The IJCAI-16 Workshop on General Game Playing

General Intelligence in Game-Playing Agents, GIGA’16

New York City, USA, July 2016
Proceedings
Artificial Intelligence (AI) researchers have for decades worked on building game-playing agents capable of matching wits with the strongest humans in the world, resulting in several success stories for board games like chess and checkers and computer games such as StarCraft, Pac-Man and Unreal Tournament. The success of such systems has been partly due to years of relentless knowledge-engineering effort on behalf of the program developers, manually adding application-dependent knowledge to their game-playing agents. The various algorithmic enhancements used are often highly tailored towards the game at hand.

Research into general game playing (GGP) aims at taking this approach to the next level: to build intelligent software agents that can, given the rules of any game, automatically learn a strategy for playing that game at an expert level without any human intervention. In contrast to software systems designed to play one specific game, systems capable of playing arbitrary unseen games cannot be provided with game-specific domain knowledge a priori. Instead, they must be endowed with high-level abilities to learn strategies and perform abstract reasoning. Successful realization of such programs poses many interesting research challenges for a wide variety of artificial-intelligence sub-areas including (but not limited to):

- computational creativity
- computational game theory
- evaluation and analysis
- game design
- imperfect-information games
- knowledge representation
- machine learning
- multi-agent systems
- opponent modeling
- planning
- reasoning
- search

These are the proceedings of GIGA’16, the first ever(!) 5th workshop on General Intelligence in Game-Playing Agents, following the inaugural GIGA Workshop at IJCAI’09 in Pasadena (USA) and the follow-up events at IJCAI’11 in Barcelona (Spain), IJCAI’13 in Beijing (China), and IJCAI’15 in Buenos Aires (Argentina). This workshop series has been established to become the major forum for discussing, presenting and promoting research on General Game Playing. It is intended to bring together researchers from the above sub-fields of AI to discuss how best to address the challenges and further advance the state-of-the-art of general game-playing systems and generic artificial intelligence.

These proceedings contain the eight papers that have been selected for presentation at this workshop. All submissions received three independent and anonymous reviews by the members of a distinguished international program committee. The accepted papers cover a multitude of topics such as boosting the efficiency of reasoning about game rules, general game playing with imperfect information, Monte-Carlo tree search, and general video game playing.

We thank all the authors for responding to the call for papers with their high quality submissions, and the program committee members and other reviewers for their valuable feedback and comments. We also thank IJCAI for all their help and support.

We welcome all our delegates and hope that all will enjoy the workshop and through it find inspiration for continuing their work on the many facets of General Game Playing!

July 2016

Stephan Schiffel
Michael Thielscher
Julian Togelius
Workshop Chairs

**Stephan Schiffel**, Reykjavík University, Iceland

**Michael Thielscher**, University of New South Wales, Australia

**Julian Togelius**, New York University, USA

Program Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yngvi Björnsson</td>
<td>Reykjavík University</td>
</tr>
<tr>
<td>Tristan Cazenave</td>
<td>Université Paris-Dauphine</td>
</tr>
<tr>
<td>Michael Genesereth</td>
<td>Stanford University</td>
</tr>
<tr>
<td>Lukasz Kaiser</td>
<td>Université Paris Diderot</td>
</tr>
<tr>
<td>Simon Lucas</td>
<td>University of Essex</td>
</tr>
<tr>
<td>Jacek Mańdziuk</td>
<td>Warsaw University of Technology</td>
</tr>
<tr>
<td>Diego Perez</td>
<td>University of Essex</td>
</tr>
<tr>
<td>Ji Ruan</td>
<td>Auckland University of Technology</td>
</tr>
<tr>
<td>Abdallah Saffidine</td>
<td>University of New South Wales</td>
</tr>
<tr>
<td>Spyridon Samothrakis</td>
<td>University of Essex</td>
</tr>
<tr>
<td>Tom Schaul</td>
<td>Google DeepMind</td>
</tr>
<tr>
<td>Sam Schreiber</td>
<td>Google Inc.</td>
</tr>
<tr>
<td>Nathan Sturtevant</td>
<td>University of Denver</td>
</tr>
<tr>
<td>Mark Winands</td>
<td>Maastricht University</td>
</tr>
</tbody>
</table>

Additional Reviewer

Chiara Sironi
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizing Propositional Networks</td>
<td>7</td>
</tr>
<tr>
<td>Chiara F. Sironi and Mark H. M. Winands</td>
<td></td>
</tr>
<tr>
<td>Grounding GDL Game Descriptions</td>
<td>15</td>
</tr>
<tr>
<td>Stephan Schiffel</td>
<td></td>
</tr>
<tr>
<td>A General Approach of Game Description Decomposition for General Game Playing</td>
<td>23</td>
</tr>
<tr>
<td>Aline Hufschmitt, Jean-Noël Vittaut and Jean Méhat</td>
<td></td>
</tr>
<tr>
<td>GDL-III: A Description Language for Epistemic General Game Playing</td>
<td>30</td>
</tr>
<tr>
<td>Michael Thielscher</td>
<td></td>
</tr>
<tr>
<td>Stochastic Constraint Programming for General Game Playing with Imperfect Information</td>
<td>38</td>
</tr>
<tr>
<td>Frederic Koriche, Sylvain Lagrue, Eric Piette and Sebastien Tabary</td>
<td></td>
</tr>
<tr>
<td>Generating Sokoban Puzzle Game Levels with Monte Carlo Tree Search</td>
<td>46</td>
</tr>
<tr>
<td>Bilal Kartal, Nick Sohre and Stephen Guy</td>
<td></td>
</tr>
<tr>
<td>An Investigation into the Effectiveness of Heavy Rollouts in UCT</td>
<td>54</td>
</tr>
<tr>
<td>Steven James, Benjamin Rosman and George Konidaris</td>
<td></td>
</tr>
<tr>
<td>Evolving UCT Alternatives for General Video Game Playing</td>
<td>62</td>
</tr>
<tr>
<td>Ivan Bravi, Ahmed Khalifa, Christoffer Holmgård and Julian Togelius</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Propositional Networks

Chiara F. Sironi and Mark H. M. Winands
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 analyses 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 a 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 on 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 on the game. A description and performance evaluation of available GDL reasoners is given in [Schiffel and Björnsson, 2014].

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) [Coulom, 2007; Björnsson and Finnsson, 2009]. 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 will improve the quality of the chosen actions.

A faster GDL reasoner, that in a given amount of time can analyse a higher number of game states than other reasoners, can positively influence Monte-Carlo based search. Propositional Networks (PropNets) [Schkufza et al., 2008; Cox et al., 2009] 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 general game players use a PropNet-based reasoner [Schreiber, 2013; Darper and Rose, 2014; Emsile, 2015].

The purpose of this paper is to analyse the performance of the implementation of the PropNet-based reasoner provided in the GGP-Base framework [Schreiber, 2013], 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 rest 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 Section 6 concludes and indicates potential future work.

2 Background
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 [Love et al., 2008]. In GDL a game state is defined by specifying
(role player)
(light p) (light q)
(<= (legal player (turnOn ?x)) (not (true (on ?x))) (light ?x))
(<= (next (on ?x)) (true (on ?x)))
(<= terminal (true (on p)) (true (on q)))
(<= (goal player 100) (true (on p)) (true (on q)))

Figure 1: Example of GDL game description.

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 very 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) [Schkufza et al., 2008; Cox et al., 2009] can be seen as a graph representation of GDL. Each component in the PropNet represents either a proposition or a 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 Figure 1.

3 PropNet Implementation

To create the PropNet the algorithm already provided in the GGP-Base framework is used. This algorithm is implemented in the create(List<PropNet> 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 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 behaviour 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 [Schiffl and Björnsson, 2014].
4 Optimizations

The PropNets built by the algorithm given in the GGP-Base framework [Schreiber, 2013] contain usually many components that are not strictly necessary to reason on the game. This section presents four optimizations that can be performed on the PropNet structure to reduce the number of this components. Opt0 removes components that are known to have a constant truth value. Opt1 removes propositions that do not have a particular meaning. Opt2 detects more constant components and removes them, and Opt3 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.

Algorithm 1 shows the main steps of this procedure. 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 procedures \( \text{REMOVEFROMTRUE}(O_T, O_F) \) and \( \text{REMOVEFROMFALSE}(O_T, O_F) \) (Lines 5 and 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 functions 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.

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 1 Remove constant-value components

1: \( \text{OPT0(propnet)} \)
2: \( O_T \leftarrow \text{outputs}(\text{TRUE}) \)
3: \( O_F \leftarrow \text{outputs}(\text{FALSE}) \)
4: while \( O_T \neq \emptyset \) or \( O_F \neq \emptyset \) do
5: \( \text{REMOVEFROMTRUE}(O_T, O_F) \)
6: \( \text{REMOVEFROMFALSE}(O_T, O_F) \)
7: end while

Algorithm 2 Remove true components

1: procedure \( \text{REMOVEFROMTRUE}(O_T, O_F) \)
2: while \( O_T \neq \emptyset \) do
3: \( c \leftarrow \text{removeComponent}(O_T) \)
4: switch \( \text{cType}(c) \) do
5: case \( \text{TRANSITION} \)
6: if \( |\text{outputs}(c)| = 0 \) then
7: \( \text{propnet.remove}(c) \)
8: end if
9: case \( \text{NOT} \)
10: \( \text{connect outputs}(c) \) to FALSE
11: \( O_F \leftarrow O_F \cup \text{outputs}(c) \)
12: \( \text{propnet.remove}(c) \)
13: case \( \text{AND} \)
14: if \( |\text{inputs}(c)| = 1 \) then
15: \( \text{connect outputs}(c) \) to TRUE
16: \( O_T \leftarrow O_T \cup \text{outputs}(c) \)
17: \( \text{propnet.remove}(c) \)
18: else if \( |\text{inputs}(c)| = 2 \) then
19: \( \text{connect outputs}(c) \) to other input
20: \( \text{propnet.remove}(c) \)
21: end if
22: case \( \text{OR} \)
23: \( \text{connect outputs}(c) \) to TRUE
24: \( O_T \leftarrow O_T \cup \text{outputs}(c) \)
25: \( \text{propnet.remove}(c) \)
26: case \( \text{PROPOSITION} \)
27: \( \text{connect outputs}(c) \) to TRUE
28: \( O_T \leftarrow O_T \cup \text{outputs}(c) \)
29: if \( \text{pType}(c) \in \{\text{OTHER, BASE}\} \) then
30: \( \text{propnet.remove}(c) \)
31: end if
32: end switch
33: end while
34: end procedure

Algorithm 3 Remove anonymous propositions

1: \( \text{OPT1(propnet)} \)
2: for all \( p \in \text{propositions(propnet)} \) do
3: if \( \text{pType}(p) = \text{OTHER} \) then
4: \( \text{connect input}(p) \) with \( \text{outputs}(p) \)
5: \( \text{propnet.remove}(p) \)
6: end if
7: end for
Algorithm 4 Detect and remove constant-value components

1: 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: \( O_c \leftarrow \text{TOOUTPUTVALUESET}(c, P_i)\)
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 \( o \in \text{outputs}(c) \) do
10: \( \quad S.push(o, P_c)\)
11: end for
12: if \( cType(c) = \text{PROPOSITION} \) then
13: \( \quad \text{if } pType(c) = \text{LEGAL} \text{ then } \)
14: \( \quad \quad i \leftarrow \text{correspondingInput}(c)\)
15: \( \quad \quad S.push(i, P_c)\)
16: end if
17: end if
18: end while
19: for all \( c \in \text{components(propnet)} \) do
20: \( \quad \text{if } V_c = T \text{ or } V_c = F \text{ then } \)
21: \( \quad \quad \text{Connect } c \text{ to the appropriate constant}\)
22: end if
23: end for
24: OPT0(propnet)

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 \( T \) and \( F \) 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 such that 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.
- \( T = \{ \text{true} \} \): if the corresponding component can only be \text{true} during all the game.
- \( F = \{ \text{false} \} \): if the corresponding component can only be \text{false} during all the game.
- \( B = \{ \text{true}, \text{false} \} \): 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 outputs components. This process 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 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 \text{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 \text{false} value. These parameters are used to detect when an AND gate and an OR gate can assume respectively the \text{true} (if \( TI_a = |\text{inputs}(a)| \)) and the \text{false} (if \( FI_o = |\text{inputs}(o)| \)) 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 its 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 \( P_i \subseteq V_c \). Such values must be propagated to the component \( c \). At the beginning the stack is filled with the following pairs:

- \( (\text{TRUE}, T) \), the \text{TRUE} constant can assume value \text{true}.
- \( (\text{FALSE}, F) \), the \text{FALSE} constant can assume value \text{false}.
- \( (i, F) \), for each INPUT proposition \( i \) in the PropNet. Each INPUT proposition can be \text{false} since we assume that no game exists where one player can only play a single move for the whole game.
- \( (b_i, T) \), for each BASE proposition \( b_i \) in the PropNet that is \text{true} in the initial state.
- \( (b_i, F) \), for each BASE proposition \( b_i \) in the PropNet that is \text{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 \text{NOT} component \( v_i \), for example, has \( V_{v_i} = T \) if its input \( i \) has \( V_i = F \). Moreover, for an AND gate \( a \), if \( T \in V_a \) then \( \text{true} \in V_a \). For each \text{OR} gate \( o \) (Line 8) and records on the stack that they have to be propagated to all the outputs \( o \) of \( c \) (Line 10). Here the algorithm treats each \text{LEGAL} propositions as if it was a direct input of the corresponding INPUT proposition, thus whenever the truth values set of a \text{LEGAL} proposition changes, the values are propagated to the corresponding INPUT proposition (Lines 12-17).

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 21) and if it equals the set \( T \) or \( F \) it is certain that the component will always be respectively \text{true} or \text{false}. It can then be disconnected from its input(s) and connected to the correct constant (Line 22).
Algorithm 5 Remove output-less components

1: OPT3(propnet)
2: Q ← components(propnet)
3: while Q ̸= ∅ do
4: c ← removeElement(Q)
5: switch cType(c) do
6: case AND, OR, NOT, OTHER PROPOSITION
7: if |outputs(c)| = 0 then
8: Q ← Q ∪ inputs(c)
9: propnet.remove(c)
10: end if
11: end switch
12: end while

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 25).

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

5.1 Setup

To evaluate the performance of the PropNet multiple series of experiments are performed. Each of them compares the performance of the PropNet with different optimizations and combinations of optimizations. Each series of experiments poses the bases to decide which other combinations of optimizations to check. Furthermore, the optimized PropNet that shows the best overall performance is compared with the GGP-Base Prover. Both reasoners are also tested with the addition of a cache that memorizes the queries results.

The reasoners are tested using flat Monte-Carlo Search (MCS) on a set of heterogeneous games. For each reasoner 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: Amazon, 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 [Schreiber, 2016].

One of the reasons behind the choice of repeating each experiment multiple times for each game 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 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.

Another series of experiments matches two MCTS-based players, one that uses the Prover and one that uses the PropNet, 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 [Sturtevant, 2008]. 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 amount of times until at least 100 matches have been played.

The final series of experiments further investigates the effect of the cache. It matches two MCTS-based players that use the Prover, one with cache and one without, against each other, and two MCTS-based players that use the best optimized PropNet, one with cache and one without, against each other. The settings are the same as in the previous experiment. At the end of each match the speed 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.

Before running any of the described experiments, the PropNet and all its optimized versions were tested against the Prover for consistency. For each game in the GGP-Base repository [Schreiber, 2016], 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 experiment that further investigates the effects of the cache on complete games was performed on an AMD Opteron 6274 2.2-GHz. All other experiments were performed on an AMD Opteron 6174 2.2-GHz.

5.2 Comparison of Single Optimizations

In the first series of experiments we compared with the basic version of the PropNet (BasicPN) the performance of each of the previously described optimizations applied singularly (Opt0, Opt1, Opt2, Opt3). Columns 2 to 6 of Table 1 show 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
Amazons achieve the best performance in the decrease in the number of components in the PropNet. As can be seen, none of the optimizations outperforms the BasicPN for all games, however Opt1 is the most suitable to be selected. More interestingly, we can see 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.

If we consider the speed as main choice criterion, Opt0 and Opt2 are the ones that, on average, produce the highest speed increase. However, the high average is due to the considerable relative increase that they produce in Othello. If we consider as starting point for the next series of experiments 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.

Table 1: Comparison of the optimizations and some of their combinations
## 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 compare the combinations of optimizations Opt13, Opt12 and Opt102 with the single optimization Opt1. 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 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 7, 8 and 9 of Table 1. Regarding the speed, Opt12 seems to be the one achieving the best overall performance. However, the performance of Opt102 is rather close, as expected, since these two combinations should remove the same number of components in each PropNet. The difference in performance is probably due to some variance in the data. Moreover, running Opt0 before Opt2 helps reducing the initialization time for large games, while it seems to have almost no effect on smaller games.

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.

## 5.4 Comparison of PropNet and Prover

In this series of experiment only one more interesting combination of optimizations is 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.

### 5.5 Game Playing Performance

In this series of experiments an MCTS player that uses the PropNet reasoner is matched against one that uses the Prover. Table 3 shows the win percentage of the PropNet-player against the Prover-player.

The last column of Table 1 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.

