than a duplicate tile), nor hand histories. However, the boards of players P2 and P3 are different from each other’s and from Players P0 and P1.

This was caused by the alteration of the winning message. Player2 quit the game instantly after placing down their final tile while other players were unaware of that fact and continued playing. The next turn, that same issue occurred to Player3, as they placed down their final tile in their hand and assumed they won the game. P0 and P1 continued playing the game normally again as they were also unaware of the other players quitting. After both P0 and P1 had completed their turns, they got stuck waiting for P2 to do their turn.

Because the game is coded to force-quit after 10 seconds of inactivity, the threads exited eventually and the inconsistency was calculated. Because of the force-quit, both Player0 and Player1 “think” that there was no winner to this game.

```
Total Inconsistency: 5
Global Inconsistency: 7
```

Figure 20. Domino Test output 4. Inconsistency values

Because the seeds were not altered, the Total Inconsistency for this test case was calculated only using the *winner* information. Out of six unique pairs of players in this game only Player0 and Player1 agreed on the winner, even though they were in the wrong. Based on the calculation, established in this paper, the Total Inconsistency metric for this test is 5.

This case is interesting because the only two players that do not have local inconsistency between themselves are still incorrect with their result. However, in author's opinion, that is not vital in an inconsistency calculation.

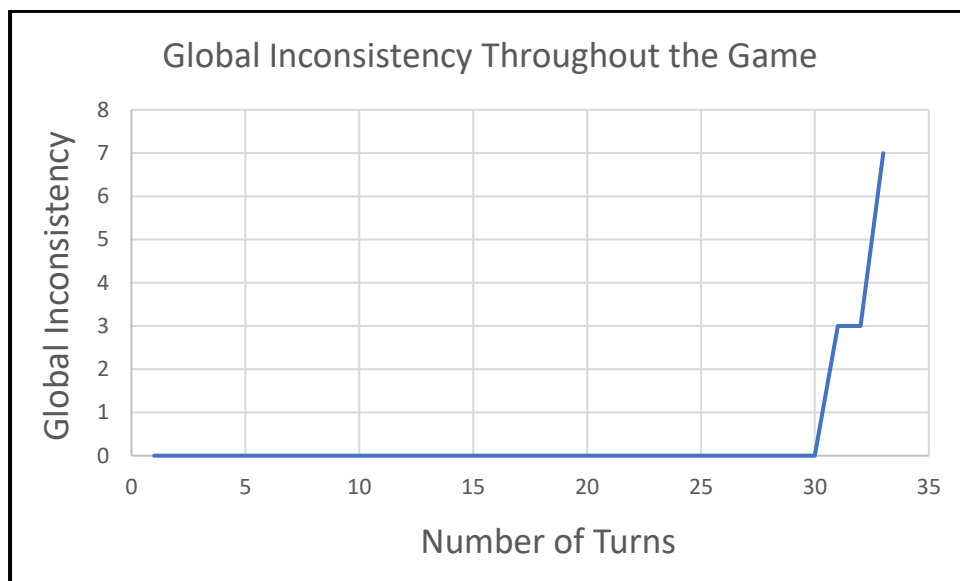


Figure 21. Domino Test output 4. All Global Inconsistency values

The Global inconsistency only starts rising from 0 to 3 at the end of the game, when Player2 wins but Player3 puts another tile on the board. And then again when P3 wins and P0 and P1 are left to play one more turn each. Since, during their last turn, P0 was forced to draw, the difference in the players' boards is smaller than it could have been.

## Domino Test #5

**Key Points:** 4 Players, 100-game average, Synchronised boards.

**Aims:** Displaying the average global inconsistency of 100 games at every turn. Measuring the average number of turns it takes to complete a game.

**Test Parameters:**

Number of Players – 4	Alter Seed – true
Number of Cycles – 100	Alter Winner – true
Drop/Alter Chance – 2 (50%)	Synchronize the Boards – true
Tiles in Hand – 5	
Min/Max Tile Number - [0, 6] (28 Tiles)	

**Introduction:**

The previous tests all focused on analysing the flow and output of a single game, however, any of those games could have been outliers that did not produce an output that a normal case for that test would.

For that reason, this test focuses on calculating the average Global Inconsistency of 100 games. In this test 100 Domino games have been run, using the same parameters. The Global Inconsistency of each game was recorded. Furthermore, the Global Inconsistency of each turn was also be recorded. This test calculates the average inconsistency of a 4-player Domino game, that uses the parameters above, and showcases the flow of the average Global Inconsistency throughout the turns.

All of the output for this and future tests will be uploaded into a spreadsheet and turned into a graph.

**Output:**

After launching 100 Domino games, the average inconsistency of A 4-player game with seed and winning message alteration was calculated to be **19.34**. This calculation included the games that were not injected with any inconsistency or were not affected by the injected inconsistency at all. Below is a graph containing the average game inconsistency after every turn.

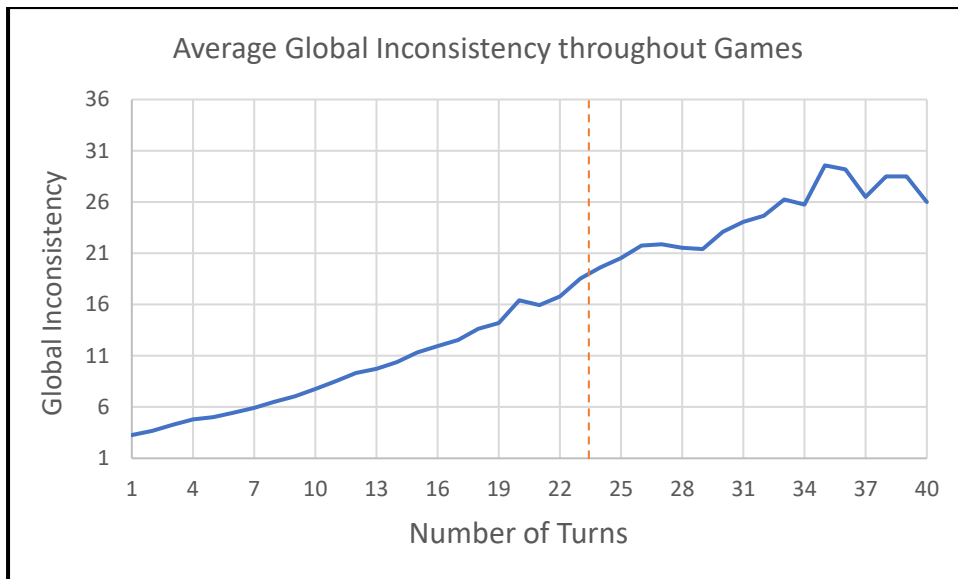


Figure 22. Domino Test output 5. Average Global Inconsistency throughout 100 games

The blue line graph in Figure 22 shows the average Global Inconsistency at every turn of all 100 games that were tested. It can be seen steadily going up throughout the course of the games and reaching its peak at Turn 35 with the value of **29.5**.

