School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Optimising Deep Learning and Search for Imperfect-Information Games in General Game Playing

by

Joel West[1]
Student ID: z5311058

Thesis submitted as a requirement for the degree of
Bachelor of Advanced Computer Science in Computer Science

Supervisor: Professor Michael Thielscher[2]
Submitted: 7 August 2024

[1]joel.west@student.unsw.edu.au
[2]mit@unsw.edu.au

Abstract

Within general game playing, computer systems are able to play any arbitrary game given only its rules and time to train. General game playing in imperfect-information games (e.g., Poker) is a new and particularly challenging area of research. Furthermore, the field is seen as a step towards general purpose artificial intelligence with games that are more akin to real-life tasks where not all information is accessible (e.g., self-driving cars). This research attempts to re-implement a previous imperfect-information general game playing framework that utilises self-play, deep reinforcement learning and search to play larger games and more optimally.

Acknowledgements

Special thanks to my supervisor, Professor Michael Thielscher, for his tremendous expertise and guidance throughout my thesis. Additionally, I'd like to thank my family and friends who've supported me throughout my time at university, especially in this final year.

Abbreviations

AI

Artificial Intelligence

ReBeL

Recursive Belief-Based Learning

CFR

Counterfactual Regret Minimisation

CFR-D

Counterfactual Regret Minimisation-Decomposition

DLS

Depth Limited Solving

EV

Expected Value

GDL

Game Description Language

GDL-II

Game Description Language with Incomplete Information

GGP

General Game Playing/Player

HUNL

Heads-Up No-Limit Texas Hold'em Poker

MCCFR

Monte Carlo Counterfactual Regret Minimisation

MCTS

Monte Carlo Tree Search

MSNE

Mixed Strategy Nash Equilibrium

RPS+

Modified Rock, Paper, Scissors

UCT

Upper Confidence Bounds for Trees

Contents

1 Introduction

Since the inception of computers in the 1940s and 1950s, game playing computer systems have served as a reproducible way to benchmark and develop AI techniques [23]. Through combined advancements in hardware and AI techniques, computer systems have continuously improved. One historic milestone is IBM's Deep Blue [7], which in 1997 defeated then World Chess Champion Garry Kasparov; something that was previously considered impossible in a game as large and complex as chess. However, Deep Blue required many game-specific components such as an evaluation function and a database of Grandmaster games [7]. Furthermore, experts handcraft much of the logic behind Deep Blue and other similar systems, thus many of the techniques developed have little applicability to real-world tasks.

General Game Playing (GGP) seeks to address this in an effort to move towards general purpose AI, where computer systems perform a variety of tasks at a superhuman level. In GGP, programs receive rules to games at runtime; thus eliminating any reliance on game specific strategies and knowledge [12]. Policies must instead be learnt by the GGP through various techniques such as self-play. Historically, GGP has been applied to perfect information games wherein all information is public and games are deterministic. A further generalisation to GGP is their application to imperfect-information games, where some information may be private and games may be non-deterministic, a superset of perfect information games.

Multiple GGP frameworks for imperfect-information games currently exist [9, 11, 19, 21], however each exhibit significant drawbacks as the field hasn't yet matured. Of particular note is a previous framework using deep reinforcement learning and search for GGP in imperfect-information games created by Partridge and Thielscher [19], which adapts ideas from current Poker AI research to GGP. The framework, written in Python, displays strong performance in a number of difficult games such as Blind Tic-Tac-Toe and Liar's Dice; however, the implementation remains unoptimised and is not yet scaleable to larger games. The goal of this research is to improve upon this framework in key areas in order to play stronger and converge more quickly to an optimal strategy. We achieve this by firstly re-implementing the framework in C++, a more performant language typically used in game playing. Secondly, hardware utilisation was improved by enabling GPU support and parallelising critical areas.

The improved framework yields significantly stronger performance in search, with convergence occurring orders of magnitude faster; enabling more complex games to be trained on and played. However, the overall results in terms of strength benchmarks against other agents show mixed results with only small or no improvements over the original implementation observed.

2 Background

2.1 Game Theory Terminology

Game theory is foundational to game playing. Definitions are shown below [3, 16, 20].

Number of Players (n)

The number of players within the game.

Player Set (P)

The set of all players. For simplicity, assume $P = \{1, 2, ..., n\}$.

State (s)

A state within a game contains all information about a given position, both public and private.

Infostate (s_i)

An infostate of player $i \in P$ is a history of all actions and observations visible to i. There are often many possible states given an infostate.

Subgame

A subgame is a partition of the original game starting at some root state.

Public Belief State (β)

A public belief state is a probability distribution each player's infostates. I.e., if ΔS_i is a probability distribution over each of player i's possible infostate $s_i \in S_i$, then the current public belief state is $\beta = (\Delta S_1, \Delta S_2, \ldots, \Delta S_n)$.

Action Set (A_i)

The set of possible actions that can be taken by player $i \in P$.

Policy (p_i)

The policy (or strategy) of player $i \in P$. Defined as a probability distribution over all actions $a_i \in A_i$. Formally, p_i must satisfy the following conditions.

1. $p_i \in \mathbb{R}^{|A_i|}$
2. $\sum_{a_i \in A_i} p_i(a_i) = 1$
3. $\forall a_i \in A_i$, $p_i(a_i) \in [0, 1]$

If the infostate the policy corresponds to is arbitrary or unclear, it can optionally be written as $p_i(s_i, a_i)$.

Policy Set (ΔA_i)

The set of possible policies that player $i \in P$ can choose.

Joint Policy Excepting i (p_{-i})

The joint policies of all players besides player $i \in P$. I.e., $p_{-i} = (p_1, p_2, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n) \in \Delta A_1 \times \Delta A_2 \times \cdots \times \Delta A_{i-1} \times \Delta A_{i+1} \cdots \times \Delta A_n$.

Joint Policy (p)

The joint policy of all players. I.e., $p = (p_i, p_{-i}) = (p_1, p_2, \dots, p_n) \in \Delta A_1 \times \Delta A_2 \times \dots \times \Delta A_n$.

Utility Function (u_i)

A function where $u_i(p) = u_i(p_i, p_{-i})$ signifies the *expected* utility (reward) player $i \in P$ will receive when joint policy p is played.

Mixed Strategy Nash Equilibrium (MSNE)

A variant of Nash equilibrium which allows non-deterministic policies, as described here. A joint policy is a MSNE given no player can gain any expected reward by unilaterally deviating from p. More formally, $p = (p_i, p_{-i})$ is a Nash equilibrium if $\forall i$, $p_i \in \arg\max_{x \in \Delta A_i}(u_i(x, p_{-i}))$.

Reach Probability (π_i^p)

A function where $\pi_i^p(s_i)$ signifies the probability that player i will reach infostate s_i given that players play according to p throughout the game. Similarly, $\pi_i^p(s_i, s_i')$ represents the probability of reaching infostate s_i' given the player is currently in s_i.

2.2 GGP in Perfect Information Games

2.2.1 Monte Carlo Tree Search and Upper Confidence Bounds for Trees

Monte Carlo Tree Search (MCTS) is a framework that allows for randomised exploration of a game tree by iteratively expanding the tree as the search progresses [8]. Basic MCTS performs the following procedure for each iteration:

1. **Selection** begins at the root node and traverses down the tree by recursively selecting child nodes based on data stored in the current node. There are numerous methods to perform selection that balance exploitation and exploration such as Upper Confidence Bounds for Trees (UCT), which is discussed shortly.

2. **Expansion** occurs once selection reaches a node outside the tree. The new node added to the tree.

3. **Simulation** entails choosing actions at random from the new node before reaching a terminal state.

4. **Backpropagation** involves propagating the terminal state's value up the game tree whilst updating the data within each node visited during the selection phase.

Upon termination, the algorithm takes the most commonly visited child of the root node as the optimal next state and the player plays the action corresponding to this optimal state. See below for an illustration of this process.

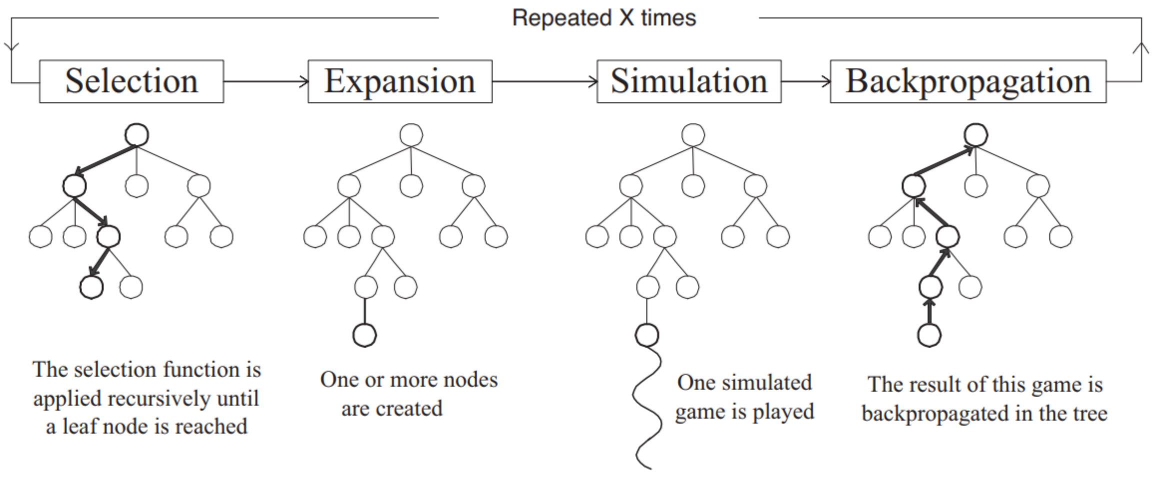

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: Stages of MCTS [8].

UCT provides an efficient way to recursively select nodes during the selection stage of MCTS whilst balancing exploitation and exploration [5]. This requires each node i to store $Q(i)$, the sum of all rewards that have passed through i, and $N(i)$, the number of times i has been visited. At node i, UCT selects a node from the set,

$$\underset{j \in \text{ set of } i\text{'s children}}{\arg\max} \left(\frac{Q(j)}{N(j)} + C_p \sqrt{\frac{\log(N(i))}{N(j)}} \right),$$

where $C_p > 0$ is a constant [5]. Similar optimisations can be made to MCTS during simulation phase by using heuristics (e.g., a neural network) to guide the search towards more promising sequences [8].

MCTS boasts several advantages over traditional search techniques. Most notable is the lack of domain knowledge required in creating an evaluation function which has significant value within GGP [8].

2.2.2 AlphaZero

AlphaZero is an algorithm developed by DeepMind that combines MCTS and deep reinforcement learning [24]. A deep neural network f with parameters θ is learnt via self-play. $f_\theta(s) = (p_i, v)$ then predicts player i's optimal policy p_i (where it is i's turn currently) and the value v of given a state s. At runtime, f_θ augments MCTS by informing the selection phase with p_i and removing the simulation phase by instead backpropagating v. Despite receiving no domain specific knowledge, the algorithm outperformed previously state-of-the-art opponents Stockfish, Elmo and AlphaGo Zero in Chess, Shogi and Go respectively.

2.2.3 Game Description Language

In order to partake in GGP, computer systems require a precise and machine-processable language capable of describing arbitrary games [26]. The Game Description Language (GDL) is a language that belongs to the logical programming paradigm and can be used to describe any deterministic perfect information game. The keywords of GDL are detailed below [13].

`role(player)`
: `player` is a role within the game.

`input(player, action)`
: `action` is a possible action for `player`.

`base(proposition)`
: `proposition` is a base proposition for the game.

`init(proposition)`
: `proposition` is true at the beginning of the game.

`true(proposition)`
: `proposition` is true in the current state.

`does(player, action)`
: True if `player` has played `action` in the current state.

`next(proposition)`
: `proposition` is true in the next state.

`legal(player, action)`
: `action` is a legal action for `player`

`goal(player, value)`
: `player` gets utility `value` in the current state.

`terminal`
: The current state is terminal.

An example that defines Tic-Tac-Toe is shown in section 7.1.

2.2.4 Propositional Networks

General game players can convert games given in GDL into a propositional network, which provides an interface that models said game [14] whilst being more computationally efficient [10]. Formally, they consist of a bipartite graph [10] (i.e., nodes can be partitioned into two disjoint sets with edges existing only between nodes of differing sets). Nodes represent propositions (forming one disjoint set), boolean gates or transitions (with boolean gates and transitions forming the remaining disjoint set). Propositions derive their truth value from the inputs of players, transitions or boolean gates. Transitions (also known as views) determine information's passage between states and boolean gates apply a logical function to their inputs (e.g., *AND*, *OR* and *NOT*) to compute a truth value [10].

