Optimizing Propositional Networks

Chiara F. Sironi(✉) and Mark H. M. Winands

Games & AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, The Netherlands
{c.sironi,m.winands}@maastrichtuniversity.nl

Abstract. General Game Playing (GGP) programs need a Game Description Language (GDL) reasoner to be able to interpret the game rules and search for the best actions to play in the game. One method for interpreting the game rules consists of translating the GDL game description into an alternative representation that the player can use to reason more efficiently on the game. The Propositional Network (PropNet) is an example of such method. The use of PropNets in GGP has become popular due to the fact that PropNets can speed up the reasoning process by several orders of magnitude compared to custom-made or Prolog-based GDL reasoners, improving the quality of the search for the best actions. This paper analyzes the performance of a PropNet-based reasoner and evaluates four different optimizations for the PropNet structure that can help further increase its reasoning speed in terms of visited game states per second.

1 Introduction

The aim of General Game Playing (GGP) is to develop programs that are able to play any arbitrary game at an expert level by being only given its rules. These programs must devise a playing strategy without having any prior knowledge about the game. Moreover, the rules are given to the player just before game playing starts and usually for each game step only few seconds are available to choose a move. Thus, the player has to learn an appropriate playing strategy on-line and in a limited amount of time.

To be able to play games, a GGP program has two main components: a way to interpret the game rules, written in the Game Description Language (GDL), and a strategy to choose which actions to play.

Regarding the first component, many different approaches have been proposed to parse the game rules. Three main methods to interpret GDL can be identified: (1) Prolog-based interpreters that translate the game rules from GDL into Prolog and then use a Prolog engine to reason about them, (2) custom-made interpreters written for the sole purpose of interpreting GDL rules, and (3) reasoners that translate the GDL description into an alternative representation that the player can use to efficiently reason about the game. A description and performance evaluation of available GDL reasoners is given in [7].

Regarding the second component, most of the approaches that proved successful in addressing the challenges of GGP are based on Monte-Carlo simulation techniques and especially on Monte-Carlo Tree Search (MCTS) [1, 2]. For Monte-Carlo methods the choice of the best action to play is based on game statistics collected by sampling the state space of the game. The number of samples that Monte-Carlo methods can collect directly influences their performance. A higher number of samples in general improve the quality of the chosen actions.

A faster GDL reasoner, which in a given amount of time can analyze a higher number of game states than other reasoners, can positively influence Monte-Carlo based search. Propositional Networks (PropNets) [3, 8] have become popular in GGP because they can speed up the reasoning process by several orders of magnitude compared to custom-made or Prolog-based GDL reasoners. Nowadays, all the best GGP programs use a PropNet-based reasoner [4, 5, 9].

The purpose of this paper is to analyze the performance of the implementation of the PropNet-based reasoner provided in the GGP-Base framework [9], discuss four optimizations of the structure of the PropNet and empirically evaluate their impact on the speed of the reasoning process. The performance of the custom-made GDL reasoner provided in the GGP-Base framework, called GGP-Base Prover, has been used as a reference.

The reminder of the paper is structured as follows. Section 2 gives a short introduction to GDL and PropNets. Sections 3 and 4 give some details about the PropNet implementation and a description of the PropNet optimizations respectively. Section 5 presents the empirical evaluation of the PropNet and Sect. 6 concludes and indicates potential future work.

2 Background

PropNets are one of the promising representations that can be used to reason about GDL descriptions. Subsection 2.1 gives a brief introduction to GDL and Subsect. 2.2 briefly describes the structure of a PropNet.

2.1 The Game Description Language

The Game Description Language (GDL) is a first order logic language used in GGP to represent the rules of games [6]. In GDL a game state is defined by specifying which propositions are true in that state. A set of reserved keywords is used to define the characteristics of the game.

Figure 1 shows as an example the GDL description of a simple game, where a player can independently turn on two lights (p and q). After being turned on, each light will remain on. The game ends when both lights are on and the player achieves a goal with score 100. In the figure, the GDL keywords are represented in bold.

2.2 The PropNet

A Propositional Network (PropNet) [3, 8] can be seen as a graph representation of GDL. Each component in the PropNet represents either a proposition or a

```
(role player)
(light p) (light q)
(<= (legal player (turnOn ?x)) (not (true (on ?x))) (light ?x))
(<= (next (on ?x)) (does player (turnOn ?x)))
(<= (next (on ?x)) (true (on ?x)))
(<= terminal (true (on p)) (true (on q)))
(<= (goal player 100) (true (on p)) (true (on q)))
```

Fig. 1. Example of GDL game description.

logic gate. Propositions can be distinguished into three types: *input* propositions that have no input components, *base* propositions that have one single *transition* as input, and *view* propositions that are the remaining ones. The truth values of *base* propositions represent the state of the game. The dynamics of the game are represented by *transitions* that are identity gates that output their input value with one step delay and control the truth values of *base* propositions in the next step. The truth value of every other component is a function of the truth value of its inputs, except for *input* propositions, for which the game playing agent sets a value when choosing the action to play. Figure 2 shows as an example the PropNet that corresponds to the GDL description given in Fig. 1.