The orange dashed vertical line marks the average number of turns the games took to complete (**23.42**). This is relevant because not all of the 100 games took the same number of turns, meaning that in the latter turns a single game holds more weight on the average value than in the earlier. This explains why the graph values fluctuate up and down on the vertical axis.

For example, the turn after a game with a relatively high inconsistency value finishes, the average inconsistency might go down, as that value is not counted in the graph anymore. This can be seen once before the orange line is crossed, between turn 20 and 21, and multiple times after.

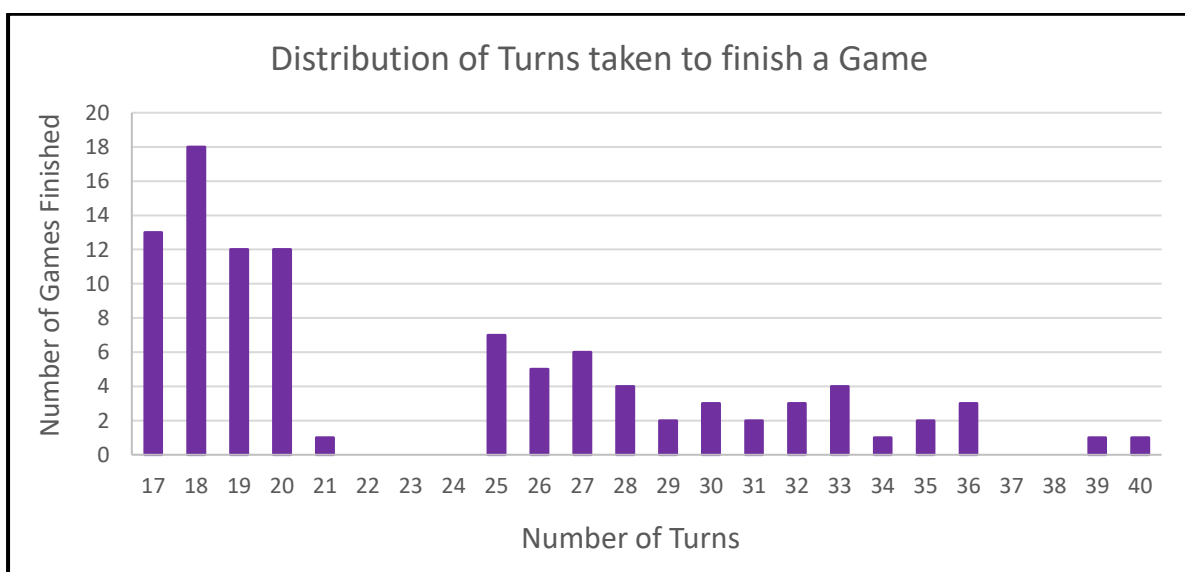


Figure 23. Domino test output 5. Distribution of Turns taken to finish a Game

Figure 23 depicts the distribution of turns it took to finish a Domino game. It can be seen that most games took between 17 and 20 turns, with the highest number of games, 18, taking 18 turns to finish.

It is worth noting that it is impossible to end a 4-player game in less than 17 turns under the conditions this test ensures, as the fastest way to finish a game would be for the first player to place all 5 of their tiles on the board in their first five turns. This further testifies to the correctness of this implementation.

On the other side of the graph, it can be observed that only two games took longer than 36 turns. That means that the final few turns of the graph depicted in Figure 22 were calculated using the average of only two games and the dip on the final turn was due to one game finishing and only one being left.



## Domino Test #6

**Key Points:** 4 Players, 100-game average, Desynchronized boards

**Aims:** Calculating the average global Inconsistency. Measuring the average number of turns required to finish a game. Comparing results with Domino Test 5

**Test Parameters:**

Number of Players – 4	Alter Seed – true
Number of Cycles – 100	Alter Winner – true
Drop/Alter Chance – 2 (50%)	Synchronize the Boards – false
Tiles in Hand – 5	
Min/Max Tile Number - [0,6] (28 Tiles)	

**Introduction:** After measuring the average inconsistency of 100 4-player games with the players' boards synchronized, it would be interesting to compare these results with a desynchronized approach.

It is interesting because this would measure the average impact that an uncaught early inconsistency can have on a game. Furthermore, comparing it with the average metric of games with a partially resolved inconsistency should help highlight the importance of reducing inconsistencies and their negative effects on the gameplay experience.

In this test another 100 4-player games are run using the parameters shown above. The output produced by each of these games is measured for the average Global inconsistency per turn and the average turn number, taken to complete a game. At the end of the test, the Global Inconsistency values are directly compared to the results of Domino Test 5 to better illustrate the impact synchronising the boards has on this game.

**Output:**

After launching 100 Domino games, the average inconsistency of A 4-player game with seed and winning message alteration was calculated to be **42.01**. That is more than twice of the previous test, further indicating that synchronizing the boards has a positive effect on the gameplay experience and helps reduce the spread of perceived inconsistency throughout the flow of the game. Below a graph illustrating the average Global Inconsistency of each turn is shown.

