Final Year Project Report

General Game Player

James Keating

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Dr. Arthur Cater

UCD School of Computer Science
University College Dublin

September 28, 2017

Project Specification

0.1 General Details

Project Title General Game Player
Academic Supervisor: Dr. Arthur Cater
Project Mentor: Dr. Arthur Cater
Subject: Game AI
Project Type: Design and Implementation
Software Requirements: Java or other HLL
Hardware Requirements: Students own laptop or PC
Preassigned: Yes

0.2 Project Description

The idea of a general game player (GGP) is the rules of a game can be expressed in a formal logic-programming style, and a deductive database can apply rules of inference to determine the legality and outcome of moves. Furthermore, the inference process may be able to support the automatic selection of good moves, even for a game that the GGP has neither played before nor been coached in.

The aim of the project is to create a GGP capable of accepting inputs in the standardized GGP language, and use them to support play of several games whose rules have been expressed in this language. It should be able to identify all moves that are possible at a moment in play, to validate or reject moves chosen by a human player, and make choices (possibly dreadful choices) of moves of its own.

Ideally the GGP should be able to create, from a set of rules for a 2-player 2-D game of perfect information, an interface through which the range of choices can be presented to a human opponent, through which the opponent can select a move, and through which an opponent can be informed of the state of a game in a reasonably natural way (not a set of logic expressions but a more diagrammatic presentation).

0.3 Mandatory Goals

- Develop a parser for the standard GGP language, without extensions.
- Develop a simple deductive database system of standard design.
- Express in GGP language the rules of at least three little-known 2-D games, for example from the opening chapters of Winning Ways For Your Mathematical Plays.
- Develop the inferential mechanism of the deductive database system to be able to generate exhaustive lists of possible moves for legal positions in at least two of those games.
- Provide a way for a users text input to be validated or rejected by attempting to match it against the legal-move list.
- Provide for a player to input moves for both sides in at least one 2-player game, determining when the game is finished and (if applicable) which side wins.

0.4 Discretionary Goals

- Create descriptions of at least six little-known 2-D games.
- Provide for move list generation, user move validation, and play to completion of all those games.
- Develop an interface which can display to a user the moves available, can accept an input move by mouse selection from the list, and for 2-D games based on a regular finite rectangular grid can display graphically the state of a game.

0.5 Exceptional goals

- Extend the deductive database to handle rules of games with a chance element.
- ~~Extend the interface to handle graphical display strictly based on the formal statement of game rules of at least one 2-D game that is not based on a rectangular grid.~~
- Implement game state representations using propositional networks.

Abstract

General Game Playing (GGP) is the playing of a wide variety games you may have never seen before, by being told nothing but the rules of the game at run time. This sets it apart from traditional specific game players, like the famous chess player Deep Blue. Deep Blue can beat the world chess champion at chess however, it has absolutely no idea how to play checkers. It is designed for one particular game and cannot adapt to rule changes, and certainly cannot play entirely different games. The goal of this project is to create a program that will play a wide variety of 2d games given descriptions of their rules without the creator of the program having ever known of the games. This report will cover the design and implementation of this project, as well as the background research performed and reflections on the outcome of the project.

Acknowledgments

First I would like to thank my fellow classmates Gary Mac Elhinney & Nidhi Kamat for always being there to offer support and encouragement throughout the past year regardless of how busy they found themselves.

Finally and most importantly I would like to sincerely thank and express my gratitude to my project supervisor Prof. Arthur Cater. He has been incredibly helpful and supportive for over a year now, ever since I began specifying this project. I cannot thank him enough for his time and effort he invested in helping me get to this point.

Table of Contents

Chapter 1: **Introduction**

General game playing (GGP) programs are designed to play games that both they and their programmer have never seen before, by being given only the game rules at run time. This is something humans are very good at. If someone were to hand you a rule book for a relatively simple new game they have made and ask you to play, most people would be able to play legally without much difficulty, possibly even well. However, this remains a challenging task for computer programs. Though humans have been surpassed by AI in most games today, in this area of general game playing humans still reign supreme.

There is a long and storied history of humans programming artificial game players, ranging from the Mechanical Turk in the 1770s all the way up to the famous chess-playing program Deep Blue that has beaten renowned world chess champions. The field of Artificial intelligence has focused on specific game playing programs for a long time. Such programs have far surpassed the best human opponents, in games ranging from Connect Four to Chess and more recently Go. Yet, such programs are helpless when you change the rules of the game or present them with entirely new games.

Specific game playing programs traditionally have the rules of the game hard-coded into them and heuristics which allow them to evaluate how good a particular state of the game is. These programs can search a game tree to examine many sequences of future moves, predicting where the game will go and choosing their moves in order to eventually reach states with high heuristic values. If everything goes according to plan, this continual search for states of high value, will allow the programs to constantly improve their position and ultimately win the game.

Though these specific game players are very effective, it is highly questionable whether they are actually intelligent. The real work and analysis used to understand the game and its strategy is done long before the program ever begins running. The players often simply follow strategies and heuristics that their original programmers devised. The systems themselves might as well be tele-operated. This shows the original programmers understand the game, but it doesn't show that the players understand the game in any meaningful way.

General game playing aims to build programs which can play any arbitrary game given its rules. The programs are written without knowing what games they are going to play in advance, so they have to be able to play any game that they are presented with. Go players can only play go; Chess players can only play chess; but a general game playing program can play any game.

1.1 How Do General Game Players Work

The three core components of any GGP program are the game representation, search and evaluation.

Game Representation: Since game rules cannot be coded before run time as the player has no knowledge of the game, players must be able to take a description of a game as an input and represent all the possible game positions based on it. Most modern GGP do this using the game description language (GDL). It is the most well-known effort at standardizing GGP AI, and is generally seen as the standard for GGP systems. GDL descriptions are interpreted by the player often as a state machine or a propositional network from which it can generate legal moves, apply moves, detect the end of the game, determine the score for each player. GDL is the language used for this project and the descriptions are interpreted as a propositional network.

Search And Evaluation Search refers to the ability to think/look ahead in the game. Evaluation refers to the method for assessing the pros and cons of each game state which arises during the search. The main challenge faced by the players in this area since they are not just focused on one game, is that the game-specific knowledge necessary for high-level play, be it for the search or the evaluation of game states, must be discovered during playing of the program itself and cannot be hard coded before by the programmer. Traditional players achieve this using minimax based game-tree search augmented with an automatically learned heuristic evaluation function [1]. Minimax works well for 2-player games, particularly zero sum games. However, many modern general game players have started to use variants of the Monte Carlo search instead [2]. For this project a Monte Carlo Tree Search based on a variant of UCT (Upper Confidence bounds applied to Trees) has been implemented for move selection.

1.2 Aims And Scope Of Project

This was a self proposed project with the aim of building a general game player which could play a wide variety of 2-d games described in the standard game description language used for the international general game playing competition. There are some limitations to what games can be described which are covered in section 2.2.

This project aimed to implement the player by parsing GDL text files to create a propositional network representing the game, and choose valid legal moves for each game state using UTC (Upper Confidence bounds applied to Trees)

1.3 Modification To Initial Project Specification

In the course of this project one of the exception goals in the project specification was modified. This was done in accordance with the project supervisor Dr Arthur Cater. The following goal was changed to more aptly reflect the aims and goals of this project:

Extend the interface to handle graphical display strictly based on the formal statement of game rules of at least one 2-D game that is not based on a rectangular grid.
$\Rightarrow$
Implement game state representations using propositional networks.

There were two primary factors for this change.

- Initially a theorem prover was intended to be implemented to determine the facts which represented the state of a game. However, in the course of implementing the theorem prover it became apparent that it would not be efficient enough to process game states with the desired speed. It was at this point, the alternative approach of implementing a propositional network was explored. The implementation of this network became a major focus and goal of the project and as such it was deemed appropriate to reflect this in the project respecification.
- The GUI component of this project was viewed as a secondary goal to the actual game playing program itself. By changing the goal related to the graphical component of the project to a goal relating to the playing of games the focus of the project was better represented by the project specification.

1.4 Report Structure

There are six chapters remaining in this report. Below you will find an outline of the content covered in each chapter order by the chapter number.

2. **Background Research:** this chapter covers the research which was conducted prior to the beginning of the implementation of the general game player. The approach and design used for this project, has been informed and heavily influenced by the findings of this research.
3. **Project Approach:** this chapter will cover the approach taken to complete this project. It will discuss all of the major steps taken and the reasons for those steps.
4. **Design Aspects:** this chapter will cover how each of the major components of the general game player were designed and how they work.
5. **Detailed Design & Implementation:** this will provide an in depth explanation of specific components of systems explored at a higher level in chapter 4. These are components which have been implemented in an interesting or non-standard manner.
6. **Testing & Evaluation:** this chapter covers the testing this software has undergone to ensure it is working as intended and to measure its performance.
7. **Conclusion & Future Work:** this covers the overall achievements of the project and the weaknesses that could be expanded upon in the future.

Chapter 2: **Background Research**

2.1 Initial Research For Project Specification

As previously stated this was a self proposed project. As such this section will cover the initial research which led to the specification and proposal of the *General Game Player* project.

Research began in the field of game playing in general, looking for something of note to base this project on. It was found that when people first started building AI to play games they thought it would lead to a deeper understanding of human thinking, by trying to replicate the way humans process problems and play games. However, in reality the best solutions to playing the most heavily researched games such as chess or go were completely different to how humans approach these games and no real insight into human thinking was gained.

This led to the thought that it would be interesting to build a game player which would more closely emulate a human rather than use openings and heuristics pre-programmed by expert players. At first it was considered to do this using a genetic algorithm. An AI would use it to learn to play a game like chess or checkers, starting with no knowledge or insight but the legal moves available at each game state. The idea was that it would teach itself to play the game. It would be learning in the same way a human would if you gave someone a rule book and locked them in a room to do nothing but play against themselves. However, the AI would be able to play and hence learn so much faster than a human.