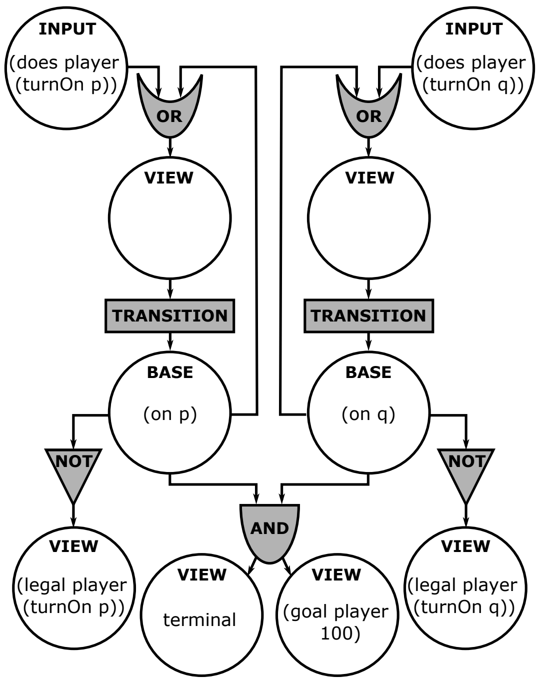

Fig. 2. PropNet structure example.

3 PropNet Implementation

To create the PropNet the algorithm provided in the GGP-Base framework is used.[1] This algorithm is implemented in the *create(List<Gdl> description)* method of the *OptimizingPropNetFactory* class and builds the PropNet according to the rules in the given GDL description.

The final product of the algorithm is a set of all the components in the PropNet, each of which has been connected to its input and output components. This set can then be used to initialize a PropNet object. The algorithm distinguishes six different types of components: *constants* (TRUE and FALSE), *propositions*, *transitions* and three different *gates* (AND, OR, NOT).

The GGP-Base framework also provides a PropNet class that can be initialized using the created set of components. We used this class as a starting point and implemented some changes to the initialization process to ensure that the PropNet respects certain constraints that are needed for the optimizations algorithms to work consistently. The first step of the initialization iterates over all the components in the PropNet and inserts them in different lists according to their type. While iterating over all the components, the following are the main actions that the initialization algorithm performs:

- Identify a single TRUE and a single FALSE constant, creating them if they do not exist, or removing the redundant ones.
- Identify the type of each proposition. Each proposition must be associated to one type only. A proposition that has a *transition* as input is identified as BASE type and a proposition that corresponds to a GDL relation containing the *does* keyword is identified as INPUT type. The propositions corresponding to GDL relations containing the *legal*, *goal* or *terminal* keyword are identified as LEGAL, GOAL and TERMINAL type respectively. To all other propositions the type OTHER is assigned.
- Make sure that all the INPUT and LEGAL propositions are in a 1-to-1 relation. If a proposition is detected as being an INPUT but there is no corresponding LEGAL in the PropNet, then it can be removed since we are sure that the corresponding move will never be chosen by the player. On the contrary, if there is a LEGAL proposition with no corresponding INPUT, the INPUT proposition is added to the PropNet, since the LEGAL proposition might become true at a certain point of the game and the player might choose to play the corresponding move.
- Make sure that only constants and INPUT propositions have no input components. If a different component is detected as having no inputs, set one of the two constants as its input. This action is needed because as a by-product of the PropNet creation some OR gates and non-INPUT propositions might have no inputs. The behavior of the PropNet has been empirically tested to be consistent when such components are connected to the FALSE constant.

[1] We have used a more recent and improved version than the one tested in [7].

4 Optimizations

The PropNets built by the algorithm given in the GGP-Base framework [9] contain usually many components that are not strictly necessary to reason about the game. This section presents four optimizations that can be performed on the PropNet structure to reduce the number of these components. Opt0 (Subsect. 4.1) removes components that are known to have a constant truth value, Opt1 (Subsect. 4.2) removes propositions that do not have a particular meaning, Opt2 (Subsect. 4.3) detects more constant components and removes them, and Opt3 (Subsect. 4.4) removes components that have no output and are not influential. All the optimization algorithms except the last one are already provided in the GGP-Base framework. The algorithms described here contain some minor modifications with respect to the original GGP-Base version in order to adapt them to the changes that were performed on the PropNet class structure.

4.1 Opt0: Remove Constant-value Components

This optimization removes from the PropNet the components that are known to be always *true* or always *false* and at the same time do not have a particular meaning for the game. For example an AND gate that has an input that is always *false* will also always output *false*, thus the gate can be removed and all its outputs can be connected directly to the *FALSE* constant of the PropNet.

Algorithm 1 shows the main steps of this optimization. The sets O_T and O_F, at any moment, contain respectively the outputs of the *TRUE* and the outputs of the *FALSE* constant that still have to be checked for removal. At the beginning O_T contains all the outputs of the *TRUE* constant and O_F contains all the outputs of the *FALSE* constant (Lines 2 and 3).

The procedure $\textsc{RemoveFromTrue}(propnet, O_T, O_F)$ (Line 5) and the procedure $\textsc{RemoveFromFalse}(propnet, O_T, O_F)$ (Lines 6) check the outputs of the *TRUE* and of the *FALSE* constant respectively. Algorithm 2 shows exactly which components the first procedure removes. The algorithm for the second procedure removes the outputs of the FALSE constant in a similar way. In the case of the FALSE constant, also always false GOAL and LEGAL propositions are removed since they will never be used. Moreover, whenever a LEGAL proposition is removed also the corresponding INPUT proposition is removed, since it is certain that the corresponding move will never be played.

Note that whenever a component is removed or detected as having always a constant value, it means that also its output is constant, thus its output components are connected directly to one of the two constants. In this case each output component will be added to the appropriate set (either O_T or O_F) to be checked in the next steps.