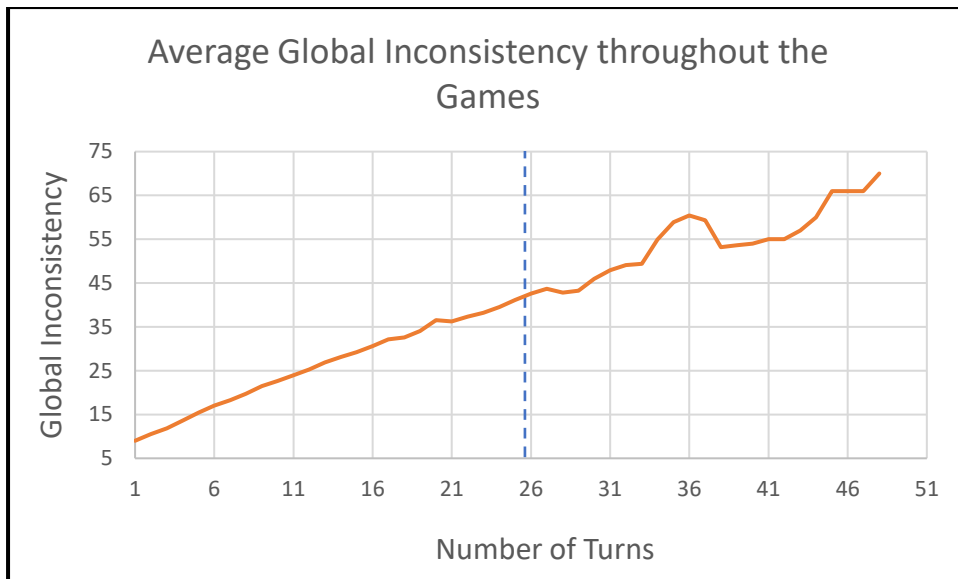


Figure 24. Domino Test output 6. Average Global Inconsistency throughout 100 Games

Figure 24 shows the average Global inconsistency for each turn number in the 100 games. It is illustrated by the orange line graph that can be seen steadily going upwards. The blue dashed vertical line illustrates the average number of turns it took to complete a game (25.62). Just like in the previous test, this line is placed there to symbolize the effect a single game has on the average inconsistency calculation. After crossing this point, the orange line is observed to lose its relatively consistent upward movement and to begin wavering up and down. Figure 25 helps in explaining this behaviour.

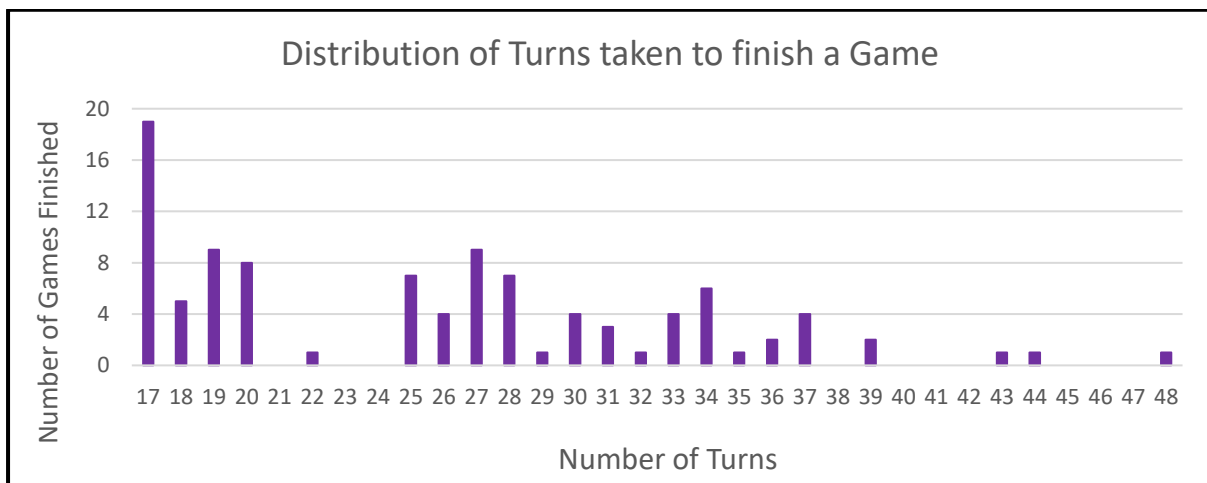


Figure 25. Domino Test output 6. Distribution of Turns taken to finish a Game

The figure above displays how many games took a specific number of turns to finish. It can be seen that the vast majority of games took between 17 and 28 turns to end. Specifically, when adding the values up, it can be calculated that 51% of the games had finished before or on turn 26. This further reinforces the importance of the blue vertical line in Figure 24.

In that figure the wavering of the inconsistency value between turns 33 and 38 can be explained by looking at the bar chart above. It can be seen that many games finish within that range of turns and only 5 are left after turn 37. This means that when losing a great percentage of the remaining games, the average calculation changes wildly, as the remaining games hold more weight within it. After turn 38 the graph seems to smoothen out and continue rising more calmly, meaning that the remaining few games had similar values and did not affect the calculation as much.

The figure below compares the results of this test with the results of Domino Test 5.

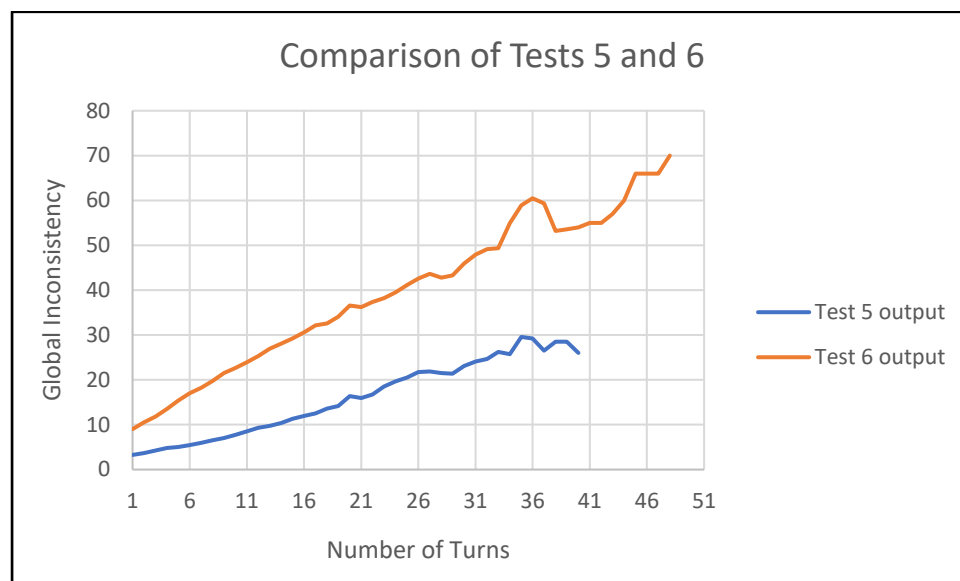


Figure 26. Domino Test output 6. Comparison between Domino Tests

From this figure, the impact of the board desynchronization between players is clearly visible. By turn 16 the desynchronized games have reached an average inconsistency equal to the peak average inconsistency the synchronized games only reached by turn 35. It is worth noting that at turn 16 no games have completed yet, meaning that that measurement is a "true" average, that would occur in 4-player games, and not an outlier.