The research into this idea led to the discovery that such approaches for complex games like chess or go would require unrealistic computational power. Although computers could play many more games much faster than a human, with typical genetic or evolutionary approaches you would need several thousand hours of CPU time for a single generation [3]. As computing power increases this approach may become more feasible, but right now this would be very difficult.

In spite of this the idea of playing a game with no prior knowledge was still intriguing. Research into other non evolutionary ways to achieve this was continued. This led to discovering the field of general game playing. Stanford's general game playing project was the first material which was explored. The Stanford Logic Group have been the most successful at standardizing general game playing so that everyone uses the same language for defining the rules of games they play. The General Game Playing competitions at the annual Association for the Advancement of Artificial Intelligence Conference even use their language now.

After reviewing the course material for Stanford's general game playing course and the book Synthesis Lectures on Artificial Intelligence and Machine Learning [4] it was decided that building a general game player would be an appropriate task for this final year project.

2.2 Game Descriptions

In order to build a general game player it was crucial to know how the rules of the games to be played are going to be represented. Therefore the first task of this project was to research and evaluate the best approach to describing games.

There are three major approaches to writing game descriptions Metagamer, Zillions of Games, and the Game Description language (GDL). However, both Metagamer and Zillions of Games developed in the 1990s have become rather outdated. Today GDL has become the standardized language used for GGP and is even used in the AAAI's annual general game playing competition, which is the biggest GGP competition in the world. The reason Metagamer and Zillions of Games have become outdated is not due to them being particularly worse than GDL rather it is due to Stanford driving the general game playing community to use a common language. For this reason GDL was used for this project and researched in depth.

GDL describes the state of a game in terms of a set of initially true facts and a set of logical rules. It then uses the set of logical rules to determine the set of facts which will be true in the next state based on what is currently true and the moves of the players. It also contains constructs for distinguishing the initial state of the game, goal states and terminal states. In this way, a game description in GDL defines a state machine.

This means given a game description in GDL, and all the moves made by all players in the game, it is possible to completely define the set of facts true that are currently true in the game and the facts in the next state of the game. Also it is possible to completely define the current set of legal moves for each player and if the game is in a terminal state. An example game description which I have documented and explained can be found in Appendix I . There are also some requirements for games described in GDL [5]:

- **Termination:** all sequences of moves from the legal state must reach a terminal state in a finite number of moves.
- **Playability:** every player must have at least one legal move from each non terminal state.
- **Complete Information:** standard GDL cannot describe games with an element of chance or when players do not have complete information about a game state, for example the cards in an opponent's hand. GDL can be extended to handle this however: it requires using what is referred to as GDL-II which has the additional keywords *RANDOM* and *SEES*.

2.3 Deductive Database

A deductive database is a finite collection of facts and rules. By applying the rules of a deductive database to the facts in the database, it is possible to infer additional facts. Datalog is probably the language most commonly used to specify facts, rules and queries in a deductive database [6]. For this project we instead use GDL which is itself based heavily on Datalog. The rules in a deductive database using this language must obey two key restrictions. The first is *safety*. A rule is safe if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal [7]. The second is *stratified negation* as these rules have potential ambiguities. This means there are no negative arcs (there is a negative arc from one proposition to another if and only if the former proposition appears in a negative subgoal of a rule in which the latter proposition appears in the head of the rule) [7] in any cycle in the dependency graph. For example $X(a,b) := \neg X(b,a)$ contains a negative arc.

2.4 Propositional Networks

In GGP games are traditionally represented as a state machine with a finite number of states. Each state is a series of GDL facts and the players actions transition from one state to another. Propositional networks, often abbreviated to propnets, are an alternative approach to representing games. A propnet is a graph containing a node for every proposition that can be true according to the game rules. It also contains nodes representing logic gates such as AND, OR and NOT which are applied to the proposition nodes. These logic gates represent the propositions' effects on each other. Finally there are transitional nodes which act like flip flops in a circuit taking outputs from one state and giving them as inputs in the next.

Using this approach it is possible to represent the game as a graph of propositions and actions rather than states. The benefit of this over traditional state machines is compactness. A set of n propositions corresponds to a set of 2^N states. Thus, it is often possible to characterize the dynamics of games with graphs that are much smaller than the corresponding state machines by using a propnet. This can lead to dramatic performance increases. Testing performed at Stanford on the time taken to search the entire game tree of tic-tac-toe showed a 92% reduction in run time when using propnet representation over a standard state machine, going from 130 seconds to 10. This time was further reduced to 0.2 seconds by compiling the propnet into machine code. It is because of the tremendous possible performance increases propositional networks can achieve, that they have been implemented in the *General Game Player* project. [8].

2.5 Game Tree Search

This section focuses on how general game players actually decide on which legal move they should make. This is done in general by looking ahead a certain number of moves into the game and evaluating how good or bad that position in the game is. This is done for many sequences of moves and the best move from the players current position is selected.

2.5.1 MiniMax

For many games the game tree is too large to be exhaustively searched, so instead the fixed-depth minimax algorithm can be used. Most successful early general game players including a number of the first winners of the AAAI's general game playing competition used variations of this approach. [2]

By using the minimax algorithm we make assumptions about the actions of the other players. We assume that every other player will always perform the worst possible action for our own player. This allows our player to make the best move based on what it is predicting its opponent will do.

This works well for specific game playing where it is easier to create a heuristic function to determine the value of moves. Although for some games such as go this can still prove challenging. For general game playing this heuristic evaluation of non terminal states is extremely difficult but there are ways it can be done to varying degrees of success which are discussed in the heuristic section.

2.5.2 Alpha-Beta

This is a variant of the basic minimax which achieves the same results as minimax but searches less of the game tree. This is the variant of minimax which would have been used for this project had it eventually been decided to use a version of minimax for game tree search.

It works almost the same as minimax, however it also dynamically keeps track of the best and worst move it has found at some point in the game. Using this it can disregard branches of the tree which are worse than the moves it has currently found since it knows the player would never rationally choose them as it can do something better.

Alpha-Beta Search can save a significant amount of work over full Minimax. In the best case, given a tree with branching factor b and depth d, Alpha-Beta Search needs to examine O($b^{d/2}$) nodes to find the maximum score instead of O(b^d). This means that an Alpha-Beta player can look ahead twice as far as a Minimax player in the same amount of time. Looked at another way, the effective branching factor of a game in this case is sqrt(b) instead of b. It would be the equivalent of searching a tree with just 5 branches at each node instead of 25. [9].

2.5.3 Search Depth

The previous two sections discussed how not all game trees can be searched exhaustively due to their size. Instead an incomplete search to a certain depth is performed. However, there are a lot of problems and questions to be answered when determining this depth. The same depth might not make sense for two different games. Go has many more possible moves than checkers. If you used the optimum depth for checkers on go your program would not finish in time as you would have searched too deep and had too many moves to process. If you used the optimum depth for Go on checkers you will not have searched as far as you could and will not get the best results.

Two potential solutions to this problem were explored in the course of this project. One approach was to use breadth-first search instead of a depth-first search. The downside of this is it requires a huge amount of space when done with large trees, in many cases even greater than storage capacity of the computer. Another problem with this type of search is it cannot utilize an alpha-beta search to reduce the search space.

Another possible solution explored was using iterative deepening to explore the game tree. This involves repeatedly exploring the game tree at increasing depths until there is no time left. This is somewhat wasteful as portions of the tree may be explored more than once. However, this waste is normally limited by a small constant factor which may be reduced even further by utilizing an alpha-beta search.

So far only searching the game tree to a uniform fixed depth has been discussed. However, it is also possible to search the tree to variable depths. This means you explore certain sequences of moves (branches of the game tree) more than others. For example in chess, it may be hard to evaluate a game state unless a piece is taken. So, one could search until a piece is captured and then evaluate the game state. That could be one move for some branches of the game tree or much more for others.

Once again however the problem with this arises with coming up with appropriate heuristics for evaluating all games. The next section discusses some of these heuristics which could be used.

2.5.4 General Game Heuristics.

In GGP the games being played are not known in advance, due to this it is very difficult to evaluate a game state as what is considered good in one game may not be good in another. A common approach to this problem is to try to find heuristics which have merit across all games. There is no guarantee when using heuristics like these that they will always be good. It is almost always possible to find a game in which a general heuristic does not apply or make sense. However, it is very often the case that these general heuristics will have merit. [10].

- **Mobility:** this idea is that the more options/moves available to a player the better. In this case the heuristic would count the number of moves available to a player at a given state. The implementation of this heuristic in particular was given much consideration for this project but ultimately rejected.
- **Focus:** this is the inverse of mobility. It is the idea, that it is better to limit the number of possible moves available to players. This allows you to search to a much greater depth in the game tree, since there are fewer possible moves to explore. It will be possible to search to terminal states much faster and hence, more easily identify the best move.

  However, this contradicts the concept of mobility. Due to this programmers will often try to strike a balance between the two ideas by limiting the opponent's moves, reducing their mobility and the search space, while still maximizing the players mobility.
- **Goal Proximity:** Goal proximity is a measure of how similar a given state is to a desirable terminal state. There are many approaches to trying to compute this. Fluxplayer, the winner of the second ever AAAI's GGP competition used a heuristic function based on goal proximity. They calculated the proximity to the goals or terminal states by assigning 1 if true or else 0 to all of the atoms which made up the complex descriptions of their goals and terminal states. They then applied standard t-norm formulas to these descriptions to determine how true they were [11].

2.5.5 Monte Carlo

This is the search method which was implemented in this project for evaluating game states and selecting moves. After comparing its performance with the other methods discussed in this section the conclusion reached was that a form of Monte Carlo search would be the most effective search based on two key points:

- It does not recognize or take into account boards, pieces, piece count or any other features of a game that might form the basis of game-specific heuristics. The evaluation process is based solely on the winning or losing of a game. This is something which can be applied to virtually every game unlike the heuristics in previous sections which only apply to a lot of games.
- It has had success in other general game playing programs. While nearly all successful early general game players used the minimax algorithm combined with a general heuristic function to decide their moves most modern general game players have instead started to incorporate at least some variant of Monte Carlo search [1]. Using a variant of this approach for example CadiaPlayer won the International General Game Playing competition three times.