Table 2 shows the comparison of the speed of Opt1023 with the Prover. For both of them also a cached version is tested. The GGP-Base framework [Schreiber, 2013] provides a cache structure that memorizes the results returned by the underlying reasoner and prevents it from computing the same queries multiple times. As can be seen, the cache seems to be helping only in a few games, but this is due to the simulation being run only for the first step of the game (see Subsection 5.6). For games with many states and legal moves the cache takes more time and more game steps to be filled with enough entries and really make an impact on the speed.

Looking at Table 2 it is clearly visible how the optimized PropNet achieves a much better performance than the Prover in the considered games, and Table 1 shows how it performs also better than the basic version.

As further test to evaluate the robustness of the PropNet, we compared its performance with the one of the Prover also on all the games (about 300) in the GGP-Base repository [Schreiber, 2016]. For each considered reasoner we performed once for each game the Monte-Carlo search with a time limit of 60s. On average the basic version of the PropNet was 418 times faster than the Prover and the version with all optimizations, Opt1023, was 698 times faster.

### Table 3: Win percentage of the PropNet-player against the Prover-player

<table>
<thead>
<tr>
<th>Game</th>
<th>Opt1023</th>
<th>Game</th>
<th>Opt1023</th>
</tr>
</thead>
<tbody>
<tr>
<td>battle</td>
<td>100.0(±0.0)</td>
<td>c-check.6</td>
<td>69.4(±7.67)</td>
</tr>
<tr>
<td>breakthr.</td>
<td>100.0(±0.0)</td>
<td>connect4</td>
<td>99.0(±1.38)</td>
</tr>
<tr>
<td>c-check.2</td>
<td>98.0(±2.76)</td>
<td>pentago</td>
<td>100.0(±0.0)</td>
</tr>
<tr>
<td>c-check.3</td>
<td>83.3(±7.27)</td>
<td>skirmish</td>
<td>100.0(±0.0)</td>
</tr>
<tr>
<td>c-check.4</td>
<td>70.5(±8.48)</td>
<td>ticTacToe</td>
<td>50.0(±0.0)</td>
</tr>
</tbody>
</table>

The last column of Table 1 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.

Table 2 shows the comparison of the speed of Opt1023 with the Prover. For both of them also a cached version is tested. The GGP-Base framework [Schreiber, 2013] provides a cache structure that memorizes the results returned by the underlying reasoner and prevents it from computing the same queries multiple times. As can be seen, the cache seems to be helping only in a few games, but this is due to the simulation being run only for the first step of the game (see Subsection 5.6). For games with many states and legal moves the cache takes more time and more game steps to be filled with enough entries and really make an impact on the speed.

Looking at Table 2 it is clearly visible how the optimized PropNet achieves a much better performance than the Prover in the considered games, and Table 1 shows how it performs also better than the basic version.

As further test to evaluate the robustness of the PropNet, we compared its performance with the one of the Prover also on all the games (about 300) in the GGP-Base repository [Schreiber, 2016]. For each considered reasoner we performed once for each game the Monte-Carlo search with a time limit of 60s. On average the basic version of the PropNet was 418 times faster than the Prover and the version with all optimizations, Opt1023, was 698 times faster.

### 5.5 Game Playing Performance

In this series of experiments an MCTS player that uses the PropNet reasoner is matched against one that uses the Prover. Table 3 shows the win percentage of the PropNet-player against the Prover-player. In most of the games the PropNet-player achieves a win percentage close or equal to 100%. The games in which the performance of the 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 PropNet-player is the lowest, but it is still significantly better than the one of the Prover-player. The game of Tic Tac Toe is the only exception, since 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.

No results are shown for Amazons and Othello since for both games the Prover-player could not complete even one iteration of the MCTS algorithm in the given time limit. The
<table>
<thead>
<tr>
<th>Game</th>
<th>Prover</th>
<th>CachePr.</th>
<th>Opt1023</th>
<th>CacheOpt1023</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg. speed (nodes/second)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amazons</td>
<td>5.7</td>
<td>2316</td>
<td>28.1</td>
<td>30519</td>
</tr>
<tr>
<td>battle</td>
<td>45.2</td>
<td>2457</td>
<td>38656</td>
<td>36607</td>
</tr>
<tr>
<td>breakhtr.</td>
<td>235</td>
<td>241</td>
<td>56275</td>
<td>51569</td>
</tr>
<tr>
<td>e-check.1</td>
<td>2273</td>
<td>466014</td>
<td>532426</td>
<td>862408</td>
</tr>
<tr>
<td>e-check.2</td>
<td>1478</td>
<td>93251</td>
<td>159935</td>
<td>258639</td>
</tr>
<tr>
<td>e-check.3</td>
<td>1105</td>
<td>28300</td>
<td>118160</td>
<td>133733</td>
</tr>
<tr>
<td>e-check.4</td>
<td>536</td>
<td>32684</td>
<td>82955</td>
<td>117017</td>
</tr>
<tr>
<td>e-check.6</td>
<td>607</td>
<td>5744</td>
<td>57008</td>
<td>53230</td>
</tr>
<tr>
<td>connect4</td>
<td>180</td>
<td>2455</td>
<td>122325</td>
<td>207508</td>
</tr>
<tr>
<td>othello</td>
<td>3.2</td>
<td>5502</td>
<td>649</td>
<td>80328</td>
</tr>
<tr>
<td>pentago</td>
<td>152</td>
<td>155</td>
<td>93185</td>
<td>75998</td>
</tr>
<tr>
<td>skirmish</td>
<td>26</td>
<td>4081</td>
<td>2997</td>
<td>3946</td>
</tr>
<tr>
<td>ticTacToe</td>
<td>1650</td>
<td>287380</td>
<td>225127</td>
<td>547398</td>
</tr>
</tbody>
</table>

Table 4: Effect of the cache on Prover and PropNet over complete games

6.6 Cache Effect on Complete Games

Table 4 shows the results of the comparison of the Prover and the PropNet with and without cache over complete games. It is visible how the cache actually provides some benefits if used for the whole game. For the Prover it increases the speed for all the games, while for the PropNet it increases the speed for most of them. The increase is especially relevant for the games of Amazons and Othello. These results, together with the ones in Table 2, are probably an indication that usually the cache slows down the performance in the initial steps of the game, but then balances this loss towards the endgame, when the chance of finding queries results in the cache increases.

Comparing with the Prover over complete games also helps putting the PropNet into perspective with the other GDL reasoners analysed in the paper [Schiffel and Björnsson, 2014]. 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.

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.

Also the use of a cache proved to be useful in some 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. Future work could investigate a way of detecting for each game if and when the cache could be 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


Grounding GDL Game Descriptions*

Stephan Schiffel
School of Computer Science / CADIA
Reykjavik University, Iceland
stephans@ru.is

Abstract

Many state-of-the-art general game playing systems rely on a ground (propositional) representation of the game rules. We propose a theoretically well-founded approach using efficient off-the-shelf systems for grounding game descriptions given in the game description language (GDL).

1 Introduction

Games in General Game Playing are generally described in the game description language (GDL) [Love et al., 2008]. While allowing to describe a large class of games and being theoretically well-founded, reasoning with GDL is generally slow compared to game specific representations [Schiffel and Björnsson, 2014]. This limits the speed of search, both for heuristic search methods, such as Minimax, as well as for simulation-based approaches, such as Monte Carlo Tree Search. Thus, an important aspect of General Game Playing is to find a better representation of the rules of a game that facilitates both fast search in the game tree as well as efficient meta-gaming analysis. Propositional networks [Schkufza et al., 2008] and binary decision diagrams (BDDs) [Edelkamp and Kissmann, 2011] have been proposed for faster reasoning with the game rules. Both approaches, require that game descriptions be grounded, that is, translated into a propositional representation. Other meta-gaming approaches could also benefit from having a propositional description of the game rules as input. Some examples are finding symmetries in games [Schiffel, 2010], discovering heuristics for games (e.g., [Michulke and Schiffel, 2013]), proving game properties [Schiffel and Thielscher, 2009a; Haufe et al., 2012] and factoring games [Cox et al., 2009; Cerexhe et al., 2014].

The GGP-Base framework [Schreiber and Landau, 2016], which is the basis for a number of general game players, contains code for generating a propositional network [Schkufza et al., 2008] representing the game rules. This code requires computing ground instances of all rules in the game description. However, the code seems ad-hoc and it is not obvious whether and for which class of game descriptions it maintains the semantics of the game rules. With this paper, we propose to transform the game description into an answer set program [Baral, 2003] and use the grounder of a state-of-the-art answer set solving system to compute a propositional representation of the game. This has the advantage of using a highly optimized and well-tested system, as well as being theoretically well-founded. Our system also turns out to be able to handle more games than the GGP-Base framework and being significantly faster for many games.

2 Related Work

In [Kissmann and Edelkamp, 2010], the authors report two methods for grounding GDL, one using Prolog and another using dependency graphs. Both methods have some deficiencies and the authors only manage to ground 96 out of the 171 tested games. While the authors do not report on the size of the grounded game descriptions, they report on one of their methods to produce game descriptions that are unnecessarily big.

In [Haufe et al., 2012], The authors use Answer Set Programming (ASP) [Baral, 2003] to prove properties of games by transforming a GDL description into an answer set program and adding constraints that encode the properties to be proven. The system they have implemented uses the Potassco ASP solver [Gebser et al., 2011] which relies on grounding the answer set program, and thus, indirectly the game description.

3 Game Description Language (GDL)

The game description language [Love et al., 2008; Schiffel and Thielscher, 2009b] is a first-order-logic based language that can be seen as an extension of Datalog permitting negations and function symbols. Thus, a game description in GDL is a logic program. The game specific semantics of GDL stems from the use of certain special relations, such as for describing the initial game state (init), detecting (terminal) and scoring (goal) terminal states, for generating legal moves (legal) and successor states (next). A game state is represented by the set of terms that are true in the state (e.g., cell(1,1,b)) and the special relations true(f) and does(r,m) can be used to refer to the truth of f being in the current state and role r doing move m in the current state.

*This work was supported by the Icelandic Centre for Research (RANNIS).
role(xplayer).
role(oplayer).
init(cell(1, 1, b)).
init(cell(1, 2, b)).
init(cell(3, 2, b)).
init(control(xplayer)).
legal(W, mark(X, Y)) :-
true(cell(X, Y, b)),
true(control(W)).
legal(oplayer, noop) :-
true(control(xplayer)).
next(cell(M, N, x)) :-
control(xplayer),
control(xplayer).
row(M, X) :-
next(control(xplayer)) :-
true(control(xplayer)).
next(M, X) :-
true(cell(M, 1, X)),
true(cell(M, 2, X)),
true(cell(M, 3, X)).
line(X) :-
row(M, X).
line(X) :-
diagonal(M, X).
line(X) :-
diagonal(X).
goal(xplayer, 100) :-
line(x).
goal(xplayer, 0) :-
line(o).
terminal :-
line(x).

Figure 1: A partial GDL game description for the game Tic-tactoe (reserved GDL keywords are marked in bold)

transition. Figure 1 shows a partial GDL description for the game Tic-tactoe.

GDL allows to describe a wide range of deterministic perfect-information simultaneous-move games with arbitrary number of adversaries. Turn-based games are modeled by having the players that do not have a turn return a move with no effect (e.g., noop in Figure 1).

To ensure an unambiguous declarative interpretation, valid GDL descriptions need to fulfill a number of restrictions:

Definition 1. The dependency graph for a set \( G \) of clauses is a directed, labeled graph whose nodes are the predicate symbols that occur in \( G \) and where there is a positive edge \( p \rightarrow q \) if \( G \) contains a clause \( p(\overline{x}) \leftarrow \ldots \land q(\overline{t}) \land \ldots \), and a negative edge \( p \leftrightharpoons q \) if \( G \) contains a clause \( p(\overline{x}) \leftarrow \ldots \land \neg q(\overline{t}) \land \ldots \).

To constitute a valid GDL specification, a set of clauses \( G \) and its dependency graph \( \Gamma \) must satisfy the following.

1. There are no cycles involving a negative edge in \( \Gamma \) (this is also known as being stratified [Apt et al., 1987; van Gelder, 1989]).
2. Each variable in a clause occurs in at least one positive atom in the body (this is also known as being allowed [Lloyd and Topor, 1986]).
3. If \( p \) and \( q \) occur in a cycle in \( \Gamma \) and \( G \) contains a clause
\[ p(s_1, \ldots, s_m) \leftarrow b_1(\overline{t}_1) \land \ldots \land b_n(\overline{t}_n) \]
then for every \( i \in \{1, \ldots, k\} \):
   - \( v_i \) is variable-free, or
   - \( v_i \) is one of \( s_1, \ldots, s_m \), or
   - \( v_i \) occurs in some \( \overline{t}_j \) \( (1 \leq j \leq n) \) such that \( b_j \) does not occur in a cycle with \( p \) in \( \Gamma \). 

4 Restrictions

As mentioned in [Haufe et al., 2012] (Section 3.2), there is no finite grounding of a GDL description in general. While the restrictions from Definition 1 ensure that reasoning about single states or state transitions is finite, the restrictions are not strong enough to ensure finiteness or decidability of reasoning about the game in general, such as, whether the game will terminate or is winnable for some player.

In fact, without further restrictions, GDL as defined in [Love et al., 2008] or [Schiffel and Thielscher, 2009b] is Turing complete [Saffidine, 2014]. Thus, some restrictions to the language are necessary in order to be able to ground a game description and only game descriptions that adhere to these restrictions can be grounded. The restriction used in [Haufe et al., 2012] and termed bounded GDL in [Saffidine, 2014] is the following:

Definition 2. Let \( G \) be a GDL specification. Let \( G' \) be \( G \) extended with the following three rules:

\[
\begin{align*}
true(F) & \leftarrow \text{init}(F). \\
true(F) & \leftarrow \text{next}(F). \\
does(B, M) & \leftarrow \text{legal}(B, M).
\end{align*}
\]

\( G \) is in the bounded GDL fragment of GDL descriptions, if, and only if, \( G' \) satisfies the recursion restriction.

As discussed in [Saffidine, 2014], this restriction makes bounded GDL decidable and therefore truly less expressive than (unbounded) GDL. However, this is of little practical consequence as all of the game descriptions currently available in GDL belong to the bounded fragment.

5 Grounding

To obtain a ground version of a game description, we transform it into an answer set program \( P \), ground \( P \) using very optimized grounder for answer set programs and extract the ground version of the game rules from the grounded answer set program.

Specifically, the program \( P \) that we create consists of
- the game description itself,
- a state generator,
- an action generator, and
- rules that encode all possible state terms and moves in the game.
The following definitions are based on [Haufe et al., 2012], but simplified for our purpose.

**Definition 3.** A state generator for a valid GDL specification $G$ is an answer set program $P^{gen}$ such that

- The only atoms in $P^{gen}$ are of the form $\text{true}(f)$, where $f \in \Sigma$ or auxiliary atoms that do not occur elsewhere; and
- for every reachable state $S$ of $G$, $P^{gen}$ has an answer set $\mathcal{A}$ such that for all $f \in \Sigma \text{true}(f) \in \mathcal{A}$ iff $f \in S$.

We use the following state generator

\[
\text{true}(F): \text{base}(F).
\]

where, intuitively, $\text{base}(f)$ encodes all possible terms $f$ that might appear in a state of the game.

**Definition 4.** Let $A(S)$ denote the set of all legal joint moves in $S$, that is,

\[
A(S) \overset{\text{def}}{=} \{ A : R \rightarrow \Sigma | l(r, A(r), S) \}
\]

An action generator for a valid GDL specification $G$ is an answer set program $P^{legal}$ such that

- The only atoms in $P^{legal}$ are of the form $\text{does}(r, m)$, where $r \in R$ and $m \in \Sigma$, or auxiliary atoms that do not occur elsewhere;
- for every reachable (non-terminal) state $S$ of $G$ and every joint move $A \in A(S)$, $P^{legal}$ has an answer set $\mathcal{A}$ such that for all $r \in R$: $\text{does}(r, A(r)) \in \mathcal{A}$; and
- for every reachable (terminal) state $S \in T$ of $G$, $P^{legal}$ has an answer set.

We use the following action generator

\[
1 = \{ \text{does}(R, M) : \text{input}(R, M) \} :- \text{role}(R).
\]

where, intuitively, $\text{input}(r, m)$ encodes all possible moves $m$ of role $r$ in the game. Thus, our action generator does admit answer sets that might not be legal joint moves for a specific state. However, this is not a problem, since we are not interested in the answer sets, but only the grounded answer set program.

For several years, games in the international general game playing competition contain definitions of $\text{base}$ and $\text{input}$ predicates as used above. However, there is no formal definition of the semantics of those predicates in GDL and many older game descriptions do not have those predicates. We argue for the following definition:

**Definition 5.** A game description $G$ is said to have well defined base and input definitions if, and only if,

- for every reachable state $S$ of $G$, for every $f \in S$, $G \vdash \text{base}(f)$; and
- for every reachable state $S$ of $G$ with $S \notin T$, role $r \in R$ and move $m \in \Sigma$, if $l(r, m, S)$ then $G \vdash \text{input}(r, m)$.