## Domino Test #7

**Key Points:** 7 players, 100-game average, Synchronized boards

**Aims:** Showcasing the average inconsistency of games with bigger decks and more players. Calculating the average number of turns needed to complete a larger game.

### Test Parameters:

Number of Players – 7

Alter Seed – true

Number of Cycles – 100

Alter Winner – true

Drop/Alter Chance – 2 (50%)

Synchronize the Boards – true

Tiles in Hand – 5

Min/Max Tile Number - [0, 8] (45 Tiles)

**Introduction:** Similar to tests 5 and 6 this test measures the inconsistency metric and the average number of turns taken to finish for 100 Domino games. Most previous tests used a relatively normal set of Domino tiles [6-6], that is most commonly used worldwide. However, this test measures the performance of a larger game of Domino, using an [8-8] set of tiles or, in this project's terms – tiles with numbers ranging from 0 to 8. Here the inconsistency metrics are measured for a larger game to see what effect increasing the scale can have. It is expected to see an increase in the average Global Inconsistency values, as the increased scale provides more chances for possible errors to occur.

### Output:

As usual, 100 games have been played and their average inconsistency values displayed in the graph below. The average final turn inconsistency for this set of games was **54.8**.

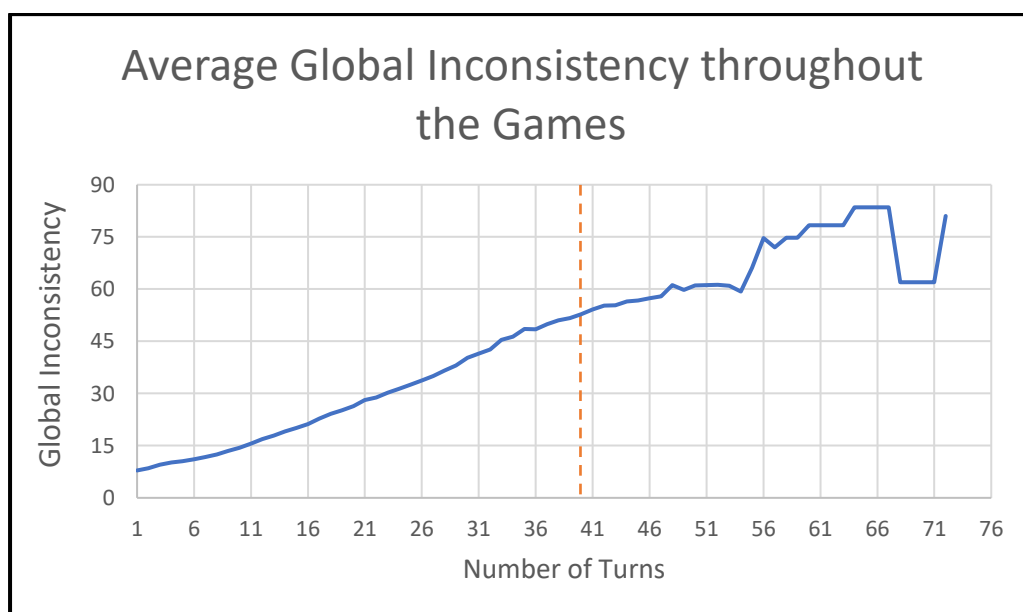


Figure 27. Domino Test output 7. Average Global Inconsistency throughout 100 Games

As with previous tests, the graph portrays a steady rise of Global Inconsistency throughout the games. An orange dashed vertical line, marking the average number of turns taken to complete a game (**39.93**), can be seen in the graph as well. After crossing this line, the average inconsistency value starts plateauing. This lasts for roughly twelve turns and, after that, it starts frantically wavering due to shift in weight a game has in the average calculation.

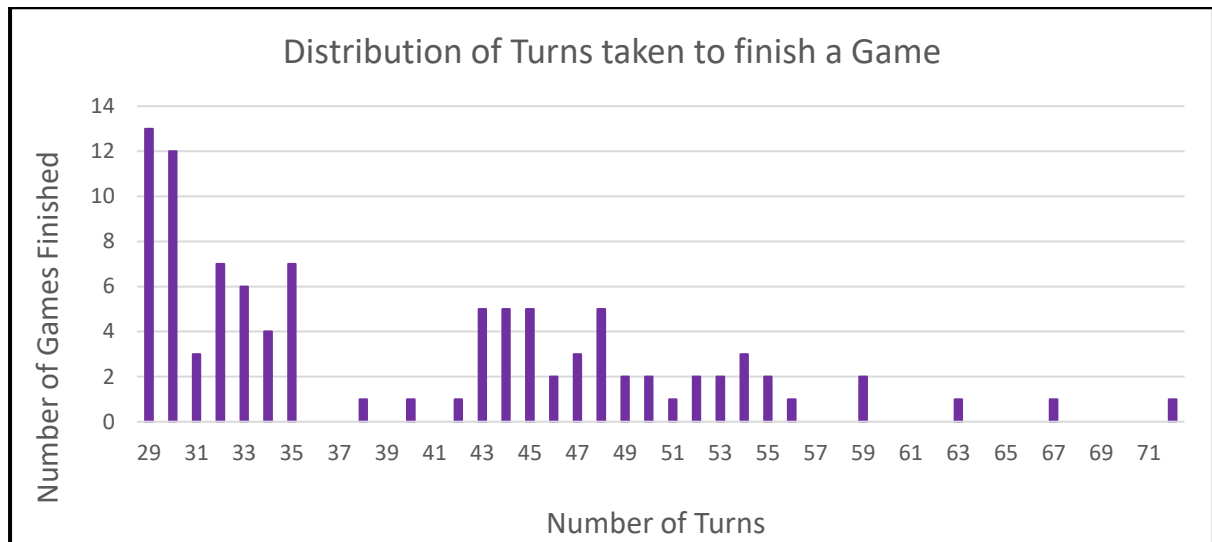


Figure 28. Domino Test 7. Distribution of Turns taken to finish a Game

Figure 28 illustrates a distribution of turns taken to finish a game. This is similar to the previous 4-player tests as the vast majority of games are shown to have ended early and only a very small percentage of games last for a longer time.