The basic idea behind the search is that it evaluates a non terminal state by "probing" to terminal states several times and getting the average value of those terminal states for the player. Probing refers to making a series of random moves for each player only considering one move each so it can do so very fast.

While this is a very powerful approach there are weaknesses:

- The Monte Carlo search does not take into account the structure of a game. For example, it cannot recognize symmetries or independences that could substantially decrease the size of the search space.
- Unlike the minimax algorithm it assumes opponents are playing randomly when in fact, it is very likely they are not and will make the best moves they can. This issue is addressed to some extent in a variation of Monte Carlo which I have implemented for this project, the algorithm known as UCT (Upper Confidence bounds as applied to Trees).

2.5.6 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a variation of Monte Carlo search. Both variants are based on the same principle of rapidly preforming random playouts of games to evaluate a game state. However, they differ on how they expand the game tree.

A pure Monte Carlo search expands the game tree uniformly. The MCTS uses a more sophisticated approach. The search biases the selection of which nodes to expand based on two factors known as exploitation and exploration.

- **Exploitation:** refers to the results of previous searches. If previous searches had good results when a node was selected it is more likely to be reselected.
- **Exploration:** refers to the number of times a node has been visited. The more times a node is visited the less likely it is to be revisited.

The idea behind selecting nodes based on these two factors is to try and strike a balance between refining the search in promising areas of the tree and exploring new areas of the tree.

These preferred nodes are more likely to be expanded and explored. In this way more promising nodes are explored more often and deeper than others, whilst still seeking confidence that the other moves are inferior.

Chapter 3: **Project Approach**

The goal of this project was to build a general game playing machine. In order to achieve this goal a divide and conquer style approach was taken. Four major tasks were identified which needed to completed:

Define games → Represent game states → Play games → Display games

3.1 Defining Games

3.1.1 Defining The Language

In order to play a game, a player must at some point be told the rules of the given game. Hence, in order to build a general game player it is essential, to in some way describe the rules of a game to the player. This led to the need to formalize the descriptions of games which became the first major component of this project.

In section 2.2 various approaches to describing games were researched. From that research one approach in particular stood out: the game description language (GDL). GDL is a logical language which can be used to describe the rules of arbitrary games provided they fulfill certain conditions as discussed in section 2.2. This language was adopted to describe games in this project. It was chosen primarily due to two factors:

- **Documentation:** of the three languages considered (Zillions of Games, Metagamer and GDL) GDL is by far the most well documented language.
- **Future Competition:** GDL is used in virtually every general game playing competition today even the AAAI's annual competition. Using GDL gives this player the potential to compete in these competitions in the future.

Unfortunately the standard GDL language was missing some functionality which was required to meet all the goals of this project. This required two extensions be made to the base language:

- **RANDOM:** a keyword used in order to describe games of incomplete information (games with random or unknown events).
- **DrawIt:** a novel keyword created specifically for this project. This is not a common extension for the GDL. It is used to describe the graphical component of games.

The rest of the definition of the language followed the standard GDL specification [5]. An example of a documented game definition used in this project can be found in Appendix I.

3.1.2 Building The Parser

Game descriptions were formalized in this language to be used as test cases these games can be found in Section 6.2.1. The next step was to find a way for the player to process the game descriptions from text files to something more meaningful.

To achieve this the text description of a game needed to be parsed by the player and stored accordingly. To determine the type of parser which would be required the grammar below was first formalized based on the game description language.

S $\rightarrow$ Description S
S $\rightarrow \epsilon$
Rule $\rightarrow$ (Fact)
Fact $\rightarrow$ Atom Fact
Fact $\rightarrow$ Rule Fact
Fact $\rightarrow \epsilon$
Atom $\rightarrow$ variable | keyword | identifier

The result above is an LL(1) grammar. As such it was decided that a recursive descent parser would be built to parse it. This decision was made as a recursive descent parser is one of the most simple parsers to implement, while still sufficiently powerful to handle the grammar. Finally once the game descriptions were successfully parsed and stored accordingly. The next stage of the project was ready to begin.

3.2 Representing Game States

Once games could be described to the player, the next step was to take that information and use it to represent each state in the game. The representation needed to tell the player everything that it would need to know in order to play the game; the moves it could make, whose turn it was, the number of players, if the game was in a terminal state etc.

To do this using GDL a series of facts needs to be produced. All propositions in the game description which are true need to be identified and presented to the player. For example in the case of a game of tic-tac-toe shown in Figure 3.1 the facts to the right of the figure are required.

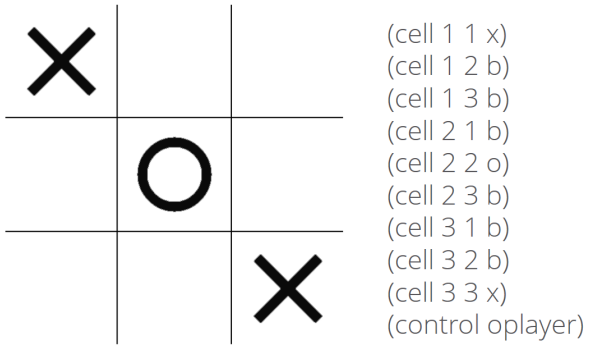


Figure 3.1: Sample game state representation in GDL

Two approaches to determining these facts were attempted. The first was to use a *theorem prover* and the second was to build a *propositional network*.

3.2.1 Theorem Proving

The initial approach taken for this task was to build a theorem prover. The theorem prover would programmatically examine each of the logical rules in the game description. Based on the other rules and propositions which were currently true, it would produce a new list of facts that could currently be proved. This new list would represent the next state of the game.

However, in the process of implementing the theorem prover it became apparent that the speed at which it would be possible to process game states would be far slower than desired. The algorithm used to determine the players' moves (Monte Carlo Tree Search) requires the game to be played out hundreds or even hundreds of thousands of times each turn to make good moves. The theorem proving approach was proving too slow to do this in a reasonable time. So alternative solutions were explored. The approach which was ultimately chosen was to replace the theorem prover with a *propositional network.*

3.2.2 Propositional Network

When the theorem prover implemented proved to be slower than desired a propositional network (propnet) was implemented to replace it. This decision was made based on experimentation conducted by Micheal Genesereth [12] of Stanford University, it was concluded that game states could be processed using a propnet much closer to the speed that was required to achieve good gameplay in a reasonable time. In spite of the potential performance increase of propnets there were three major drawbacks to this approach, which is why a theorem proving approach was attempted first.

- **Complexity:** the implementation of a propnet is a very difficult and intensive programing task. It was thought that the more simple theorem proving approach would be sufficient for our player's needs.
- **Build Time:** before a game can be played by this player the propnet must be built. For more complex games such as chess or go this can take many hours. Although, games can be played much faster than a theorem proving player once the network is built.
- **Additional Description:** in order to build a propnet additional information is required in the game description than normal. The description must contain a list of the possible values of variables within propositions. It it possible to programmatically generate these values for small games, however it is not computationally feasible to do so for complex games.

3.3 Move Selection

Once a propositional network representing game states had been built all the information which was required in order to actually play the game was available to the player. It was at this stage of the project that the logic for selecting the player's moves was implemented.

To begin a random legal player was built. This player simply selected random legal moves each turn. Some simple game management logic was then added to allow this player to play games to completion against itself or a human player.

Once this infrastructure had been built and tested, the next step taken was to extend the legal player, to select moves in an 'intelligent' manner. There are a variety of potential approaches to this problem. Each consists of two common components:

- **Searching** the game tree to determine the results of future moves.
- **Evaluating** the state of the game after certain moves have been made.

However, there are many ways in which these search and evaluations steps can be performed. For this player a UCT search was used. It was selected based on several factors:

- **Evaluation style:** most approaches to evaluating a game state use some form of heuristic function. This is very effective in specific game players as they can apply expert knowledge to the game. In general game playing this is much harder as what is good in one game may not be in another. In section 2.5.4 some possible heuristics for GGP were explored. While these have merit in many games there are still some games where they will not. MCTS does not rely on any heuristic function to evaluate game states. It instead uses the number of wins and losses after a move is made. This evaluation has merit in virtually every game which cannot be said for the other possible evaluation methods explored.
- **Performance of other players:** this decision was influenced by the results of other successful general game players, in particular by looking at the winners of the AAAI's annual general game playing competition. These players represented the best players in the world at the time. The first winners of this competition did not implement any form of Monte Carlo search. The first winner to do so successfully was Cadiaplayer [1] in 2013. The Cadiaplayer then went on to become the only player to ever win this competition 3 times. Almost all of the winners since then have implemented some variation of the Monte Carlo search [2].
- **Flexible runtime:** many other searches were required to be run to completion hence needing a fixed amount of time to choose a move. Using the UCT search any time can be allotted to selecting a move and the search can simply terminate at the desired time. The more time that is allocated the better the selection of moves becomes.

At first a standard Monte Carlo search was implemented. This search was then extended to what is known as the Monte Carlo Tree Search (MCTS). The MCTS is a more sophisticated version of the standard Monte Carlo search as discussed in section 2.5.6 This change was made due to research which suggested it could provide a substantial performance increase [13].

3.4 Graphical User Interface

The final component of this project was to build a GUI which would display what was happening in a given game to the user. Most of the general game players which were researched for this project did not have a built in graphical component of this nature.

A separate piece of software was generally required for each game. The player would generally provide the external software with its moves as inputs so users can see its moves and play against it either themselves or using another AI.

It was decided for this project to take an alternative approach to displaying games to a user. A graphical interface for each game is programmatically generated based entirely on an extension to the game description. This was done to allow new games to be added more easily as specific graphics would not need to be developed. It was also done to avoid having any game specific code required to play games as this goes against the core philosophy of general game playing.

To achieve this a new keyword was created to extend the GDL language as mentioned in section 3.1.1. The required information such as images and coordinates could then be given in the game descriptions using this extension.