Algorithm 1 alternates between the two procedures mentioned above until both sets, O_T and O_F, are empty. This repetition is needed because of the NOT gate. Whenever this gate is removed from the outputs of a constant, its outputs are connected to the other constant, thus the set of outputs to be checked for that constant will still have at least one element.

Algorithm 1 Remove constant-value components

```
procedure OPT_0($propnet$)
    $O_T \leftarrow propnet.TRUE.outputs$
    $O_F \leftarrow propnet.FALSE.outputs$
    while $O_T \neq \emptyset$ or $O_F \neq \emptyset$ do
        REMOVEFROMTRUE($propnet, O_T, O_F$)
        REMOVEFROMFALSE($propnet, O_T, O_F$)
    end while
end procedure
```

Algorithm 2 Remove true components

```
procedure REMOVEFROMTRUE($propnet, O_T, O_F$)
    while $O_T \neq \emptyset$ do
        $c \leftarrow O_T.removeElement()$
        switch $c.compType$ do
            case TRANSITION
                if $|c.outputs| = 0$ then
                    $propnet.remove(c)$
                end if
            case NOT
                connect $c.outputs$ to FALSE
                $O_F \leftarrow O_F \cup c.outputs$
                $propnet.remove(c)$
            case AND
                if $|c.inputs| = 1$ then                    ▷ Only TRUE as input
                    connect $c.outputs$ to TRUE
                    $O_T \leftarrow O_T \cup c.outputs$
                    $propnet.remove(c)$
                else if $|c.inputs| = 2$ then               ▷ Only 2 inputs, one is TRUE
                    connect $c.outputs$ to other input
                    $propnet.remove(c)$
                else                                        ▷ More than 2 inputs, one is TRUE
                    disconnect $c$ form TRUE
                end if
            case OR
                connect $c.outputs$ to TRUE
                $O_T \leftarrow O_T \cup c.outputs$
                $propnet.remove(c)$
            case PROPOSITION
                connect $coutputs$ to TRUE
                $O_T \leftarrow O_T \cup c.outputs$
                if $c.propType \in$ {OTHER, BASE} then
                    $propnet.remove(c)$
                end if
        end switch
    end while
end procedure
```

4.2 Opt1: Remove Anonymous Propositions

This optimization is trivial, nevertheless useful as it removes many useless components from the PropNet. The algorithm for this optimization (Algorithm 3) simply iterates over all the propositions in the PropNet and removes the ones with type OTHER, connecting their input directly to each of their outputs. These propositions can be safely removed as they do not have any special meaning for the game.

Algorithm 3 Remove anonymous propositions

```
1: procedure OPT1(propnet)
2:     for all $p \in propnet.propositions$ do
3:         if $p.propType =$ OTHER then
4:             connect $p.input$ with $p.outputs$
5:             $propnet.remove(p)$
6:         end if
7:     end for
8: end procedure
```

4.3 Opt2: Detect and Remove Constant-value Components

This optimization can be seen as an extension of Opt0 where, before removing from the PropNet the constant value components directly connected to the *TRUE* and *FALSE* constant, the algorithm detects if there are other constant value components that have not been discovered yet.

This optimization (see Algorithm 4) associates to each component c in the PropNet a set V_c that contains all the truth values that such component can assume during the whole game. There are only four possible sets of truth values, namely:

– $N = \emptyset$: if the corresponding component can assume *neither* of the truth values.
– $T = \{true\}$: if the corresponding component can only be *true* during all the game.
– $F = \{false\}$: if the corresponding component can only be *false* during all the game.
– $B = \{true, false\}$: if the corresponding component can assume *both* values during the game.

The idea behind the algorithm is to start from the components for which the truth value that they will assume in the initial state of the game is known. It then propagates this value to each of their outputs o and updates the corresponding truth value set V_o. Whenever the truth values set of a component is updated, the algorithm propagates such changes on to its output components. This process

Algorithm 4 Detect and remove constant-value components

```
 1: procedure Opt2($propnet$)
 2:     Initialize all the parameters and the stack $S$
 3:     while $S \neq \emptyset$ do
 4:         $(c, P_i) \leftarrow S.pop()$
 5:         $O_c \leftarrow$ ToOutputValueSet$(c, P_i)$
 6:         $P_c \leftarrow O_c \setminus V_c$
 7:         if $P_c \neq N$ then
 8:             $V_c \leftarrow V_c \cup P_c$
 9:             for all $o \in c.outputs$ do
10:                 $S.push(o, P_c)$
11:             end for
12:             if $c.compType$ = PROPOSITION and $c.propType$ = LEGAL then
13:                 $i \leftarrow c.correspondingInput$
14:                 $S.push(i, P_c)$
15:             end if
16:         end if
17:     end while
18:     for all $c \in propnet.components$ do
19:         if $V_c = T$ or $V_c = F$ then
20:             Connect $c$ to the appropriate constant
21:         end if
22:     end for
23:     Opt0($propnet$)
24: end procedure
```

will eventually end when the truth values sets of all components stop changing. Termination is guaranteed since only the truth values just added to the truth values set of a component are propagated to its outputs and the number of possible truth values is finite.

When the algorithm starts, the set V_c of each component c is set to N, since it is not known yet which values the component can assume. For each AND gate a the algorithm keeps track of TI_a, i.e. the number of inputs of a that can assume the *true* value. Similarly, for each OR gate o the algorithm keeps track of FI_o, i.e. the number of inputs of o that can assume the *false* value. This parameters are used to detect when an AND gate and an OR gate can assume respectively the *true* (if $TI_a = |a.inputs|$) and the *false* (if $FI_o = |o.inputs|$) value. These values are initialized to 0 for all the gates.