Just like the 4-player games have to last at least 17 turns, a 4-player game with 5 tiles in the starting hand has to take at least 29 turns. The formal calculation for this metric would be

$$f(n, h) = n * (h - 1) + 1, \quad \text{where}$$

$n$  – number of players  
 $h$  – number of tiles in the starting hand.

It is also interesting to note that in Figure 27 the average inconsistency value can be seen dropping harshly after turn 67, before harshly rising back up after turn 71. By simply looking at the inconsistency graph, it would be fair to assume that a low-inconsistency game finished at turn 71, leading to the average value of the remaining games to spike up. However, Figure 28 depicts no such game and, moreover, that only one game lasted more than 67 turns. This means that the average inconsistency values of turns 68-72, including the spike at the end, were caused by a single game. This game is a definite outlier to the rest of the testing data. It would be safe to assume that the massive spike of inconsistency is caused by the game's winner information being altered. This would cause a there to be a distance between player's boards, similarly to what was observed in Domino Test 4, and, if the tile that was placed during that turn was a duplicate, that would increase the Global Inconsistency even more.

## Domino Test #8

**Key Points:** 7 players, 100-game average, Desynchronized boards

**Aims:** Measuring the effects of desynchronizing the boards in larger games. Calculating the average turns taken to complete a game. Comparing the results with the previous test.

**Test Parameters:**

Number of Players – 7	Alter Seed – true
Number of Cycles – 100	Alter Winner – false
Drop/Alter Chance – 2 (50%)	Synchronize the Boards – false
Tiles in Hand – 5	
Min/Max Tile Number - [0, 8] (45 Tiles)	

**Introduction:** Similarly to the previous tests, this test will measure the negative effects that having desynchronized boards can have on the Domino games by calculating the average Global Inconsistency for each turn played. It will also calculate the average number of turns a game took to finish and display the distribution of the turns that games have taken.

However, seeing these tests, specifically 5, 6, and 7, it can be easy to assume that a longer game always means a more inconsistent game. This is especially true when looking at the average upward trend of the inconsistencies throughout the game. And while the inconsistency of a single game can only rise and not go down, in many of the test cases covered so far, the inconsistency graph begins to waver and fluctuate up and down towards the end.

This is explained as just having games with lower inconsistency that take more turns in the game data. However, having a way to see the distribution of inconsistency values throughout games, based on the number of turns they took to complete, would be useful to support this theory. This final test will introduce this type of output and will cover the results behind it.

Finally, just how Domino test 6 contained a comparison of its data to the data of Domino Test 5, this test will contain a comparison between its inconsistency values and the values from Domino Test 7. This will illustrate the expectedly negative impact an early uncaught inconsistency has on the flow of larger games between more players.

**Output:**

The test begins from running 100 games with the parameters listed above. For each of the 100 games the Global Inconsistency value is calculated after every turn. The average Global Inconsistency value for a single game in this test was **217.98**. As expected, this value is much higher than any previous test. However, while this value in Test 6 was around twice the size of the same value in Test 5, this test provided an average Global Inconsistency value nearly four times greater than it was in Domino Test 7. This further solidifies the importance of

reducing inconsistencies as early as possible, as the increase in game's scale massively increased the difference in inconsistency between equivalent game runs.

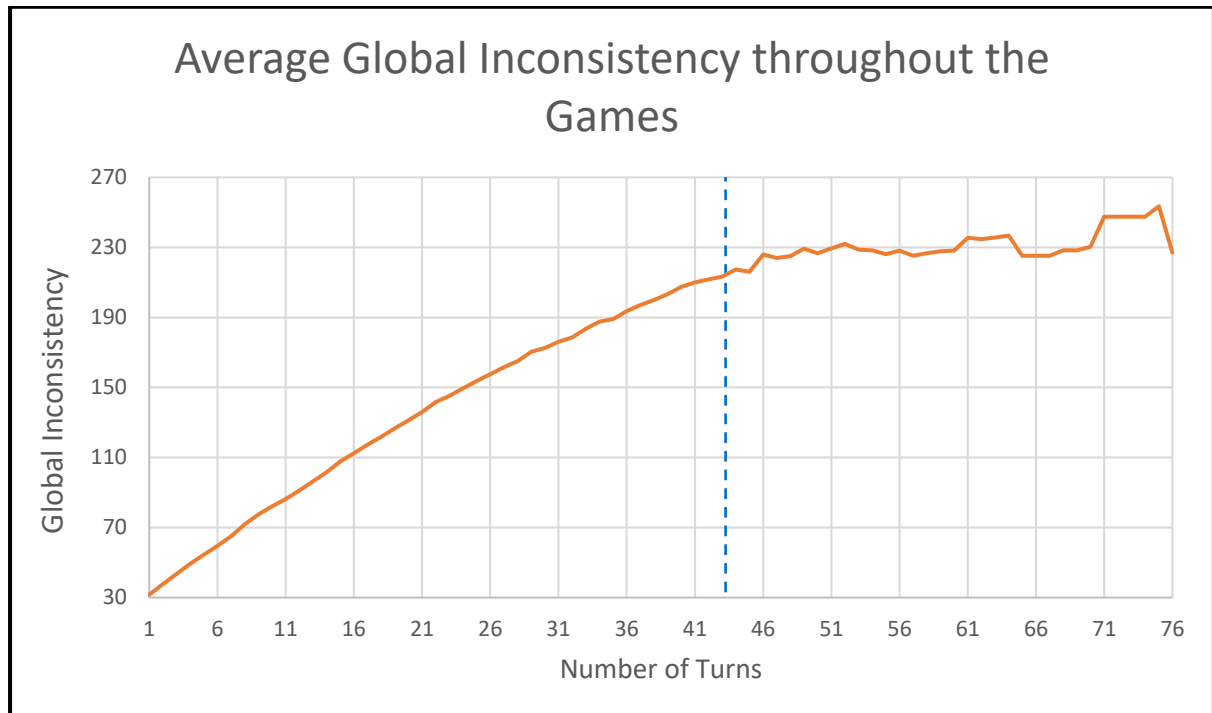


Figure 29. Domino Test output 8. Average Global Inconsistency throughout 100 games