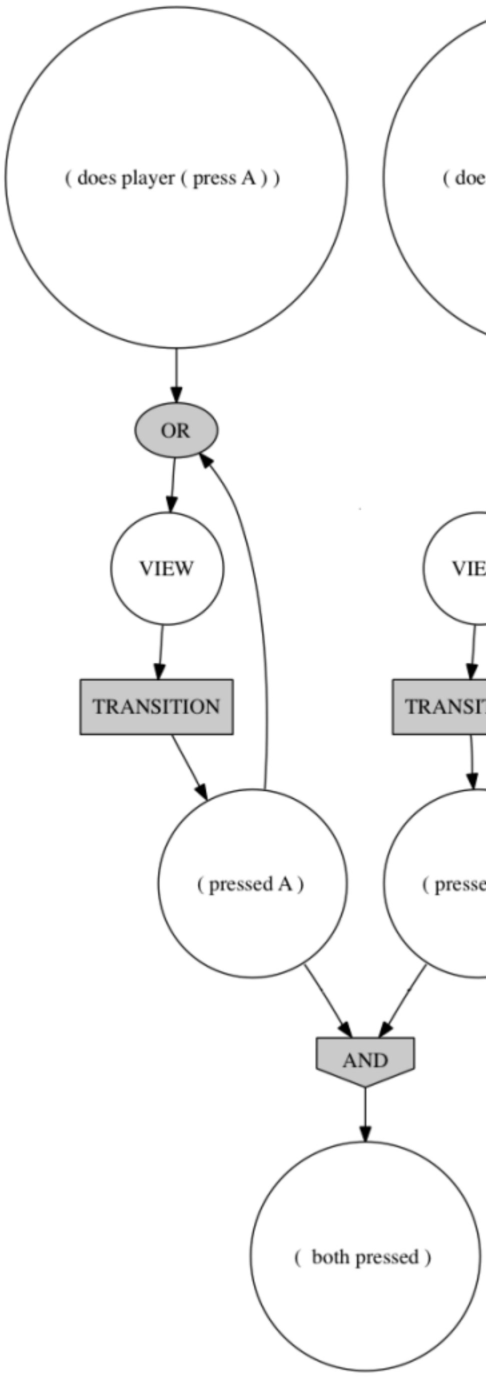


Figure 2: Example of a propositional network of a game where two buttons A and B can be pressed. Once pressed, the buttons cannot be unpressed [10].

A publicly available program, `ggp-base` [22], has previously been extended to be able to transform GDL-II files into Python-based propositional networks [14, 19]. These Python files are then loaded and interpreted by an interface written in Cython, a super-set of Python which can be transformed into C code and compiled [2]. The interface allows for queries from each player about visible information for the current state and can transition to new states using inputs taken from players.

2.2.5 Generalised AlphaZero

Whilst AlphaZero can play many games based only on rules, it is limited to two-player, zero-sum, symmetric, turn based, perfect information games [14]. Furthermore, AlphaZero requires input from a handcrafted board. Generalised AlphaZero adapts AlphaZero by relaxing these limitations for a GGP context. See below for the most relevant extensions to this research.

Handcrafted Board

AlphaZero receives input for its neural network via a handcrafted board. This is impossible for GGP as programs must play games with rules given at runtime. Generalised AlphaZero replaces this with a propositional network.

Symmetry

AlphaZero has a single neural network, f_θ, common for all players. Generalised AlphaZero's neural network instead has a separate head for each player which accounts for asymmetric roles and policies. Importantly, it is unnecessary for each player to possess its own feature extraction as the networks will simply extract the same information from the propositional network.

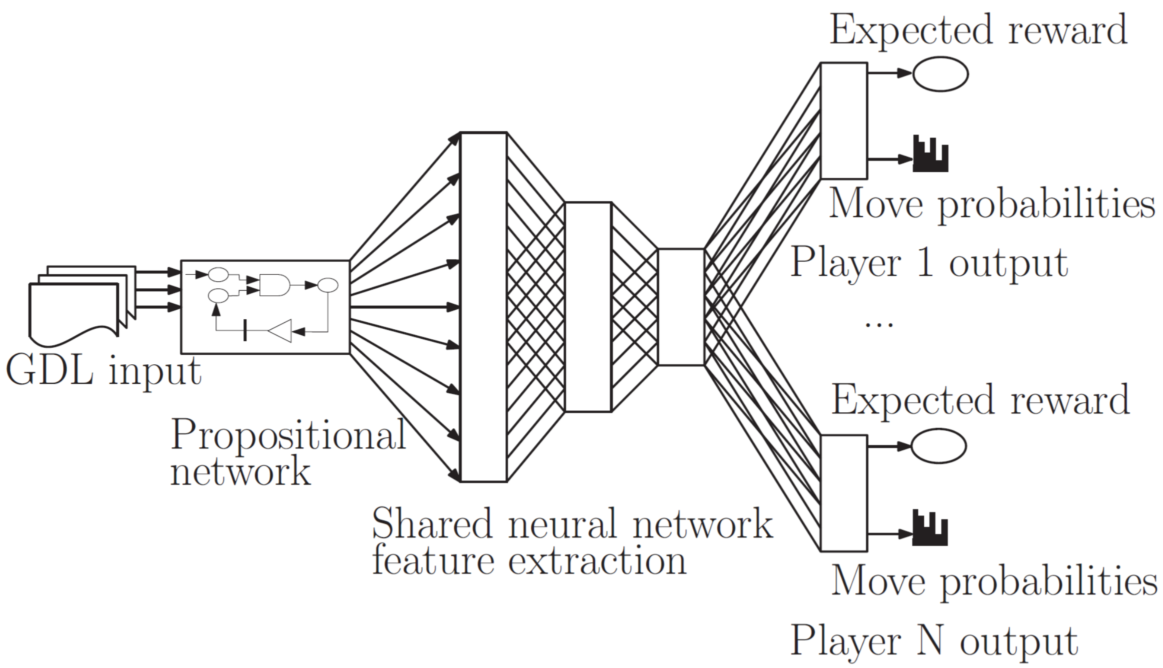


Figure 3: Neural network framework of Generalised AlphaZero [14].

With these and other adaptations Generalised AlphaZero is able to play any perfect information game defined by GDL. Furthermore, the algorithm outperformed a basic UCT agent in a range of games [14].

2.3 Game Playing in Imperfect-Information Games

Within the real world, the vast majority of tasks have imperfect-information (e.g., driving, modelling financial markets, negotiating etc.). Thus, concepts and strategies within imperfect-information games are more relevant in creating useful, real world AI systems.

2.3.1 Flaws of Perfect Information Search in Imperfect-Information Games

Unfortunately, imperfect-information games are fundamentally different to perfect information games because state values do not have a natural definition [4]. As a consequence, many of the previously mentioned techniques like MCTS do not perform well in imperfect-information games. For example, consider a game of Modified Rock, Paper, Scissors (RPS+) where if either agent plays scissors the magnitude of utility lost or gained is doubled for both players. Importantly, in RPS+ player 2 cannot see player 1's action until after they've chosen their action.

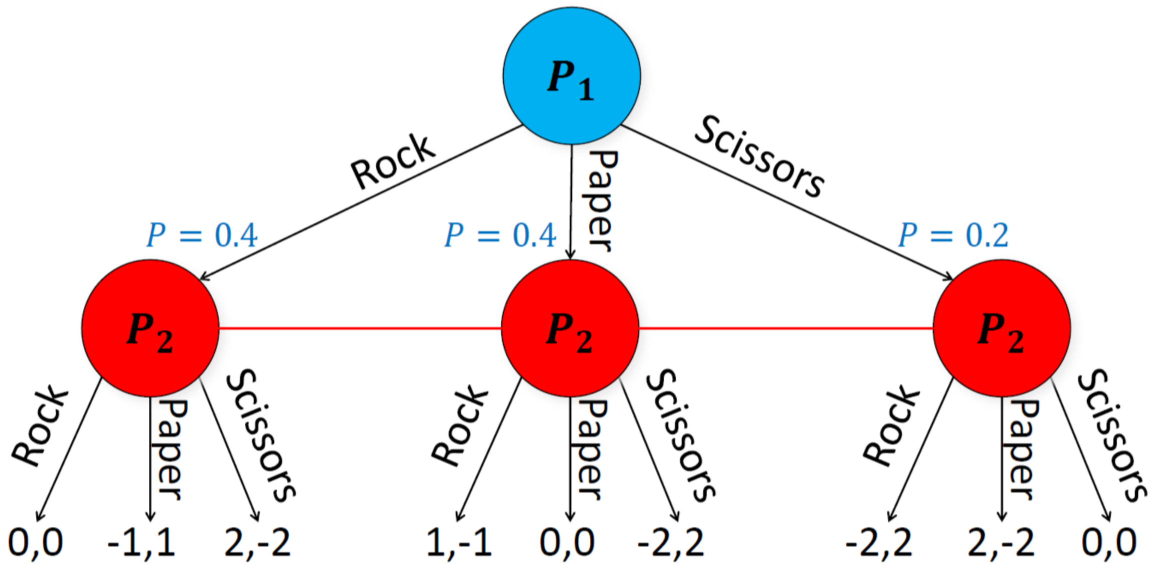


Figure 4: Game tree of RPS+ showing the payoffs and MSNE strategy [4].

Let the probability of player i playing rock, paper and scissors be r_i, p_i and s_i respectively. Furthermore, let player i's policy be $p_i = (r_i, p_i, s_i) \in P_i$. The MSNE of RPS+ is $p_1 = p_2 = (0.4, 0.4, 0.2)$. Assuming player 2 plays according to this MSNE, perfect information searches will assign values to states as shown below.

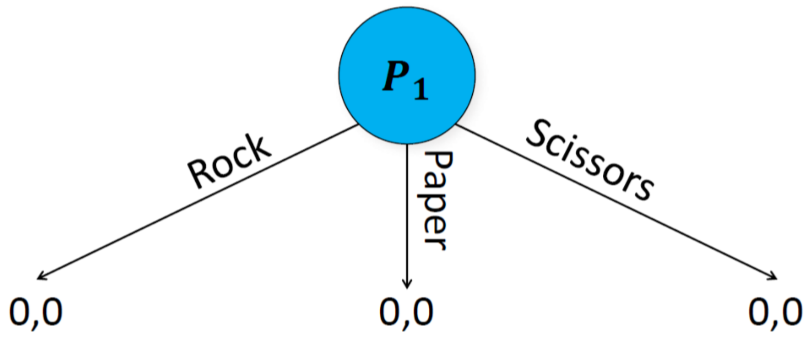


Figure 5: Game tree of RPS+ with player 2's state values propagated up the tree [4].

Given only these state values it's impossible for player 1 to determine the MSNE $p_1 = (0.4, 0.4, 0.2)$. Crucially, the optimal policy for player 1 depends on player 2's policy and, assuming opponents can adapt, the value of each action depends on the probability with which it is chosen. Thus, the values of states as defined within perfect information games, are insufficient to reconstruct a MSNE.

2.3.2 Counterfactual Regret Minimisation

Counterfactual Regret Minimisation (CFR) is a stochastic gradient descent algorithm that exhibits very strong performance when searching imperfect-information games. Before beginning a search, variables corresponding to all infostates for all players are created. In the simplest version of CFR each infostate variable stores the policy used on each iteration as well as the cumulative regret across all iterations, which is initially 0. Firstly, define the counterfactual value $v_i(s_i, a_i)$ and counterfactual regret $r_i(s_i, p, a_i)$ via,

$$v_i(s_i, p) = \sum_{z_i \in Z_{s_i}} \pi^p_{-i}(s_i)\pi^p_i(s_i, z_i)u_i(z_i),$$

$$r_i(s_i, p, a_i) = v_i(p, s_i) - v_i(a_i, s_i),$$

where $z_i \in Z_{s_i}$ is a terminal infostate that can result from s_i [16]. Intuitively, the counterfactual regret $r_i(s_i, p, a_i)$ for player i is the expected utility lost by playing some policy p_i in information set s_i instead of always taking a single action a_i. Furthermore, on some iteration T define the cumulative counterfactual regret $R^T_i(s_i, a_i)$ and positive cumulative counterfactual regret $R^{T,+}_i(s_i, a_i)$ as [16],

$$R^T_i(s_i, a_i) = \sum_{t=1}^{T} r_i(s_i, p^t, a_i) \text{ and } R^{T,+}_i(s_i, a_i) = \max(R^T_i(s_i, a_i), 0).$$

Starting at a grounded state, on each iteration T CFR plays against itself by first computing and saving a new policy p_i^T for each player i in each information set s_i through a process of regret-matching. In regret-matching actions are weighted by positive cumulative counterfactual regret [16],

$$p_i^T(s_i, a_i) = \begin{cases} \frac{R_i^{T,+}(s_i,a_i)}{\sum_{a'_i \in A_i} R_i^{T,+}(s_i,a'_i)} & \text{if the denominator} > 0, \\ \frac{1}{|A_i|} & \text{otherwise.} \end{cases}$$

These policies are used to recursively compute the counterfactual regret for each infostate, which is added to the cumulative counterfactual regret. This process can be repeated indefinitely. Once the search is terminated on iteration T the average policy $\bar{p_i}^T$ is calculated,

$$\bar{p_i}^T(s_i, a_i) = \frac{\sum_{t=1}^{T} \pi_i^{p^t}(s_i) p_i^t(s_i, a_i)}{\sum_{t=1}^{T} \pi_i^{p^t}(s_i)},$$

which provably converges to an epsilon approximate MSNE [16].

There are many optimisations that can be made to CFR including pruning [16] to reduce the search space and other variations that provide specific advantages. Variants of particular interest to this research include CFR-Decomposition (CFR-D) which allows subgames to be solved independently in a theoretically sound manner [6] and Monte Carlo CFR (MCCFR) which samples states instead of conducting a full game tree search [16] as Vanilla CFR does.