The algorithm exploits a stack structure S to keep track of the components for which the set of truth values that their input(s) can assume is changed. A pair (c, P_i) is added to the stack when the algorithm detects that an input i of the component c can also assume the values in the set $P_i \subseteq V_i$, and such values must be propagated to the component c. At the beginning the stack is filled with the following pairs:

– $(TRUE, T)$, the $TRUE$ constant can assume value *true*.
– $(FALSE, F)$, the $FALSE$ constant can assume value *false*.

– (i, F), for each INPUT proposition i in the PropNet. Each INPUT proposition can be *false* since we assume that no game exists where one player can only play a single move for the whole game.
– (b_j, T), for each BASE proposition b_j in the PropNet that is *true* in the initial state.
– (b_j, F), for each BASE proposition b_j in the PropNet that is *false* in the initial state.

During each iteration, the algorithm pops a pair (c, P_i) from the stack (Line 4) and checks if, given the new truth values P_i that the input i can assume, also the truth values V_c of its output c will change. Note that not for each type of component the set of truth values that its input can assume corresponds to the set of truth values that the component itself can output. The NOT component n, for example, has $V_n = T$ if its input i has $V_i = F$. Moreover, for an AND gate a, $true \in V_a \Leftrightarrow true \in V_i, \forall i \in a.inputs$. The same holds for the $false$ value for an OR gate. This means that the algorithm must first change the values in P_i according to the type of the component c, obtaining the new set of truth values O_c that c can output. This is done at Line 5 by the function ToOutputValueSet(c, P_i). Subsequently, the algorithm checks if in O_c there are some values P_c that were not in V_c yet (Line 6), and if so, it adds them to the set V_c (Line 8) and records on the stack that they have to be propagated to all the outputs o of c (Lines 9-11). Here the algorithm treats each LEGAL propositions as if it was a direct input of the corresponding INPUT proposition, thus whenever the truth values set of a LEGAL proposition changes, the values are propagated to the corresponding INPUT proposition (Lines 12-15).

When no more changes are detected in the truth values sets (Line 3), the process terminates. At this point, the truth values set of each component is checked (Line 19) and if it equals the set T or F it is certain that the component will always be respectively *true* or *false*. It can then be disconnected from its input(s) and connected to the correct constant (Line 20).

The last step the algorithm performs consists in running the same algorithm that was proposed as Opt0 to remove all the newly detected constant components (Line 23).

4.4 Opt3: Remove Output-less Components

This optimization is also quite trivial, but helps remove some more useless components. Algorithm 5 shows this procedure: all the components in the PropNet are checked, if they are gates, or propositions of type OTHER and they have no output they are removed from the PropNet. Every time a component is removed, its inputs are added again to the set of components to be checked, since removing their outputs might have made them output-less.

5 Empirical Evaluation

In this section an empirical evaluation of the performance of the PropNet and its optimizations is presented. Subsection 5.1 describes the setup of the performed

Algorithm 5 Remove output-less components

```
 1: procedure OPT3($propnet$)
 2:     $Q \leftarrow propnet.components$
 3:     while $Q \neq \emptyset$ do
 4:         $c \leftarrow Q.removeElement()$
 5:         if $((c.compType = \text{PROPOSITION}$ and $c.propType = \text{OTHER})$
              or $c.compType \in \{\text{AND, OR, NOT}\})$ and $|c.outputs| = 0$ then
 6:             $Q \leftarrow Q \cup c.inputs$
 7:             $propnet.remove(c)$
 8:         end if
 9:     end while
10: end procedure
```

experiments. Subsections 5.2 and 5.3 discuss the results of the experiments that compare the performance of single optimizations and combinations of them respectively. The combination of PropNet optimizations that performs overall best is then compared with the default Prover. Subsection 5.4 presents a comparison of PropNet and Prover in terms of their speed, while Subsect. 5.5 presents a comparison in terms of their game-playing performance.

5.1 Setup

To evaluate the performance of the PropNet multiple series of experiments are performed. Each of them tests the performance of the PropNet with different optimizations and combinations of them. Each series of experiments poses the bases to decide which other combinations of optimizations to check.

The different PropNet optimizations and their combinations are tested using flat Monte-Carlo Search (MCS) on a set of heterogeneous games. For each optimized PropNet the search is run from the initial state of the game with a time limit of 20s. This experiment is repeated 100 times for each of the chosen games. Such games are the following: *Amazons*, *Battle*, *Breakthrough*, *Chinese Checkers* with 1, 2, 3, 4 and 6 players, *Connect 4*, *Othello*, *Pentago*, *Skirmish* and *Tic Tac Toe*. The GDL descriptions of these games can be found on the GGP-Base repository [10].[2]

One of the reasons behind the choice of repeating each experiment multiple times for each game is that for each repetition of the game a different seed is used for the random number generator that controls the random exploration of the search tree with the MCS algorithm. Thus, for different seeds different results might be obtained and different parts of the search space explored.

Another reason is that the number of components that the PropNet of a game has when created by the basic algorithm (i.e. without optimizations) is not always constant. This variance in the number of components could be due

[2] The GDL descriptions used for the experiments were downloaded from the repository on 03/02/2016.

to the non-determinism of the order in which game rules are translated into PropNet components for different runs of the algorithm. This can cause a different grounding order of the GDL description, originating more or less propositions and can also cause gates and propositions to be connected in different equivalent orders.