Figure 29 shows the steady increase in the average Global Inconsistency. As with the tests seen before, the graph starts fluctuating more and more after crossing the line representing the average turn value it took to finish a game (**43.25**).

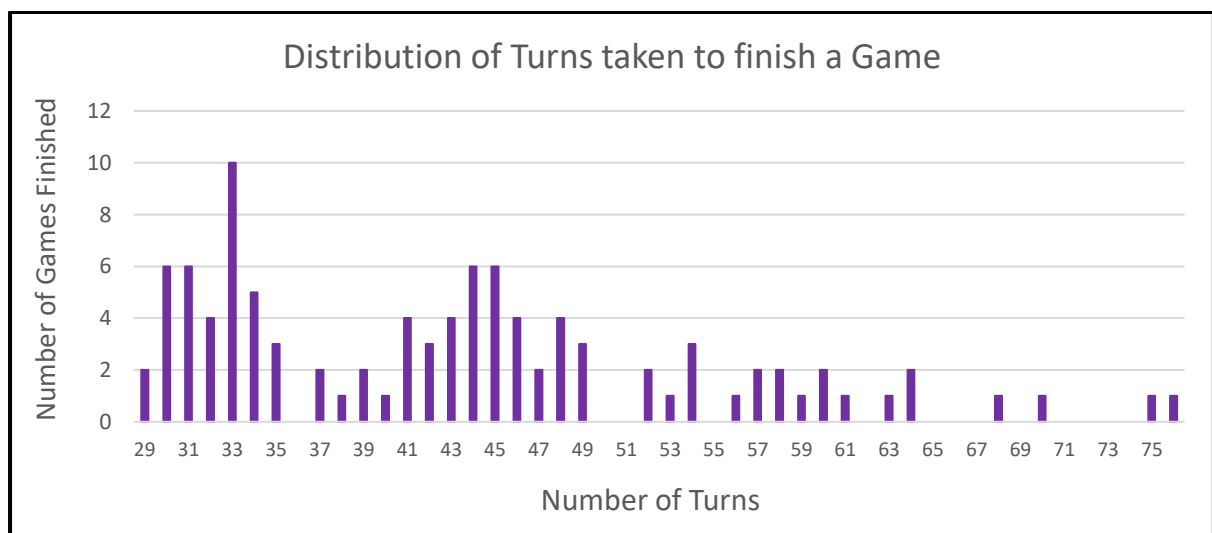


Figure 30. Domino Test output 8. Distribution of Turns taken to finish a Game

Figure 30 illustrates the distribution of turns taken to finish a game. As with previous tests, most of the games were shorter. However, in this case there are surprisingly few games that have finished in 29-32 games. This is much lower than the equivalent metrics in other tests. As 29 turns are the minimum a game may take, an assumption to explain these results could

be that in a lot of short games the winning message alteration had kicked in and expanded the game by up to six turns.

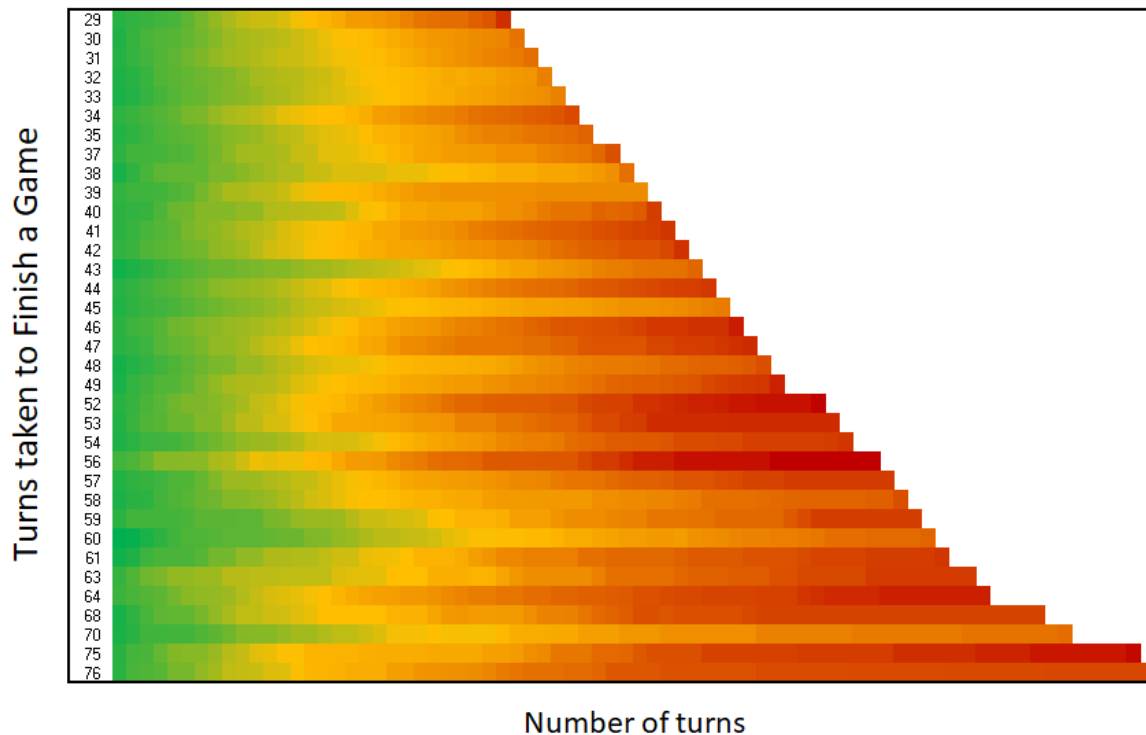


Figure 31. Domino Test output 8. The inconsistency of games, categorized by their length

As mentioned in the introduction of this test, Figure 31 illustrates the average inconsistency values of the games, measured in this test, categorized by the number of turns they took to finish. The number next to each row in this heatmap represents the category of turns a game lasted. Each row next to the number has that number of coloured squares in it. The “heat” of each squares represents the Global Inconsistency value per that category per that turn. The darker the square is, the larger inconsistency it represents.