There are many further sub-variants which each determine the sampling procedure used and offer slightly different advantages such as external sampling, which does not require the reach probabilities to be explicitly calculated (as they're implicitly factored in through the sampling) and thus simplifies the implementation within GGP. Additionally, MCCFR with external sampling empirically converges significantly faster than Vanilla CFR, particularly in games with a high branching factor [16].

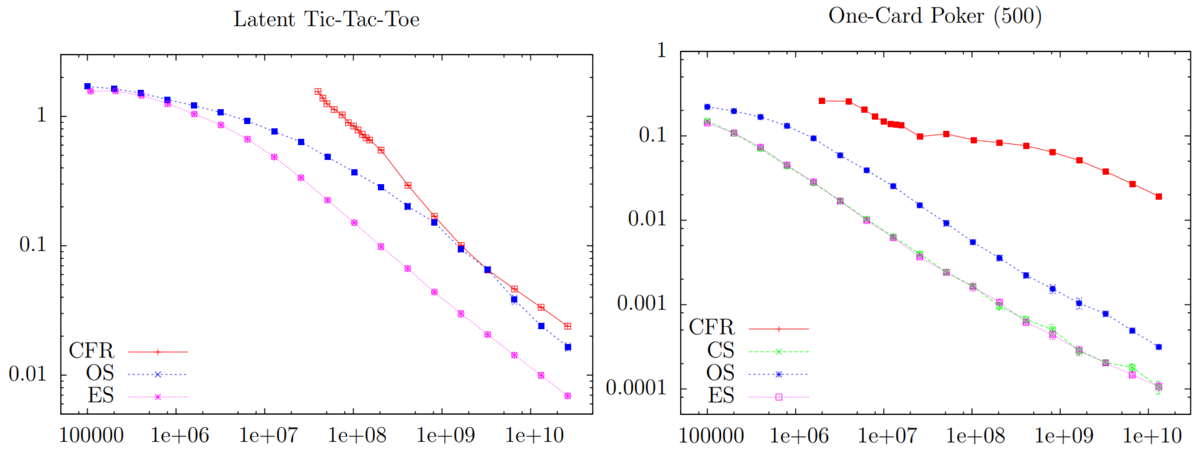


Figure 6: Comparison between Vanilla CFR (CFR) and MCCFR using various sampling techniques including Change Sampling (CS), Outcome Sampling (OS) and External Sampling (ES). y-axis is logarithmic in exploitability and x-axis is logarithmic in number of nodes searched. Exploitability in this context is the sum of expected utility lost compared to a known MSNE from all players [16].

2.3.3 Depth Limited Solving

Brown et al. provide a methodology that approximates a MSNE named Depth Limited Solving (DLS) [4]. Originally tested by playing Heads-Up No-Limit Texas hold'em poker (HUNL), DLS is applicable to any imperfect-information game. Furthermore, given only a fraction of the computing power the algorithm defeated state-of-the-art-opponents whilst exhibiting low exploitability.

During training DLS approximates a MSNE for the entire game referred to as a blueprint strategy. Additionally, the program generates a set of approximately 10 strategies for the opponent either by biasing the opponent's blueprint strategy or allowing them to be learnt via self-play.

At runtime DLS expands the current state to a predetermined depth-limit to create a depth-limited subgame. For each infostate at the leaf state the opponent chooses of strategy. Next, the algorithm simulates the game to a terminal state from each leaf state and propagates values up the game tree to determine an approximate MSNE.

2.3.4 Recursive Belief-Based Learning

Recursive Belief-Based Learning (ReBeL) is an AlphaZero-like algorithm developed to play HUNL that utilises both deep reinforcement learning and search [3]. It achieved superhuman performance, outperforming previous Poker AIs by a significant margin [4, 17]. It does this by transforming imperfect-information games into a high-dimensional continuous state and action space perfect information games [3]. The public belief state β is embedded within states with players announcing their policy for all sampled infostates from β; an appropriate and common approach within game theory as agents assume their policy is public to avoid being modelled and exploited. A referee then selects an action for each player using their actual infostate and announced policy and updates β based on any public observations.

Similar to AlphaZero, ReBeL trains a value function $\hat{v} : B \to \mathbb{R}^{|S_1|+|S_2|}$, where B is the set of all β and S_i is the set of all infostates of player i given β, via deep learning and self-play. The output $\hat{v}(\beta) = \boldsymbol{v}$ is a value for each possible infostate every player could be within given β. Optionally, another function $\hat{\Pi} : B \to (\Delta A)^{|S_1|+|S_2|}$ can be learnt to similarly predict policies.

The procedure used for both training and playing is common [3]. Assume that in a two-player game player 1 is the program and player 2 is the opponent. Player 1 searches from a root state with public belief state β by initialising it's policy p_1 with either uniform random or $\hat{\Pi}(\beta)$. Next, ReBeL searches a depth-limited subgame with leaf values estimated using $\hat{v}(\beta)$ using CFR-D, although any imperfect-information search algorithm such as other CFR algorithms or fictitious self play are appropriate. The algorithm will iteratively solve these depth-limited subgames with the average policy converging to a MSNE.

2.4 GGP in Imperfect-Information Games

2.4.1 Game Description Language with Imperfect-Information

The Game Description Language with Incomplete Information (GDL-II) extends GDL in order to allow for both stochastic events and imperfect-information with the addition of just two keywords [25].

`random`

A player that chooses actions at uniform random.

`sees(player, percept)`

`player` gets information on the state of `percept`.

This addition allow any stochastic and imperfect-information game to be represented [26]. An example for how to define the Monty Hall problem can be seen in section 7.2 of the appendix.

2.4.2 Hidden Information GGP with Deep Learning and Search

Partridge and Thielscher describe a framework that generalises ReBeL for use in GGP [19]. Although depth-limited solving was considered, it ultimately was not utilised presumably because of the precomputation required to determine a blueprint strategy and a set of opponent strategies; a difficult task without domain knowledge. ReBeL was instead chosen likely due to the lack of domain knowledge required despite only provably converging to a MSNE within two-player zero-sum games [3].

The adaptations made to ReBeL are analogous to those made to AlphaZero to form Generalised AlphaZero. A propositional network was added to create an interface that can be generated at run-time. Furthermore, ReBeL contains a single head for its neural networks which does not account for asymmetric games. Instead, each player is given a different head which allows for differing policies and values depending on their roles. Adaptations can be broadly broken into subsections.

Extension of Propositional Network for GDL-II

`ggp-base`, which was modified by the authors of Generalised AlphaZero to transform GDL files into a Python-based propositional network, was further extended to handle GDL-II by accounting for the additional `sees` keyword. Note that as `random` is merely a special type of `role` it does not need to be accounted for to process GDL-II games. Python propositional network files are then loaded dynamically at runtime using `importlib`.

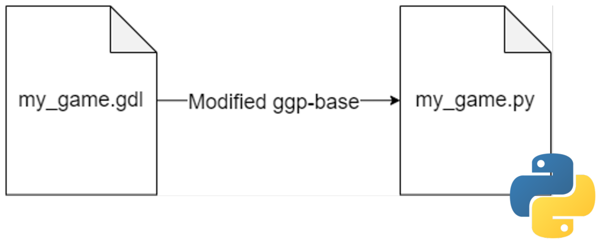


Figure 7: Creation of Python propositional network from GDL file.

Sampling Grounded States from Agent Histories

Authors describe multiple methods to sample states during training or runtime given an agents history of observations and actions. For example, one methodology conducts the following:

1. At the beginning of the game, create an empty cache for states incompatible with the agents observations.
2. For each sample begin at the initial state.

3. For the agent play the action recorded in it's history. For all other agents, play actions according to policies estimated via a neural network. Continue this process until the agents history is exhausted, at which point the current state is a sample.
4. If at any point the current state is contained within the invalid state cache, observations from the current state made are inconsistent with the recorded history or all possible proceeding states have been marked as invalid deem the current state as invalid. Furthermore, add the current state to the cache to avoid later unnecessary computation and backtrack to the previous state.

See the original paper for greater detail [19].

Vanilla CFR Search

Once a grounded state is sampled a time and depth limited Vanilla CFR search is conducted utilising the previously mentioned neural network which estimates the optimal policies and value for each role given a grounded state. Policy estimations are used to initialise the policies of unencountered nodes to speed up search convergence whilst value estimations are assigned to leaf nodes at the depth limit. Once the search terminates the resulting policy is saved to a buffer.

Search Consolidation

Once a pre-determined number of states are sampled and searched the resulting policies found are summed to create a single policy. This cumulative policy is then sampled to produce the resulting action.

To illustrate how each component described interacts pseudo-code for how an agent implementing the framework plays is provided below.

Algorithm 1 Function to get action from agent

```
1:  procedure getAction(propnet, state)
2:      observations ← propnet.getAgentObservations(state, role)
3:      cumulativePolicy ← $\vec{0}$
4:      for $i$ in 1 . . . sampleSize do
5:          sampledState ← sampler.sampleState(propnet, model, history, observations)
6:          policy ← depthLimitedCFR(propnet, model, sampledState, role)
7:          cumulativePolicy ← cumulativePolicy + policy
8:      end for
9:      action ← sample(cumulativePolicy)
10:     history.add(observations, action)
11:     return action
12: end procedure
```

The aforementioned neural network model used by the agent is trained via self-play utilising a similar methodology to that used at runtime. For each state, a depth limited search is conducted on the true state with the resulting policy and expected value added to a replay buffer. Next, using the runtime methodology actions for each player are generated. The model is then trained on batches of size 128 sampled from the replay buffer for 5 epochs. Finally, the actions previously computed are used to transition to the next state where the process repeats. Whenever a terminal state is reached the game is restarted at the initial state. See the below pseudo-code modified from [19] for more context into how training is conducted.

Algorithm 2 Training loop

```
1: procedure TRAIN(game, timeLimit)
2:     propnet ← loadPropnet(game)
3:     model ← createModel(propnet)
4:     agents ← createAgents(propnet, model)
5:     endTime ← nowTime() + timeLimit
6:     replayBuffer ← ∅
7:     while nowTime() < endTime do
8:         state ← propnet.initialState()
9:         while propnet.gameNotOver(state) do
10:            (policies, values) ← depthLimitedCFR(propnet, model, state)
11:            replayBuffer.add(state, policies, values)
12:            actions ← ∅
13:            for agent in agents do
14:                action ← agent.getAction(propnet, state)
15:                actions.add(action)
16:            end for
17:            state ← propnet.nextState(state, actions)
18:            model.train(replayBuffer)
19:        end while
20:    end while
21:    model.save()
22: end procedure
```

The framework yields impressive results, often outperforming handcrafted algorithms within games such as Blind Tic-Tac-Toe and Liar's Dice [19]. However, several significant drawbacks exist as discussed below [19].

1. **Opponent modelling** may lead to insights into flaws in opponent behaviour and knowledge. Using these insights, the program could deviate from a what it deems an "optimal" policy to a "maximal" policy in order to exploit an opponents and achieve a greater reward.

2. **Checking game exploitability**, as the program performs poorly in games that are partially or non-exploitable. For example, in a non-exploitable game like the Monty Hall problem the program switches only $\frac{2}{3}$ of the time, whereas the ideal strategy is to always switch. The authors

suggest this could be remedied by utilising a methodology [15] to check if a game is exploitable whilst being adequately efficient for a GGP context.

3. **Optimisations** could allow larger games to be played which are currently infeasible. Furthermore, optimisations may allow for more complete or accurate searches given the same resources leading to an increase in performance. Suggested optimisations include determining the value of hyperparameters, parallelisation of key areas and the use of a better suited CFR variant. Additionally, the framework is written entirely in Python and thus would benefit greatly from being rewritten in a faster and more suitable language.

The focus of this research will be optimising the original implementation of this framework.

3 Methodology

Full implementation along with documentation and instructions for running and installation can be found here on GitHub.

3.1 Re-implementation in C++

For many games estimating an optimal policy is an extremely resource intensive task due to the oftentimes high depth and branching factor. As a result the choice of technologies used is critical in creating a competitive agent. Thus, as the original framework was written entirely in Python, it was decided to completely rewrite the system in a more suitable language. C++ was selected due to it's prevalence in both industry and current game playing research, lightweight abstractions, compiled nature and mature libraries which together contribute to a sizeable speedup.

3.1.1 Propositional Network Reformatting

As previously discussed, the original framework stores propositional networks as Python files (see 7.3 for an example) which are then dynamically loaded as a module. This approach is sufficient for programs written in Python, but becomes infeasible with propositional networks written in other languages as cannot be dynamically loaded. Thus, to facilitate the frameworks re-implementation in C++, a script was created to transform Python files into JSON format, the result of which can be seen in 7.4.

The script dynamically loads propositional network files in the same manner as the original framework, adds additional more readable metadata such as the type names to help with debugging and adds additional pre-computation which would either be extremely tiresome using C++, which includes:

1. Computing the topological ordering to inform the update order of network nodes. This is done so that when transitioning to a new state nodes only need to be updated once, beginning at nodes that have no inputs and ending at nodes that have no outputs. The result of this can be seen under the entry `topologically_sorted` in the appendix.

2. Parsing GDL to both determine the values of goals and to link related nodes (such as corresponding input and legal nodes), seen under `goals` and `legal_to_input` respectively. This is done by using regular expressions on GDL statements such as `(goal player1 75)` to extract required information.