Instead of putting the burden of writing well defined base and input definitions, we propose to generate them from the remaining rules. The idea for this is that the possible instances of $\text{base}(f)$ should comprise all possible instances of $\text{init}(f)$ and all possible instances $\text{next}(f)$ for all possible state transitions. Similarly, the possible instances of $\text{input}(r, m)$ must contain all instances of $\text{legal}(r, m)$ in any reachable non-terminal state.

The idea is to compute a static version $P^{base}$ of the rules that define next states and legal moves of the players. Here, static means a relaxation of the rules that is independent of true and does, defined as follows.

**Definition 6.** Let $G$ be a GDL specification. We call a predicate $p$ in $G$ static iff $p \notin \{ \text{init, next, legal, does} \}$ and $p$ does neither depend on true nor does in the dependency graph of $G$.

Furthermore, let $p^{static}$ be a predicate symbol which represents a unique name for the static version of predicate $p$. By definition

\[
\text{init}^{static} = \text{base}, \quad \text{true}^{static} = \text{base}, \quad \text{next}^{static} = \text{base}, \quad \text{does}^{static} = \text{input}, \quad \text{legal}^{static} = \text{input}, \quad p^{static} = p, \text{if } p \text{ is static}
\]

For each rule $p(X)$:

\[
\begin{align*}
\{ q^{static}(\vec{Y}) & : q(\vec{Y}) \in B \} \cup \\
\{ \neg q^{static}(\vec{Y}) & : \neg q(\vec{Y}) \in B \wedge \text{is static} \}
\end{align*}
\]

where $B^{static}$ comprises the following literals:

\[
\{ q^{static}(\vec{Y}) : q(\vec{Y}) \in B \} \cup \\
\{ \neg q^{static}(\vec{Y}) : \neg q(\vec{Y}) \in B \wedge \text{is static} \}
\]

As an example, the following rules form $P^{base}$ as generated for the Tictactoe game (Figure 1):

\[
\begin{align*}
1 & \text{base}(\text{cell}(1, 1, \text{b})). \\
2 & \ldots \\
3 & \text{base}(\text{cell}(3, 3, \text{b})). \\
4 & \text{base}(\text{control}(\text{xplayer})). \\
5 & \text{base}(\text{cell}(M, N, \text{x})): - \\
6 & \text{input}(\text{xplayer}, \text{mark}(M, N)), \\
7 & \text{base}(\text{cell}(M, N, \text{b})). \\
8 & \text{base}(\text{cell}(M, N, \text{o})): - \\
9 & \text{input}(\text{oplayer}, \text{mark}(M, N)), \\
10 & \text{base}(\text{cell}(M, N, \text{b})). \\
11 & \text{base}(\text{cell}(M, N, \text{c})): - \\
12 & \text{base}(\text{cell}(M, N, \text{c})), \text{c}! = \text{b}. \\
13 & \text{base}(\text{control}(\text{xplayer})): - \\
14 & \text{base}(\text{control}(\text{oplayer})). \\
15 & \text{base}(\text{control}(\text{oplayer})): - \\
16 & \text{base}(\text{control}(\text{xplayer})). \\
17 & \text{base}(\text{cell}(M, N, \text{b})): - \\
18 & \text{base}(\text{cell}(M, N, \text{b})), \\
19 & \text{input}(R, \text{mark}(X, Y)), \text{m}! = \text{X}.
\end{align*}
\]

GIGA’16 Proceedings 17
The answer set program $P$ that we generate from a game description $G$ is defined as $P = G \cup P^{base} \cup$

1. $(true(F)) : base(F)$.
2. $l = \{ does(R, M) : input(R, M) \} :- role(R)$.

As can easily be seen, our definition of $P^{base}$ (and thus $P$) fulfills the restrictions for a valid GDL description (Definition 1) if the original game description $G$ belongs to the bounded fragment of GDL (Definition 2). The reason is that in $P^{base}$, we introduce recursions involving true$^{static}$ and next$^{static}$ (which are both base) and therefor also between legal$^{static}$ and does$^{static}$ (both input). Thus, all bounded GDL programs can be grounded using this method in principle. GDL descriptions not fulfilling the restrictions for bounded GDL, can lead to an infinite ground representation.

That said, grounding bounded GDL can still lead to an exponential blowup in the size of the representation which can make grounding infeasible. Especially games containing rules with many variables suffer from this problem.

Optimizations Before grounding the answer set program $P$, we apply optimizations to it similar to the ones described in [Haufe et al., 2012] (Section 6.2). That is, we try to reduce the resulting grounding by removing existential variables and removing unnecessary rules, as illustrated in the following paragraphs.

As an example, consider the rule $p(X, Z) :- q(X, Y), r(Y), s(Z)$. The variable $Y$ in the body is existentially quantified (does not appear in the head). We replace this rule by

1. $p(X, Z) :- q(X, Y), s(Z)$.
2. $qr(X) :- q(X, Y), r(Y)$.

where $qr$ is a new predicate symbol and obtain two rules with two variables each instead of one rule with three variables. This reduces the number of ground rules that are generated (unless the domains of the variables are singletons).

Some rules in the game descriptions are unnecessary and can be removed. For example, Tic-Tac-Toe (see Figure 1) contains the rules for line$(X)$. In those rules $X$ can be replaced with any of $\{x, o, b\}$, however only line$(x)$ and line$(o)$ appear in the body of another rule. Thus, the ground rules that would be generated for line$(b)$ are irrelevant as are the ground instances of row, column and diagonal where $X$ is replaced with $b$. We prevent these unnecessary rules from being generated in the first place, by instantiating the $X$ in the rules for line$(X)$ with $x$ and $o$ and handing these partially instantiated rules to the grounder.

6 Experiments
We ran experiments on 231 games from the GGP server [Schiffel, 2016]. For each game we ran our grounder and recorded

- the time it took to generate the answer set program $P$;
- the total runtime for grounding (including the time for generating $P$);
- the size of the resulting ground description in terms of number of resulting clauses;
- the number of components of a propositional network created from the those clauses without optimizations.

For comparison, we used the GGP-Base framework [Schreiber and Landau, 2016] to generate a propositional network (prophenet) using the OptimizingPropNetFactory class. Generating a prophenet includes grounding the game description as a first step and we measured only the time for this step without the remaining time that is spent on optimizing the propositional network. However, these two are somewhat intertwined such that a complete separation is not possible. GGP-Base makes use of base and input predicates. Since most games on the GGP server do not contain base and input definitions, we added the base and input definitions that were generated by our own grounder to the game rules that were given as input to the prophenet generator. For GGP-Base we recorded the runtime and the number of resulting components, where each component represents a conjunction, disjunction, negation or proposition in the (grounded) rules. Thus, this number is roughly comparable to the number of clauses (including facts) in the grounded game description.

The ASP-based grounder can ground 226 of the 231 tested games within the time limit of 1h and memory limit of 4GB. The median runtime was 1.4s (average 4.5s), which includes the time for starting an external process for the grounder and reading the resulting grounded game description. For comparison, the GGP-Base grounder can ground 218 of the tested games with a median runtime of 2.4s (average 5.9s). Since no external process needs to be started, this runtime does not include any process communication overhead. There was no game that could be grounded using GGP-Base but not using ASP. Most of the games that could be handled by the ASP-based grounder but not by GGP-Base feature heavy use of recursive rules. Neither system could ground laikLee_hex, merrills, mummymaze1p, ruledepthquadratic or small_dominion. All of those games feature recursive rules, except for mummymaze1p, which could be grounded by the ASP system resulting in about 5 million clauses, but processing the ground clauses and generating the prophenet took too much time. The games farmers, god, Goldrush, kalaha_2009, quad_5x5, SC_TestOnly, sudoku_simple and uf20-01.cnf.SAT could be grounded by the ASP-based grounder, while the GGP-Base grounder exceeds the run-time limit. Of those games, only farmers and kalaha_2009 can be considered complex games taking 25.4s and 21.7s to ground respectively and resulting in more than 100000 components. The other games could all be grounded by the ASP-based grounder in under 5s resulting in no more than 13000 components.
Figure 2 shows the runtimes of the GGP-Base vs. ASP-based grounder for all 231 games on a logarithmic scale. We can see that most games can be grounded in less than 1 minute with both grounders. In 173 cases the ASP-based grounder is faster than the GGP-Base grounder (some of them within the margin of error). On average, the ASP-based grounder is 20.4% faster than the GGP-Base grounder.

We compared the size of the grounded description with both systems in Figure 3. As can be seen, there is little difference between both systems, but the GGP-Base system creates significantly larger groundings in few selected games (smallest, logistics, mastermind, crossers3, othello-comp2007, othello-suicide, racer, racer4, battlebrushes). The number of clauses of the grounded game descriptions range from 26 (from troublemaker01) to 2038583 (for battlesnakes1509) with a median of 2444. The number of components in the generated propositional networks is similar (between 46 and 2715284). The median number of generated components is 2455 for the ASP-based grounder vs. 2518 for the GGP-Base grounder.

In Figure 4, we plotted the size of the ground representation (propositional network) compared to the size of the original GDL rules for each of the games. We used the smaller of the two groundings for each game and measured the size of the rules by taking the sum of the number of literals of all rules. As can be seen in the graph, the propositional network is generally some orders of magnitude larger than the GDL rules, except for some edge cases where the rules are essentially already grounded. However, we can not see a general trend indicating an exponential blowup in size. That is, although this blowup is theoretically possible, it seems to happen rarely in the games we looked at. In fact, the best fit to the data (excluding the abnormal cases) seems to be a power law which puts the propnet size at slightly more than a quadratic function of the size of the GDL rules.

7 Conclusion

Grounding game descriptions using a state-of-the-art answer set programming system is a viable alternative to the GDL specific approach implemented in the GGP-Base framework. The system we presented is able to handle more games and is typically faster despite the overhead of transforming GDL.
Figure 3: Number of components of a propositional network created from the grounding resulting from GGP-Base vs. our ASP-based grounder. Points above the diagonal denote games where the GGP-Base grounder creates larger propnets. The horizontal and vertical lines show the median numbers of components for GGP-Base and ASP-based grounder, respectively.

into a different format and starting and communicating with a separate process. Furthermore, our grounding of a game description is well-founded theoretically by the transformation into answer set programs. This allows to optimize the descriptions further without changing their semantics. In the future, we plan to look into further optimizations of the grounding to allow grounding of more complex game descriptions. Additionally, these optimizations will likely reduce the size of the grounded descriptions which generally leads to faster reasoning with the grounded game descriptions, for example, in the form of propositional networks. However, even with those optimization there will likely be games where the potential exponential blowup will prevent grounding from being feasible. In those cases it is necessary to fall back on reasoners that do not require a propositional representation (e.g., Prolog).

References


Figure 4: Number of components of the propositional network compared to the size of the original GDL description (without base and input rules) measured in number of literals for all 226 games that could be grounded. The dots denote games that are abnormal in that the GDL rules are larger than the propnet. All of these games turned out to be either test case games made to test certain aspects of GDL reasoners (as opposed to general game players) or games that are essentially already ground. The line shows the best matching regression.


A General Approach of Game Description Decomposition for General Game Playing

Aline Hufschmitt and Jean-Noël Vittaut and Jean Méhat
LIASD - University of Paris 8, France {alinehuf.jm,jnv}@ai.univ-paris8.fr

Abstract
We present a general approach for the decomposition of games described in the Game Description Language (GDL). In the field of General Game Playing, the exploration of games described in GDL can be significantly sped up by the decomposition of the problem in sub-problems analyzed separately. Our program can decompose game descriptions with any number of players while addressing the problem of joint moves. This approach is used to identify perfectly separable sub-games but can also decompose serial games composed of two subgames and games with compound moves while avoiding, unlike previous works, to rely on syntactic elements that can be eliminated by simply rewriting the GDL rules. We tested our program on 40 games, compound or not, and we can decompose 32 of them successfully in less than 5 seconds.

1 Introduction
Despite incentives from [Genesereth and Björnsson, 2013] to encourage the development of GGP players able to discern structure of compound games and therefore to dramatically decrease search cost, very little research exists in this area.

Cox et al., 2009] prove conditions under which a global game represents multiple, simultaneous independent subgames, but the practical implementation of a GGP player using decomposition presents two major issues: the first is to detect and decompose a compound game, the second is to combine local subgame solutions into a global one.

[Cerexhe et al., 2014] provide a systematic approach for single player games to solve this second difficulty which they refer to as the composition problem. However, identifying and decomposing game is not within the scope of their paper.

[Güntther, 2007; Güntther et al., 2009] propose a decomposition approach for single player games by building a dependency graph between fluents and actions: the connected parts of the graph represent the different subgames. Potential preconditions, positive and negative effects between fluents and actions are used to build this dependency graph while action-independent fluents are isolated in a separate subgame to prevent them from blocking the decomposition.

[Zhao et al., 2009; Zhao, 2009] propose a similar approach for multiplayer games using partially instantiated fluent and action terms. Serial games and games with compound actions are handled separately.

These approaches present different shortcoming we will details below such as an heavy reliance on certain syntactic structures in game descriptions.

We propose a more general approach to decompose games with any number of players while addressing the problem of joint moves, compound moves and serial games without relying on syntactic elements that can be eliminated by simply rewriting the GDL rules. The result of our decomposition can be used to solve the game by an approach like the one of [Cerexhe et al., 2014] ; it is a non-trivial problem outside the scope of this paper.

We begin (§2) with a brief introduction of the Game Description Language and the different types of compound games that can be found on the different online servers and that our approach can decompose. Then we present the different aspects of our method to handle these different types of games (§3). We present results on 40 games, compound or not (§4). Finally, we conclude and present future work (§5).

2 Preliminaries
We present here some details about the Game Description Language and the different types of compound games that our approach can decompose.

2.1 The Game Description Language
We assume familiarity of the reader with the General Game Playing [Genesereth et al., 2005] as well as with the Game Description Language (GDL) [Love et al., 2008]. A GDL game description takes the form of a set of assertions and of logical rules which conclusion describes: the transition to the next position (next predicate); the legality of actions (legal); the game termination (terminal); and the score (goal). The rules are expressed in terms of actions (does) and fluents (true) describing the game state.

Rule premises can also include auxiliary predicates, specific to the game description itself, which truth is defined by rules also using true and does premises. In the rest of this article, we will refer to auxiliary predicates, exclusively defined in terms of fluents (true) (does never appear in their
premises), which have an important role in our decomposition approach (§3.3, §3.5).

2.2 Types of compound games

Among games available on the different General Game Playing servers (http://games.ggp.org), different types of compound games can be identified. The types we distinguish represent specific issues for the decomposition and are not directly related to the formal classification proposed by [Cerexhe et al., 2014].

For example, Parallel games like Dual Connect 4 or Double Tictactoe Dengji are composed of two subgames played in parallel that can be synchronous or asynchronous, but this difference has no influence on the decomposition approach to use. Decomposing these games do not present any particular difficulty.

However, in some synchronous parallel games like Asteroids Parallel each player’s action is a compound moves corresponding to two simultaneous actions played in each subgame. These create a strong connection between subgames and represent a specific difficulty for decomposition.

Serial games like Blocker Serial are composed of two sequential subgames i.e. the second starts when the first is completed. As the two games are linked together, identifying the boundary between them is a specific issue for decomposition.

Multiple games like Multiple Buttons And Lights are composed of several subgames, only one of them being involved in the score calculation or the game termination. The other subgames only increase the size of the game tree to explore. Identifying those useless subgames allows to avoid unnecessary calculations. Note that in the game Incredible, contemplate actions are detected as noop actions by our decomposition program and does not constitute a useless subgame.

Games using a stepper to ensure finite games like Eight Puzzle may be considered as compound games (synchronous). In these games, different descriptions of a position can vary only by the value of the stepper (step counter). To allow a programmed player to exploit these near-perfect transpositions, it is necessary to operate a game decomposition to separate the stepper from the game itself. This stepper is then an action independent subgame.

Some impartial games, like Nim starting with several piles of objects, may also be considered as compounds games (asynchronous) as they can be decomposed in several subgames, one for each pile, each of them being an impartial game [Zhao, 2009]. Identifying that these subgames are impartial, subsequently allows to use known techniques for the resolution of the global game.

3 Method

Our approach is based on the [Günther, 2007] idea and consists in using a dependency graph between actions and fluents, and then to identify the connected parts of the graph representing the subgames. As nothing in the GDL specification prohibits the use of completely instantiated rules or prevents that fluents or actions be reduced to simple atoms, we identify relations between totally instantiated fluents $f$ and actions $a$ and rely neither on their predicates names nor their arguments.

For the analysis of these relations, we use the following definitions:

**Definition 1** Let $F$ be the set of all the instantiated fluents $f$ appearing in $\text{true}(f)$ or $\neg\text{true}(f)$.

**Definition 2** $R$ being the set of all the roles $r$ and $O$ the set of all options $o$ of these roles, let $A = R \times O$ be the set of all the instantiated player actions $a = (r, o)$.

**Definition 3** Let $C$ be the set of all the possible conjunctions of atoms of the form $\text{true}(f)$, $\neg\text{true}(f)$, $\text{does}(r, o)$ or $\neg\text{does}(r, o)$.