In this heatmap, the values of each turn appear to be rising, as expected. Comparing this figure with Figure 30 we can observe that many of the outliers seen in the heatmap are categories, that do not contain many games within them. However, even when checking the categories that have multiple games in them, like 33, 44, and 45, not many observations can be made. However, disregarding a few outliers, it does appear as if there is a tendency of the longer turns having a larger inconsistency. In theory, this assumption would hold true, as the more turns the game lasts, the more chances for inconsistency there would be.



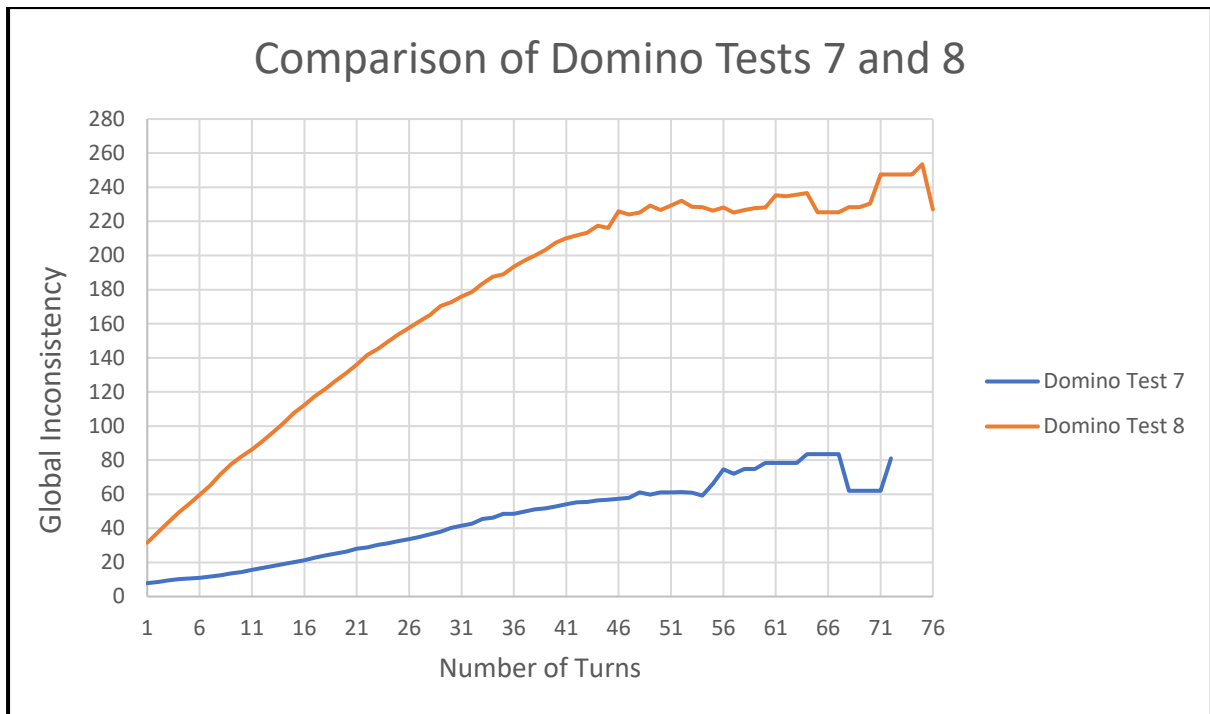


Figure 32. Domino Test output 8. Comparison of tests 7 and 8

Finally, the Figure 32 displays the difference in the increase of the inconsistency throughout games from test 7 and test 8. The inconsistency of the unsynchronized boards can be seen being vastly higher than when using the synchronized approach. the difference between the values of these approaches is so grand that Test 8's values overtake the highest inconsistency value reached by Test 7 roughly after Turn 11.

Comparing this with the results gotten from the Test 5 and 6 comparison solidifies the statement that, when increasing the game's scale, without even partially resolving the possible inconsistencies, the game's inconsistency metric will be disproportionately increased.

## Evaluation of the results

Tests 1-4 have explored the inconsistency effects and calculation in detail. Looking at Domino Test #2 it can be seen that an uncaught inconsistency early in the game can lead the game to be borderline unplayable. Comparing this to a partially fixed scenario, presented in Test 3, we can see the difference even a partial inconsistency resolution can make on the gameplay.

By synchronizing the player boards and no longer giving the players an opportunity to place tiles that cannot be placed on other players' boards, it fixed the incoherent flow of the game. Fixing this issue also made the game more consistent from the perspective of network inconsistencies as well as in the sense of following through on the actions the players take and not discarding them without their knowledge. As talked about in the test, despite the leftover inconsistencies, partial inconsistency resolution made the game playable by making sure that the action a player takes on their local state will be consistently transmitted to other players.

The impact of partial inconsistency resolution is further explored in Domino tests 5-8. Here it is observed how much of a difference the fixes can make of the average inconsistency metric of a game. We see that with an attempt to partially resolve the inconsistency, the average Global inconsistency metric significantly dropped.

In addition to that, these tests explore the impact an early inconsistency can have on the game itself. From the graphs in each of these tests we see that an uncaught inconsistency, that occurred very early in the progress of the game, can continue causing more and more damage as the game progresses. This further solidifies the importance of inconsistency resolution.

Furthermore, tests 7 and 8 showed a huge increase in the inconsistency metrics when the scale of the game is increased. When comparing these tests with their 4-player counterparts – tests 5 and 6 – it is clear how much the scale affects the rise of inconsistency metrics throughout the games. Comparing the results of tests 7 and 8 with tests 5 and 6 also proved that not only the impact of the inconsistencies but also the impact of the partial inconsistency resolution greatly rises with the scale of the game being increased. However, the average effect the partial resolution had on a larger scale game was by far greater than in the smaller games depicted in tests 5 and 6.