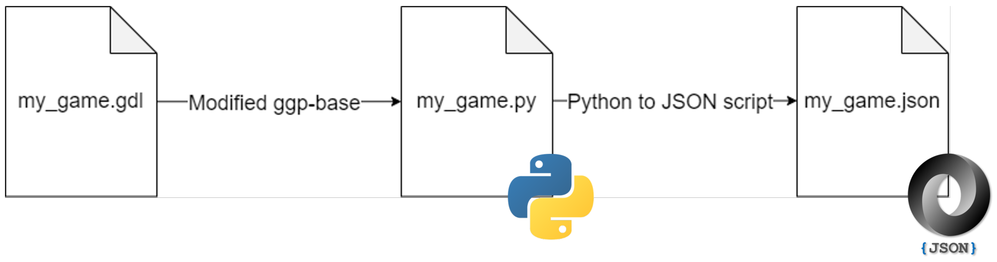


Figure 8: Reformatting of Python propositional network into JSON.

3.1.2 Scope

The framework has been largely re-implemented in C++, with no Python code remaining at train or play time. This includes:

1. Propositional network.
2. Sampling of grounded states.
3. Search.
4. The neural network, including the training loop.
5. The player itself.
6. Other simple agents.
7. A client to remotely play against other GGP programs implementing a propositional network.

There remain a several sections of the original framework which are yet to be translated into C++; including the policy head of the neural network originally used to speed up CFR search convergence by providing initial values and by some sampling methods which is currently untrained and unused.

3.2 MCCFR Search

The original framework uses Vanilla CFR without any game tree pruning optimisations commonly implemented [16]. In practice, as described in section 2.3.2, MCCFR converges significantly faster to equilibrium compared to Vanilla CFR in all games tested [16] due to time being allocated more efficiently by searching segments of the game tree that are more likely to occur and are thus more significant to the overall result. This, along with the considerably simpler implementation that external sampling benefits from by not having to calculate reach probabilities of terminal states, makes MCCFR a suitable choice of search algorithm. Another possible choice that remains unimplemented is CFR-D described in [6] which is utilised by the ReBeL framework mentioned in section 2.3.4.

Pseudo-code for a non-depth limited external sampling MCCFR is shown below, which has been adapted from [16] to integrate and contextualise the capabilities of a propositional network.

Algorithm 3 External sampling MCCFR implementation adapted for a propositional network

```
1: procedure externalSamplingMCCFR(propnet, state, infostates, traversingPlayer)
2:     if propnet.gameOver(state) then
3:         return propnet.getUtility(state, traversingPlayer)
4:     end if
5:     playerToMove ← propnet.playerToMove(state)
6:     if playerToMove == random then
7:         legalActions ← propnet.getLegalActions(random)
8:         action ← sample(legalActions)
9:         nextState ← propnet.nextState(state, {action})
10:        return externalSamplingMCCFR(propnet, nextState, infostates, traversingPlayer)
11:    end if
12:    infostate ← infostates.getInfoState(state)
13:    currentPolicy ← infostate.regretMatch()
14:    if playerToMove == traversingPlayer then
15:        legalActions ← propnet.getLegalActions(traversingPlayer)
16:        utility ← array indexed by each action in legalActions
17:        $\text{utility}_{\text{expected}} \leftarrow 0$
18:        for action in legalActions do
19:            nextState ← propnet.nextState(state, {action})
20:            utility[action] ← externalSamplingMCCFR(propnet, nextState, infostates, traversingPlayer)
21:            $\text{utility}_{\text{expected}} \leftarrow \text{utility}_{\text{expected}} + \text{currentPolicy[action]} \times \text{utility[action]}$
22:        end for
23:        for action in legalActions do
24:            $\text{regret} \leftarrow \text{utility[action]} - \text{utility}_{\text{expected}}$
25:            infostate.addToCumulativeRegret(regret)
26:        end for
27:        return $\text{utility}_{\text{expected}}$
28:    else
29:        action ← sample(currentPolicy)
30:        nextState ← propnet.nextState(state, {action})
31:        utility ← externalSamplingMCCFR(propnet, nextState, infostates, traversingPlayer)
32:        infostate.addToCumulativePolicy(currentPolicy)
33:        return utility
34:    end if
35: end procedure
```

This implementation of MCCFR cannot be called directly and requires a dedicated interface to control the traversing player, time limit and initialise key variables. Pseudo-code for this interface is presented below.

Algorithm 4 External Sampling MCCFR Interface

```
1: procedure SEARCH(propnet, state, timeLimit)
2:     infostates ← createBlankInfostates(propnet.allInfostates())
3:     endTime ← nowTime() + timeLimit
4:     while nowTime() < endTime do
5:         for traversingPlayer in propnet.allPlayers() do
6:             externalSamplingMCCFR(propnet, state, infostates, traversingPlayer)
7:         end for
8:     end while
9:     infostate ← infostates.getInfoState(state)
10:    return infostate.getAveragePolicies()
11: end procedure
```

As in the original framework, when performing a depth limited search the values of leaf node are are estimated using a neural network.

3.3 Hardware Utilisation

The original implementation was built to run on a laptop with no GPU; thus, the code base is written to be single-threaded and use CPU-only machine learning modules. When more resources are available this can lead to unnecessary bottlenecks. To improve hardware utilisation, the new implementation can optionally be run on a GPU and key areas of the program suggested by [19] have been parallelised.

3.3.1 Sampling and Searching

The sampler used to generate possible states has been made thread-safe by ensuring critical sections around shared data (i.e., invalid state caches) are appropriately protected from race conditions. This allows a single sampler instance is shared among multiple threads; allowing the caches to be populated more quickly. Additionally, as each search is independent of one another, states are searched in parallel with results added asynchronously to a buffer once terminated. This process repeats until either the buffer is fully populated or time expires.

Algorithm 5 Updated function to get action from agent

```
1: procedure getAction(propnet, state)
2:     observations ← propnet.getAgentObservations(state, role)
3:     cumulativePolicy ← $\vec{0}$
4:     sampleCount ← 0
5:     for parallel $i$ in $1 \ldots$ threadCount do
6:         while sampleCount < sampleSize do
7:             sampledState ← sampler.sampleState(propnet, model, history, observations)
8:             policy ← depthLimitedMCCFR(propnet, model, sampledState, role)
9:             cumulativePolicy ← cumulativePolicy + policy
10:            sampleCount ← sampleCount + 1
11:        end while
12:    end for
13:    action ← sample(cumulativePolicy)
14:    history.add(observations, action)
15:    return action
16: end procedure
```

Importantly, as previously mentioned, the sampling methodology used is not the sampling methodology described by the original authors in [19].

3.3.2 Training Loop

The training loop has been parallelised in two separate ways:

1. Since each agent determines their action independently of one another searches on the current state can be done in parallel as opposed to sequentially.

2. Whilst training multiple games can be played in parallel in a similar manner to described in [1].

Algorithm 6 Parallelised Training Loop

```
1: procedure TRAIN(game, concurrentGames, time)
2:     propnet ← loadPropnet(game)
3:     model ← createModel(propnet)
4:     agents ← createAgents(propnet, model)
5:     end ← now() + time
6:     replayBuffer ← $\emptyset$
7:     while now() < end do
8:         for parrallel $i$ in $1 \ldots$ concurrentGames do
9:             state ← propnet.initialState()
10:            while propnet.gameNotOver(state) do
11:                (policies, values) ← async MCCFR(propnet, model, state)
12:                actions ← $\emptyset$
13:                for parrallel agent in agents do
14:                    action ← agent.getAction(propnet, state)
15:                    actions.add(action)
16:                end for
17:                replayBuffer.add(state, policies, values)
18:                state ← propnet.nextState(actions)
19:            end while
20:        end for
21:        model.train(replayBuffer)
22:    end while
23:    model.save()
24: end procedure
```

Given the search improvements, the MCCFR search which has its results added to the replay buffer is no longer depth limited. Importantly, due to games being played in parallel, training cannot be done once each move is made (as was originally done) without synchronising moves across threads. This is a viable approach, and may yield better results, but instead it was decided to simply counteract this effect by manually increasing epoch size on a per-game basis from the original 5.

4 Results

4.1 Testing Environments

Search benchmarks

Performed on a machine running WSL2 (Ubuntu) on Windows 10 with an AMD Ryzen 5 5600X (6 cores, 12 threads) processor and 64 GB of RAM. Programs were run as root and given the highest niceness score (-20) to minimise the chance of preemption and, in cases where benchmarks took longer than 10 hours, run two at a time. Timing data taken is the real time (not program time) and does not include collecting or writing to disk.

Strength benchmarks

As the machine used for search benchmarks doesn't include CUDA-enabled GPU, agents were trained and benchmarked for their strength on a rented machine running Ubuntu with an AMD Ryzen 9 7950X3D (16 cores, 32 threads) processor, 128 GB of RAM and an NVIDIA RTX 4090 GPU. Models were trained for a maximum of 16 hours using 10 concurrent games with both agents and the full search being given 30 seconds for each move. Furthermore, agents were given a single thread to sample and search using with a time limit of 3 seconds per sample. Multiple benchmarks were conducted at a time with agents given 10 threads; each of which was limited to 20 seconds to sample and search one possible state.

4.2 Payoff-Modified Rock, Paper, Scissors

Rock, paper, scissors is a very simple two player zero-sum game where players simultaneously choose between rock, paper and scissors. This variant of Rock, Paper, Scissors has payoffs modified to bias the MSNE away from the uniform strategy found in regular rock, paper, scissors which has a MSNE of $(r, p, s) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

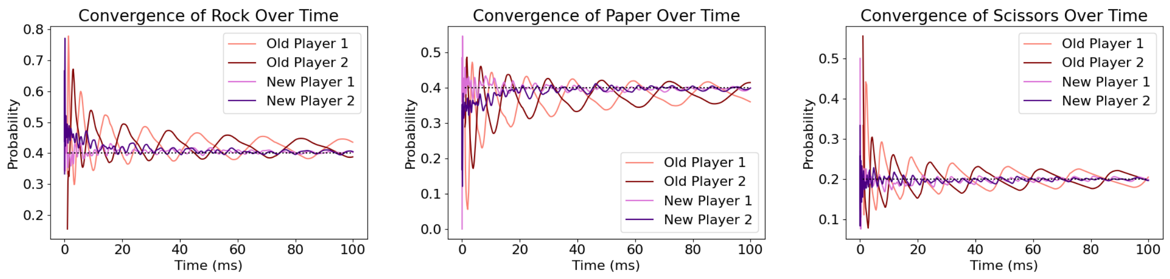
Importantly, Payoff-Modified Rock, Paper, Scissors has slightly differently modified payoffs compared to the previously mentioned RPS+, however the MSNE of $(r, p, s) = (0.4, 0.4, 0.2)$ is identical. Furthermore, playing this policy yields an EV of 50 against any policy an opponent may play. See the payoff matrix below for details on how payoffs have been modified.

		Player 2		
		R	P	S
	R	$(50, 50)$	$(25, 75)$	$(100, 0)$
Player 1	P	$(75, 25)$	$(50, 50)$	$(0, 100)$
	S	$(0, 100)$	$(100, 0)$	$(50, 50)$

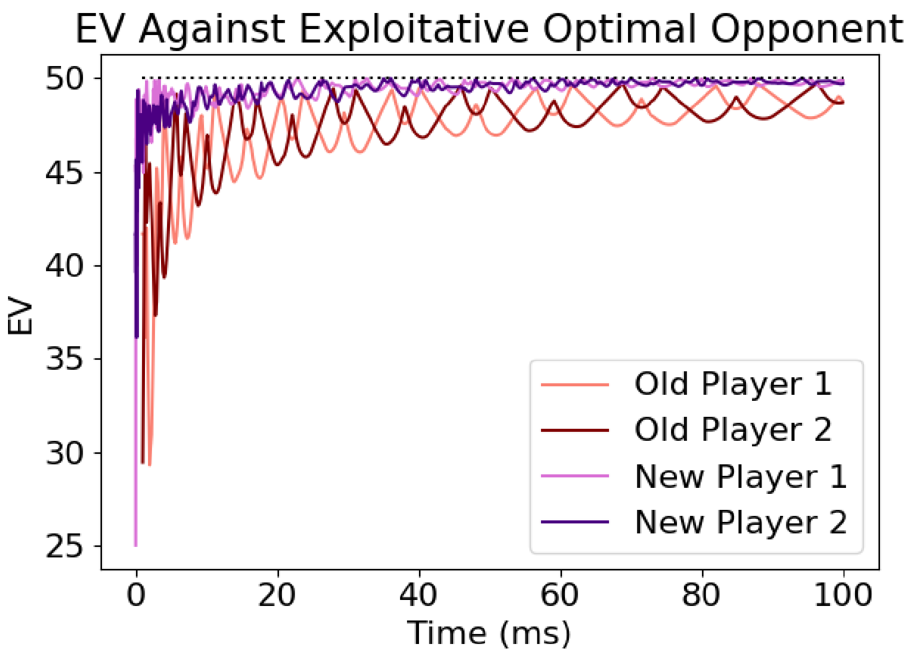
Table 1: Matrix of payoffs for rock (R), paper (S) and scissors (S) in Payoff-Modified Rock, Paper, Scissors.