3.1 Grounding and creation of a logic circuit

To instantiate completely the rules (grounding), we carry out a fast instantiation using Prolog with tabling [Vittaut and Méhat, 2014] and use these instantiated rules to build a logic circuit similar to a prover [Schkufza et al., 2008]. Conclusions of legal, next, goal or terminal rules are the outputs of the circuit and only depends on fluents ($\text{true}$) and actions ($\text{does}$) at the inputs.

It is possible, according to the GDL specifications, to produce a description with fully developed rules using no auxiliary predicate at all. However, these predicates, like column1, diagonal2 or game1over in Tictactoe, may be necessary for some specific stages of our process of decomposition (§3.3, §3.5). To ensure that these auxiliary predicates will be available even when not specified in the GDL description, we proceed to a factorization of the conjunctions, disjunctions and use De Morgans laws to reduce the number of negations in the circuit. As a perfect factorization is an NP-hard problem, our program uses a greedy approach where the first common factor is used. Factorization and application of De Morgans laws are iterated until the circuit reaches a minimum size.

We identify the needed auxiliary predicates as these are represented by internal logic gates of the circuit, depending only on input fluents and representing important expressions in the logic of the game i.e. these expressions are used several times, several logic gates use their outputs.

After the factorization, the GDL description is a set of formulas under disjunctive normal form of which atoms are fluents, actions, and auxiliary predicates. In the following we say that these formulas are under DNF form.

Other stages of the decomposition process need a description of the game under canonical form. By recursively replacing auxiliary predicates by their expression we obtain a new set of formulas in disjunctive normal form describing the same game where all the auxiliary predicates have been eliminated. In the following we say that these formulas are under DNFD form.

3.2 Building a dependency graph

To build our dependency graph and to identify the different subgames, we start with a set of vertices which are the fully instantiated actions and fluents. We then identify different relations between these fluents and actions that we define below. For each of these relations we add an edge between the involved actions and fluents vertices. These relations correspond to preconditions or effects of the actions.
Unfortunately, GDL does not explicitly describe action effects unlike STRIPS or PDDL languages used for planning domains. A fluent being false by default, an action present in a next rule can have an effect or not. For example, let us consider the legal actions does(r, a), does(r, b) and does(r, c), in the rule next(f) := ¬true(f) ∧ (does(r, a) ∨ does(r, b)), a and b have an effect if the rule means “The cell will contain a pawn if r does one of the 2 actions moving a pawn in it” and c has an effect if it means “the boat will sink if r does anything else than action c (hailing)”. A similar example can be found for any next rule with an action (in a negation or not) and regardless of the value of the fluent f and its presence or not in the rule premises.

It is thus possible to produce GDL descriptions in which the actions present in a next rule body belong to another subgame than the fluent in the rule head. We can only address this using heuristics similar to those of [Günther et al., 2009].

They propose to consider that an action a has a negative effect on a fluent f if this action does not keep the fluent true i.e. if next(f) does not contain true(f) ∧ does(a) in its premises. However in a game like Double Tictactoe, there is no rule like this to indicates that actions of a subgame does not change the value of the other subgame fluents. Consequently, fluents of a subgame can be considered as negative effects of the second subgame actions and the decomposition fails.

In our approach we use slightly different heuristics which work well for existing composed games to find potential effects of actions:

**Definition 4** The fluent f is a potential negative effect of the action a = (r, o) if next(f) under DNFD has a clause where ¬does(r, o) appears.

The fluent f is a potential positive effect of the action a = (r, o) if next(f) under DNFD has a clause containing the does(r, o) literal and not containing the true(f) literal.

In case of joint moves from several players, it is necessary to identify if the action of each player is responsible of the observed effect on the rule conclusion to avoid linking unrelated action with the conclusion.

To solve this problem [Zhao et al., 2009] propose to compare the arguments used in a next rule head with the ones used in the moves (does). For example, in the following rule from Blocker Serial, we can see that the action from crosser is the only one that is likely to affect the conclusion:

next(cell2(XC, YC, crosser)) :=
distinctcell(XC, YC, XB, YB)
∧ does(crosser, mark2(XC, YC))
∧ does(blocker, mark2(XB, YB)).

However, GDL specification allows to use completely instantiated rules and simple atoms to represent fluents and moves. For example, we can replace the previous rule by some instantiated rules:

next(f) := does(crosser, o1) ∧ does(blocker, o2).

next(f) := does(crosser, o1) ∧ does(blocker, o3).

... With fluents like f and moves like does(r, o), their approach is no longer able to deal with joint moves.

To identify which action has an effect without relying on syntactic elements, we compare, for each player, the different actions used in conjunction with the same fluents and actions of other players in the clauses of each next rule.

Suppose that next(f) ← C_f is in DNFD. Let us consider a specific option o’ for player r’. We consider the set E(o’) of the different options of the role r when r’ choose the o’ option:

\[ E(o') = \{ o \in O_r \mid \exists c \in C_f, \exists b \in C, c = does(r, o) \land does(r', o') \land b \} \]

We define E(o) the same way by exchanging the role of (r, o) with (r’, o’).

If all the options of the r are present in conjunction with the same action of r’; these options have probably no effect i.e. the result is the same regardless of the option chosen. On the contrary, if a single option of r is present, it is probably responsible for the observed effect. We then use the following heuristics:

**Definition 5** The action a = (r, o) ∈ A is potentially responsible for an effect on f if:

- \( \text{card}(E(o')) = 1 \), or
- \( E(o') \subseteq O, \text{ and } \text{card}(E(o)) \neq 1 \)

For example, in the game BlockerSerial, the term next(cell1(2, 3, crosser)) is true if blocker choose any option but mark1(2, 3) and crosser choose the mark1(2, 3) option. All the options of blocker are not represented but, as crosser has a single possible option, its action is considered responsible for the effect while actions of blocker are not linked to the cell1(2, 3, crosser) fluent.

Even if this approach sometimes put aside actions related to the conclusion, we did not observe any over-decomposition. At least one of the actions is indeed related to the conclusion and edges between fluents and actions added in the dependency graph to represent preconditions relations are redundant with those added for effect relations.

Therefore a fluent is a potential effect of an action if this action has a potential positive or negative effect on this fluent and if this action is potentially responsible for this effect in presence of joint moves. From the potential effect of actions we can deduce fluents that are action-independent, such as step or control fluents, and actions that are fluent-independent such as noop actions:

**Definition 6** A fluent f is action-independent if it is not the potential effect of any action a.

An action a is fluent-independent if no fluent f is the potential effect of this action.

Then we can identify fluents that are potential preconditions of an action in the same subgame and create a link in the graph between them:

**Definition 7** The fluent f is a potential precondition in the same subgame of the action a = (r, o) if:

- a is not fluent-independent, and
- f is not action-independent, and
- one of the two following conditions holds:
  - legal(r, o) under DNFD has a clause where true(f) or ¬true(f) appears, or
– it exist \( f' \) which is a potential effect of \( a \), such that \( \text{next}(f') \) under DNFD has a clause containing \( \text{does}(r, o) \land \text{true}(f) \) or \( \text{does}(r, o) \land \lnot \text{true}(f) \).

An action-independent fluent can be present in the premises of all legal rules, it is then a precondition of all actions but belongs to another subgame which is action-independent.

### 3.3 Subgoal-predicates to fix over-decomposition

Edges between actions and fluent vertices corresponding to preconditions or effects of these actions may not be sufficient to connect all the elements of a subgame. For instance, in a subgame like Tictactoe, an action has an effect on a cell and the state of this cell is a precondition to this action. However, no link exists through actions between fluents describing different cells.

In the game Double Tictactoe given as an example by [Zhao et al., 2009] the auxiliary predicates line1/1 or line2/1 are present in the premises of some legal rules. All the fluents in the premises of these predicates are then preconditions of the corresponding actions and create a link between the cells of each subgame. However, in games like Tictactoe Parallel, Connect4 or Rainbow no such predicate is present in the legal rules and an over-decomposition occurs.

The logic link between elements of a subgame is in the goal to reach and this goal is usually a condition for the termination of the global game. We need to distinguish an auxiliary predicate corresponding to a subgoal in one subgame from one corresponding to different subgoals from different subgames because the second one can prevent the decomposition. To address this problem of over-decomposition we use the following heuristic to identify potential subgoal-predicates corresponding to only one subgame:

**Definition 8** Let \( g \) be the maximum possible score of \( r \). An auxiliary predicate \( b \) is a potential subgoal-predicate if:

- terminal depends on the logical value of \( b \) and 
- \( \text{goal}(r, g) \) under DNFD has a clause where \( b \) appears.

or

- All the roles play in different subgames, and
- \( \text{goal}(r, g) \) under DNFD has a clause where \( b \) appears, and for all roles \( r' \neq r \), \( \text{goal}(r', g') \) under DNFD has no clause where \( b \) appears.

In games like Dual Rainbow or Dual Hamilton, subgoal-predicates appear only in the premises of goal rules. Since these games are composed of single player subgames, an auxiliary predicate present in the goal rule of a single player involves only this player and therefore only one subgame.

The first part of the definition holds in the games where the victory in one of the subgames terminates the game as it is generally the case in compound games. Otherwise, the subgames may be connected by the use of a misidentified subgoal-predicate.

Once a subgoal-predicate is identified, we add edges in our dependency graph between fluents that appear in a same clause in its formulas under DNFD.

### 3.4 Compound moves and meta-action sets

A compound move is composed of two or more actions related to different subgames. For example, in the game Asteroid Parallel the compound move legal(ship, do(clockcounter)) corresponds to a clockwise move in a first subgame and a counterclockwise move in a second subgame. Such an action creates a link between the different subgames and can interfere with the decomposition process.

To detect compound moves, [Zhao et al., 2009] use the same approach as that applied to the problem of joints move. For example, in the following rule from Tictactoe Parallel we can see that only the first two arguments of the action have an effect on the rule conclusion:

\[
\text{next}((cell(X1, Y1, o)) \leftarrow \text{do}(@\text{oplayer}, \text{mark}(X1, Y1, X2, Y2))).
\]

Once again, the rule has just to be rewritten to defeat detection: \( \text{next}(f) \leftarrow \text{do}(@\text{oplayer}, o) \).

In games with compound moves, the set of all actions is a combination of the sets of all actions of each subgame. Then in a game composed of two subgames, for each action in the first subgame, there is \( N \) compound moves corresponding to this action combined to the \( N \) possible actions in the second subgame. To identify the different parts of compound moves, we distribute actions into meta-action sets. An action can belong to one or several meta-action sets which depend only on a role \( r \), a fluent \( f \in F \) and two clauses \( c \in C \) and \( c' \in C \).

**Definition 9** An action \( a = (r, o) \) belongs to the meta-action set \( P(r, f, c, c') \) if:

- \( f \) is a potential effect of \( a \), and
- \( \text{next}(f) \) under DNFD has a clause \( (\text{does}(r, o) \land c) \), and
- if \( c' \) is empty, legal\((r, o)\) must always be true, or if \( c' \) is not empty, it contains only action-dependent literals and appears in at least one clause of legal\((r, o)\) under DNFD.

Therefore a meta-action set is a group of actions with an identical effect on a fluent of a particular subgame, the same preconditions in the corresponding next rule and at least one precondition in common in their legal rules.

For example, in the game Blocks World Parallel we can find the meta-action set \{do(robot, do(stack, stack, a, b, +, *)), do(robot, do(stack, stack, a, b, +, *))\} corresponding to the action stack\((a, b)\) in the first subgame. These actions have an effect in common on true\((onl(a, b))\), same preconditions \{true\((table1(a))\), true\((clear1(b))\), true\((clear1(a))\)\} in the next\((onl(a, b))\) clauses and are always legal.

In a game with compound actions, each action is placed in \( M \) meta-action sets corresponding to \( M \) effects. If a game contains no compound action but some actions with an identical effect in the same situation, these actions are grouped in the same meta-action set. And finally, if all actions in a game have a different effect, each one constitutes a meta-action singleton. The use of meta-action sets is then compatible with all games.

---

\(^1\)the * represents different possible values, the whole meta-action set contains 12 compound moves
In our dependency graph, we then encapsulate all actions into meta-action sets to avoid compound actions from connecting different subgames. The links between actions and fluents are replaced by links between action sets and fluents i.e. in the dependency graph, edges are added between a meta-action set and its effect $f'$ and preconditions $f \in c \cup c'$

3.5 Serial games

In serial games an auxiliary predicate describing the terminal situation of the first subgame determines the legality of all actions of the second subgame. Consequently, it creates links between first subgame fluents and second subgame actions. We must detect it and avoid these links to separate both subgames.

[Zhao, 2009] uses a separate special detection: the desired auxiliary predicate must be false to authorize the first subgame actions and true to authorize the second ones, like gameover in Tictactoe Serial:

$$\text{legal}(\text{PLAYER, mark}(X, Y)) \iff \neg \text{gameover} \land \ldots$$

$$\text{legal}(\text{PLAYER, mark2}(X, Y)) \iff \neg \text{gameover} \land \ldots$$

with gameover depending on line1($x$) v line1($o$) v open1. However, someone can defeat this approach by simply rewriting the first subgame legal rules with a different precondition: $\text{legal}(\text{PLAYER, mark}(X, Y)) \iff \text{ongoing1} \land \ldots$ with ongoing1 depending on $\neg \text{line1}(x) \land \neg \text{line1}(o) \land \text{open1}$.

To generalize the approach of [Zhao, 2009], we consider that a pivot between two serial subgames is composed of two auxiliary predicates that can be the negation of each other or two completely different predicates. We use our circuit representing the game to test the influence of each auxiliary predicate detected during the circuit creation on the actions legality and look for a couple of predicates that parts the fluent-dependent actions in two groups.

If such a couple of auxiliary predicates is found, then it is a pivot and the latter predicates are directly used as action preconditions instead of the fluents included in them. In our dependency graph, fluents of the first subgame are then encapsulated in these auxiliary predicates to ensure that they will not connect the different subgames with direct links to actions (meta-action sets) of the second subgame. This approach works for existing games that are limited to two serial subgames.

Unfortunately, we cannot generalize this approach and identify a pivot in case of more than two serial subgames without risking an over-decomposition of games with movable parts. In a pivot, each auxiliary predicate is necessary to allow the legality of some actions and may prevent the legality of other actions. If a third subgame is present, its actions are not affected by both auxiliary predicates. In a game with movable pawns, an auxiliary predicate may be used to describe the state of a cell; this predicate may also prevent some moves from this cell, prevent some moves to this cell and does not concern other moves of the game, consequently it may be confused with a part of a pivot. Therefore, if we try to identify pivots for more than two serial subgames with a generalization of this approach, a game with movable pawn may be over-decomposed, each cell being a small serial subgame leading to the next ones.

3.6 Multiple games and useless subgames

Some subgames are involved in the calculation of the score or can cause the end of the game when some position is reached. A subgame may also be played to allow another subgame to start in the case of serial subgames.

Definition 10 Let $V_S$ be the set of vertices of a connected part of the dependency graph representing a subgame $S$. $S$ is considered useful if:

- $S$ is played before another subgame in a serial game and is necessary to start it, or
- it exists $f \in F \cap V_S$ such that terminal depends on the logical value of $true(f)$, or
- it exists $f \in F \cap V_S$ such that goal$(r,g)$ depends on the logical value of $true(f)$.

In multiple games, all the subgames that are not identified as useful can be ignored and remain unexplored. However, a useless action (noop) can be sometime strategically useful to avoid a zugzwang in another subgame. Actions of these subgames can then be flagged as noop actions, be considered equivalently useless, and only one of them need to be explored (if legal) for each position of the game.

4 Experiments

We evaluated our decomposition program on a panel of 40 descriptions of games, compound or not, from the servers of Dresden, Stanford and Tiltyard. We took all the available compound games except for the redundant ones. We added the original version of games commonly used as subgames and a representative panel of games with different characteristics (movable parts, steppers, asymmetry, impartiality) and complexity. The experiments were run on one core of an Intel Core i7 2.7GHz with 8G of 1600MHz DDR3.

For each game, we measured the mean time necessary for each stage of the decomposition on a set of 100 decomposition tests. To limit the duration of the experiments, a decomposition test was aborted after 60 minutes. The longest stages of the decomposition are grounding the rules, factorizing the circuit and calculating completely developed disjunctive normal forms (DNFD). The column 5 of table 1 indicates the total time needed to decompose each game and shows that the DNFD calculation can be very time consuming.

We try to compute DNF without developing the auxiliary predicates identified during the circuit construction. As we can see it in column 6, the time saved is really significant and allows the successful decomposition of 32 games among 40 in less than 5 seconds. The major part of the total time necessary for the decomposition using DNF corresponds to the rules grounding and circuit factorization.

Unfortunately, the use of partially developed DNF presents a shortcoming: if a rule containing variables is already instantiated in the original GDL description of a game and if some of these instances only are expressed in terms of auxiliary predicates, actions may occur in conjunction with different but equivalent premises: a group of fluents or an equivalent auxiliary predicate. The factorization of the circuit should restore auxiliary predicates in all rules instances but as we use a
Games at the top of the table are composed of only one action-dependent subgame and sometimes a stepper detected as a useful action-independent subgame. The useless action-independent subgame detected for games like Breakthrough or Sheephound and Wolf corresponds to the control fluents which indicate the active player in an alternate moves game and does not represent a playable game per se.