The optimized PropNet that showed the best overall performance in the previous series of experiments is compared with the GGP-Base Prover in another series of experiments. Both reasoners are also tested with the addition of a cache that memorizes the queries results.

This series of experiments matches two MCS-based players that use the Prover, one with cache and one without, against each other, and two MCS-based players that use the best optimized PropNet, one with cache and one without, against each other. We use the same 13 games that were used for the other experiments. Each player has 10s per move to perform the search. A new PropNet is built for each match in advance, before the game playing starts. For each game, if r is the number of roles in the game, there are 2^r different ways in which 2 types of players can be assigned to the roles [11]. Two of the configurations involve only the same player type assigned to all the roles, thus are not interesting and excluded from the experiments. Each configuration is run the same number of times until at least 100 games have been played in total.

At the end of each game repetition the speed of the reasoners is computed by dividing the total number of nodes visited by the total time spent on the search during the whole game. Since we are only interested in the reasoning speed, for this experiment we do not consider the 10s search time per move strictly, but we allow each player to finish the current simulation when this time expires.

The final series of experiments aims at evaluating the impact of the reasoners on the win rate of game playing agents. This experiments match two MCTS-based players, one that uses the fastest version of the Prover (i.e. with the cache) and one that uses the fastest optimized PropNet (also with the cache), against each other. The settings are the same as in the previous experiment, except the minimum number of played games that is increased to 200. Moreover, for this experiment the 10s search time per move is considered strictly.

Before running any of the described experiments, the PropNet and all its optimized versions were tested against the Prover for consistency. For each game (about 300) in the GGP-Base repository [10], for a duration of 60s, the same random simulations were performed querying both the Prover and the currently tested version of the PropNet for next states, legal moves, terminality and goals in terminal states. The results returned by the PropNet were compared with the ones returned by the Prover for consistency. All the PropNet versions passed this test on all the games in the repository, except for 12 games for which the PropNet construction could not be completed in the given time.

In all experiments, a limit of 10 minutes was given to the program to build the PropNet. The experiments that compare the speed of PropNet and Prover with and without cache were performed on an AMD Opteron 6174 2.2-GHz. All other experiments were performed on an AMD Opteron 6274 2.2-GHz.

5.2 Comparison of Single Optimizations

The first series of experiments compares with the basic version of the PropNet (BasicPN) the performance of each of the previously described optimizations applied singularly (Opt0, Opt1, Opt2, Opt3). Table 1 shows the obtained results. For each PropNet variant, for each game the first block of the table gives the average simulation speed in nodes per second, the second block gives the average number of components and the third block gives the average total initialization time (creation+optimization+state initialization) in milliseconds. The line at the bottom of each block reports the average over the 13 games of the percentage increase of the values considered in the block, relative to the basic version of the PropNet (BasicPN).

The main interest is the speed increase that the optimizations induce on the PropNet, however the other two aspects are also relevant. A low number of components means less memory usage, and a shorter initialization time means more time for metagaming at the beginning of a match (or more chances to avoid timing out when the start clock time is short). From the table it seems that for most of the games, as expected, the increase in the simulation speed is related to the decrease in the number of components in the PropNet.

As can be seen, none of the optimizations outperforms the others in speed for all games. Opt0 and Opt2 seem to have the best performance in *Amazons*, *Battle*, *Othello* and *Connect 4*, while Opt1 performs best in the other games. When looking at the initialization time, Opt2 is the one that increases it the most for almost all the games. Another observation is that the performance of Opt2 is overall better than the one of Opt0. This was expected because Opt2 is an extension of Opt0, thus for the same PropNet it always removes at least the same number of components as Opt0.

The speed is used as main criterion to choose which of the four optimization to use as starting point for further experiments that involve testing combinations of optimizations. If we consider the speed, Opt0 and Opt2 are the ones that, on average, produce the highest increase. However, the high average is due to the considerable relative increase that they produce in *Othello*. If we consider the optimization that produces the highest speed in most of the games, then Opt1 is the most suitable to be selected. Moreover, Opt1 is the optimization that reduces the most the number of components of the PropNet without consistently slowing down the initialization process.

5.3 Comparison of Combined Optimizations

In this series of experiments Opt1 is combined with other optimizations applied in sequence. In general, when we refer to OptXY we refer to the PropNet optimization obtained by applying OptX and OptY in sequence. These experiments first compare the combinations of optimizations Opt13, Opt12 and Opt102. The combination Opt10 has been excluded from the test since it is considered less interesting. As also previously mentioned, Opt0 always removes a subset of the components that are removed by Opt2, thus Opt10 is expected to perform less