4.2.1 Search Benchmarks

Using search only, the updated implementation converges significantly faster than the previous implementation given the same resources. This is true for all moves in Payoff-Modified Rock, Paper, Scissors as well as overall EV versus an exploitative opponent. See below for plots of the speed of convergence for both the original and updated search implementation, where known optimal values are shown as a black dotted line.



(a) Time-series plots showing convergence of moves to equilibrium from previous and updated search implementations.



(b) Time-series plots showing EV from previous and updated implementations.

Figure 9: Comparison of convergence speed between previous and updated search implementations in Payoff-Modified Scissor, Paper, Rock.

4.2.2 Strength Benchmarks

Unfortunately, benchmarking the strength of agents in Payoff-Modified Scissor, Paper, Rock is of little value due to the EV of optimal policy being a constant 50 no matter the opposing policy. This can be illustrated by plotting the EV of the original search over time versus a static optimal agent (as is attempted in this research).

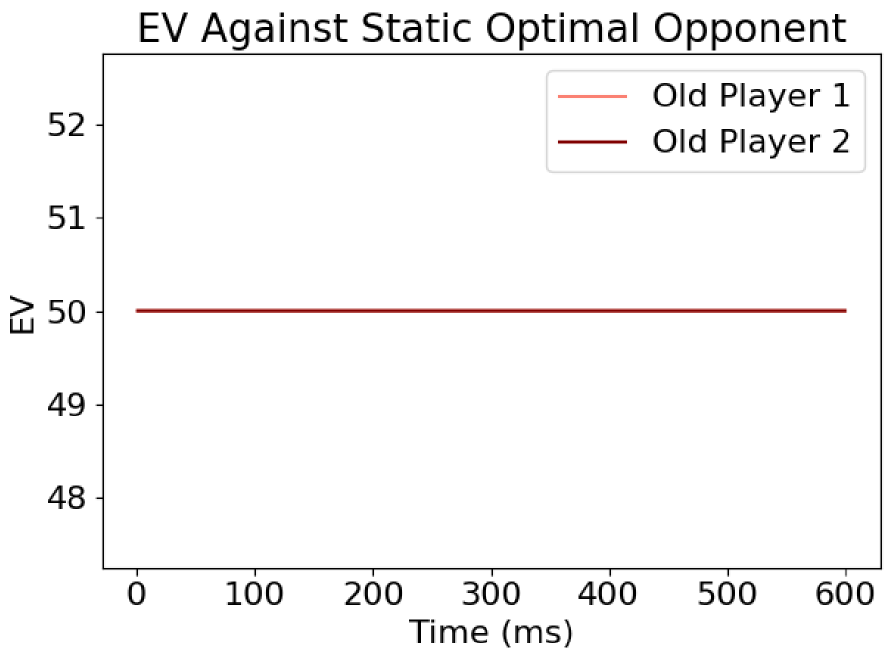


Figure 10: Original implementations EV over time against a static optimal opponent in Payoff-Modified Scissor, Paper, Rock.

4.3 Blind Tic-Tac-Toe

Blind Tic-Tac-Toe is a toy variant of Tic-Tac-Toe where players take actions simultaneously. Furthermore, the only observations players receive are if their previous moves were successful. After players make their moves, one of a following cases happen:

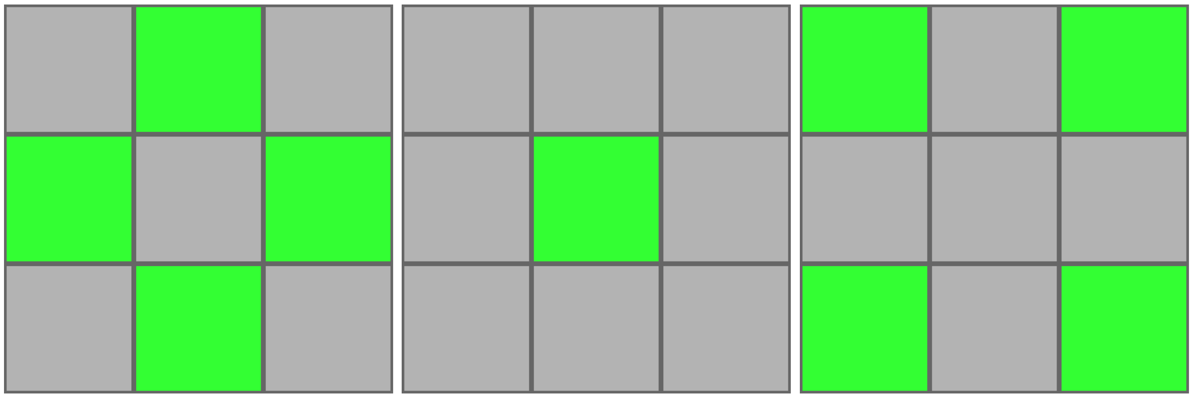
Player selects an unmarked tile
Player's mark is successfully placed, which they're informed of.

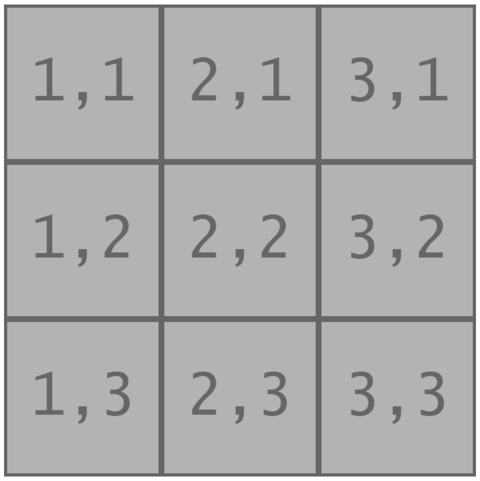
Player selects marked tile
Player's mark is not placed, which they're informed of.

Both players simultaneously select an unmarked tile
A tiebreak decides which players mark shall be successful. The odds of X and O being successful in the tiebreak differ between variants of Blind Tic-Tac-Toe which can lead to asymmetric policies.

The game is won in an identical way to the original Tic-Tac-Toe when three of a single player's markings form a line. Blind Tic-Tac-Toe, along with other blind or phantom style games, remain an unsolved problem in game playing as they lead to exceptionally large infostates. They are particularly difficult for our framework to accurately sample due to the lack of information to narrow down possible states. Furthermore, frameworks often actively assume all agents actions are public information (e.g., ReBeL is only applied to Poker and Liar's Dice [3]) to avoid this complexity.

Although we were unable to find literature on the variant of Blind Tic-Tac-Toe described here, a similar variant where agents instead take turns to make their moves and are able to re-select actions if they're unsuccessful has approximately 10^{10} distinct histories and 5.6×10^{6} infostates [16]. As Tic-Tac-Toe is symmetric it could be abstracted to reduce the number of histories and infostates, however this is extremely difficult to achieve in GGP.


(a) Different groupings of symmetric Tic-Tac-Toe moves referred to as "middle-side", "middle-middle" and "corner" respectively.



(b) Tic-Tac-Toe square labels.

Figure 11: Terminology used to describe both different classes of squares and specific squares in following results.

The previous implementation is unable to meaningfully search Blind Tic-Tac-Toe due to these factors. Fortunately, improvements made to the updated search implementation mean it's able to fully search even from the initial state of Blind Tic-Tac-Toe games, albeit slowly (1-2 seconds per iteration, with multiple thousand iterations required for convergence) and requiring a large amount of memory (in excess of 12 GB per search). Each search was run for 100000 iterations over the course of approximately 40 hours.

4.3.1 Regular

In Regular Blind Tic-Tac-Toe tiebreaks favour both players equally i.e., there's $\frac{1}{2}$ chance for either the x or o player of being successful. We therefore expect symmetry between players for both search and strength benchmarks.

4.3.1.1 Search Benchmarks

Symmetry between similar squares regardless of the role is observed with middle-middle move and middle-side moves converging rather quickly to be in the range $[0.316, 0.320]$ and $[0.012, 0.022]$ respectively. Corner moves took significantly longer to converge with probabilities at termination in the range $[0.148, 0.165]$.

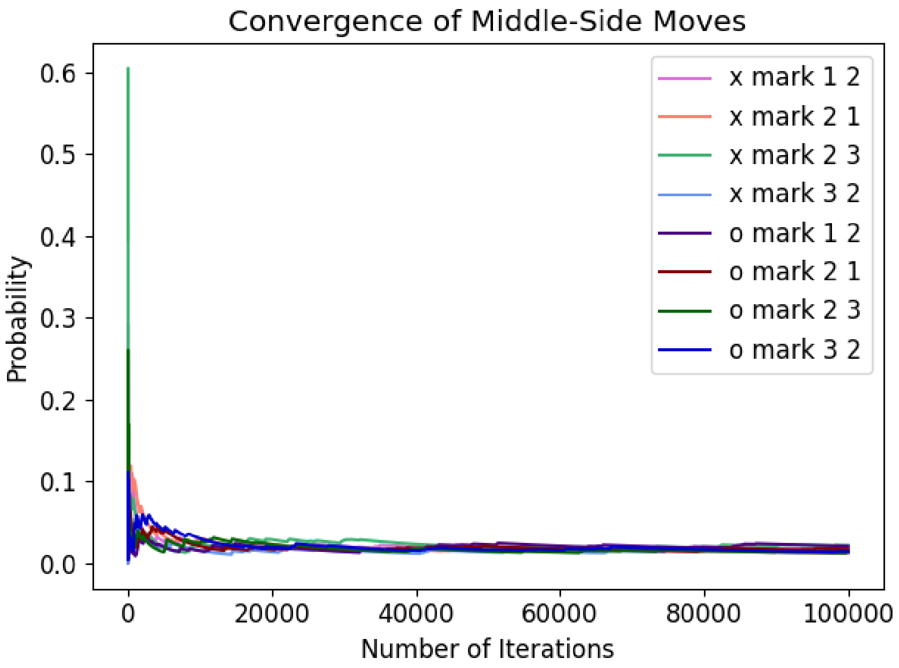


Figure 12: Plots showing convergence of playing middle-side moves from updated search implementation in Regular Blind Tic-Tac-Toe.

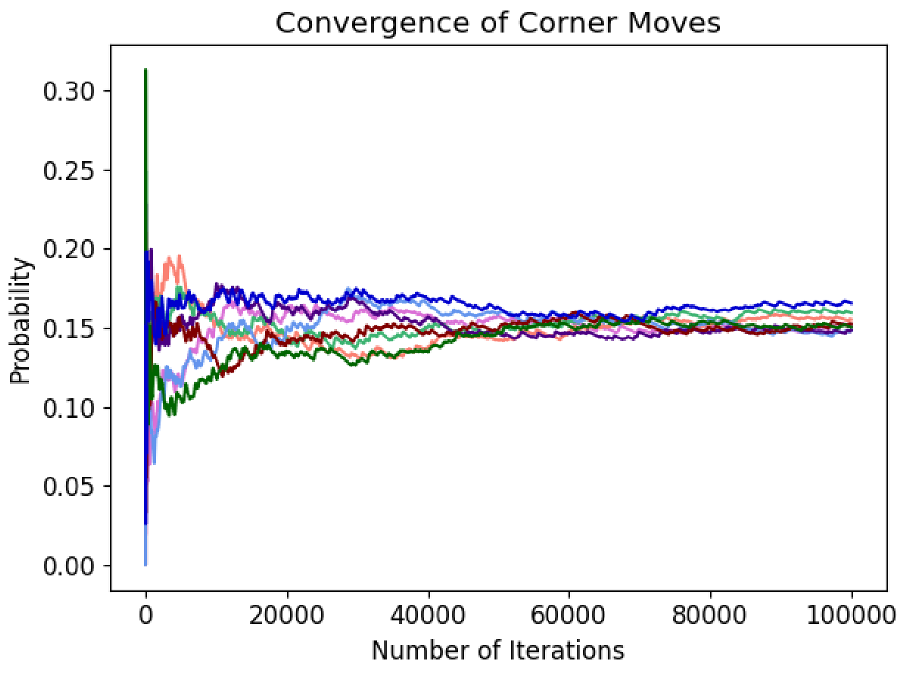


Figure 13: Plots showing convergence of playing corner moves from updated search implementation in Regular Blind Tic-Tac-Toe.