Useless subgames in multiple games are correctly identified. We remark that for *Multiple Tictactoe*, the number of useless subgames is particularly large because these subgames have been over-decomposed as no auxiliary predicate creates a link between their cells.

For the game of *Nim*, our program has detected an action-independent subgame not involved in the end of the game (it is not a stepper) while it is the only subgame useful for the calculation of the score: this is an important clue indicating that this game is impartial.

Except for the special case of *Chomp*, all the detected subgames are the expected ones and correspond to what would have been obtained by a manual decomposition. *Chomp* is a challenge on games on which the heuristics used for the action effects detection do not work properly. Other actions than eating the poisoned chocolate square have only implicit negative effects which are not detected. These actions are considered as *noop* actions and would be evaluated as equivalent during the game: this could not allow the player to prevent the fatal outcome. Fortunately, such a wrong detection of the action effects is visible in the resulting dependency graph as a huge proportion of fluents and actions are isolated vertices. So we can prevent this error from affecting the game solving.

### 5 Conclusion and future work

In this paper we presented a general approach for the decomposition of games described in the *Game Description Language (GDL)*. Our program decomposes descriptions of games, compound or not, with any number of players while addressing the problem of joint moves. It decomposes parallel games, games with compound moves and serial games composed of two subgames. It also identifies steppers, useless subgames in multiple games, and unlike previous works, without relying on syntactic elements that can be eliminated by simply rewriting GDL rules. We tested our program on 40 games, compound or not, and have decomposed 32 of them with success in less than 5 seconds which is a time compatible with GGP competition setups.

Using Meta-action sets is an efficient way to the problem raised by compound moves (§3.4). However, it requires the completely developed disjunctive normal form of the next rules which is computationally expensive. We are seeking another approach to avoid this need or to minimize it’s computation time. Beside this, we plan to eliminate the ad-hoc heuristics used to identify action effects (§3.2) and to avoid over-decomposition (§3.3). We will also address the problem of the decomposition of more than two sequential subgames.

Finally, using these decomposed games to solve the *composition problem* for any games with any number of players remains an open problem.

---

**Table 1:** Result of the decomposition for a panel of 40 games descriptions from the servers of Dresden (D), Stanford (S) and Tiltyard (T) with comments on subgames (SG) found.

The greedy approach (§3.1), it is not guaranteed. Therefore, meta-action sets detection may be hindered. Nevertheless, this case is sufficiently specific to successfully use the auxiliary predicates in DNF, in most cases.

For *Hex* and *Blocker Parallel*, the time required to compute the ground rules, the factorization and the DNFs still remains too large. The factorization does not allow to sufficiently reduce the complexity of *Hex* and, in *Blocker Parallel*, the presence of compound actions combined with joint moves for both players brings a large number of combinations.

Note that *LeJoueur* of Jean Noël Vittaut, which won the 2015 Tiltyard Open, is on average 8.5 times faster to ground for both players bringing a large number of combinations. This indicates the potential scope for improving these steps.

Table 1 also shows the total number of subgames discovered for each of the 40 games and among them, the ones that are action-dependent and action-independent. The figures in parenthesis indicate the number of discovered subgames considered as useless.
References


GDL-III: A Description Language for Epistemic General Game Playing

Michael Thielscher
University of New South Wales, Australia
mit@unsw.edu.au

Abstract

We present an extension of the standard game description language for general game playing to include epistemic games, which are characterised by rules that depend on the knowledge of players. A single additional keyword suffices to define GDL-III, a game description language with imperfect information and introspection. We define syntax and semantics and present an execution model for this extended language. We develop a provably correct answer set program (ASP) for reasoning about GDL-III games. We also show how the extended language can be used to formalise general epistemic puzzles and how those can be solved using a mere game controller for GDL-III, that is, without requiring any game-playing intelligence beyond the ability to generate legal play sequences.

1 Introduction

The game description language GDL has become the standard for describing the rules of games to general game-playing systems [Love et al., 2006; Genesereth and Thielscher, 2014]. Its syntax is a variation of logic programming with specific keywords for specifying the initial game states, legality and effects of moves, as well as termination and winning conditions. While descriptions of games with simultaneous moves are supported, the language assumes that players have complete knowledge of the game state after every round [Genesereth et al., 2005]. The extension GDL-II has been developed with the aim to include general imperfect information games [Schiffel and Thielscher, 2014]. The logical-epistemic foundations of this extended language have been analysed [Ruan and Thielscher, 2014] and general game-playing systems for imperfect information have been developed on the basis of GDL-II [Edelkamp et al., 2012; Schofield and Thielscher, 2015].

Although the extended description languages can be used to describe games with imperfect and incomplete information [Schiffel and Thielscher, 2014], it does not support the specification of games with epistemic goals [Ågotnes et al., 2013] or, more generally, with rules that depend on the epistemic state of players. This is so because while we can mathematically reason about knowledge of players in imperfect-information games, we cannot refer to knowledge within a GDL-II rule. As an example, consider the game NUMBER-GUESSING from the GDL-II track at the AI’12 general game playing competition, in which the goal for a single player is to repeatedly ask yes/no questions to determine an initially unknown number. However, the player can win merely by guessing correctly. GDL-II is not expressive enough to specify, as a necessary winning condition, that the player actually knows the number from the preceding percepts.

Another example of epistemic games are so-called Russian Cards Problems [Cordón-Franco et al., 2013; van Ditmarsch et al., 2006], in which the goal of two cooperating players is to inform each other about their hands through public announcements without a third player being able to learn anything from their communication. GDL-II cannot be used to express these goal rules because the language does not provide means to refer to the fact that a player knows that another player knows their cards, nor that it is common knowledge that a third player does not. Yet other examples beyond the expressiveness of GDL-II are games in which players are obliged, by the official rules, to always truthfully answer questions about what they know. This also requires the conditioning of the legality of a move on players’ epistemic states.

The purpose of this paper is to overcome these limitations by developing a formal language suitable for general game playing that supports the specification of game rules which need to refer to the epistemic states of the players. This is a useful addition to GDL-II whenever players’ knowledge is not present in the current state but follows implicitly from their past percepts. We will show that a single additional keyword suffices to define GDL-III, a general description language for games with imperfect information and introspection. The new keyword can be used to express both individual, possibly nested knowledge, e.g. that player A knows that her cards are known to player B, as well as common knowledge, e.g. that player C does not know of any card held by another player and everyone knows this (and also that everyone knows that everyone knows etc). We will formally define the syntax and the semantics of GDL-III as an extension of the existing language GDL-II. We will also develop a provably correct answer set program (ASP) for reasoning

\footnote{\text{\url{see ai2012.web.cse.unsw.edu.au/ggp.html}}}
about GDL-III games. This can either be used as the basis for a legal player that can handle the extended language, or as a game controller to administer the execution of arbitrary GDL-III games.

While the main purpose of our language extension is to allow for the description of epistemic games for the purpose of general game playing, we will furthermore demonstrate how GDL-III can be used to encode, and automatically solve, epistemic puzzles like Cheryl’s Birthday, which recently acquired public fame [Chang, 2015]. Notably, this does not even require an intelligent general game player; a mere game controller that is capable of computing legal playouts suffices to solve these puzzles.

The remainder of the paper is organised as follows. After providing the necessary background in Section 2, we will introduce the syntax of the extended language and its semantics in Section 3. The ASP-based controller for executing GDL-III games will be presented in Section 4. In Section 5, we will use the example of Cheryl’s Birthday to show how the extended language can also be used to formalise general epistemic puzzles that can then be automatically solved using the ASP-based implementation of a naive game controller. In Section 6, we will present experimental results to test the scalability of the ASP-based game controller and two of the examples considered in this paper, NUMBERGUESSING and CHERYLSBIRTHDAY. Finally, in Section 7 we compare and contrast GDL-III to other existing languages and conclude.

2 General Game Playing With GDL

The declarative Game Description Language (GDL) is a formal language for specifying the rules of strategy games to a general game-playing system [Genesereth et al., 2005]. It uses a prefix-variant of the syntax of logic programs along with the following special keywords.

\[
\begin{align*}
\langle \text{role R} \rangle & \quad \text{R is a player} \\
\langle \text{init F} \rangle & \quad \text{feature F holds in the initial position} \\
\langle \text{true F} \rangle & \quad \text{feature F holds in the current position} \\
\langle \text{legal R M} \rangle & \quad R \text{ has move } M \text{ in the current position} \\
\langle \text{does R M} \rangle & \quad R \text{ does move } M \\
\langle \text{next F} \rangle & \quad \text{feature F holds in the next position} \\
\langle \text{terminal} \rangle & \quad \text{the current position is terminal} \\
\langle \text{goal R V} \rangle & \quad \text{player R gets payoff } V \\
\langle \text{sees R P} \rangle & \quad \text{player R is told } P \text{ in the next position} \\
\langle \text{random} \rangle & \quad \text{the random player (aka. Nature)}
\end{align*}
\]

The bottom two keywords have been added in GDL-II to enable the specification of games with randomised moves and imperfect information [Schiffel and Thielscher, 2014].

Example 1 (NUMBERGUESSING) The rules in Figure 1 formalise a simple number guessing game. Lines 1–2 introduce the rules. Line 7 defines the initial game state. The moves are specified by the rules for legal: In the first round, a number between 1 and 32 is randomly chosen (rule 12–13). The player can then repeatedly ask yes/no questions (lines 15–16) or attempt a guess (lines 17–18). The guesser’s perceptions are truthful replies to his questions (rule 21–24), using a recursive axiomatisation of less (rule 19–20). The remaining rules specify the state update (next), the conditions for the game to end (terminal), and the payoff for the players (goal).

Syntactic and Semantics

In order to admit an unambiguous interpretation, so-called valid GDL-II game descriptions must obey certain general syntactic restrictions; for details we refer to [Love et al., 2006; Schiffel and Thielscher, 2014]. Under these restrictions, any valid GDL-II game description \( G \) determines a state transition system as follows.

To begin with, the derivable instances of \( \langle \text{role R} \rangle \) define the players, and the initial state consists in the derivable instances of \( \langle \text{init F} \rangle \). In order to determine the legal moves of a player in any given game state, this state has to be encoded first, using the keyword true: Let \( S = \{ f_1, \ldots, f_n \} \) be a state (more specifically, a finite set of ground terms over the signature of \( G \)), then \( G \) is extended by the \( n \) facts

\[
S_{\text{true}} \equiv \{ \text{(true } f_1 \text{) } \cdots \text{ (true } f_n \text{)} \} \quad (1)
\]

Those instances of \( \langle \text{legal R M} \rangle \) that follow from \( G \cup S_{\text{true}} \) define all legal moves \( M \) for player \( R \) in position \( S \).

In the same way, the clauses with terminal and \( \langle \text{goal R V} \rangle \) in the head define, respectively, termination and goal values relative to the encoding of a given position.

Determining a position update and the percepts of the players requires the encoding of both the current position and a joint move. Specifically, let \( M \) denote that players \( r_1, \ldots, r_k \) take moves \( m_1, \ldots, m_k \), then

\[
M_{\text{does}} \equiv \{ \text{(does } r_1 m_1 \text{) } \cdots \text{ (does } r_k m_k \text{)} \} \quad (2)
\]

All instances of \( \langle \text{next F} \rangle \) that follow from \( G \cup M_{\text{does}} \cup S_{\text{true}} \) compose the updated position; likewise, the derivable instances of \( \langle \text{sees R P} \rangle \) describe what a player perceives when the given joint move is done in the given position. All this is summarised below, where “\( |= \)” denotes entailment wrt. the unique stable model of a stratified set of clauses.

Definition 1 The semantics of a valid GDL-II game description \( G \) is given by

- \( R = \{ r: G \models (\text{role } r) \} \) (player names);
- \( s_0 = \{ f: G \models (\text{init } f) \} \) (initial state);
- \( t = \{ S: G \cup S_{\text{true}} \models (\text{terminal}) \} \) (terminal states);
- \( l = \{ (r, m, S) : G \cup S_{\text{true}} \models (\text{legal } r m) \} \) (legal moves);
- \( u(M, S) = \{ f : G \cup M_{\text{does}} \cup S_{\text{true}} \models (\text{next } f) \} \) (update);
- \( \mathcal{T} = \{ (r, M, S, p) : G \cup M_{\text{does}} \cup S_{\text{true}} \models (\text{sees } r p) \} \) (players’ percepts);
- \( g = \{ (r, v, S) : G \cup S_{\text{true}} \models (\text{goal } r v) \} \) (goal values).

GDL-II games are played using the following execution model [Schiffel and Thielscher, 2014].

\[2\]A rule for a percept no can of course be added but is not necessary, because not receiving a yes suffices as an answer.
1. All players receive the game description $G$.
2. Starting with $s_0$, in each state $S$ each player $r \in R$ selects a legal move from $\{m : I(r, m, S)\}$.
3. The update function (synchronously) applies the joint move $M$ to the current position, resulting in the new position $S' = u(M, S)$. Furthermore, each of the roles $r$ receives their individual percepts $\{p : I(r, M, S, p)\}$.
4. This continues until a terminal state is reached, and then the goal relation determines the result for all players.

Based on this protocol, legal play sequences are defined [Schiffel and Thielscher, 2014] as sequences $\mu_1, \ldots, \mu_n$ of joint moves $\mu_i$, that is a move $\mu_i(r)$ for each $r \in R$, which satisfy the following: There is a sequence of states $s_0, s_1, \ldots, s_n$ such that, for all $i \in \{1, \ldots, n\}$,

- $(r, \mu_i(r), s_{i-1}) \in I$ for all $r \in R$ (legality of moves);
- $s_i = u(\mu_i, s_{i-1})$ (position update).

Furthermore, $\{s_0, \ldots, s_{n-1}\} \cap t = \{\}$, that is, only the last state may be terminal. Two legal play sequences $\delta, \delta'$ of the same length $n$ are called indistinguishable for a player $r \in R$ if $r$'s moves and percepts are the same [Schiffel and Thielscher, 2014], that is, for all $i \in \{1, \ldots, n\}$:

- $\mu_i(r) = \mu'_i(r)$;
- $\{p : (r, \mu_i, s_{i-1}, r', p) \in I\} = \{p' : (r, \mu'_i, s'_{i-1}, r', p') \in I\}$.

These definitions based on the objective game rules about the percepts of players, as given by a GDL-II game description, assume that players have perfect recall [Schiffel and Thielscher, 2014].

3 GDL-II + Introspection

The general game description language with imperfect information is expressive enough to model games that give rise to complex epistemic models including players’ knowledge of the knowledge of other players [Ruan and Thielscher, 2014]. However, the language does not support any references to players’ knowledge in the game rules themselves, for example, in order to specify knowledge goals or to require players to be truthful when asked about their knowledge. In this section, we will extend the syntax of GDL-II by one additional pre-defined language element to facilitate such specifications.

3.1 GDL-III Syntax

We define the syntax of GDL-III as that of GDL-II augmented by a new keyword for introspection:

```
(knows R P) player R knows P in the current position
(knows P) P is common knowledge
```

The new keyword comes with the following additional, syntactical restrictions on valid GDL-III game descriptions $G$:

1. knows only occurs in the body of clauses, and neither role nor init depend on knows.
2. There is a total ordering $>$ on all predicate symbols $P$ that occur as argument of knows in $G$ such that $P > Q$ whenever $P$ itself depends on (knows R Q) or (knows Q) in $G$.
3. If $P$ occurs as argument of knows in $G$ then $P$ does not depend on does in $G$.

Formally, the new keyword uses the syntactic concept of reification, whereby a defined predicate, $P$, is used as an argument of another predicate. While the basic syntax does not support direct nesting within the new keyword, as in (knows a (knows b P)), nested knowledge can be expressed with the help of auxiliary predicates, for example, (knows a kbp) along with $<=$ kbp (knows b P). The syntactical restrictions then ensure that nested knowledge is both hierarchical (condition 2) and confined to state-dependent properties (condition 3). The former simply disallows circular definitions, as in $<=$ P (knows Q), $<=$ Q (knows P), while the latter restriction ensures that knowledge only refers to the current state and not to future actions.

Example 1 (cont’d) With the help of the new keyword, the objective of the number guessing game can be reformulated as a true knowledge goal. This results in a variant of the game that may appropriately be called NUMBERDETERMINATION. This is achieved by replacing the termination and goal conditions (rules 32–36 in Figure 1) as follows:
This, furthermore, allows us to simplify gameplay in the original game description by removing the final action of explicitly guessing the number, which is no longer needed (rules 17–18, 29–30 can be deleted).

GDL-III uses derived predicates such as (num ?n) as arguments of the knowledge predicate—in-stead of allowing for the use of state features such as (secret ?n)—since this enables specifications of non-atomic knowledge conditions and nested knowledge of different players, as in the following example.