Table 1. Comparison of single optimizations

	Game	BasicPN	Opt0	Opt1	Opt2	Opt3
Avg. speed (nodes/second)	Amazons	35.1	41.4	32.7	41	40.2
	Battle	34957	49666	37877	51257	35276
	Breakthrough	50557	50932	65518	51357	51058
	Chinese Checkers 1P	426374	427773	550230	444671	424516
	Chinese Checkers 2P	125581	128623	189368	128910	127519
	Chinese Checkers 3P	155886	157242	169352	161000	159267
	Chinese Checkers 4P	105766	106738	127886	107153	105660
	Chinese Checkers 6P	119650	118547	126863	113700	118783
	Connect 4	110081	113484	105081	112920	109672
	Othello	290	1610	235	1604	295
	Pentago	76336	76786	116065	76721	96782
	Skirmish	5887	6022	6780	6230	6151
	Tic Tac Toe	223403	228056	248769	234915	222952
	Avg. relative increase	-	40.59%	15.51%	41.44%	3.95%
Avg. number of components	Amazons	1497649	1254742	741874	1192364	1023913
	Battle	51197	14267	36863	14262	50721
	Breakthrough	10745	10678	5933	10678	10584
	Chinese Checkers 1P	793	785	559	785	789
	Chinese Checkers 2P	1540	1524	1179	1524	1532
	Chinese Checkers 3P	2411	2389	1845	2236	2400
	Chinese Checkers 4P	3159	3119	2465	2999	3133
	Chinese Checkers 6P	4451	4411	3473	4123	4431
	Connect 4	2164	2063	1724	1291	2114
	Othello	1311988	274940	1033197	274940	1305515
	Pentago	3696	3706	1470	3708	2111
	Skirmish	126019	124267	108171	124267	78575
	Tic Tac Toe	312	291	249	291	302
	Avg. relative increase	-	-14.28%	-29.21%	-18.62%	-9.49%
Avg. total init. time (ms)	Amazons	311335	313719	314455	417097	315637
	Battle	5756	6027	5897	6303	5869
	Breakthrough	3989	4007	4012	4358	3910
	Chinese Checkers 1	2699	2651	2659	2653	2707
	Chinese Checkers 2	2848	2773	2810	2873	2775
	Chinese Checkers 3	3162	3140	3159	3251	3149
	Chinese Checkers 4	3258	3261	3241	3473	3244
	Chinese Checkers 6	3225	3203	3204	3639	3205
	Connect 4	2437	2465	2456	2698	2430
	Othello	35756	36486	37074	39417	36544
	Pentago	4249	4230	4278	4390	4232
	Skirmish	11887	11702	11664	12089	11824
	Tic Tac Toe	1525	1529	1523	1522	1508
	Avg. relative increase	-	0.13%	0.24%	7.69%	-0.19%

than Opt12. However, Opt0 has less negative impact than Opt2 on the total initialization time. This is why these experiments include the test of Opt102: we want to see if the application of Opt0 before Opt2 can speed up the process of Opt2 that will then run on a smaller PropNet.

The results of this series of experiments can be seen in columns 3, 4 and 5 of Table 2. The structure of this table is the same as Table 1. The average percentage increase reported in the last line of each block is still computed with respect to the basic version of the PropNet (BasicPN).

As the table shows, regarding the speed, Opt12 seems to be the one achieving the best overall performance. However, the performance of Opt102 is rather close, as expected, because these two combinations should reduce each PropNet to the same number of components. The small difference in performance is probably due the reasons already mentioned in Sect. 5.1. Both the difference in the random seed used for each repetition of the game and the variance in the number of components generated by the algorithm that creates the initial PropNet can influence the performance.

One more thing that can be noticed from Table 2 is that running Opt0 before Opt2 helps reducing the initialization time for large games, while it seems to have almost no effect on smaller games. Moreover, Opt13 is the one that, regarding the speed, performs worse in this series of experiments, thus it has been excluded from further tests. Among Opt12 and Opt102, it has been chosen to keep testing on top of Opt102 because of its shorter initialization time for games with large PropNets, given that its speed is still comparable with the one of Opt12.

Using Opt102 as starting point, there is only one more interesting combination of optimizations left to test: Opt1023. No further gain in performance can be obtained by repeating the same optimizations multiple times in a row, since no further change will take place in the structure of the PropNet. Thus, it is not interesting to evaluate combinations of optimizations that extend Opt1023.

The last column of Table 2 shows the statistics for Opt1023. For most of the games, Opt1023 seems to be the fastest. It is also the one that reduces the number of PropNet components the most. As for the initialization time, this optimization is between a few milliseconds and a bit more than 1 second slower that the basic version of the PropNet, except for *Amazons*. Optimizing the large PropNet of *Amazons* can slow down the initialization time by more than a minute.

5.4 Comparison of PropNet and Prover

In this series of experiments the overall fastest combination of optimizations among the tested ones (Opt1023) is compared with the Prover. More precisely, Opt1023 and the Prover are compared measuring their speed over complete games (as opposed to previous experiments that were comparing the speed only on the first step of the game).

Moreover, for both of them also a cached version is tested (i.e. CachedProver and CachedOpt1023). The GGP-Base framework [9] provides a cache structure