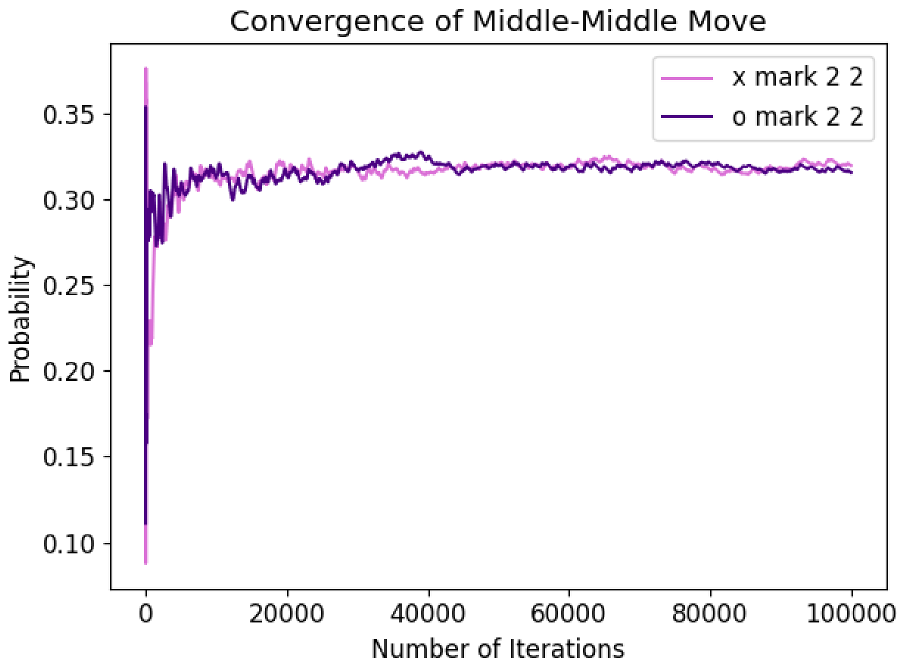
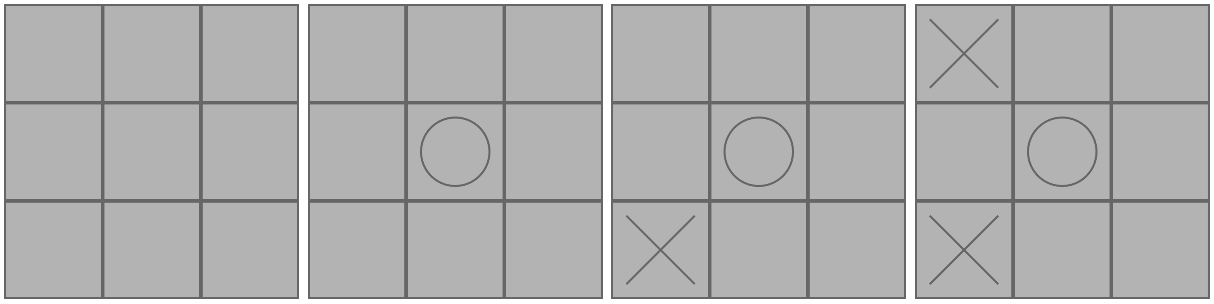


Figure 14: Plots showing convergence of playing middle-middle move from updated search implementation in Regular Blind Tic-Tac-Toe.

4.3.1.2 Model Evaluations

Given the slightly modified training methodology, wherein searches used for training are not depth limited, the new neural network learns slightly different values for given states. See example states and their corresponding values below.



State	New Value for x	Old Value for x
1	0.48	0.50
2	0.20	0.27
3	0.39	0.27
4	0.65	0.50

Table 2: State values estimated by neural network for x player to the nearest 2 decimal places. A value of 0 indicates a lost state and 1 a won state.

Interestingly the old network correctly identifies the initial state (state 1) as being worth 0.5 for both players. Whilst the new network correctly identifies the initial state is equal for both players it incorrectly predicts the value at 0.48. Results of other states are similar, however the new methodology is able to identify that the value of state 3 is strictly greater than that of state 2 whereas previously they were estimated to be of identical value.

4.3.1.3 Strength Benchmarks

Strength results for Regular Blind Tic-Tac-Toe are unremarkable, with the old framework slightly outperforming. This is likely due to the unfinished sampling methodology used and is reflected in later benchmarks.

x Player	o Player				
	Random	New	MCCFR$_{(\text{new})}$	Old	CFR$_{(\text{old})}$
Random	(50.5, 49.5)	(27.0, 73.0)	(36.0, 64.0)	(20.5, 79.5)	(40.0, 60.0)
New	(74.0, 26.0)	(49.0, 51.0)	(58.0, 42.0)	-	-
MCCFR$_{(\text{new})}$	(66.5, 33.5)	(47.5, 52.5)	(55.5, 44.5)	-	-
Old	(79.5, 20.5)	-	-	-	-
CFR$_{(\text{old})}$	(54.0, 46.0)	-	-	-	-

Table 3: Cumulative reward for various players over 100 games of Regular Blind Tic-Tac-Toe played.

4.3.2 Biased

Biased differs from Regular Blind Tic-Tac-Toe with an asymmetric tiebreaks which favours x $\frac{2}{3}$ of the time and o the remaining $\frac{1}{3}$. This skews the optimal strategy for the x player towards more favourable squares such as the middle-middle square (which is involved in more lines than any other square), with the converse being true for the o player. Additionally, we expect strength benchmarks to be slightly in the x players favour given this advantage.

4.3.2.1 Search Benchmarks

Search converges similarly quickly to Regular Blind Tic-Tac-Toe and the effect of the bias is clearly apparent. For example, in Regular middle-middle moves were played by both player with a probability of ≈ 0.318 but are now played at frequencies 0.329 for x and 0.203 for o respectively.

Similar trends are seen in probabilities of playing each middle-side move, which weren't favoured by either player in the Regular variant at ≈ 0.017. In Biased however, as x favours these squares even less at ≈ 0.005 each and o is more inclined with each being in the range $[0.074, 0.091]$

As corner moves are somewhere in between, they don't experience as polarising of an effect.

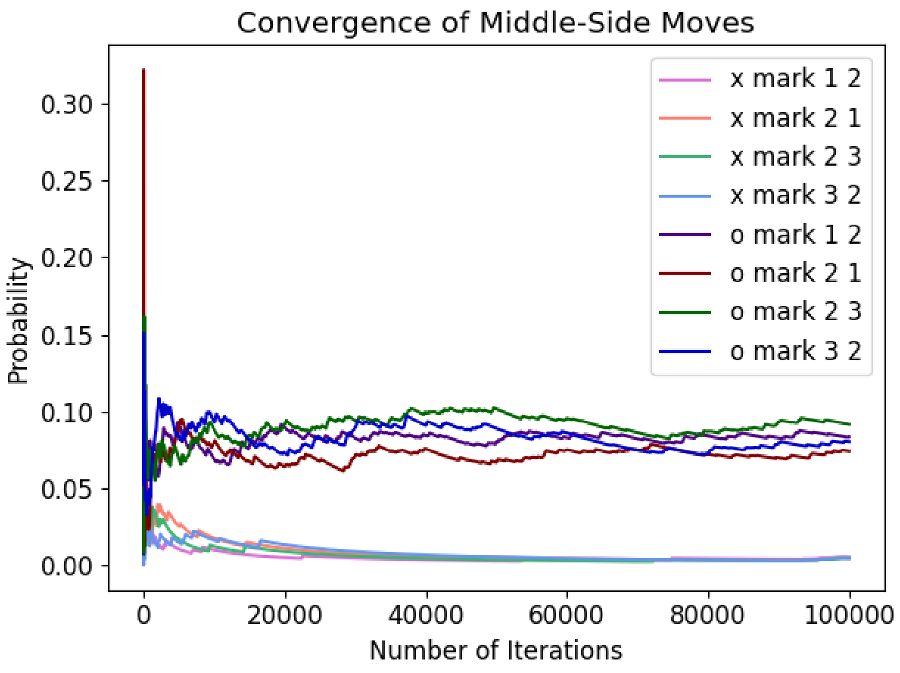
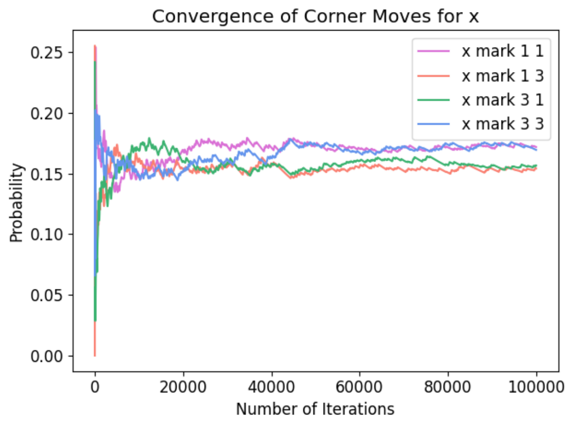


Figure 15: Plots showing convergence of playing middle-side moves from updated search implementation in Biased Blind Tic-Tac-Toe.



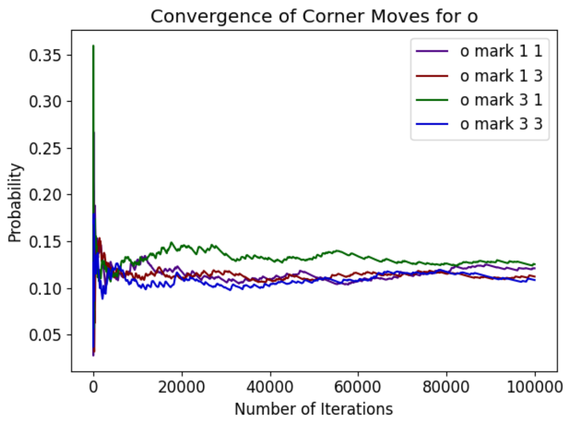


Figure 16: Plots showing convergence of playing corner moves from updated search implementation in Biased Blind Tic-Tac-Toe.

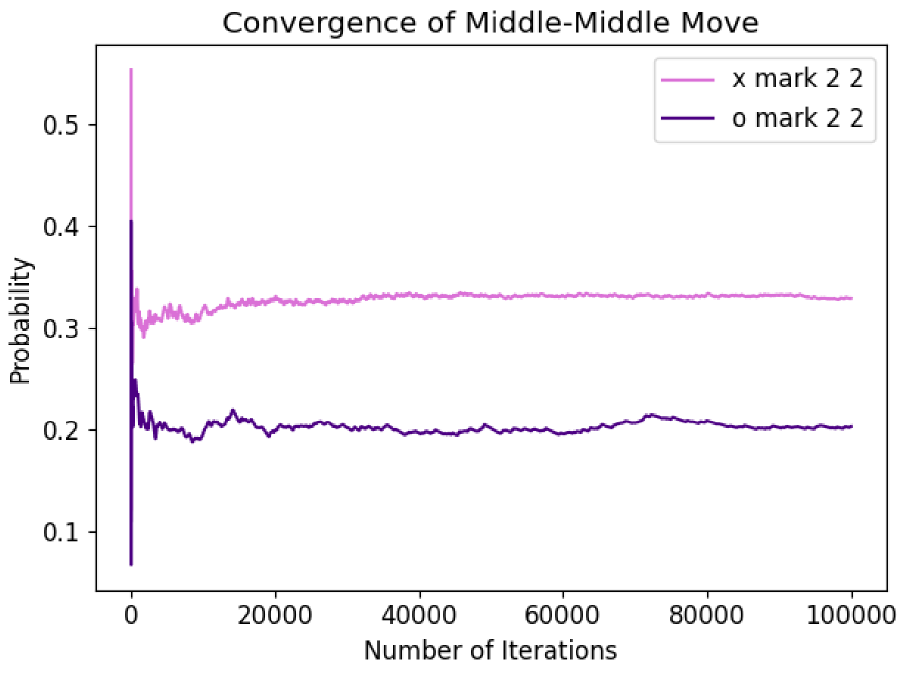


Figure 17: Plots showing convergence of playing middle-middle moves from updated search implementation in Biased Blind Tic-Tac-Toe.

4.3.2.2 Strength Benchmarks

The new framework, and even the search only variant, outperforms the old when playing as x. However, as o the old framework performs best against a random agent. Strangely the old framework as player x performs slightly worse against a random player than it did in the non-biased variant.

x Player	o Player				
	Random	New	MCCFR$_{\text{(new)}}$	Old	CFR$_{\text{(old)}}$
Random	(54.5, 45.5)	(38.5, 61.5)	(41.5, 58.5)	(26.5, 73.5)	(43.0, 57.0)
New	(84.0, 16.0)	(56.0, 44.0)	(62.0, 38.0)	-	-
MCCFR$_{\text{(new)}}$	(73.5, 26.5)	(53.5, 46.5)	(56.0, 44.0)	-	-
Old	(76.5, 23.5)	-	-	-	-
CFR$_{\text{(old)}}$	(60.5, 39.5)	-	-	-	-

Table 4: Cumulative reward for various players over 100 games of Biased Blind Tic-Tac-Toe played.

4.3.3 Very Biased

Very Biased Blind Tic-Tac-Toe sees a further biasing of tiebreaks with player x always winning. We expect policies to reflect this with a more extreme deviation away from the symmetrical policies found in Regular Blind Tic-Tac-Toe.

4.3.3.1 Search Benchmarks

As anticipated, the middle-middle move is favoured even further by x and less by o, settling at 0.394 and 0.137 respectively. Similarly, x plays middle-side moves even less in the range $[0.002, 0.003]$ and with o favouring it in the range $[0.071, 0.080]$. Corner moves continue to remain rather unchanged, although in the search for x corners (1, 1) and (3, 3) diverged from (1, 3) and (3, 1). This result was not observed in any other search benchmark run for fewer iterations and is therefore assumed to be an outlier.