With this information it was possible to begin building the GUI. It was decided there would be five key features, which were implemented in the order below:

1. Selecting the game description by file directory.
2. Assigning an AI or human to each player.
3. Providing a list of valid moves which human players could select from.
4. Display the current game state based on the description e.g. the board, pieces, cards
5. Preview users moves when they had selected one from the list, if the game state was not reliant on random or unknown events.

Chapter 4: **Design Aspects**

The software for this project was developed using a top down design. The overall system was viewed as a single entity and decomposed into the four major components:

Parser → Propositional Network → Monte Carlo Tree Search → GUI

Each of these components was in turn viewed as a system in its own right and decomposed further. Below is a UML class diagram modeling the interactions between the main components of the software and an accompanying explanation of the overall design at a high level.

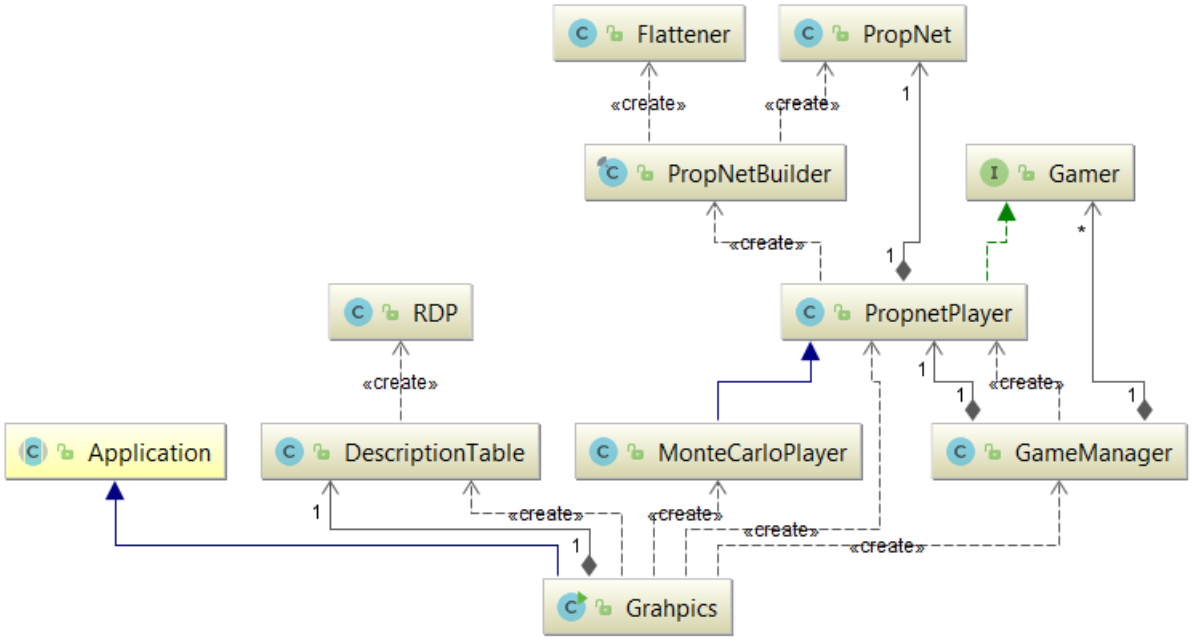


Figure 4.1: UML class diagram of major software components

The $Graphics$ class creates the GUI from which users can load and play games. Once a user has selected a game, the $DescriptionTable$ is created and the file is parsed and stored as appropriate in the $DescriptionTable$.

Finally once the user is ready to play the $GameManager$ and players are created. The $GameManager$ builds a single $PropNetPlayer$ which is used to manage the game and determine the outcome of random events. All additional players required then share that players $PropNet$ so only one has to be built, as that is a time consuming process.

4.1 Game Description Parser

This system was designed as a general purpose parser with three major sub-systems. It was not designed to parse a particular grammar rather it can take any LL(1) grammar as an input and process a text file accordingly.

This design approach was taken so that modifications could be easily made to the grammar without any system code being altered. This proved to be a wise decision as several changes were made to the initially defined grammar over the course of this project.

4.1.1 Lexical Analyzer

Lexical analysis is the first phase in the parser. It takes the directory of a text file containing a GDL description. The lexical analyzer then combines the characters of the file into a series of tokens. This is done by reading the character stream from the game description and feeding into the deterministic finite automaton (DFA) illustrated below.

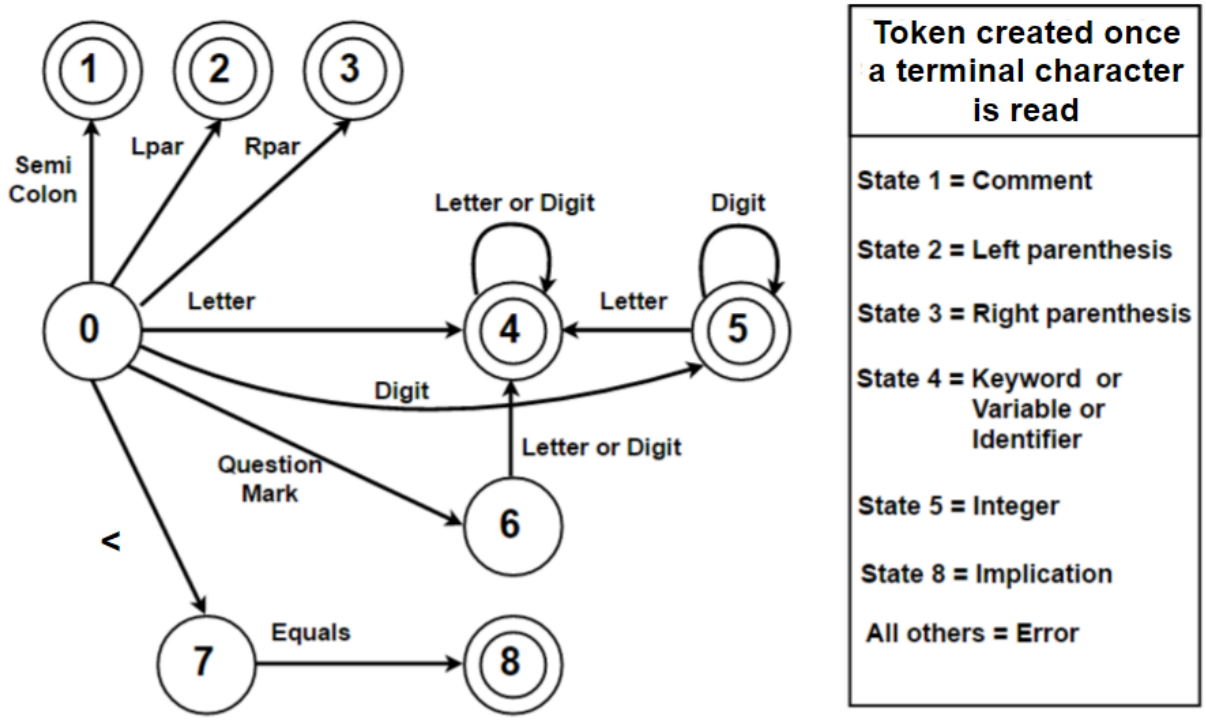


Figure 4.2: DFA used by the lexical analyzer

The DFA is walked through based on the characters presented until the end of a token. By then examining the state of the DFA the lexical analyzer knows if the token is valid and its type. Based on the result it can generate an error and terminate the program or generate the appropriate token if it is valid. If the file can be tokenized successfully, a list of the valid tokens is sent to the parser to perform syntactic analysis.

4.1.2 Syntactic Analysis

The syntactical meaning of the game description is validated in the parser. Parsers do this by examining the token stream produced by the lexical analyzer and comparing it to the grammar which it is provided. The way in which this is done depends on how the parser is designed. For this project a *top down, back tracking, recursive descent parser* was implemented.

- **Top Down:** the parser constructs the parse tree beginning with the start symbol. It then attempts to transform that symbol into the token stream produced by the lexical analyzer.
- **Recursive Descent:** this is a style of parsing which uses recursive procedures associated with grammar non terminals to process the input. It determines which grammar production to use by trying each production in turn. This leads to certain limitations. The main issue is it can only parse grammars with certain properties. For example, a grammar containing left recursion cannot be parsed by this parser.
- **Back Tracking:** This parser requires backtracking. This means it may process certain inputs more than once to find the required production. If one derivation of a non terminal fails, the parser restarts the process trying different productions of same non terminal.

Once the parser has validated the syntactic meaning of a game description. It then groups the tokens into facts and rules as appropriate and stores them accordingly for future use. A detailed description and example of the parsing process used can be found in Section 5.1.

4.2 Propositional Network

A propositional network (propnet) is a type of graph. The graph is made of propositions (statements about the game which can become true or false) with logical connectives (inverters, and-gates, or-gates, and transitions) representing their effects on each other. The location of networks which this player has built and instructions on how to view them can be found in Appendix II.

This system was designed to programmatically map a GDL description of a game to an equivalent propnet. This was necessary as game descriptions must be written manually. This can be easily done using compact descriptions in GDL. However, manually defining a propnet is an extremely difficult task. This design decision was made as it can be very difficult to manually define a propnet. The propnet built for TicTacToe by this player contains 3206 nodes (Section 6.2.2), many of which have multiple inputs and outputs. Due to this complexity it was decided to manually define games in GDL and then programmatically generate the propnet from the description. There were two major components to this task; flattening the original description and building the propnet from the flattened description.

4.2.1 Flattening The Description

A propnet must contain a node for each unique proposition, which could potentially become true based on the game description. However, when rules or facts are defined in GDL descriptions, they will often contain propositions whose values are not specifically defined. Instead, they will contain a variable which could represent many different values. This is done to describe games compactly. Rather than replicating the same rule potentially hundreds of times changing only one value, the rule can be written just once with a variable.

This means that before the propnet can be built each rule containing a variable must be replaced by an equivalent set of grounded rules (rules with no variables). This is done by the Flattener class. The domain of all variable are explicitly specified in the base propositions of a GDL description. This gives the player access to the potential domain of each variable. An example of this can be found in Appendix I

The flattener examines each non-grounded rule in the description. It then will recursively attempt to instantiate the rule with every possible combination of valid values. This will be determined based on two factors:

- **Domain** of the variables. The possible values each variable in a proposition could have based on the game description.
- **Consistent instantiation** of the variable. In a rule consisting of multiple propositions, a variable ?x must have the same value for each occurrence in the rule.