Table 2. Comparison of combined optimizations

	Game	BasicPN	Opt12	Opt102	Opt13	Opt1023
Avg. speed (nodes/second)	Amazons	35	38.5	41.4	32.3	41
	Battle	34957	59308	59697	39981	60419
	Breakthrough	50557	66943	66551	66833	66991
	Chinese Checkers 1P	426374	570858	562737	541682	561634
	Chinese Checkers 2P	125581	194442	192048	190161	193752
	Chinese Checkers 3P	155886	175410	176162	170722	176185
	Chinese Checkers 4P	105766	130362	130279	129194	130451
	Chinese Checkers 6P	119650	127535	128111	127619	129000
	Connect 4	110081	127053	126535	105978	129272
	Othello	290	1934	1894	245	1979
	Pentago	76336	116353	115064	117127	121108
	Skirmish	5887	7075	7042	7403	7600
	Tic Tac Toe	223403	259980	257285	247246	257525
	Avg. relative increase	-	70.32%	69.39%	17.38%	73.48%
Avg. number of components	Amazons	1497649	623460	623460	711596	596240
	Battle	51197	11084	11077	36676	10902
	Breakthrough	10745	5900	5900	5869	5836
	Chinese Checkers 1P	793	556	556	559	556
	Chinese Checkers 2P	1540	1172	1172	1179	1172
	Chinese Checkers 3P	2411	1718	1718	1845	1718
	Chinese Checkers 4P	3159	2362	2362	2465	2362
	Chinese Checkers 6P	4451	3238	3238	3473	3238
	Connect 4	2164	1063	1063	1724	1056
	Othello	1311988	208510	208510	1031580	206846
	Pentago	3696	1464	1473	1338	1337
	Skirmish	126019	107296	107296	62427	61552
	Tic Tac Toe	312	239	239	249	239
	Avg. relative increase	-	-42.34%	-42.32%	-32.52%	-45.65%
Avg. total init. time (ms)	Amazons	311335	411905	400113	312793	401559
	Battle	5756	6367	6233	5968	6329
	Breakthrough	3989	4354	4415	3982	4328
	Chinese Checkers 1P	2699	2693	2654	2707	2652
	Chinese Checkers 2P	2848	2848	2843	2817	2842
	Chinese Checkers 3P	3162	3214	3186	3160	3167
	Chinese Checkers 4P	3258	3405	3330	3275	3379
	Chinese Checkers 6P	3225	3423	3430	3207	3395
	Connect 4	2437	2536	2555	2417	2525
	Othello	35756	39170	36689	35359	37804
	Pentago	4249	4269	4286	4308	4325
	Skirmish	11887	12386	12285	11870	12577
	Tic Tac Toe	1525	1532	1535	1524	1555
	Avg. relative increase	-	6.38%	5.18%	0.18%	5.66%

Table 3. Comparison of the PropNet with the Prover and effect of the cache

	Game	Prover	CacheProver	Opt1023	CacheOpt1023
Avg. speed (nodes/second)	Amazons	5.7	2316	28.1	30519
	Battle	45.2	2457	38656	36607
	Breakthrough	235	241	56275	51569
	Chinese Checkers 1P	2273	466014	532426	862408
	Chinese Checkers 2P	1478	93251	159935	258639
	Chinese Checkers 3P	1105	28300	118160	133733
	Chinese Checkers 4P	536	32684	82955	117017
	Chinese Checkers 6P	607	5744	57008	53230
	Connect4	180	2455	122325	207508
	Othello	3.2	5502	649	80328
	Pentago	152	155	93185	75998
	Skirmish	26	4081	2997	3946
	Tic Tac Toe	1650	287380	225127	547398
	Avg. relative increase	-	22139%	-	9321%

that memorizes the results returned by the underlying reasoner and prevents it from computing the same queries multiple times.

The results of these experiments are shown in Table 3. The last row of this table reports for both CachedProver and CachedOpt1023 the average percentage increase of the speed with respect to their non-cached versions.

From the table it is visible how the optimized PropNet achieves a much better performance than the Prover in the considered games. When adding the cache to both reasoners the difference in performance is reduced, however the PropNet is still faster in all games but one, *Skirmish*, for which the speed of the cached PropNet and the cached Prover are quite close.

The use of a cache provides some benefits increasing the overall performance of both reasoners with respect to their non-cached version. However, the cache gives more benefits to the Prover. For the Prover the speed is increased for all the games, while for the PropNet it is increased for most, but not all of them. To be noticed is that the increase in speed provided by the cache is especially relevant in the games of *Amazons* and *Othello*.

Moreover, observing the results for all the *Chinese Checkers* versions it is clear that the speed of the cached Prover and the speed of the cached PropNet both decrease when increasing the number of players. However, for the PropNet this decrease is slower. For *Chinese Checkers* with 1 player the cached PropNet is about 2 times faster than the cached Prover, while for the version with 6 players it is almost 10 times faster.

When performing the experiments it was also noticed that in many games the cache decreases the speed of the PropNet reasoner during the initial steps. This loss is then balanced towards the endgame, when the chance of finding cached query results increases. It takes some time for the cache to be filled with a sufficient number of entries and thus have a positive impact on the speed of the PropNet.

The same effect was not observed for the Prover. For the first steps of the games the cache did not decrease the speed of the Prover for any of the games, and for some of them increased it. The explanation for this is that the time for computing the answer of a query with the Prover is in general much higher than the one of the PropNet. Thus, for the Prover finding in the cache even a small number of query results saves enough computational time to compensate the extra time spent looking in the cache for results that are not present yet.

Finally, the results of Table 3 also help putting the PropNet into perspective with the other GDL reasoners analyzed in the paper [7]. Even if that paper uses different experimental settings than ours, we can still make some general observations. Considering the performance of the reasoners that, like the PropNet, rely on an alternative representation of the GDL description, it seems that our implementation of the PropNet provides for most of the games a speed increase of the same order of magnitude when compared to the Prover. Moreover, for *Amazons*, *Othello* and *Chinese Checkers* with 4 and 6 players, it seems that our optimized PropNet, especially with the cache, could even achieve a better performance in similar circumstances.

5.5 Game Playing Performance

In this series of experiments an MCTS player that uses the cached PropNet reasoner with the fastest combination of optimizations (Opt1023) is matched against an MCTS player that uses the cached Prover. Because Sect. 5.4 showed the cache to be overall beneficial for both reasoners, it has been included in this experiment.

Table 4 shows the win percentage of the cached PropNet-player against the cached Prover-player. The table does not include the results for the single-player version of *Chinese Checkers* because this game is tested separately and the score is used to measure the performance of the players. This game has a relatively small search space, so both players achieved the maximum score in every match.