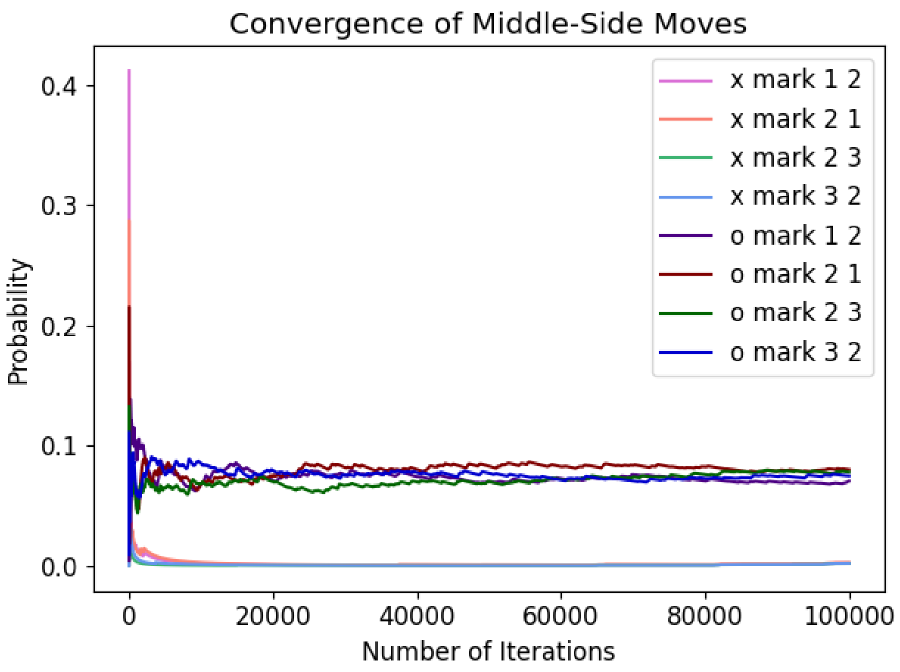


Figure 18: Plots showing convergence of playing middle-side moves from updated search implementation in Very Biased Blind Tic-Tac-Toe.

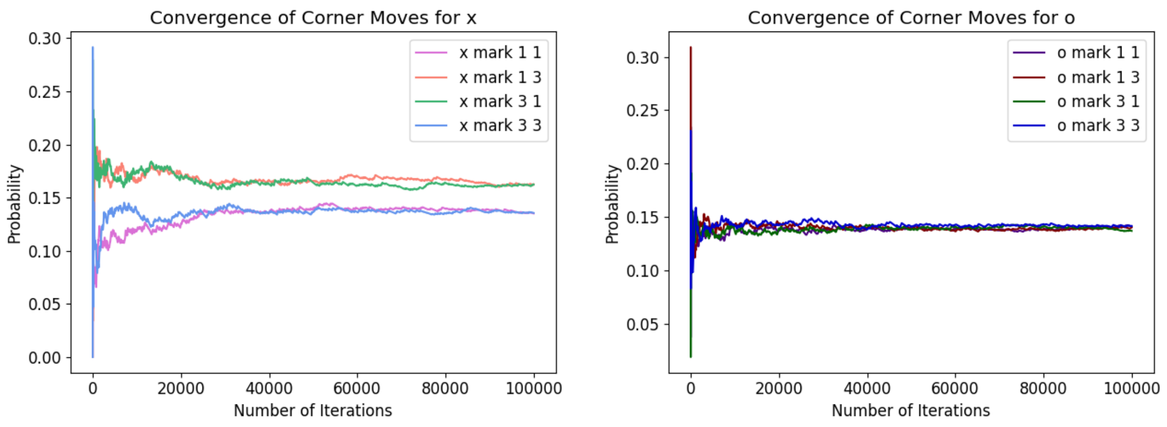


Figure 19: Plots showing convergence of playing corner moves from updated search implementation in Very Biased Blind Tic-Tac-Toe.

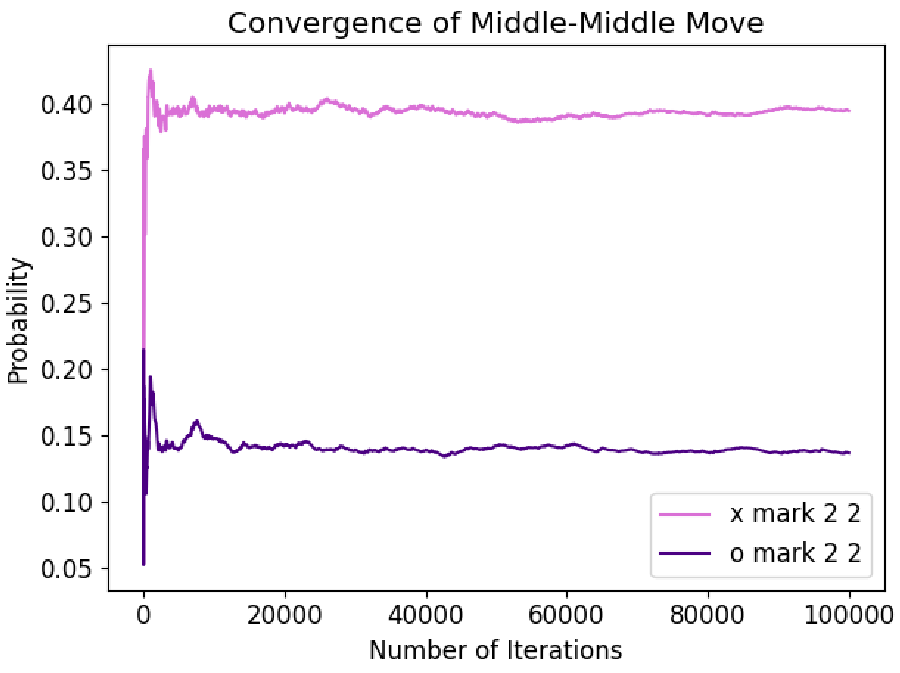


Figure 20: Plots showing convergence of playing middle-middle move from updated search implementation in Very Biased Blind Tic-Tac-Toe.

4.3.3.2 Strength Benchmarks

The new and old frameworks perform comparably. However, using search only the new variant significantly outperforms the old when playing as *o*.

x Player	o Player				
	Random	New	MCCFR$_{(\text{new})}$	Old	CFR$_{(\text{old})}$
Random	(64.0, 36.0)	(44.0, 56.0)	(48.5, 51.5)	(48.0, 52.0)	(64.0, 36.0)
New	(87.5, 13.5)	(75.0, 25.0)	(79.5, 20.5)	-	-
MCCFR$_{(\text{new})}$	(81.5, 18.5)	(69.0, 31.0)	(74.5, 25.5)	-	-
Old	(91.5, 8.5)	-	-	-	-
CFR$_{(\text{old})}$	(84.0, 16.0)	-	-	-	-

Table 5: Cumulative reward for various players over 100 games of Very Biased Blind Tic-Tac-Toe played.

4.4 Liar's Dice

Liar's Dice (also known as Meier, Bluff, Dudo and Perudo [16]) is a game of bluff. Our variation has two players who each begin by rolling their own 6-sided dice, keeping the result a secret from their opponent [19]. Players next then bid on the outcome of their dice rolls present. Bids continue to be increases in turn until one of the players believes their opponent is bluffing, at which point the game ends. The dice are revealed, with the game being won by the calling player if their opponent was indeed bluffing.

4.4.1 Strength Benchmarks

It was found that, with the contributions in this work, that the variant of Liar's Dice described here could be fully searched at play time. Using a search only approach the new implementation significantly outperforms both the trained player and search only implementation from the old framework.

Player 1	Player 2			
	Random	MCCFR$_{(\text{new})}$	Old	CFR$_{(\text{old})}$
Random	(35.0, 65.0)	(4.0, 96.0)	(20.0, 80.0)	(20.0, 80.0)
MCCFR$_{(\text{new})}$	(98.0, 2.0)	(96.0, 4.0)	-	-
Old	(97.0, 3.0)	-	-	-
CFR$_{(\text{old})}$	(83.0, 17.0)	-	-	-

Table 6: Cumulative reward for various players over 100 games of Liar's Dice played.

5 Conclusion

5.1 Summary

A previous framework that exhibits strong performance in imperfect-information GGP was iterated upon and improved by:

1. Re-implementing entirety of original framework in C++
2. Changing CFR variant from Vanilla CFR to MCCFR with external sampling
3. Parallelising key areas such as the state sampler and training loop
4. Optionally allowing GPU support for neural networks.
5. Adding documentation and ensuring portability for ease of use.

5.2 Discussion

Improvements are significant and clear, particularly in search performance which outperforms the original implementation by a large margin in all metrics and benchmarks; converging orders of magnitude faster in some games and allowing games to be searched when it was previously impractical to do so. This along with other advancement has allowed for neural networks to be trained more precisely in a shorter time by parallelising key areas.

Whilst the new implementation distinctly outperforms the previous implementation in the more traditional of Liar's Dice, results against other agents in variants of Blind Tic-Tac-Toe are lacklustre. These results can be explained through important areas of the framework being only partially implemented due to a lack of time, however the root cause is currently unknown for certain and requires investigation.

5.3 Future Work

Despite the work completed there remain many significant potential improvements to the framework. See below for a list of future work to be conducted in the field of GGP with imperfect-information, many of which weren't or were only partially addressed from the previous framework [19].

5.3.1 Completion of Re-implementation

As previously mentioned several components originally described in [19] have only been translated into C++ such as the policy head of the neural network and more sophisticated sampling methodologies. This is simply due to a lack of time and it's probable that implementing these would allow more interesting states to be sampled; thus improving and likely comprehensively beating the original framework in strength benchmarks.

5.3.2 Adaptation of ReBeL Framework into GGP

The ReBeL framework discussed in section 2.3.4 assumes actions are public knowledge to maintain a public belief state. Whilst this assumption hasn't been made in this research, by doing so a faithful translation of ReBeL into GGP could be completed. This is particularly appealing as ReBeL has proven superhuman performance in multiple games and requires no expert knowledge to train making it ideal for GGP.

5.3.3 Methodology Improvements

5.3.3.1 Soundness

The current framework isn't theoretically sound as policies within searches are biased towards the agents observations. For example imagine a game of Poker where the agent is dealt aces. The agent begins sampling possible states and searches them as described in section 2.4.2. As the agent will naturally be holding aces in all samples found, during the search all opponents will implicitly (from the utilities observed during each iteration) learn that the agent has aces; therefore biasing their policies to be more passive than they ought to given their observations. CFR-D, which is able to solve subgames independently in a theoretically sound manner [6], could potentially illicit greater performance than MCCFR by eliminating the above issue.

5.3.3.2 Parameters

As in the previous implementation, many parameters haven't been rigorously tested to find optimal values [19]. A non-exhaustive list, many of which [19] also mentions, includes training options (number of concurrent games, time given to full CFR search and the batch size etc), player options (search depth limit, number of threads, sample size, CFR search iteration or time limit etc) and neural network paramters (LRU cache size, number of layers, layer types, individual layer parameters, sizes of hidden layers etc). It's likely that adjustments to these parameters would yield greater results using less computational power.

5.3.3.3 Speed

Despite improvements discussed throughout this work many games, including blind or phantom games more complicated than Tic-Tac-Toe such as Connect Four as well as more traditional games such as Backgammon and Stratego, remain infeasible. Future work to improve the time complexity of the methodology presented could allow these currently intractable games to be played although it's unclear how this would be accomplished.

5.3.4 Exploitability Checking

Previous authors note that the framework is not able to recognise games that are non-exploitable and thus performs suboptimally by unnecessarily choosing a mixed strategy [19]. For example, consider Monty Hall problem; a non-exploitable game given there's only a single player. On the second move when deciding to switch or stay the are two possibilities:

The player has already chosen the correct door

Occurs with $\frac{1}{3}$ probability with the player deducing that they should stay.

The player hasn't chosen the correct door

Occurs with $\frac{2}{3}$ probability with the player deducing that they should switch.

Using the existing framework, the agent will sample, search and weight these states according to the probability they occur; therefore determining that it should switch $\frac{1}{3} \times 0 + \frac{2}{3} \times 1 = \frac{2}{3}$ of the time. This logic is akin to that used by many humans, however does not produce the optimal policy to always switching [19]. Critically, in these non-exploitable games the agent can choose to play a pure strategy by simply choosing the action with the highest EV (which in this case is switching).

One overly simplistic that identifies a small subset of non-exploitable games is to check if the number of players present. This is insufficient to catch more complex non-exploitable games however such as a multiplayer Monty Hall problem. Additionally, this wouldn't even consider games that are both exploitable and non-exploitable depending on what stage of the game; for example a single two player game where the result Rock, Paper, Scissors determines who progresses into the Monty Hall problem [19].

The original authors identify a more sophisticated methodology that can test the exploitability of each state encountered whilst avoiding a full game tree traversal [15] which may be feasible within GGP.

5.3.5 Opponent Modelling

Opponent modelling seeks to forecast an opponent's behaviour or hidden information (which can be both within or outside the game e.g., hidden cards, search technique used), with the intent to gain a greater reward for the player [18]. Players use past observations of their opponent over a period of time to create a model that can predict future behaviour. Players may then adapt their strategy to integrate their opponent's biases; balancing exploitation and safety as they seek to further understand the value of said information [18].

To illustrate the intended behaviour a concrete example against a suboptimal opponent in the game of RPS+ is given below.