4.2.2 Building The Network

Once the game description has been flattened the network itself needs to be built by the PropNetBuilder class.

Each proposition in the game description is assigned a unique node at first with no inputs or outputs. These nodes can be propositions if their values change or constants if they do not.

Then the head of each rule (the proposition which is proved by the rule) is given an *And-gate* as an input. The outputs of all nodes in the body of the rule are then connected to the *And-gate*. This means when the body of a rule is true its head becomes true. A *Transition* node is then given as an output. The transition node controls flow of information from one step to the next. It acts as one step delay similar to a flip-flop in digital circuitry.

Next *Not-gates* are inserted after components which are negated in the game description. Finally any proposition with multiple inputs has an *Or-gate* inserted between itself and its inputs.

4.3 Gameplay / Move Selection

This system has been designed to manage gameplay through the GameManager class in addition to various extensions of the PropNetPlayer class. The PropNetPlayer is the most basic form of game player on which all other players are built as extensions. There are currently three extended players Human, Pure Monte Carlo and Monte Carlo Tree Search.

The GameManager takes a list of these players and assigns them each to a role in the game. It then initializes the propositional network i.e. the game. It will then ask all players for their move each turn. In some cases this move may even be to do nothing that turn. This continues until a terminal state is reached and the GameManager can terminate the game and start a new game without rebuilding the propositional network.

Once the manager has all the moves for a turn it generates the outcome of random elements in the game and then updates the new game state created by the player's actions.

4.3.1 Monte Carlo Tree Search

The most sophisticated and successful player designed for this project to date implements the Monte Carlo Tree Search (MCTS). This player uses the MCTS to determine its move selection each turn.

There are four stages to this algorithm which are discussed below. A more detailed explanation of the implementation of the selection can be found in Section 5.1

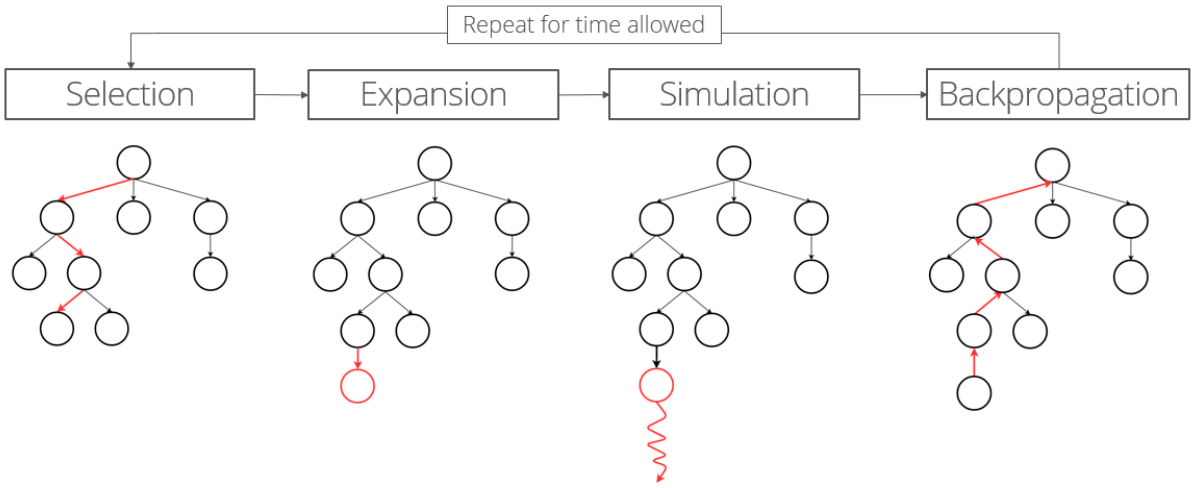


Figure 4.3: Stages of the Monte Carlo Tree Search

- **Selection:** the player begins at the root of the game tree (the current state of the game). It then begins to select child nodes until it reaches a leaf node in the tree. However, it does not select these child nodes at random. The selection is biased by two factors discussed in Section 2.5.6 *exploration* and *exploitation*. By looking at both these factors the aim is to strike a balance between refining the search in promising areas of the tree and exploring new areas.

  These two factors are used to generate a score for each child node. The nodes are then each assigned a probability of being selected based on this score. The better their score the more likely they are to be chosen.

- **Expansion:** once a node has been selected that node must then be expanded. Nodes are created for each of its children i.e. for each possible move from that state of the game. These nodes are then added to the tree.

- **Simulation:** This is the step which tries to evaluate each game state. Its results are used as part of the selection phase to *exploit* nodes which do well in this phase.

  From the selected node a random playout of the game is performed to termination. Since the playouts are random this can be done very fast. Neither player spends time thinking about which move it should make and only one branch of the game tree must be explored at each depth.

- **Back Propagation:** once a playout has reached termination the results for each player are propagated backwards along the path to the root. Each node along the path is updated with the results of the playout and the one extra time it has been visited. This will cause them to have a new score in the next selection phase.

There are some drawbacks to this design. In order for this algorithm to be successful it is essential that the player is fast enough to simulate a very large number of games during simulation phase. Performing a single random playout of a game will generally give a very poor indication of how good that position is for the player. This is because the random moves selected may have been terrible moves that no rational player would ever make. Over a small set of simulations luck simply plays too large a role in estimating the value of a game state. However, as the simulation set increases the results become more and more reliable as the number of lucky runs gets balanced by equally unlucky runs. The more time allotted and hence simulations performed the better this algorithm performs.

This was a major factor in the decision to switch to propnet design for game representation. It would have been extremely difficult to successfully use this type of move selection in a reasonable time without the propnet as processing game states would be too slow.

Another drawback is that it can struggle in games in which a loop can be entered, for example in a game such as the 8-puzzle where the only terminating condition is completing the puzzle. Though unlikely a player could potentially choose an infinite series of moves which would not reach the terminal condition i.e. move tile left, move it back and repeat. This can be counteracted to a degree by, forcibly terminating simulations taking much longer than expected.

4.4 Graphical User Interface

This graphical user interface (GUI) for this project was implemented using javaFX. It has been designed to generate graphics entirely based on the game description. This means that there is no game specific code required to display games, although game descriptions do need an extension to allow this facility to be exploited.

Two javaFX scenes have been designed for the GUI which can be seen below. They are the *Selection Scene* and the *Playable Scene*.

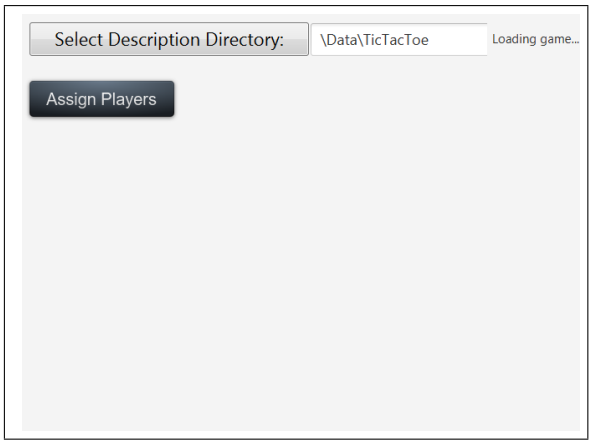


Figure 4.4: Selection scene file section.

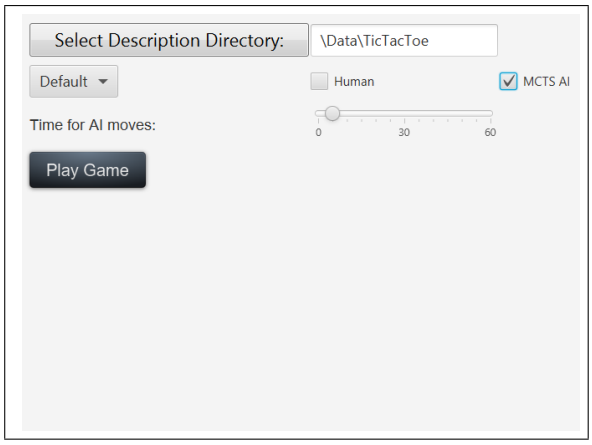


Figure 4.5: Selection scene assign player.

There are two stages to the selection scene. First as seen in Figure 4.3 the user can select a file directory using a FileChooser or by inputing it manually. Once the directory to the game description has been selected the user has the option to assign a AI or Human player to each role in the game. This can be seen in Figure 4.4. There is a drop down menu to allow the user to select a specific role or all roles and they can assign the player type using the check boxes. Finally the user can press play game to display the playable scene shown below.