Example 2 In Russian card games [Cordón-Franco et al., 2013], two players cooperate to learn each other’s hands through open communication without revealing enough information for a listening third player to know of any of their cards who holds it. The extended expressiveness of GDL-III can be used to specify this as a complex goal for the two cooperating players, namely, that they both know that they know each other’s cards and that it is common knowledge that the third player does not know of any of their cards. The axiomatisation below uses the following domain-dependent predicates for this purpose:

\[
\begin{align*}
&\text{(holds R C)} & \text{player R holds card C} \\
&\text{(kholdsc R S C)} & \text{player R knows that player S holds C} \\
&\text{(kholds1 R S)} & \text{player R knows some card of player S} \\
&\text{(k_all R S)} & \text{player R knows all cards of player S} \\
&\text{(ignorant R S)} & \text{player R does not know any card of S} \\
&\text{(ignorant1 R S)} & \text{player R does not know all cards of S}
\end{align*}
\]

The intuitive meaning of these predicates is formalised through the following rules of a GDL-III description suitable to define the goal in Russian card games:

\[
\begin{align*}
&\text{(kholdsc ?r ?s ?c)} & \text{(knows ?r (holds ?s ?c))} \\
&\text{(kholds1 ?r ?s)} & \text{(holds ?s ?c) (kholdsc ?r ?s ?c))} \\
&\text{(k_all ?r ?s)} & \text{(k_all ?r ?s ?c))} \\
&\text{(ignorant ?r ?s)} & \text{(ignorant1 ?r ?s))}
\end{align*}
\]

This definition can be used to specify the goal of a cooperating player, let us call the two of them Alice and Bob and their opponent Eve, as follows:

\[
\begin{align*}
&\text{(goal alice 100)} \\
&\text{(knows alice (k_all bob alice))} \\
&\text{(knows bob (k_all alice bob))} \\
&\text{(knows (ignorant eve alice))} \\
&\text{(knows (ignorant eve bob))}
\end{align*}
\]

Put in words, Alice needs to know that Bob knows all her cards and vice versa while it is common knowledge that Eve does not know any of the cards of either Alice or Bob. It is easy to verify that the predicates used as arguments of the knowledge operator are not defined circularly and hence satisfy the requirement of being hierarchical, e.g. by the ordering ignorant > k_all > kholdsc > holds.

We refrain from providing a complete formalisation of a specific instance of Russian card games in this paper and just note that this requires formal rules to define the range of communicative actions available to Alice and Bob.

3.2 GDL-III Semantics

The semantics for the new keyword can be defined in two stages. First, the logical interpretation of the rules according to Definition 1 is extended by incorporating the encoding of a given set \(K = \{(\text{knows } r_1 p_1) \ldots (\text{knows } r_n p_n)\}\) instances of the knowledge predicate. The legality of moves, percepts of players, updated states as well as the termination and goal conditions in GDL-III are all evaluated relative to a given set \(K\).

Definition 2 The pre-semantics of a valid GDL-III game description \(G\) is given by

- \(R = \{r: G \models \text{role}(r)\}\)
- \(s_1 = \{f: G \models \text{init}(f)\}\)
- \(t = \{(S, K): G \cup S^{\text{true}} \cup K \models \text{terminal}\}\)
- \(l = \{(r, m, S, K): G \cup S^{\text{true}} \cup K \models \text{legal}(r, m)\}\)
- \(u(M, S, K) = \{f: G \cup M^{\text{means}} \cup S^{\text{true}} \cup K \models \text{sees}(r, p)\}\)
- \(g = \{(r, v, S, K): G \cup S^{\text{true}} \cup K \models \text{goal}(r, v)\}\)

Example 1 (cont’d) Consider Number Determination from above, state \(S = \{(\text{secret } 9), (\text{step } 5)\}\), and sets \(K = \{}\) and \(L = \{(\text{guesses } \text{guesser} (\text{num } 9))\}\). Following rule 38–39, we have that \((S, K) \not\in t\) because there is no known instance of \((\text{num } ?n)\) in \(K\) nor has step 12 been reached in \(S\). On the other hand, \((S, L) \in t\) because in \(L\) the player knows the target. For the same reason, \((\text{guesser}, 0, S, K) \in g\) and \((\text{guesser}, 100, S, L) \not\in g\) by rules 40–43.

The second step in the definition of the semantics for GDL-III consists in an inductive characterisation of legal play sequences and their resulting knowledge states. Common knowledge is defined using the notion of the transitive closure \(~^+\) of a given family of indistinguishability relations \(~\), (one for every \(r \in R\) of a set of roles \(R\)). More formally, suppose given a set of legal play sequences and individual indistinguishability relations \(~_r\), then \(~^+\) is the smallest relation such that for all \(\delta, \delta' \delta'':\)

- \(\delta \sim^+ \delta\) and
- if \(\delta \sim^+ \delta' \delta' \sim_r \delta''\) for some \(r \in R\) then \(\delta \sim^+ \delta''\).

Definition 3 Let \(G\) be a game description along with all the sets and relations it describes according to Definition 2.

- Empty sequence \(\varepsilon\) is the only legal play sequence of length 0 and satisfies \(\varepsilon \sim_{\varepsilon}\), for all \(r \in R\), and results in state \(s_0\) and knowledge state \(K_0\), the latter being the smallest set that satisfies

\[
K_0 = \{(\text{knows } r p): r \in R, G \cup S^{\text{true}} \cup K_0 \models p\} \\
\cup \{(\text{knows } p): G \cup S^{\text{true}} \cup K_0 \models p\}
\]
For the inductive definition, let \( \delta \) be a legal play sequence of length \( n \geq 0 \), then \( \delta \) followed by \( \mu \), written \( \delta, \mu \), is a legal play sequence of length \( n + 1 \) if

\[
\begin{align*}
\delta & \text{ is a legal play sequence of length } n \text{ resulting in } (s_n, K_n) \text{ and} \\
\mu & \text{ is a legal play sequence of length } n \text{ resulting in } (s_n, K_n) \text{ and} \\
(\mu(r), s_n, K_n) & \in I \text{ for all } r \in R.
\end{align*}
\]

Sequence \( \delta, \mu \) results in state \( s_{\delta, \mu} = u(\mu, s_n, K_n) \). For the resulting knowledge state, we first define two sequences (of length \( n + 1 \)) to satisfy \( \delta, \mu \sim_r \delta', \mu' \) for \( r \in \mathcal{R} \) if

\[
\begin{align*}
\delta & \sim_r \delta', \\
\mu(r) & = \mu'(r), \text{ and} \\
\{p : (r, \mu, s_n, K_n, p) \in \mathcal{T}\} & = \{p' : (r, \mu', s_n, K_n', p') \in \mathcal{T}\}.
\end{align*}
\]

The knowledge state \( K_{\delta, \mu} \) resulting from \( \delta, \mu \) is then obtained as the smallest set that satisfies

\[
K_{\delta, \mu} = \{(\text{knows } r \ p) : G \cup s_{\delta, \mu}^{\text{true}} \cup K_{\delta, \mu} \models p \text{ for all } (\delta', \mu') \sim_r (\delta, \mu)\}
\]

\[
\cup \{(\text{knows } p) : G \cup s_{\delta, \mu}^{\text{true}} \cup K_{\delta, \mu} \models p \text{ for all } (\delta', \mu') \sim^+(\delta, \mu)\}
\]

(3)

This is well-defined for any GDL-III game description \( G \) with hierarchically defined knowledge predicates, in which case \( K_{\delta, \mu} \) can be "constructed" by first evaluating all \( (\text{knows } r \ p) \) and \( (\text{knows } p) \) instances for which \( p \) itself does not depend on \( \text{knows} \) and then evaluating the other instances in accordance with the hierarchy.

Definition 3 can be understood as follows: All players have perfect knowledge of all predicates in the initial state \( (K_0) \). The legality of moves and the updated states are determined inductively from the preceding (actual and knowledge) state \( s_n \) and \( K_n \), respectively. Two legal play sequences \( \delta, \mu \) and \( \delta', \mu' \) cannot be distinguished by a player after the last move if they were indistinguishable beforehand (i.e., \( \delta \sim_r \delta' \)) and if the player made the same legal move in both \( \mu \) and \( \mu' \) and obtained the same percepts. This (in-)distinguishability relation in turn determines the evaluation of the knowledge predicates, including common knowledge, for the resulting knowledge states \( K_{\delta, \mu} \).

4 Automated Reasoning for GDL-III

In order to be able to play games specified in the extended game description language, any general game-playing system needs an automated reasoning component for evaluating the rules to determine legal moves and compute state updates. Several approaches to building reasoners for GDL and GDL-II have been described [Schiffel and Björnsson, 2013; Thielscher, 2013]. In this section, we build on previous uses of Answer Set Programming (ASP) [Gelfond, 2008] in general game playing [Thielscher, 2009; Möller et al., 2011] to develop the foundations for automated reasoners for GDL-III.

Reasoning With Game Rules The game description language and ASP have essentially the same basic semantics given by the unique stable model (a.k.a. answer set) of a stratified set of logic program rules. Hence, the basic reasoning tasks according to Definition 2 can be easily automated by mapping any given game description into ASP clauses. Since the evaluation of knowledge conditions depends on previous moves and percepts, all state-dependent predicates in a game description are augmented by two additional arguments so that a single ASP can be used to reason about different legal play sequences, \( \text{seq}(S) \), and different time points, \( \text{time}(T) \). We define \( \mathcal{P}(G) \) to be the ASP-encoding thus obtained from any given set of GDL-III rules \( G \). For example, the ASP-encoding of rule 40–41 in NUMBERGUESSING with knowledge (cf. Section 3.1) is

\[
\begin{align*}
g\text{o}\text{a}\text{l}(\text{gu}\text{e}\text{ss}e\text{r},100,S,T) :- \\
\text{k}\text{n}\text{o}\text{w}\text{s}(\text{gu}\text{e}\text{ss}\text{e}r,\text{n}\text{u}\text{m}(N),S,T), \text{seq}(S), \text{time}(T)).
\end{align*}
\]

We follow the convention of using natural numbers for time, so that (init \( F \)) is replaced by \( \text{true}(F,S,0) \) in \( \mathcal{P}(G) \) and (next \( F \)) by \( \text{true}(F,S,T+1) \).

Reasoning About Knowledge In accordance with their semantics, knowledge conditions are evaluated on the basis of the (in-)distinguishability of legal play sequences. The definition in the previous section implies that players can distinguish any two sequences in which at least one of their preceding moves or percepts differ. Otherwise, the two sequences are indistinguishable (predicate \( \text{ind} \)):

\[
\begin{align*}
d\text{i}\text{s}\text{t}\text{i}\text{n}\text{d}\text{i}\text{i}\text{s}\text{n}\text{i}\text{t}\text{i}\text{a}\text{b}\text{i}\text{l}\text{e}\text{r}(R,S1,S2,N) :- \text{time}(N), T<N, \\
do\text{e}\text{s}(R,M1,S1,T), \text{does}(R,M2,S2,T), M1!=M2. \\
d\text{i}\text{d}\text{i}\text{n}\text{s}\text{t}\text{i}\text{n}\text{i}\text{s}\text{i}\text{t}\text{i}\text{a}\text{b}\text{i}\text{l}\text{e}(R,S1,S2,N) :- \text{time}(T), T<N, \\
\text{s}\text{e}\text{e}\text{s}(R,P,S1,T), \text{not}\ \text{s}\text{e}\text{e}\text{s}(R,P,S2,T).}
\end{align*}
\]

\[
\begin{align*}
\text{i}\text{n}\text{d}(R,S1,S2,N) :- \text{role}(R), \text{seq}(S1), \text{seq}(S2), \\
\text{t}\text{i}\text{m}\text{e}(N), \text{not}\ \text{d}\text{i}\text{s}\text{t}\text{i}\text{i}\text{n}\text{s}\text{i}\text{t}\text{i}\text{a}\text{b}\text{i}\text{r}(R,S1,S2,N).
\end{align*}
\]

\[
\begin{align*}
\text{i}\text{n}\text{d}\text{n}\text{t}\text{r}\text{a}\text{n}\text{s}(S1,S3,N) :- \text{seq}(S1), \text{time}(N). \\
\text{i}\text{n}\text{d}\text{n}\text{t}\text{r}\text{a}\text{n}\text{s}(S1,S3,N) :- \text{ind}(R,S1,S2,N), \\
\text{i}\text{n}\text{d}\text{n}\text{t}\text{r}\text{a}\text{n}\text{s}(S2,S3,N).
\end{align*}
\]

The last two clauses above encode the transitive closure \( \sim^+ \) of the indistinguishability relation over all roles. It is easy to verify that the encoding of \( \text{distinguishable}(R,S1,S2,N) \) corresponds to the semantics of two sequences \( S1 \) and \( S2 \) of length \( N \) being indistinguishable by player \( R \) according to the inductive characterisation of \( \sim \), as per Definition 3.

According to the definition of resulting knowledge states, (3), a condition on an individual role’s knowledge is true if the property in question holds in all sequences that are indistinguishable by that player. On this basis, every \( (\text{knows } R \ p(x)) \) can be evaluated according to the schema

\[
\begin{align*}
\text{k}\text{o}\text{w}\text{n}\text{s}(R,p(x),S,T) :- p(x,S,T), \text{not}\ np(R,\bar{x},S,T). \\
\text{n}\text{p}(R,\bar{x},S,T) :- \text{ind}(R,S1,S1,T), \text{not}\ p(\bar{x},S1,T).
\end{align*}
\]

Put in words, if \( S \) is the actual play sequence at time \( T \), then player \( R \) knows that \( p(\bar{x}) \) just in case \( p(\bar{x}) \) actually holds and it is not the case that (predicate \( \text{np} \)) there is a sequence \( S1 \) that \( R \) cannot distinguish from \( S \) and in which \( p(\bar{x}) \) does not hold.
Similarly, according to Definition 3, a property \( p(\vec{x}) \) is common knowledge if \( p(\vec{x}) \) holds in all sequences that are in the transitive closure of the indistinguishability relation across all players. Hence, every \( (\text{knows} \ p(\vec{x})) \) can be evaluated according to the schema

\[
\text{knows}(p(\vec{x}), S, T) :- p(\vec{x}, S, T), \text{not} \ np(\vec{x}, S, T).
\]
\[
np(\vec{x}, S, T) :- \text{indtrans}(S, S_1, T), \text{not} \ p(\vec{x}, S_1, T).
\]

Let \( \mathbb{K}(G) \) be the ASP clauses thus obtained. Furthermore, for a given set of play sequences \( \mathcal{S} \), let \( \mathbb{D}(\mathcal{S}) \) be its ASP encoding as facts of the form \( \text{does}(R, M, S, T) \) with a unique identifier \( S \) for each element in \( \mathcal{S} \). The correctness of the encoding \( \mathbb{P}(G) \cup \mathbb{K}(G) \cup \mathbb{D}(\mathcal{S}) \) then follows from the fact that the clauses \( \mathbb{K}(G) \) provide a direct encoding of the definition of legal play sequences and their (in-)distinguishability.

**Example: A GDL-III Game Controller**  The ASP-based reasoning technique can be used to design a game controller [Genesereth et al., 2005] for automatically controlling the execution of GDL games \( G \) with imperfect information and introspection, as follows.

1. Send all players the game description \( G \).
2. Let \( \mathcal{S} = \{s\} ; s = e; t = 0 \).
3. Request a move \( M \) from each player \( R \). The legality of the moves can be verified using the predicate legal\( (R, M, s, t) \) against \( \mathbb{P}(G) \cup \mathbb{K}(G) \cup \mathbb{D}(\mathcal{S}) \). Let \( m \) be the joint move.
4. Let \( \mathcal{S} = \{\delta, \mu : \delta \in \mathcal{S}, \mu \text{ legal in } \delta\} ; s = s, m; t = t + 1 \).
5. Repeat steps 3 and 4 until \( \mathbb{P}(G) \cup \mathbb{K}(G) \cup \mathbb{D}(\mathcal{S}) \) entails terminal\( (s, t) \).

**5 Solving Epistemic Puzzles With GDL-III**

Epistemic puzzles are characterised by multiple agents starting off with imperfect, and in many cases asymmetric, knowledge. They draw further conclusions by logical reasoning about (and controlling the execution of) games developed in the previous section can be used with just a small modification: Rather than maintaining an actual sequence \( s \) and checking for its termination, we just need to find, in the answer set of \( \mathbb{P}(G) \cup \mathbb{K}(G) \cup \mathbb{D}(\mathcal{S}) \), a positive instance of terminal\( (s, t) \). Any such \( s \), in conjunction with the corresponding move sequence encoded in \( \mathbb{D}(\mathcal{S}) \), determines a solution. For CHERYLSBIRTHDAY with the standard 10 choices [Chang, 2015], there is only one such legal play sequence after step 4, and cheryl’s initial move gives the unique solution (jul16).

**6 Experimental Results**

In order to test how an ASP-based controller for GDL-III games scales, we ran two different sets of experiments with the example games in this paper. The experiments were carried out on a 2.8 GHz processor with 8 GB of RAM using an off-the-shelf answer set solver.\(^6\) Times are reported in seconds (CPU time).

**NUMBERDETERMINATION** (cf. Example 1) is a representative of a class of games in which the goal is for players to uncover hidden knowledge. In these games, players typically start off (after one or more unobserved, random moves) with a maximal information set, which then decreases monotonically as the game unfolds and the players acquire further information. We ran experiments with increased ranges of possible numbers and game lengths (i.e. maximal numbers of questions the player can ask). All moves were chosen randomly. The runtime for an ASP-based game controller, averaged over 1,000 runs for each problem size, are summarised in the table below, including the average size of the final information set.

\(^3\)This would also require to provide Albert and Bernard with more move options to ensure that they have a legal move in every reachable state.