Lastly, Test #4 showcases the negative effects that an alteration of a game-deciding message can cause. In contrast of the early inconsistencies that the other tests explored, it shows the impact of a late inconsistency. This affected the outcome of the game for nearly all players involved in the game. While the inconsistency metrics may not have been that high in this test, the damage, that a few message alterations caused, diminished the value of the entire game by compromising the decisive moment that was supposed to end it. Furthermore, the end message alteration affected another player's experience by falsely allowing them to assume they had won.

## Conclusion

### Project Aims and Objectives

In regards to the project aim and objectives, specified at the beginning of this paper, this project was a moderate success.

Due to the time constraints, it was unsuccessful in properly achieving the objectives 4 and 5 that put the focus on measuring the full inconsistency resolution and its overhead affecting the player's experience. However, in lieu of that, this project thoroughly explored the effects of the inconsistencies themselves and partial inconsistency resolution on the game's flow and the player's experience. Furthermore, the planned approach to fix the inconsistency midway through the game was thoroughly described in the "Inconsistency resolution" section of the "What was done and how" chapter. This also described the potential effects on the players' experience the overhead of this resolution could have. The only reason this was not implemented and tested practically was the risk of running out of time.

On the other hand, this project was successful in simulating a network environment and a multiplayer game that made full use out of it. Inconsistency was successfully injected into the game and it is the author's belief that the results produced by testing this simulation accurately represent the effects inconsistencies have on actual online multiplayer games. It successfully showcases the rise of uncaught early inconsistencies and objectively calculates the effect they can have on the flow of the game. Furthermore, it uses methods developed and described by published professionals of the network inconsistency field and adapts those approaches to the video game context.

### Reflecting on the Project

This project successfully simulated an online multiplayer game environment using execution threads as players and waiting queues as well as other concurrent programming approaches as a message passing network.

In this project we have also introduced a way to measure the inconsistency metrics of the game, using Total and Global Inconsistency metrics. These metrics could also be adapted to other games by using the general ideas behind them. How Total Inconsistency measures the potentially directly affected parts of the game's local state and Global Inconsistency measures the impact those alterations caused on the rest of the game's state, when compared to other players.

Despite not finishing the implementation, this project proposed the idea of using Chandy and Lamport's snapshot algorithm along with the consensus algorithm to locate and resolve any inconsistencies between the players' game states. Moreover, it discussed the pros and cons of using this approach.

Most importantly, this project objectively tested and measured the impact an uncaught inconsistency can cause to the progression of the game as well as the players' experience. By

doing so, it also measured the great importance of having at least a partial or a precautionary fix for network inconsistencies to prevent a total disruption of the gameplay.

## Future Work

The future of this project would involve fully finishing and testing the usage of the consensus algorithm [3] to resolve inconsistencies detected by players midway through the game. The biggest time hurdle that stopped the development of this approach was implementing the reconstruction of the board and the entire game state using board history. When that would have been implemented, a thorough testing would have been done on this feature. The tests would measure the average number of turns lost in a game by reconstructing the game state to the last consistent turn. These tests would have been an objective way to showcase the negative effect on the players' experience that inconsistency resolution can cause.

Furthermore, given more time for this project, there would be more testing focused on producing the heatmaps, seen in Domino Test #8. In that test the heatmap contains too many outliers to produce a definitive evaluation of the results. To change this, instead of using 100 games to run the tests, we would use even more of them – potentially 500 or 1000. This would produce a better distribution of turns taken to finish a game and lower the chance of outliers affecting the metric.

## References

- [1] WePC, *Video Game Industry Statistics, Trends and Data In 2022* [online] Available at: <https://www.wepc.com/news/video-game-statistics/> (Accessed 26th May 2022)
- [2] Yanev, V., *Video Game Demographics - Who Plays Games in 2022*. [online] Available at: <https://techjury.net/blog/video-game-demographics/> (Accessed 26th May 2022)
- [3] Coulouris, G.F. (2012) *Distributed Systems: Concepts and Design. 5th ed.*. Harlow: Addison-Wesley / Pearson Education Ltd.
- [4] Grant, J., Hunter, A. (2011). *Measuring Consistency Gain and Information Loss in Stepwise Inconsistency Resolution*. In: Liu, W. (eds) *Symbolic and Quantitative Approaches to Reasoning with Uncertainty. ECSQARU 2011*. Available at: <http://www0.cs.ucl.ac.uk/staff/A.Hunter/papers/ecsqaru11.pdf> (Accessed 26th May 2022)
- [5] Grant, J., Hunter, A. (2017) *Analysing inconsistent information using distance-based measures*. *International Journal of Approximate Reasoning*, 89 pp. 3-26file. Available at: <https://discovery.ucl.ac.uk/id/eprint/1492987/1/distancejournal.pdf> (Accessed 26th May 2022)
- [6] Celko, J. (2001) *The Mathematics of Dominoes* [online] Available at: <https://www.pagat.com/domino/math.html> (Accessed 26th May 2022)
- [7] Savery, C., Graham, T.C.. (2014). Reducing the negative effects of inconsistencies in networked games. *CHI PLAY 2014 - Proceedings of the 2014 Annual Symposium on Computer-Human Interaction in Play*. 237-246. Available at: [https://www.researchgate.net/publication/287069304\\_Reducing\\_the\\_negative\\_effects\\_of\\_inconsistencies\\_in\\_networked\\_games](https://www.researchgate.net/publication/287069304_Reducing_the_negative_effects_of_inconsistencies_in_networked_games) (Accessed 26th May 2022)
- [8] Chandy K. Lamport L. (1985) *Distributed Snapshots: Determining Global States of Distributed Systems*, *ACM Transactions on Computer Systems*, Vol 3
- [9] Lohman, N., <https://github.com/nlohmann/json> (Accessed 26th May 2022)
- [10] Levenshtein, V.. (1966). *Binary codes capable of correcting deletions, insertions, and reversals*. *Soviet Physics Doklady*