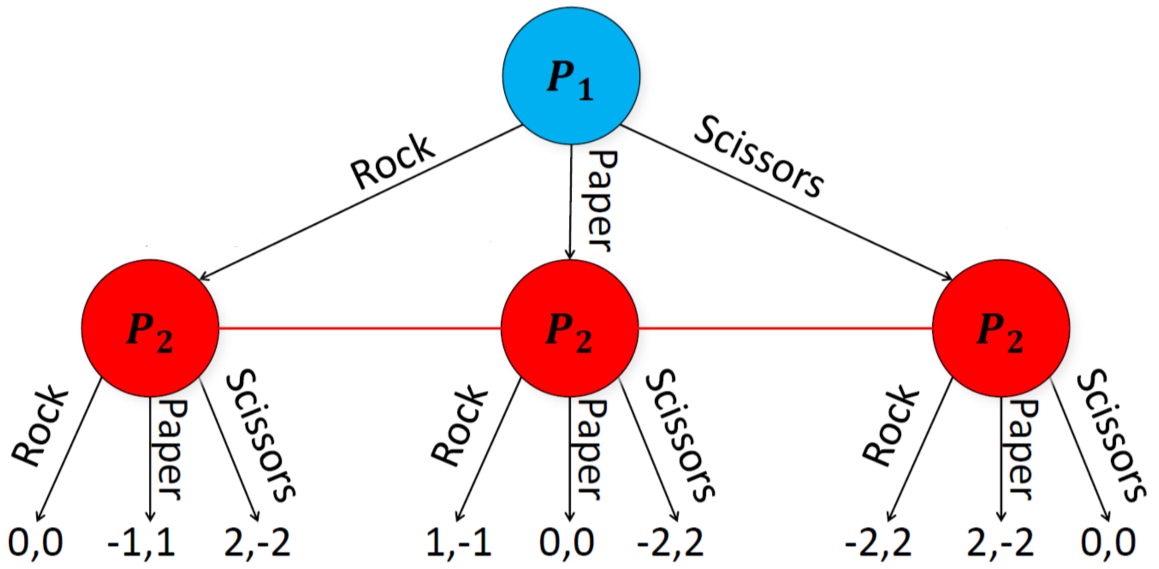


Figure 21: Game tree of RPS+ showing the payoffs [4].

Let player i play policy $p_i = (r_i, p_i, s_i)$ where r_i, p_i and s_i represent the probability of playing rock, paper and scissors respectively. Furthermore, let the program be player 1 and opponent be player 2. Recall that the MSNE policy for RPS+ is $p_1 = p_2 = (0.4, 0.4, 0.2)$ which has utility $u_1(p_1, p_2) = 0$.

Suppose the opponent instead plays $p_2' = (0.6, 0.2, 0.2)$. Assuming the agent doesn't deviate from the optimal policy, $u_1(p_1, p_2') = 0.08$. However, if the agent recognised that the opponent's policy p_2' was suboptimal and adapted to instead play $p_1' = (0, 0, 1)$ they could achieve a utility $u_1(p_1', p_2') = 0.8 = 10u_1(p_1, p_2')$. Thus, the program has achieved a significantly higher reward ($10\times$) by modelling and exploiting its opponent. Note that playing p_1' opens the program up for exploitation itself; a common drawback of adaptation and exploitation.

6 References

[1] C Alfredo and C Arjun. Efficient parallel methods for deep reinforcement learning. In *The Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*, pages 1–6, 2017.

[2] S. Behnel, R. Bradshaw, D. Woods, M. Valo, and L. Dalcín. Cython c-extensions for python. `https://cython.org/`, 2007.

[3] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17057–17069. Curran Associates, Inc., 2020.

[4] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[5] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[6] Neil Burch, Michael Johanson, and Michael Bowling. Solving imperfect information games using decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[7] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[8] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 4(1):216–217, Sep. 2021.

[9] Armin Chitizadeh and Michael Thielscher. Iterative tree search in general game playing with incomplete information. In Tristan Cazenave, Abdallah Saffidine, and Nathan Sturtevant, editors, *Computer Games*, pages 98–115, Cham, 2019. Springer International Publishing.

[10] Evan Cox, Eric Schkufza, Ryan Madsen, and Michael Genesereth. Factoring general games using propositional automata. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 13–20, 2009.

[11] Stefan Edelkamp, Tim Federholzner, and Peter Kissmann. Searching with partial belief states in general games with incomplete information. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, pages 25–36, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62, Jun. 2005.

[13] Michael Genesereth and Michael Thielscher. *General game playing*. Springer Nature, 2014.

[14] Adrian Goldwaser and Michael Thielscher. Deep reinforcement learning for general game playing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1701–1708, Apr. 2020.

[15] Michael Johanson, Kevin Waugh, Michael Bowling, and Martin Zinkevich. Accelerating best response calculation in large extensive games. In *IJCAI*, volume 11, pages 258–265, 2011.

[16] Marc Lanctot. *Monte Carlo sampling and regret minimization for equilibrium computation and decision-making in large extensive form games*. University of Alberta (Canada), 2013.

[17] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[18] Samer Nashed and Shlomo Zilberstein. A survey of opponent modeling in adversarial domains. *Journal of Artificial Intelligence Research*, 73:277–327, 2022.

[19] Zachary Partridge and Michael Thielscher. Hidden information general game playing with deep learning and search. Nov. 2021.

[20] Aaron Roth. Algorithmic game theory. `https://www.cis.upenn.edu/~aaroth/courses/agtS23.html`, 2023. Accessed: 2023-07-26.

[21] Michael Schofield and Michael Thielscher. General game playing with imperfect information. *Journal of Artificial Intelligence Research*, 66:901–935, 2019.

[22] S. Schreiber, E. Dreyfuss, E. Schkufza, K. Schwarz, S. Bills, and M. Mintz. General game playing base package. `https://github.com/ggp-org/ggp-base`, 2016.

[23] C. E. Shannon. Programming a computer for playing chess. *first presented at the National IRE Convention, March 9, 1949, and also in Claude Elwood Shannon Collected Papers*, pages 637–656, 1993.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[25] Michael Thielscher. A general game description language for incomplete information games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1):994–999, Jul. 2010.

[26] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107. Citeseer, 2011.

7 Appendix

7.1 GDL Example

An example for how Tic-Tac-Toe could be defined is shown below [13].

```
role(white)
role(black)

base(cell(row, col, x)) :-
    index(row) &
    index(col)
base(cell(row, col, o)) :-
    index(row) &
    index(col)
base(cell(row, col, blank)) :-
    index(row) &
    index(col)

base(control(white))
base(control(black))

input(player, mark(row, col)) :-
    role(player) &
    index(row) &
    index(col)
input(player, noop) :-
    role(player)

index(1)
index(2)
index(3)

init(cell(1, 1, blank))
init(cell(1, 2, blank))
init(cell(1, 3, blank))
init(cell(2, 1, blank))
init(cell(2, 2, blank))
init(cell(2, 3, blank))
init(cell(3, 1, blank))
init(cell(3, 2, blank))
init(cell(3, 3, blank))
init(control(white))
```

```
legal(player, mark(row, col)) :-
    true(cell(row, col, blank)) &
    true(control(player))
legal(white, noop) :-
    true(control(black))
legal(black, noop) :-
    true(control(white))

next(cell(row, col, x)) :-
    does(white, mark(row, col)) &
    true(cell(row, col, blank))
next(cell(row, col, o)) :-
    does(black, mark(row, col)) &
    true(cell(row, col, blank))
next(cell(row, col, symbol)) :-
    true(cell(row, col, symbol)) &
    distinct(symbol, blank)
next(cell(row1, col1, blank)) :-
    does(player, mark(row2, col2)) &
    true(cell(row1, col1, blank)) &
    distinct(row1, row2)
next(cell(row1, col1, blank)) :-
    does(player, mark(row2, col2)) &
    true(cell(row1, col1, blank)) &
    distinct(col1, col2)
next(control(white)) :-
    true(control(black))
next(control(black)) :-
    true(control(white))

goal(white, 100) :-
    line(x) &
    ~line(o)
goal(white, 50) :-
    ~line(x) &
    ~line(o)
goal(white, 0) :-
    ~line(x) &
    line(o)
```

```
goal(black, 100) :-
    ~line(x) &
    line(o)
goal(black, 50) :-
    ~line(x) &
    ~line(o)
goal(black, 0) :-
    line(x) &
    ~line(o)

line(num) :-
    row(row, num)
line(num) :-
    column(row, num)
line(num) :-
    diagonal(num)

row(row, num) :-
    true(cell(row, 1, num)) &
    true(cell(row, 2, num)) &
    true(cell(row, 3, num))
column(col, num) :-
    true(cell(1, col, num)) &
    true(cell(2, col, num)) &
    true(cell(3, col, num))
diagonal(num) :-
    true(cell(1, 1, num)) &
    true(cell(2, 2, num)) &
    true(cell(3, 3, num))
diagonal(num) :-
    true(cell(1, 3, num)) &
    true(cell(2, 2, num)) &
    true(cell(3, 1, num))

terminal :-
    line(num)
terminal :-
    ~open
open :-
    true(cell(row, col, blank))
```

7.2 GDL-II Example

Full definition of the Monty Hall problem [26].

```
role(candidate)
role(random)

init(closed(1))
init(closed(2))
init(closed(3))
init(step(1))

legal(random, hideCar(door)) :-
      true(step(1)) &
      true(closed(door))
legal(random, openDoor(door)) :-
      true(step(2)) &
      true(closed(door)) &
      ~true(car(door)) &
      ~true(chosen(door))
legal(random, noop) :-
      true(step(3))
legal(candidate, choose(door)) :-
      true(step(1)) &
      true(closed(door))
legal(candidate, noop) :-
      true(step(2))
legal(candidate, noop) :-
      true(step(3))
legal(candidate, switch) :-
      true(step(3))

sees(candidate, door) :-
      does(random, openDoor(door))

next(car(door)) :-
      does(random, hideCar(door))
next(car(door)) :-
      true(car(door))
next(closed(door)) :-
      true(closed(door)) &
      ~does(random, openDoor(door))
next(chosen(door)) :-
```

```
            does(candidate, choose(door))
    next(chosen(door)) :-
            true(chosen(door)) &
            ~does(candidate, switch)
    next(chosen(door)) :-
            does(candidate, switch) &
            true(closed(door)) &
            ~true(chosen(door))
    next(step(2)) :-
            true(step(1))
    next(step(3)) :-
            true(step(2))
    next(step(4)) :-
            true(step(3))

    terminal :-
            true(step(4))

    goal(candidate, 100) :-
            true(chosen(door)) &
            true(car(door))
    goal(candidate, 0) :-
            true(chosen(door)) &
            ~true(car(door))
```

7.3 Rock, Paper, Scissors Python Propositional Network File

Shortened propositional network file in Python format [19].

```
from constants import *

roles = [
    'player1',
    'player2',
]

entries = (
    (0, -1, PROPOSITION, [2], [], goal, '( goal player1 75 )'),
    (1, -1, PROPOSITION, [], [20, 78], input, '( does player1
       scissors )'),
    (2, -1, AND, [76, 34], [0]),
    (3, -1, AND, [74, 76], [83]),
    (4, -1, TRANSITION, [50], [71]),
    (5, -1, AND, [76, 16], [33]),
    (6, -1, AND, [79, 16], [31]),
    (7, -1, PROPOSITION, [40], [25], other, '( next ( chosen player1
       rock ) )'),
    (8, -1, AND, [54, 16], [15]),
    (9, -1, PROPOSITION, [11], [], sees, '( sees player1 ( chose
       paper ) )'),
    (10, -1, TRANSITION, [42], [34]),
    (11, -1, PROPOSITION, [], [9, 35], input, '( does player1 paper
       )'),
    (12, -1, PROPOSITION, [23], [17], base, '( true ( step 2 ) )'),
    (13, -1, AND, [54, 74], [47]),
    (14, -1, AND, [76, 34], [41]),
    (15, -1, OR, [8, 67], [39]),
    (16, -1, PROPOSITION, [72], [56, 5, 45, 6, 8, 82], base, '( true
       ( chosen player2 scissors ) )'),
    (17, -1, PROPOSITION, [12], [], terminal, 'terminal'),
        ... 65 more nodes ...
    (83, -1, OR, [3, 45, 59], [55]),
)
```

7.4 Rock, Paper, Scissors JSON Propositional Network File

Shortened propositional network file in JSON format.

```
{
    "roles": [
        {
            "role": "player1",
            "goals": [
                {
                    "goal": 0,
                    "value": 75
                },
                    ... 3 more goals ...
                {
                    "goal": 81,
                    "value": 0
                }
            ],
            "sees": [
                9,
                20,
                49
            ],
            "legal_to_input": [
                {
                    "legal": 24,
                    "input": 48
                },
                {
                    "legal": 58,
                    "input": 11
                },
                {
                    "legal": 62,
                    "input": 1
                }
            ]
        },
        {
            "role": "player2",
                ... goals, sees and legal_to_input for player2 ...
        }
    ],
    "entries": [
        {
            "id": 0,
            "type": 3,
            "type_name": "PROPOSITION",
            "proposition_type": "goal",
            "gdl": "( goal player1 75 )",
            "in": 2,
            "display": "( goal player1 75 )"
        },
        {
            "id": 1,
            "type": 3,
            "type_name": "PROPOSITION",
            "proposition_type": "input",
            "gdl": "( does player1 scissors )",
            "display": "( does player1 scissors )"
        },
        {
            "id": 2,
            "type": 1,
            "type_name": "AND",
            "ins": [
                76,
                34
            ]
        },
            ... 88 more nodes ...
        {
            "id": 91,
            "type": 9,
            "type_name": "POST TRANSITION",
            "in": 72
        }
    ],
    "topologically_sorted": [
        84,
        85,
        86,
            ... all other nodes in topological ordering ...
        0
    ]
}
```