\(^6\)http://potassco.sourceforge.net/
Channel 2: Chersyl's Birthday: a possible description of this puzzle using the syntax of GDL with introspection. Cheryl begins by picking a date (rule 13–14), of which Albert and Bernard only get to see the month and day, respectively (lines 16–17). Using three defined properties (lines 26–30), announcements are modelled by two moves (sayUnknown, sayKnown) whose preconditions require players to be truthful (lines 32–43) and which are public (rule 45–47).

The results with an ASP-based solver as described in Schiffel and Thielscher, 2014. Knowledge preconditions require players with perfect recall [Schiffel and Thielscher, 2014]. Knowledge preconditions and knowledge goals in GDL-III therefore refer to what players can and cannot know in principle given the observations and knowledge goals in GDL-II assuming players with perfect recall. Thus the axiomatisation of epistemic games does not require rules that specify how percepts affect the knowledge of players. Rather, this follows implicitly from the execution model. The semantics of the knowledge operator in GDL-III uses the same concept of indistinguishability of play sequences that has been used to characterise the evolution of knowledge in GDL-II assuming players with perfect recall [Schiffel and Thielscher, 2014]. Knowledge preconditions and knowledge goals in GDL-III therefore refer to what players can and cannot know in principle given the observations they make throughout a game.

Again, from the game description in Figure 2 it is clear that a state in NUMBERGUESSING consists of at most two features, independent of the problem size. Note also that, unlike for the results in NUMBERGUESSING, the length of a game is constant too (4 steps). The results demonstrate how the average time to find a legal play sequence increases since this game requires maintaining and evaluating the knowledge state of both roles, including what they entail about one player’s knowledge of the other player.

### Related Work and Conclusion

The extended general game description language GDL-III shares with its predecessor GDL-II the fact that all game rules are objective in the sense that they specify the observations that players make. Thus the axiomatisation of epistemic games does not require rules that specify how percepts affect the knowledge of players. Rather, this follows implicitly from the execution model. The semantics of the knowledge operator in GDL-III uses the same concept of indistinguishability of play sequences that has been used to characterise the evolution of knowledge in GDL-II assuming players with perfect recall [Schiffel and Thielscher, 2014]. Knowledge preconditions and knowledge goals in GDL-III therefore refer to what players can and cannot know in principle given the observations they make throughout a game.

The extended expressiveness of the new language is manifest in the definition of its semantics, which is considerably more involved than that for its predecessor. State transition systems that provide the full semantics for GDL-II act merely
as the pre-semantics for GDL-III, which then requires the interleaved incorporation of (nested and common) knowledge in order to provide the full semantics. This self-reference, where knowledge feeds back into the definition of legal play sequences, is not present in GDL-II.

The general game description language shares with other logic-based knowledge specification languages for actions and change the use of atomic state features in conjunction with precondition and effect axioms. It has been shown, for example, that the Situation Calculus with knowledge [Schierl and Levesque, 2003] can be used to provide an axiomatisation of the concept of (in-)distinguishable play sequences in GDL-II [Schiffel and Thielscher, 2014]. Since the semantics of the knowledge operator in GDL-III is based on the same concept of indistinguishability, this characterisation can be applied to the extended language as well.

Epistemic puzzles similar to Cheryl’s Birthday have been solved using model checking systems for Dynamic Epistemic Logic [van Ditmarsch et al., 2005]. The main difference is that this requires an explicit encoding of an epistemic structure in form of a concrete accessibility relation for the problem in question, whereas it suffices to describe the rules of how the environment evolves in a formal description with the syntax of GDL-III in conjunction with the use of a general game controller (cf. Figure 2); the necessary epistemic structure then follows implicitly from the semantics built into the general reasoner. While the proposed solution of epistemic puzzles uses GDL different from its main purpose in general game playing, because no players actually play the game, it is an interesting by-product of having an ASP-based game controller for the new language element. More generally, the formal relation between GDL-II and epistemic logic has been studied in detail [Ruan and Thielscher, 2014]. These existing results carry over to the extended language.

Our experimental results have shown how the size of the information sets of players influences the runtimes of a game controller, unlike in case of GDL-II. While explicitly maintaining the set of relevant legal play sequences, as in our ASP-based encoding, is practically viable in case of typical epistemic games, in which the goal of the players is to reduce an initially maximal information set, other imperfect-information games will require compact representations of information sets, e.g. as has been proposed for Kriegspiel [Ciancarini and Favini, 2007], or approximations via state sampling [Long et al., 2010].

GDL-III has applications in general game playing beyond the description of games with explicit knowledge conditions. For example, it could be used by imperfect-information players for the automatic construction of logical strategy rules by which they condition their moves on their knowledge.

References


Stochastic Constraint Programming for General Game Playing with Imperfect Information

Frédéric Koriche, Sylvain Lagrue, Éric Piette, and Sébastien Tabary
CRIL Univ. Artois - CNRS UMR 8188, France-62307 Lens
{koriche,lagrue,epiette,tabary}@cril.fr

Abstract
The game description language with incomplete information (GDL-II) is expressive enough to capture partially observable stochastic multi-agent games. Unfortunately, such expressiveness does not come without a price: the problem of finding a winning strategy is NP-hard, a complexity class which is far beyond the reach of modern constraint solvers. In this paper, we identify a PSPACE-complete fragment of GDL-II, where agents share the same (partial) observations. We show that this fragment can be cast as a decomposable stochastic constraint satisfaction problem (SCSP) which, in turn, can be solved using general-purpose constraint programming techniques. Namely, we develop a constraint-based sequential decision algorithm for GDL-II games which exploits constraint propagation and Monte Carlo sampling based. Our algorithm, validated on a wide variety of games, significantly outperforms the state-of-the-art general game playing algorithms.

1 Introduction
Of all human activities, games convey one of the most illustrative examples of intelligent behavior. A player has indeed to face complex tasks, such as understanding abstract rules, evaluating the current situation, choosing the best possible move and, ultimately, devising a winning strategy. In Artificial Intelligence, the General Game Playing (GGP) challenge [Genesereth and Thielscher, 2014; Genesereth et al., 2005] is to develop computer players that understand the rules of previously unknown games, and learn to play these games well without human intervention.

In General Game Playing, the rules of an input game are described using a high-level, declarative representation formalism, called Game Description Language (GDL). The first version of this language (GDL-I) is restricted to deterministic games with complete information [Love et al., 2008]. While a GDL-I game may involve multiple players, each player has complete knowledge about the current game state, the past actions of her opponents, and the deterministic effects of the joint action.

In order to alleviate these restrictions, Schiffel and Thielscher [2011; 2014] recently proposed a new game description language (GDL-II) for representing games with incomplete information. In a GDL-II game, players may have restricted access to the current game state, and the effects of their joint actions are uncertain. As such, GDL-II is expressive enough to capture partially observable stochastic games (POSGs), which cover a wide variety of multi-agent sequential decision problems. However, such expressiveness does not come without a price: from a computational viewpoint, the problem of finding a winning strategy in a POSG is PSPACE-complete [Goldsmith and Mundhenk, 2007], a complexity class which is far beyond the standard PSPACE complexity class of games with complete information. Although several algorithms have been devised for tackling specific instances of POSGs, such as Contract Bridge [Ginsberg, 2001] and Poker [Bowling et al., 2015], they are dedicated programs which heavily rely on human knowledge about game rules and evaluation functions. By contrast, the task of developing general game players for POSGs appears extremely challenging due to their complexity barrier.

Despite this theoretical issue, several GGP algorithms have been recently developed for solving restricted, yet expressive, fragments of GDL-II. They include, for example, mono-agent deterministic games with incomplete information [Geißer et al., 2014], and multi-agent stochastic games with complete information [Koriche et al., 2016]. In particular, the last approach relies on Constraint Programming techniques which have proved to be successful in practice. Our present study aims at extending the range of GDL-II games which can be expressed in terms of Stochastic Constraint Satisfaction Problems (SCSPs).

The main contribution of this article can be summarized as follows: (i) we present an important PSPACE-complete fragment of GDL-II, where players share the same (partial) observations, and which can be expressed as a SCSP; (ii) we extend MAC-UCB, a sequential decision algorithm that exploits constraint propagation and Monte Carlo sampling, to GDL-II games; (iii) we provide a comparative experimental study on various GDL games, including deterministic games, stochastic games, and partially observable stochastic games (with shared information). Experimental results show that our constraint programming technique outperforms the current general game playing algorithms.

2 General Imperfect Information Games
The problems under consideration in GGP are finite sequential and synchronous games. Each game involves a finite number of players, and a finite number of states, including one distinguished initial state, and one or several terminal states. On each round of the game, each player has at her disposal a finite number of actions (called “legal moves”); the current state of the game is
updated by the simultaneous application of each player’s action (which can be “noop” or do nothing). The game starts at the initial state and, after a finite number of rounds, ends at some terminal state, in which a reward is given to each player. In a stochastic game, a distinguished player, often referred to as “chance”, can choose its actions at random according to a probability distribution defined over its legal moves. In an incomplete information game, some aspects (called “fluent”) of the current game state are not fully revealed to the agents. We shall focus on stochastic shared information games in which, at each round, all agents have the same (possibly incomplete) information about the game state.

2.1 GDL-II Syntax

GDL is a declarative language for representing finite games. Basically, a GDL program is a set of rules described in first-order logic. Players and game objects (coins, dice, locations, pieces, etc.) are described by constants, while fluents and actions are described by first-order terms. The atoms of a GDL program are constructed over a finite set of relation symbols and variable symbols. Some symbols have a specific meaning in the program, and are described in Table 1. For example, in the TicTacToe game, legal(alice,mark(X,Y)) indicates that player alice is allowed to mark the square (X,Y) of the board. In GDL-II, the last two keywords of the table are added to represent stochastic games (random), and partially observable games (sees).

Table 1: GDL-II keywords

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role(P)</td>
<td>P is a player</td>
</tr>
<tr>
<td>init(F)</td>
<td>F holds in the initial state</td>
</tr>
<tr>
<td>true(F)</td>
<td>F holds in the current state</td>
</tr>
<tr>
<td>legal(P,A)</td>
<td>the fluent F holds in the current state and player P can take action A</td>
</tr>
<tr>
<td>does(P,A)</td>
<td>the player P performs action A</td>
</tr>
<tr>
<td>next(F)</td>
<td>F holds in the next state</td>
</tr>
<tr>
<td>terminal</td>
<td>the current state is terminal</td>
</tr>
<tr>
<td>goal(P,V)</td>
<td>the player P gets reward V in the current state</td>
</tr>
<tr>
<td>sees(P,F)</td>
<td>the player P perceives F in the current state</td>
</tr>
<tr>
<td>random</td>
<td>the “chance” player</td>
</tr>
</tbody>
</table>

The rules of a GDL program are first-order Horn clauses. For example, the rule:

sees(alice,cell(X,Y,0)) ← does(alice,mark(X,Y))

states that alice sees the effect of marking squares of the board. In order to represent a finite sequential game, a GDL program must obey syntactic conditions, defined over the terms and relations occurring in rules, and the structure of its rule set. We refer the reader to [Love et al., 2008; Thiel scher, 2010] for a detailed analysis of these conditions.

Example 1 “Matching Pennies” is a well-known game involving two players, who place a penny (or coin) on the table, with the payoff depending on whether pennies match. We consider here a variant named “Hidden Matching Pennies” in which one player chooses tails or heads for its two pennies and, during the second round, the other player chooses tails or heads for its two pennies and, during the second round, alice places a coin on the table; alice wins 100 points if all the three sides are heads or tails and 50 points if at least one side of the chance player and its side are similar. The corresponding GDL program is described in Figure 1;

2.2 GDL-II Semantics

For a positive integer n, let [n] = {1,...,n}. For a finite set S, let ΔS denote the probability simplex over S, that is, the space of all probability distributions over S. Various descriptions of incomplete information games have been proposed in the literature (see e.g. [Schiffel and Thiel scher, 2011]). We focus here on a variant of [Geißer et al., 2014].

Formally, a partially observable stochastic game with legal actions (POSG), is a tuple G = (k,S,S0,Sg,A,L,P,B,R), where:

- k ∈ N is the number of players,
- S is a finite set of states, including a distinguished initial state s0, and a subset Sg of goal (or terminal) states.
- A is a finite set of actions. As usual an action profile is a tuple α = ⟨a1,...,ak⟩ ∈ Ak, by ap, we denote the action of player p, and by αp, the action profile ⟨a1,...,ap−1,ap+1, ...,ak⟩ of the remaining players.
- L: [k] × S × A → ΔS defines the set of legal actions Lp(s) of player p at state s; we assume that Lp(s0) = ∅ for all s ∈ Sg.
- P: S × A → ΔS is the partial transition probability function, which maps each state s ∈ S and each action profile ⟨a1,...,ak⟩ ∈ L(s) to a probability distribution over S.
- B: [k] × S → ΔS is the belief function which maps every player p ∈ [k] and every state s ∈ S to a probability distribution Bp(s) over S, capturing the belief state of p at s.
- R: [k] × Sg → [0,1] is the reward function which maps every player p ∈ [k] and every goal state s ∈ Sg to a value Rp(s) ∈ [0,1], capturing the reward of p in s.

With these notions in hand, the POSG G associated to a GDL-II program G is defined as follows. Let B denote the Herbrand base (i.e. the set of all ground terms) of G; A (resp. F) is the set of all ground action (resp. fluent) terms occurring in B. We use G |= A to denote that atom A is true in the unique answer set of G. The number k of ground terms p such that role(p) ∈ G, determines the set of players.

Each state s is a subset of F. Notably, the initial state s0 is {f: G |= init(f)}, and any terminal state is a set of fluents s = {f1,...,fn} such that G∪strue |= terminal, where strue is the set of facts {true(f1),...,true(fn)}. The set Lp(s) of legal actions for player p at state s is given by G∪strue |= legal(p,a). In particular, L0(s) denotes the set of legal actions for the chance player (random). Any action profile (extended to the chance player) α = ⟨a0,a1,...,ak⟩ ∈ L0(s) × L1(s) × ... × Lk(s) determines a successor s′ of s given by {f: G∪strue |= does(f) |= next(f)}, where αdoes is the set of facts {does(a0),does(a1),...,does(ak)}. The probability distribution P(s,α−0) over all those successors is the uniform distribution, i.e. P(s,α−0)(s′) = 1/|L0(s)|.

The belief state Bp(s) of player p at any successor s′ of s is given by the joint distribution \(\prod_{f \in F} P(f)\), where P(f) = 1 if G∪strue |= sees(p,f), and P(f) = 1/2 otherwise. Finally, the reward R(s) of player p at a terminal state s is the value v such that G∪strue |= goal(p,v).

40 GIGA’16 Proceedings
2.3 A PSPACE Fragment of GDL-II

A game with shared information is any partially observable stochastic game $G$ in which, at each round, all players share the same belief state, i.e. $B_1(s) = \cdots = B_k(s)$ for all states $s \in S$. We use here $B(s)$ to denote the common belief state in $s$. Remark that any game with shared information can be converted into a fully observable stochastic game, by replacing the transition function $P$ and the belief function $B$ with a new transition function $Q: S \times A^k \rightarrow \Delta S$ defined by:

$$Q(s, a)(s') = \prod_{t \in S} P(s, a)(t) \cdot B(t)$$

for all $a \in L(s)$. In other words, $Q(s, a)(s')$ is the probability of observing $s'$ after performing the action profile $a$ in state $s$. Since any game $G$ with shared information is a stochastic game, a joint policy for $G$ is a map $\pi: S \rightarrow A^k$, where $\pi_p(s)$ is the policy of player $p$, and $\pi_{-p}(s)$ is the joint policy of other players. Given a threshold vector $\theta \in [0, 1]^k$, we say that $\pi$ is a winning policy for player $p$ if the expected reward of $p$ w.r.t. $\pi$ is greater than $\theta_p$.

Based on the notion of shared information, we now examine several restrictions of GDL-II programs which together guarantee that the problem of finding a winning policy is in PSPACE. Namely, a GDL-II program $G$ is depth-bounded if the number of ground terms in the Herbrand universe of $G$ is polynomial in $|G|$. If $G$ is bounded and each rule of $G$ has a constant number of variables, then $G$ is propositional. For an integer $T$, $G$ is of horizon $T$ if any terminal state is reachable after at most $T$ rounds. Finally, $G$ is information sharing if for every player $p$, every fluent $f$, every state $s$, and every action profile $a$, if $G \cup s^{\text{true}} \cup a^{\text{does}} \models \text{sees}(p, f)$, then $G \cup s^{\text{true}} \cup a^{\text{does}} \models \text{sees}(q, f)$, for all players $q \in [k]$.

**Theorem 1** Let $G_T \subseteq \text{GDL-II}$ be the fragment propositional, information sharing programs of horizon $T$. Then, the problem of finding a winning policy in $G_T$ is PSPACE-complete.