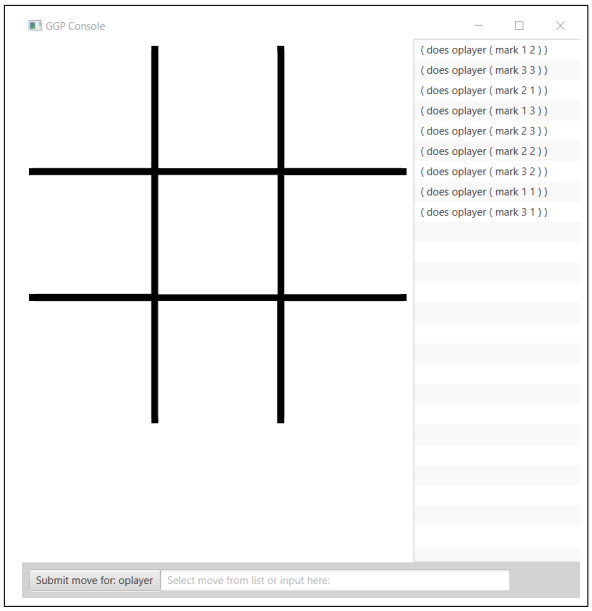


Figure 4.6: Playable scene for Tic Tac Toe

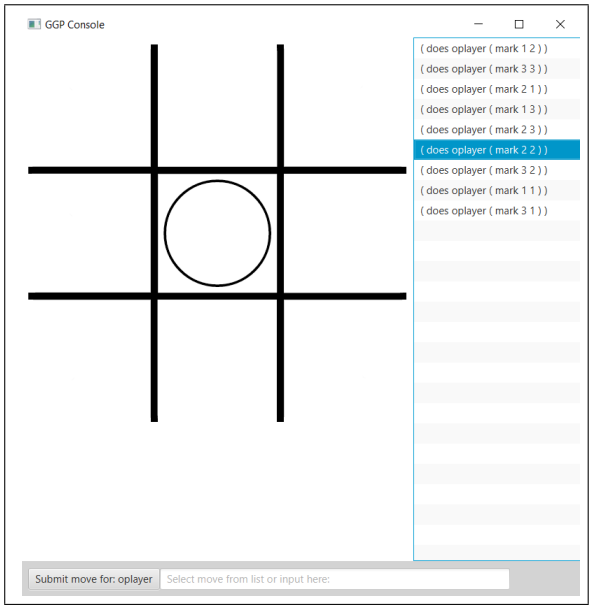


Figure 4.7: Playable scene previewing move

Once the user is ready to play the propnet is built and the playable scene in Figure 4.5 can be displayed. This scene is made of three javaFX panes enclosed in a single BorderPane:

- **Options Pane:** this pane is along the bottom of the scene and currently contains only the move submission button and a text field to enter a move manually if the user desires. This is where options which will hopefully be implemented in future iterations will be, for example undo move or hint.
- **Moves Pane:** this can be found on the right of the playable scene. It contains a scrollable list of all the legal moves for the current player. If the current player is an AI selecting these moves will have no effect they will simply show the possible moves the AI could make. However, if a human player selects a move by clicking it, as seen in Figure 4.6 the move will be previewed showing the user its effect on the game. If the outcome of the move depends on random events the move will not be previewed as the outcome is unknown.
- **Board Pane:** this is a GridPane which displays the game itself. What this displays is determined by the game description. Rules can be defined which tell the GUI the location of an image and the coordinates on the pane it should be drawn. When these rules become true the images are drawn as specified. A detailed explanation of this implementation can be found in section 5.4.

Chapter 5: **Detailed Design & Implementation**

5.1 Recursive Descent Parsing

Top-down parsers start from the root node and match the input against the production rules to replace them if they match. Here is an example of how this is implemented in this player. Take the following grammar:

S $\rightarrow$ Description S
S $\rightarrow \epsilon$
Rule $\rightarrow$ (Fact)
Fact $\rightarrow$ Atom Fact
Fact $\rightarrow$ Rule Fact
Fact $\rightarrow \epsilon$
Atom $\rightarrow$ variable | keyword | identifier

For the input string: '(example)', the parser will act as follows:

1. The lexical analyzer will convert the string to a list of tokens: [(, identifier,)]
2. It will start with the start symbol S from the production rules and will try match its yield to the left-most token of the input '('.

   S $\rightarrow$ Description S
   Description $\rightarrow$ (Fact)

   Here, it can match the first token '(' so it advances to the next token identifier.
3. It continues to try match this production by mapping Fact $\rightarrow$ identifier.

   Fact $\rightarrow$ Atom Fact
   Atom $\rightarrow$ identifier

   In order for this production to match the second fact must become ϵ (nothing)

   Fact $\rightarrow$ Atom Fact (Fails tries next production)
   Fact $\rightarrow$ Rule Fact (Fails tries next production)
   Fact $\rightarrow \epsilon$ (Successful match)

   If this production had failed the parser would have to backtrack and try a different production of Description since there would have been no valid Fact production.
4. Now that Fact is matched it can continue matching the Description production. The next token Rpar matches so the description production is now complete.
5. Finally the second S in the initial production is matched S $\rightarrow \epsilon$

   Therefore the input string is valid. However, if the second S had not been matched the search of the initial production would be abandoned and it would need to backtrack and try all other S productions from the beginning, returning an error if none matched.

5.2 Propositional Network

5.2.1 Computing A Topological Ordering

After a turn has been made the propositional network must be updated to determine the facts of the new game state. To do this all the values in network need to be propagated forward. However, the order in which the nodes are updated is very important. Imagine the scenario illustrated below:

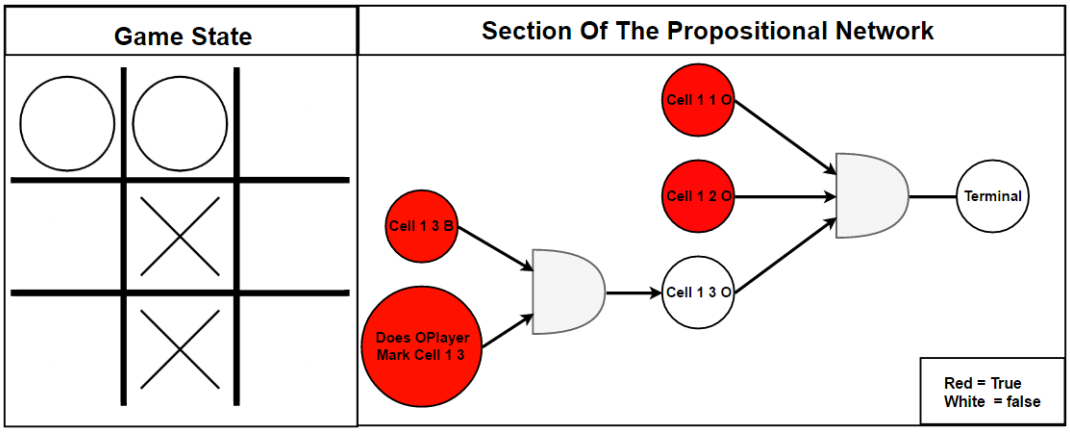


Figure 5.1: Topological ordering example.

The node representing (Cell 1 3 O) is currently false. However, the player has decided to mark that cell so next turn it will be true. This means the player will have made a line of O's so the terminal condition should become true and the game should end. If the network updates the terminal node before (Cell 1 3 O) is updated it will think cell 1 3 is still blank and continue playing.

In order to solve this problem the topological ordering of the graph can be calculated. A topological ordering is a linear ordering of a graphs vertices where for every directed edge uv, u is visited before v in the ordering. Below you can see an example of a graph which has been ordered in such a manner.

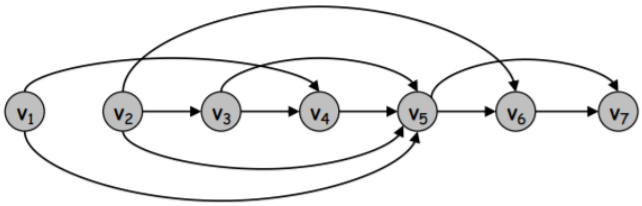


Figure 5.2: Topologically ordered graph.

This was implemented using the following algorithm:

1. Identify a proposition that has no incoming edge (no inputs).
2. Remove the node and its edges from the network and append it to the output.
3. Repeat steps one and two until every node has been removed from the original network and appended to the new network.
4. Return new ordered network.

5.2.2 Displaying The Network

In order to create a visual representation of the networks a tool called ZGRViewer was used [15]. ZGRViewer is a graph visualizer implemented in Java which can display graphs expressed using the DOT language from AT&T GraphViz. Below is an sample extract from one network built by this player. For instructions to view full networks built for this project see Appendix II.

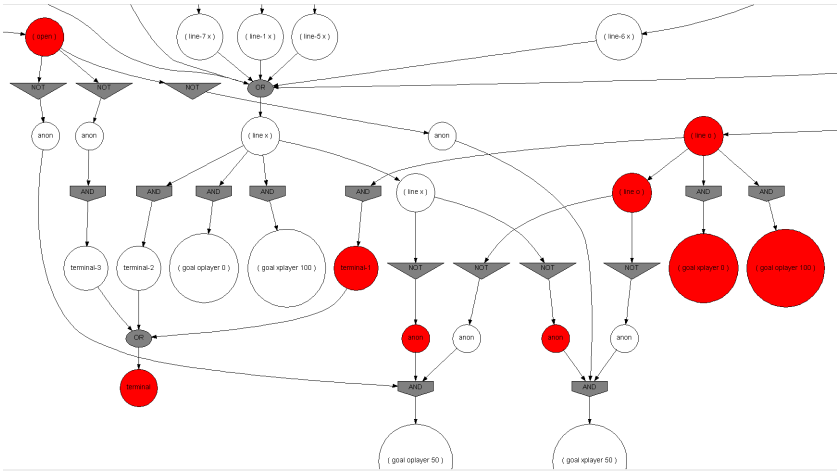
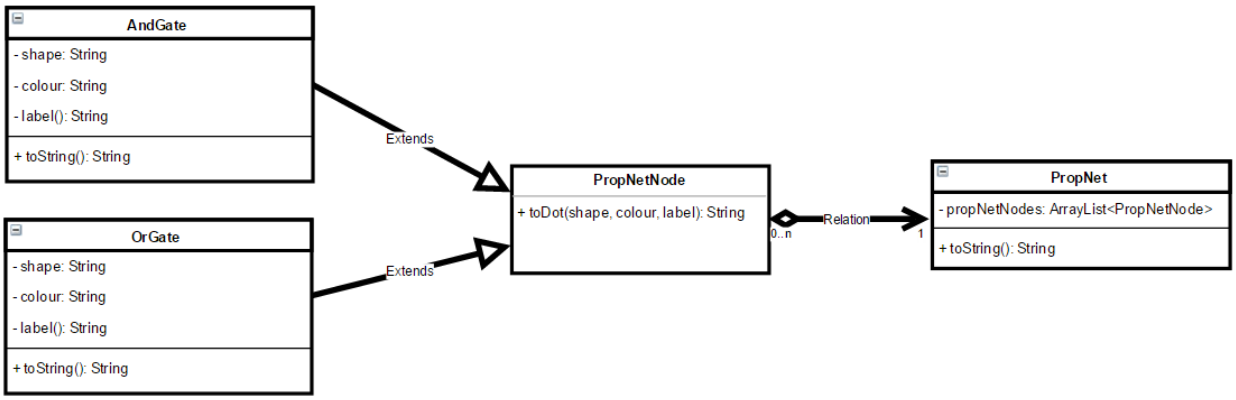


Figure 5.3: Section of a propositional network in ZGRViewer.