Table 4. Win percentage of the PropNet-player against the Prover-player

Game	Opt1023
Battle	100.0(±0.0)
Breakthrough	100.0(±0.0)
Chinese Checkers 2P	96.0(±2.72)
Chinese Checkers 3P	77.5(±5.75)
Chinese Checkers 4P	68.1(±6.32)
Chinese Checkers 6P	64.7(±5.73)
Connect 4	99.3(±1.09)
Pentago	100.0(±0.0)
Skirmish	100.0(±0.0)
Tic Tac Toe	50.0(±0.0)

Moreover, no results are shown for *Amazons* and *Othello* because for both games, during the first game steps, the cached Prover-player could not return a move within the given time limit. Even with the use of the cache, during the first game steps the number of memorized query results is not sufficient to allow the Prover to complete even one MCTS simulation within the time limit.

Looking at the results for the remaining games, for most of them the cached PropNet-player achieves a win percentage close or equal to 100%. The games in which the performance of the cached PropNet-player seems to drop are the ones with more than 2 players. *Chinese Checkers* with 4 and 6 players are the ones where the win percentage for the cached PropNet-player is the lowest, but it is still significantly better than the one of the cached Prover-player. The game of *Tic Tac Toe* is the only exception, because its state space is so small that both players can easily reach a sufficient number of simulations to play optimally and result in a tie.

The results of Table 4 are in line to what would be expected when looking at the average speed reported in Table 3 for the two cached reasoners. For all the games for which the speed of the cached PropNet is at least one order of magnitude higher than the one of the cached Prover, the cached PropNet-player achieves a significantly higher win percentage. However, for the game of *Skirmish* Table 3 reports a similar average speed for both the cached reasoners so their performance would be expected to be close. A win rate of 100% for the cached PropNet can be explained by the fact that the speed per game step of the Prover exhibits a higher variance than the speed of the PropNet. The speed of the PropNet in the initial game steps is close to the average speed. The Prover, instead, is about 340 times slower than the PropNet in this stage. Its speed increases only in the last few steps of the game. At this point the PropNet-player already gained enough advantage over the Prover-player to win the game.

6 Conclusion and Future Work

In this paper the performance of a PropNet-based reasoner has been evaluated, together with four possible optimizations of the structure of the PropNet and their impact on the performance. Even though the tested implementation of the PropNet is based on the code provided by the GGP-Base framework, the principles behind its representation and its optimizations can also be applied in general.

Experiments have shown that the use of a PropNet substantially increases the reasoning speed by, on average, at least two orders of magnitude with respect to the GGP-Base Prover. Moreover, the addition of a combination of optimizations that reduce the size of the PropNet increases the reasoning speed further. Experiments also show that the reasoning speed increase has a positive effect on the performance of the PropNet-based player. This player achieves a win rate close to 100% in most of the games for which it is matched against an equivalent player based on the Prover. Thus, it is possible to conclude that for a general game playing agent a reasoner based on a PropNet, especially when optimized, is in general a better choice than a custom-made GDL interpreter like the Prover.

Also the use of a cache proved to be useful for the PropNet in most of the games. For small games its effect is already visible in the first steps, while for most of the other games it helps only during later game steps. However, we may conclude that the use of a cache is overall positive for a PropNet reasoner.

Future work could further investigate the use of the cache with the PropNet, for example by devising a strategy to detect for each game if and when the use of a cache is helpful. Finally, another interesting aspect that future work could consider is the impact that the use of different strategies to propagate truth values among the components of the PropNet would have on the reasoning speed.

Acknowledgments. This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121.

References

1. Björnsson, Y., Finnsson, H.: CadiaPlayer: A simulation-based general game player. Computational Intelligence and AI in Games, IEEE Transactions on 1(1), 4–15 (2009)
2. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) Proceedings of the 5th International Conference on Computer and Games. Lecture Notes in Computer Science (LNCS), vol. 4630, pp. 72–83. Springer-Verlag, Heidelberg, Germany (2007)
3. Cox, E., Schkufza, E., Madsen, R., Genesereth, M.R.: Factoring general games using propositional automata. In: Björnsson, Y., Stone, P., Thielscher, M. (eds.) Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA). pp. 13–20 (2009)
4. Draper, S., Rose, A.: Sancho GGP player. `http://sanchoggp.blogspot.nl/2014/07/sancho-is-ggp-champion-2014.html` (2014)
5. Emslie, R.: Galvanise. `https://bitbucket.org/rxe/galvanise_v2` (2015)
6. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.R.: General Game Playing: Game Description Language specification. Tech. rep., Stanford University, Stanford, CA, USA (2008)
7. Schiffel, S., Björnsson, Y.: Efficiency of GDL reasoners. Computational Intelligence and AI in Games, IEEE Transactions on 6(4), 343–354 (2014)
8. Schkufza, E., Love, N., Genesereth, M.R.: Propositional automata and cell automata: representational frameworks for discrete dynamic systems. In: Wobcke, W., Zhang, M. (eds.) AI 2008: Advances in Artificial Intelligence. Lecture Notes in Artificial Intelligence (LNAI), vol. 5360, pp. 56–66. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
9. Schreiber, S.: The General Game Playing base package. `https://github.com/ggp-org/ggp-base` (2013)
10. Schreiber, S.: Games - base repository. `http://games.ggp.org/base/` (2016)
11. Sturtevant, N.R.: An analysis of UCT in multi-player games. ICGA Journal 31(4), 195–208 (2008)