This was done to allow for easier manual validation of the network. It allows the network to be examined to confirm if it was working as intended. Also it was done for demonstration purposes, to display the network to others and explain how it works.

In order to use this tool the network had to be written to a file in the Grahviz dot language [14]. Custom toString methods were implemented for the PropNet and all PropNetNode components. The implementation is illustrated below:



1. $Propnet.toString()$ appends each element of its propNetNodes list to a string.
2. Each element of the list extends PropNetNodes. Their $toString()$ calls $toDot()$ and passes its attributes.
3. $toDot()$ produces strings in the following format:
   $NodeID[shape, fill, label]NodeID \rightarrow OutputedNodeID$
4. The object's hashcode is used as its ID

5.3 Monte Carlo Tree Search Selection

The Monte Carlo Tree Search uses a selection function to assign a value to each node. The formula used for this player is:

$$\frac{NodeScore}{\#VisitsToNode} + \sqrt{2 * \log\left(\frac{\#VisitsToParent}{\#VisitsToNode}\right)}$$

The greater the value of this function for each node the more likely is to be selected. The first important fact to note is that this formula returns an infinite value if the node has never been visited, as in that case the node score is divided by 0. This means a node with no visits is always selected before its sibling nodes which have been visited.

In the case where all children have at least one visit they are assigned a range of numbers based on their values relative to one another. For example if node A had a value 10 and node B had a value of 20. A would be assigned the range 1-10 and B the range 11-30. A random number from 1-30 would then be chosen to decide which node was selected. This means since node B has double the score of A it is twice as likely to be selected.

5.4 Games Of Incomplete Information

One of the exceptional goals for this project was to build a player capable of playing games of incomplete information (games with random elements).

This is not supported by the standard game description language so some extensions to the base language had to be made. The lexical analyzer was modified to accept the new keyword RANDOM. This keyword allows a special type of player to be defined in a game description.

This special player is used to determine the outcome of all random events in the game. The possible outcomes of random events are assigned as the possible legal moves for the RANDOM player at the time of the event. In many instances this is a very simple and elegant solution. However, there were some instances where it was not instantly obvious how to describe games with this approach.

- **Multiple Events Per Turn:** since RANDOM is implemented as a regular player it can only make one move per turn. However, in some games multiple random events happen each turn. For example both players drawing a card.

  To handle this situation until all random events have been resolved non RANDOM player's only legal move is to do nothing. The GUI and GameManager recognize this and will allow the RANDOM player to make multiple moves in a row without the players having to interact in any way with the system or know this is happening. Once the random events are resolved the RANDOM player has his legal moves set to nothing and the game management returns to normal.

- **Weighted Randomness:** In some games an outcome may not be completely random. It may be that there is a 20% chance of outcome A and 80% of B. However, the RANDOM player has an equal chance of selecting each legal move (possible outcome of an event). So to replicate weighted randomness multiple copies of the same legal move are assigned. This can be expressed in the game description. The result is that the players list of legal moves would be [A,B,B,B,B] giving it the appropriate odds of selecting each outcome.

Chapter 6: **Testing & Evaluation**

6.1 Functionality Testing & Methodology

This section will cover the testing this software has undergone to ensure it is working as intended. There were four main stages to the testing methodology used for this project: unit testing, integration testing, system testing, and user acceptance testing.

6.1.1 Unit Testing

At this stage of testing, a unit refers to a function or an individual class. During this first phase of testing, specific units/components of the software were focused on to determine if they could handle known inputs and outputs correctly. The goal of this endeavor was to verify that the code actually worked. This was done partially using classical unit tests. One of the biggest benefits of these tests was they could be run each time code was modified, allowing issues to be resolved as quickly as possible. However, manual testing was used instead for certain units which were more difficult to produce automated tests for.

For example many elements of the Monte Carlo player rely on randomness which makes it more difficult to write unit tests for since the 'correct' output is unknown. Also producing a unit test to validate if a propnet was built correctly would have been a very challenging task. Each time the network was built nodes would have random NodeID's, in addition there could be thousands of nodes all interconnected making it very difficult to manually determine the correct output. This is why so much effort went into creating a graphical representation of the network as discussed in Section 5.2.2 so that networks could be examined and verified manually.

6.1.2 Integration Testing

The integration testing combined all of the units within a program and tested them as a group. This testing was designed to find defects in how multiple classes interacted. This was particularly beneficial in determining how effectively sub-systems were working together. No matter how effective each component was on its own, if they were not integrated correctly and efficiently, the overall software would have been affected.

6.1.3 System Testing / User Acceptance Testing

This is the first stage where the completely integrated system was tested. It was tested in end-to-end scenarios that users would engage in. The aim of this testing was to verify that the system met all its requirements (the goals of the project specification) and that the software was easy to understand and use. In this phase of testing a type of smoke testing was used. A checklist of possible uses was created and the software was used as a user would in the real world to find any flaws.

Finally a small alpha test was conducted. The software was given to a small number of users to experiment with. These users were then surveyed on their experience. Their feedback was used to find flaws and improve usability. For example based on this testing the previewing of moves reliant on random outcomes was removed as it confused users.

6.2 Performance Testing

The previous section discussed the testing which was done to prove the correctness of this software. This section will instead cover the testing which has been done in order to evaluate the performance of the software as a whole. Reflections on and discussions of these results can be found in section 6.3.

6.2.1 Games Tested

This shows the games which have been defined and throughly tested on the general game player. All the games in this section work as intended. A compilation video of many of these games being played has been produced to better show this [16]. The player has currently been tested for the 9 unique games listed below:

#	Name	Category	Description
1	Light Puzzle	Single Player Puzzle	There are 3 switches with different effects on the lights. The aim is to turn on all the lights pressing only seven buttons.
2	3 Puzzle	Single Player Puzzle	This is a slider puzzle in a 2*2 grid, the aim is to order the numbers on the tiles
3	8 Puzzle	Single Player Puzzle	The same as 3 Puzzle but on a 3*3 grid.
4	Tic Tac Toe	Standard 2 player game	A two player game where the aim is to make a line of X or O in a 3*3 grid.
5	Eot Cat Cit	Standard 2 player game	A variant of Tic Tac Toe where the first player to make a line loses.
6	Horseshoe	Standard 2 player game	A two player game where the aim is to box in your opponent so they cannot move.
7	Duikoshi	Standard 2 player game	A two player variant of sudoku.
8	High or low	Incomplete information	A card game. Each Player is dealt a **random** card and must guess if the card is higher or lower than the dealers.
9	Blind Tic Tac Toe	Incomplete information and simultaneous moves	S variant of Tic Tac Toe where both player make their **moves simultaneously**. If they choose the same tile the player who gets to mark it is selected **randomly**.

6.2.2 Propositional Network Performance

Below is a list of results relating to the propnets performance for each game in Section 6.2.1. These results were produced by the $PropNetPerformace$ class. A discussion of the results can be found in Section 6.3.

Game	#Nodes	Build Time	Average # States Processed Per Minute
Light Puzzle	434	0.011 Seconds	932,376 States
3 Puzzle	2,566	0.297 Seconds	139,656 States
8 Puzzle	35,287	101.621 Seconds	8,436 States
Tic Tac Toe	3,206	0.603 Seconds	146,148 States
Eot Cat Cit	3,024	.594 Seconds	135,540 States
Horseshoe	9,926	3.964 Seconds	32,904 States
Duikoshi	44,715	67.716 Seconds	5,952 States
High or Low	4,873	0.98 Seconds	141,276 States
Blind Tic Tac Toe	6,274	0.655 Seconds	85,032 States

6.2.3 Quality Of Gameplay

To test the quality/skill of the player, it was played against a random legal player. This was used as a baseline to measure the player's performance. The results of testing with five seconds allocated to making a move each turn have been included below. These results were produced by the $WinRateTest$ class and additional results with different times per move can be found in the test_results folder of the project submission.

MCTS player vs Random Player					
Game	**Time Per Move**	**#Matches**	**%Won**	**%Drawn**	**%Lost**
Tic Tac Toe	5 Seconds	1000	98%	1%	1%
Horseshoe	5 Seconds	1000	88%	0%	12%
Duikoshi	5 Seconds	1000	94%	0%	6%
Blind Tic Tac Toe	5 Seconds	1000	62%	33%	5%

6.3 Evaluation Of Results

The results of the test phase of this project have yielded very promising results as a whole.

The first set of results in section 6.2.1 has shown the range of games which have been tested on this player. These results show a diverse selection of games have been tested on the player. The fact that the player has been shown to play this wide range of games correctly to completion [16], in combination with the thorough testing methodology which was used in its development, yields a high level of confidence that the player works as intended for games that it is given a correct description of.

However, all the games tested are relatively simple. They do not indicate how the player performs with very complex games. The testing is lacking in this aspect. The reason for this is that the time taken to build the propositional network is currently very long for large games making it very difficult test them effectively. The building of the needs to be further optimized to handle this.

The testing performed on the propositional network has yielded both positive and negative results. The positive result is that the speed at which the network is capable of processing game states particularly for simpler games is exceptionally fast. For example in the game Tic Tac Toe where there are 5478 states in the entire game tree, the network can transition through every possible state in the game in approximately 2.5 seconds.

However, as mentioned before, as the games get more complex for example Duikoshi the propnet slows dramatically, as the size of the network needed to describe the game becomes very large. From manually inspecting the network it can be seen that it is in fact much larger than it needs to be. Multiple inefficiencies can be found which could be optimized to greatly improve performance. These are discussed in section 7.2.

Largely due to the speed of the propnet the quality of gameplay is very good. Since many random playouts can be performed by the MCTS it is able to select good moves. However, once again these results only indicate how it performs on simple and some moderately complex games.

The player can make mostly better than random moves very quickly with only one second per move. However, the win rates continue to increase when given more time per move. With approximately 5 seconds per turn, it can be seen that the player is making far better than random moves. It consistently produces a very high win rate across multiple games tested in section 6.2.3. For these smaller games there seems to be diminishing returns around the 5-10 second range, yet small improvements are continually made with additional time.

Chapter 7: **Conclusion & Future Work**

7.1 Conclusion

The main aim of the *General Game Player* project was to develop a piece of software which could be told the rules of many 2d games using the standard game development language and then play those games. In many ways this project has been extremely successful at meeting and even surpassing this goal. In addition every single mandatory, discretionary and exceptional goal of the project specification has been met. However, it has been limited to a certain extent in both the complexity of the games it can play and graphically display.

A completely functional general game player has been developed. The player can take a description for a game in GDL and play it to completion with any combination of human or AI players.

Nine games have been used as test cases to illustrate the wide range of games the player is capable of playing. They include card games, board games, puzzles, well known games and obscure games. They even go beyond what was originally hoped for by playing games which cannot be described in the standard game description language (games of incomplete information) which was one of the exceptional goals of the project.

The player also surpassed the goal of simply playing these games correctly. A random legal player is capable of that. The player plays these games much better than randomly as can be seen in Section 6.3. With no prior game knowledge of what is good or bad it is capable of identifying good moves and very often for simple games the best move.

These games can be played easily by the user through a graphical interface which generates unique graphics for each game and also lists the possible legal moves for the player to make.

However, the graphics that can be produced are very basic. The only graphical information which can be specified in game descriptions currently is an image and the co-ordinates it should be drawn at. Though this can be made to work for a great many games if the user is creative, it is not an elegant solution. Many more tools and options could be added to allow for more complex graphics. For example sprites, layering of images or drawing lines between coordinates.

Also as mentioned before, there are limitations to the complexity of the games this player can handle. Though theoretically this player can play any game which can be correctly defined in the game description language and even some which cannot, in practice for complex games such as chess or go it simply takes too long to build the propositional network needed to play the game. Far and above any other issue, that is the greatest limitation of this project.

In spite of this limitation, by no means should the use of propositional networks in general game playing be condemned. Once built, the ability to process game states using propositional networks has proven to be very effective even if it currently only works for more simple games.

There will always be some trade off when using a propositional network where time must be spent building the network prior to the game, so that the game can be played faster and better. However, there are several ways in which the network could be optimized to reduce this time. This is a very promising area in general game playing which requires more experimentation and could potentially yield great advancements in the field of general game playing.

7.2 Future Work

This section covers future work which could be done to improve and extend the general game player developed for this project. There were two major areas of this project that have been identified which could be improved by future work.

- **Propositional Network Optimization:** as the complexity of games increase, the size of the networks built by this player to represent them increases rapidly. To effectively use a propositional network for representing complex games this needs to be optimized to make the networks smaller. There are many ways in which this could be done. The image below is taken from a propnet built for Tic Tac Toe. It illustrates two examples of optimizations that could be made.

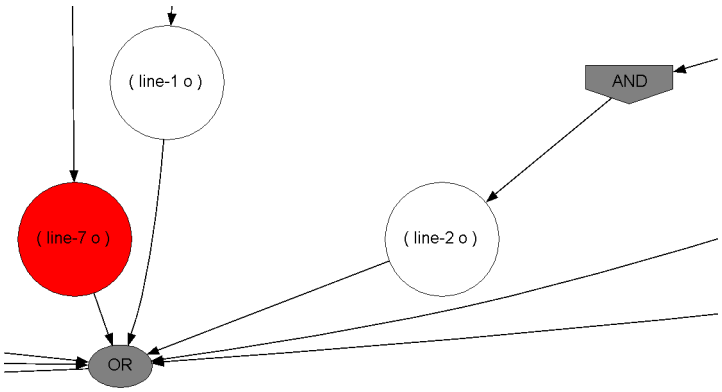


Figure 7.1: Network inefficiencies

At the top right of Figure 7.1 there is an And-Gate. It has only one input making the gate superfluous. It could be removed. There are also multiple nodes to represent the different ways in which (Line O) can become true. These could all be replaced by direct inputs to the Or-Gate. These are two examples of how the propnet could be optimized but there are other many ways the network could be better optimized in the future.

Additionally in order to mitigate the time needed to build the network serialization could be implemented to write the network to a file so it could be read later. Users would then only need the network to be built once and could quickly load it from a file in the future.

- **User Interface:** the graphical capability of the player is very basic. It can only draw static images at preassigned coordinates. Many improvements and extensions could be made to this system. Examples are allowing images to be layered, the use of sprites, highlighting cells. There are too many possibilities to list.

  Looking beyond the graphical component of the user interface there are also many functional improvements which could be added. For example options to undo moves, save the current state of the game, ask the AI for a hint. These kinds of additions could vastly improve the end user experience and are definitely worthy of future development.

Another potential use for this project which could be pursued further is the testing of various AI for general game playing. The software developed for this project could prove to be an excellent testbed for prototyping and experimenting with new player AI's, in particular if the improvements previously discussed in this section were made. New AI can be very easily implemented into this system by extending the $PropNetPlayer$ class which contains all the essential functionality for a player. Also a suite of reusable tests have been designed to quickly measure a new AI's performance across multiple games against other AI seen in Section 6.2.3. This could be very useful for testing new AI in the GGP field without needing to build an entire GGP system.

Appendix I: **Sample GDL Description**

This is a very simple game described in GDL for demonstration purposes. In this game there is one player and a single light, if the player turns the light on he wins. The **bold** *font is the actual GDL.*

role is the keyword which is used to specify players in the game.
(role player)

This is a base proposition. It is used to specify the domain of variables within propositions in the description.
(base light (on off))

This is the set of propositions which are true at the start of the game.
(init (light off))

<= indicates this is a rule. The proposition which follows it(the head of the rule) will become true if the rest of the propositions in the rule are true (the body of the rule). legal represents a move a player can make. Here it is legal for player to press switch if it is true the light is off.
(<= (legal player (press switch))(true (light off)))

next represents a proposition which will become true next turn. Here the light will be on next turn if player presses the switch.
(<= (next (light on))(does player (press switch))))

terminal represents the end of the game, if the light is on the game ends.
(<= terminal(true (light on)))

Appendix II: **ZGRViewer For Propnets**

ZGRViewer is a graph visualizer tool for displaying graphs expressed in the DOT language. Dot files for several networks have been included in the $Sample\_Propositional\_Networks$ folder of the project submission. Instructions for how to use this tool to view these networks can be found on the ZGRViewer website [16]. Alternatively a video has been produced to show a propositional network for the game of Tic Tac Toe [17] which can simply be viewed instead.

References

[1] Bjornsson, Y., & Finnsson, H. (2009). CadiaPlayer: A Simulation-Based General Game Player [Abstract]. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1), 4-15 . doi:10.1109/tciaig.2009.2018702

[2] Swiechowski, M., Park, H., Mandziuk, J., & Kim, K. (2015). Recent Advances in General Game Playing [Abstract]. *The Scientific World Journal*, 2015, 1-22. doi:10.1155/2015/986262

[3] Kendall, G., & Whitwell, G. (2001). An evolutionary approach for the tuning of a chess evaluation function using population dynamics. *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, 1-8. doi:10.1109/cec.2001.934299

[4] Genesereth, M., & Thielscher, M. (2014). General Game Playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2), 1-229. doi:10.2200/s00564ed1v01y201311aim024

[5] Love, N., Hinrichs, T., & Genesereth, M. (2016, April 4). General Game Playing: Game Description Language Specification. Retrieved April 4, 2017, from http://logic.stanford.edu/reports/LG-2006-01

[6] Jha, A. K., & Ramjas, S. M. (2006). An Introduction To Deductive Database And Its Query Evaluation. *International Journal of Advanced Computer Technology* , 4(2).

[7] Genesereth, M. (2013). Deductive Databases. Retrieved April 4, 2017, from http://logic.stanford.edu/logicprogramming/notes/ddb.html

[8] Genesereth, M. (2013, May 2). Chapter 9 - Propositional Nets. Retrieved April 4, 2017, from https://web.archive.org/web/20161220033617/http://logic.stanford.edu/ggp/chapters/chapter_09.html

[9] Genesereth, M. (2013). Chapter 6 - Small Multi-Player Games. Retrieved April 04, 2017, from https://web.archive.org/web/20130419011744/http://logic.stanford.edu/ggp/chapters/chapter_06.html

[10] Genesereth, M. (2013). Chapter 7 - Heuristic Search. Retrieved April 04, 2017, from https://web.archive.org/web/20160418091450/http://logic.stanford.edu/ggp/chapters/chapter_07.html

[11] Schiffel , S., & Thielscher, M. (2007). Fluxplayer: A Successful General Game Player. Retrieved April 4, 2017, from http://cgi.cse.unsw.edu.au/m̃it/Papers/AAAI07a

[12] Game Playing with Propnets. (2016, March 30). Retrieved April 04, 2017, from https://youtu.be/whGaYMxBu8o?list=PLoNVFS-hCert8MHid8ldxFtI9UZR9yqUC

[13] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., . . . Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 1-43. doi:10.1109/tciaig.2012.2186810

[14] Gansner, E. R., Hu, Y., & Kobourov, S. G. (2010). GMap: Drawing Graphs as Maps. *Graph Drawing Lecture Notes in Computer Science*, 405-407. doi:10.1007/978-3-642-11805-0_38

[15] Pietriga, Emmanuel . "*ZGRViewer, a GraphViz/DOT Viewer.*" ZGRViewer. INRIA, 11 Mar. 2015. Web. 06 Apr. 2017. ¡http://zvtm.sourceforge.net/zgrviewer.html¿.

[16] Keating, J. (2017, April 06). General_Game_Player_FYP_Gameplay_Compilation. Retrieved April 06, 2017, from https://vimeo.com/209013179

[17] Keating, J. (2017). PropNet. Retrieved April 06, 2017, from https://vimeo.com/209135857