**Proof (Sketch)** Since $G_T$ includes full-information stochastic games as a special case, the problem is PSPACE-hard. For any finite and depth-bounded game $G \in G_T$, a winning policy can be found in $f(|G|)$ time and $g(|G|)$ space using a stochastic-alternating Turing Machine (TM), i.e. a TM which includes stochastic states (for random), existential states (for player $p$), and universal states (for all other players). Since $G$ is propositional, the number of fluents and the number of actions are polynomial in $|G|$, which together imply that $g(|G|)$ is polynomial. At each game state, the stochastic-alternating TM can guess a game action profile using its existential states. Since $k \leq |G|$, the next game state can be found using a polynomial number of universal states and stochastic states. This, together with fact that the TM will find a terminal game state in at most $T$ rounds, implies that $f(|G|)$ is also polynomial. Finally, since any stochastic-alternating TM using polynomial time and space can be simulated by NPSPACE (see e.g. [Bonnet and Saffidine, 2014]) then, using Savitch’s theorem NPSPACE = PSPACE, it follows that $G_T$ is in PSPACE, which yields the result.

3 The SCSP Framework

Borrowing the terminology of [Walsh, 2002], stochastic constraint networks extend the standard CSP framework by introducing...
stochastic variables in addition to the usual decision variables. We focus here on a slight generalization of the original SCSP model that captures conditional probability distributions over stochastic variables.

Definition 1 A Stochastic Constraint Satisfaction Problem (SCSP) is a 6-tuple \( N = (\mathcal{V}, \mathcal{Y}, \mathcal{D}, C, P, \theta) \), such that \( \mathcal{V} = (V_1, \ldots, V_n) \) is a finite tuple of variables, \( \mathcal{Y} \subseteq \mathcal{V} \) is the set of stochastic variables, \( \mathcal{D} \) is a mapping from \( \mathcal{V} \) to domains of values, \( C \) is a set of constraints, \( P \) is a set of conditional probability tables, and \( \theta \in [0,1] \) is a threshold.

- Each constraint in \( C \) is a pair \( C = (scp_C, val_C) \), such that \( scp_C \) is a subset of \( \mathcal{V} \), called the scope of \( C \), and \( val_C \) is a map from \( \mathcal{D}(scp_C) \) to \( \{0,1\} \).
- Each conditional probability table in \( \mathcal{P} \) is a triplet \( \langle Y, scp_Y, prob_Y \rangle \), where \( Y \in \mathcal{Y} \) is a stochastic variable, \( scp_Y \) is a subset of variables occurring before \( Y \) in \( \mathcal{V} \), and \( prob_Y \) is map from \( \mathcal{D}(scp_Y) \) to a probability distribution over the domain \( \mathcal{D}(Y) \).

By \( \mathcal{X} \), we denote the set \( \mathcal{V}\setminus \mathcal{Y} \) of decision variables. If \( Y \in \mathcal{Y} \) is a stochastic variable and \( \tau \in \mathcal{D}(scp_Y) \) is a tuple of values in the conditional probability table of \( Y \), then we use \( P(Y | \tau) \) to denote the distribution \( prob_Y(\tau) \). In particular, if \( y \in \mathcal{D}(Y) \), then \( P(Y = y | \tau) \) is the probability that \( Y \) takes value \( y \) given \( \tau \).

Given a subset \( \mathcal{U} = (V_1, \ldots, V_m) \subseteq \mathcal{V} \), an instantiation \( \mathcal{I} \) is an assignment \( I \) of values \( \nu_1 \in \mathcal{D}(V_1), \ldots, \nu_m \in \mathcal{D}(V_m) \) to the variables \( V_1, \ldots, V_m \), also written \( I = \{(V_1,\nu_1),\ldots,(V_m,\nu_m)\} \). An instantiation \( \mathcal{I} \) on \( \mathcal{U} \) is complete if \( \mathcal{I} \supseteq \mathcal{U} \). Given a subset \( \mathcal{U} \subseteq \mathcal{V} \), we use \( I_{\mathcal{U}} \) to denote the restriction of \( I \) to \( \mathcal{U} \), that is, \( I_{\mathcal{U}} = \{(V_i,\nu_i) \in I : V_i \in \mathcal{U}\} \). The probability of \( I \) is given by:

\[
P(I) = \prod_{Y \in \mathcal{Y} \setminus scp_Y \subseteq \mathcal{U}} P(Y = I_Y | I_{scp_Y})
\]

Correspondingly, the utility of an instantiation \( I \) on \( \mathcal{U} \) is given by:

\[
val(I) = \sum_{C \in C : scp_C \subseteq \mathcal{U}} val(I_{scp_C})
\]

An instantiation \( I \) is called consistent with a constraint \( C \) if \( val(I_{scp_C}) = 1 \), that is, \( I \) can be projected to a tuple satisfying \( C \). By extension, \( I \) is locally consistent if \( val(I) = 1 \), that is, \( I \) satisfies every constraint in \( C \). Finally, \( I \) is globally consistent (or consistent, for short) if it can be extended to a complete instantiation \( I' \) which is locally consistent.

A policy \( \pi \) for the network \( N \) is a rooted tree where each internal node is labeled by a variable \( V \) and each edge is labeled by a value in \( \mathcal{D}(V) \). Specifically, nodes are labeled according to the ordering \( V \): the root node is labeled by \( V_1 \), and each child of a node \( V_i \) is labeled by \( V_{i+1} \). Decision nodes \( X_i \) have a unique child, and stochastic nodes \( Y_i \) have \( |\mathcal{D}(Y_i)| \) children. Finally, each leaf in \( V \) is labeled by the utility \( val(I) \), where \( I \) is the complete instantiation specified by the path from the root of \( \pi \) to that leaf. Let \( \mathcal{L}(\pi) \) be the set of all complete instantiations induced by \( \pi \). Then, the expected utility of \( \pi \) is the sum of its leaf utilities weighted by their probabilities. Formally,

\[
val(\pi) = \sum_{I \in \mathcal{L}(\pi)} P(I) val(I)
\]

A policy \( \pi \) is a solution of \( N \) if its expected utility is greater than or equal to the threshold \( \theta \), that is \( val(\pi) \geq \theta \). We mention in passing that if \( \theta = 1 \), then \( \pi \) is a solution of \( N \) if and only if \( val(I) = 1 \) for each path \( I \) in \( \mathcal{L}(\pi) \) such that \( P(I) \neq 0 \).

A (decision) stage in a SCSP is a tuple of variables \( (X_i, Y_i) \), where \( X_i \) is a subset of decision variables, \( Y_i \) is a subset of stochastic variables, and decision variables occurs before any stochastic variable [Hnich et al., 2012]. By extension:

Definition 2 A T-stage stochastic constraint satisfaction problem is an SCSP \( N = (\mathcal{V}, \mathcal{Y}, \mathcal{D}, C, P, \theta) \), in which \( \mathcal{V} \) can be partitioned into T stages, i.e. \( \mathcal{V} = \{(X_1,Y_1),\ldots,(X_T,Y_T)\} \), where \( \{X_i\}_{i=1}^T \) is a partition of \( \mathcal{X} \), \( \{Y_i\}_{i=1}^T \) is a partition of \( \mathcal{Y} \), and \( scp_Y \subseteq X_i \) for each \( i \in \{1,\ldots,T\} \) and each \( Y \subseteq Y_i \). If \( T = 1 \), \( N \) is called a one-stage SCSP, and denoted \( \mu \text{SCSP} \).

Note that the problem of finding a winning policy in a SCSP is PSPACE-complete. The problem remains in PSPACE for T-stage k-player SCSPs, as each stage of the problem is in \( \text{NP}^P \).

4 An SCSP Representation of Games

In this section, we present a constraint-based representation of games. Namely, a GD-IL game \( G \) is first cast as a stochastic constraint network \( N \), which encodes the rule of the game as a set of constraints. Then, \( N \) is enriched by a new set of constraints describing the players’ solution concepts. The final stochastic constraint network, capturing both the game rules and the players’ strategies, can be solved using standard SCSP algorithms.

4.1 Modelling Game Rules

The translation of a k-player game \( G \in G_T \) into a T-stage SCSP is specified as follows. We first convert \( G \) into a set ground rules \( G' \), whose size is polynomial in \( |G| \) because \( G \) is propositional. To \( G' \) we associate a one-stage SCSP \( N = (\langle k \rangle, G_T, R_T, \{F_i\}, \{A_i\}, A_{i,0}, \{F_{i+1}\}) \), where \( \langle k \rangle = \{1,\ldots,k\} \) is the set of players, \( G_T \) is a Boolean variable indicating whether the game has reached a terminal (goal) state, and \( R_T \) is the set of reward values of the game. \( \{F_i\} \) and \( \{F_{i+1}\} \) are the sets of fluents describing the game state at round \( i \) and \( i + 1 \), respectively; in order to respect the one-stage property, \( \{F_i\} \) is a set of decision variables, and \( \{F_{i+1}\} \) is a set of stochastic variables. \( \{A_i\} = \{A_{i,1},\ldots,A_{i,k}\} \) is a set of decision variables, each \( A_{i,j} \) describing the set of possible moves of player \( j \). Finally, \( A_{i,0} \) is a stochastic variable describing the set of possible moves of the chance player.

The Horn clauses of a GD-IL program \( G \) can naturally be partitioned into initial rules describing the initial stage, legal rules specifying the legal moves at the current state, next rules capturing the effects of actions, see rules describing the observations of each player on the game, and goal rules defining the players’ rewards at a terminal state. init, legal, and next rules are encoded into hard constraints in the network \( N \). The see rules are used to express the conditional probability table \( P(f_{i+1} | f, a) \), of each stochastic variable \( f_{i+1} \). The last stochastic variable \( A_{i,0} \) is associated to a uniform probability distribution \( P(a_{i,0} | f) \) over the legal moves of the chance player. The constraint relation is extracted in the same way as the domains of variables, by identifying all allowed combinations of constants. Similarly, goal rules are encoded by a constraint encoding player’s rewards at a terminal state.
By repeating $T$ times this conversion process, we therefore obtain a $T$-stage SCSP encoding the $T$-horizon game $G$. The threshold $\theta$ is set to $1$, indicating that all constraints must be satisfied by a solution policy.

4.2 Modelling Strategies

Based on the above translation process, the resulting SCSP $N$ encodes only the game rules of the input GDL program. In order to “solve” a game, $N$ must also incorporate, in a declarative way, the solution concepts of players. In essence, these solutions concepts or strategies are captured by two operators $\otimes$ and $\otimes$, together with a set of constraints joining them. In what follows, we write $a_{0} \in \{A_{1}\}$ as an abbreviation of $(a_{1}, \ldots, a_{t}) \in A_{1} \times \ldots \times A_{k}$, for specifying an action profile of the $k$ players (excluding the chance player $0$). By extension, $a_{\{0, p\}} \in \{A_{1}\}^{-p}$ denotes an action profile $(a_{1}, \ldots, a_{p-1}, a_{p+1}, \ldots, a_{k})$ of $k-1$ players excluding both $p$ and the chance player. The shortcut $f \in \{F_{i}\}$ is defined similarly.

To incorporate players’ strategies, each one-stage SCSP in $N$ is enriched with several constraints. The first constraint $u_{t, 0}$, defined over the scope $\{f_{0}, \{A_{1}\}, U_{0}\}$, associates to each player $p \in \{k\}$, each action description $f_{0}$, each state description $f$, and each action profile $a_{0} \in \{A_{1}\}$, the value $u_{t, 0}(p, f, a_{0})$ in $U_{0}$ given by:

$$u_{t, 0}(p, f, a_{0}) = \sum_{a_{0} \in A_{1}} \sum_{f_{0} \in \{F_{1}\}} P(a_{0} | f) P(f' | f, a) u_{t+1, 0}(p, f')$$

where $a = (a_{0}, a_{0})$ is the final action profile of all $k+1$ players (including the chance player), and $u_{t+1, 0}(p, f')$ is the utility of player $p$ at state $f'$. Intuitively, $u_{t, 0}(p, f, a_{0})$ is the expected utility of player $p$ when the joint action $a_{0}$ of the $k$ players is applied on the game position $f$. Based on this value, we can define the utility of each player’s move. To this end, we associate to each player $p$ a constraint $u_{t, p}$ defined over the scope $\{f_{1}, \{A_{1}\}, U_{p}\}$. This constraint maps each state description $f_{1}$ to $\{F_{1}\}$ and each action description $a_{1} \in \{A_{1}\}$ to the value $u_{t, p}(f, a_{1})$ in $U_{p}$ given by:

$$u_{t, p}(f, a_{1}) = \sum_{a_{-1} \in \{A_{1}\}^{-p}} u_{t, 0}(p, f, a_{0})$$

In a symmetrical way, we can also capture the player’s utility of each state. To this end, we associate to each player $p$ a constraint $u_{t, p}$ defined over the scope $\{f_{1}, U_{p}\}$. This constraint maps each state description $f$ to $\{F_{1}\}$ to a value $u_{t, p}(f)$ in $U_{p}$, defined by the following condition: if $f$ is a terminal state, then $u_{t, p}(f)$ is the reward of $p$ at $f$, as specified by the goal rules. Otherwise,

$$u_{t, p}(f) = \bigotimes_{a_{p} \in A_{1}, p} u_{t, p}(f, a_{p})$$

Finally, the optimal play is captured by simply adding the equality constraints $u_{t, p}(f) = u_{t, p}(f, a_{p})$ defined over the scope $\{U_{p}\}$. These equalities filter out any player’s move that is not optimal. Various solution concepts can be defined according the operators $\otimes$ and $\otimes$. In our experiments, we use the standard maximin strategy (for all players) given by $\otimes = \max$ and $\otimes = \min$.

5 MAC-UCB-II

Based on a fragment of SCSP for GDL games, we now present our resolution technique called MAC-UCB-II, an extension of the MAC-UCB algorithm [Koriche et al., 2016] for GDL-II. As indicated above, the stochastic constraint network of a GDL program is a sequence of $\mu$SCSPs, each associated with a game round $t$ in $\{1, \ldots, T\}$. For each $\mu$SCSP, in $\{1, \ldots, T\}$, MAC-UCB-II searches the set of feasible policies by splitting the problem into two parts: a CSP and a $\mu$SCSP (smaller than the original one). The first part is solved using the MAC algorithm [Sabin and Freuder., 1994; 1997] and the second part with the FC algorithm dedicated to SCSP [Walsh, 2002]. Then, a sampling with confidence bound (UCB) is performed to estimate the expected utility of each feasible solution of $\mu$SCSP.

Recall that the task of sequential decision associated to a strategic game is an optimization problem. Classically, this problem is addressed by solving a sequence of stochastic satisfaction problems. In practice, each time our player has to play, a slot of time is dedicated to choose the next action in which we solve as much $\mu$SCSPs as possible.

MAC-UCB-II adds conditional probabilities in each $\mu$SCSP, based on the sees rules for each fluent.

Figure 2 represents the constraint network of the $t$th $\mu$SCSP returned by our encoding procedure on the GDL program of Example 1. For the sake of clarity, the identifiers representing variables and domains were renamed: $H$ (heads), $T$ (tails), $U$ (unset) denote the different values of a side of a coin. First, the two variables terminal and score, stand for the end of the game and the remaining reward. These variables are extracted from terminal and goal $(P, V)$ keywords. Their associated domain is boolean for the first one and the set of possible rewards for the second one. From the role keyword, two variables control and control1+ determine the player for the round $t$ and $t+1$. The domain of these variables corresponds to the set of players defined with the role statement. The states of the game for the round $t$ and $t+1$ are represented by the set of variables defined from the next keyword. The domains of these variables corresponds to the different combination of values of the fluents $S$ and $S1S2$. One can extract from the legal keywords the variables choose, and play. Note that choose is a stochastic variable since it corresponds to legal move of the chance player. The second stochastic variable is coins. The conditional probability of the fluent coins, for Alice is directly associated to the corresponding sees rules (sees(alice, coins(S1S2)) ← does(random, choose(S1S2))). indicating that Alice percepts only the first coin when Random chooses its two coins. A terminal constraint links the terminal variable with the two variable coins. The game is ended when no unset value is set to a coin. In the same manner the goal constraint links the side of the different coins with the associated reward. The next constraints allow the game to change from one state ($t$) to another ($t+1$) depending on the chosen action for the player (variable play) and the chosen action for random (variable choose).
Table of Variables and Domains:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminal</td>
<td>{true, false}</td>
</tr>
<tr>
<td>score</td>
<td>{0.5, 100}</td>
</tr>
<tr>
<td>percept</td>
<td>{\text{coins}(H), \text{coins}(T)}</td>
</tr>
<tr>
<td>coin</td>
<td>{H, T, U}</td>
</tr>
<tr>
<td>coins</td>
<td>{HH, HT, TH, TT, UU}</td>
</tr>
<tr>
<td>control</td>
<td>{alice, random}</td>
</tr>
<tr>
<td>choose</td>
<td>{HH, HT, TH, TT, noop}</td>
</tr>
<tr>
<td>play</td>
<td>{H, T, noop}</td>
</tr>
<tr>
<td>coin_{t+1}</td>
<td>{H, T, U}</td>
</tr>
<tr>
<td>coins_{t+1}</td>
<td>{HH, HT, TH, TT, UU}</td>
</tr>
<tr>
<td>control_{t+1}</td>
<td>{alice, random}</td>
</tr>
</tbody>
</table>

Variable Domain

<table>
<thead>
<tr>
<th>terminal</th>
<th>{true, false}</th>
</tr>
</thead>
<tbody>
<tr>
<td>score</td>
<td>{0.5, 100}</td>
</tr>
<tr>
<td>percept</td>
<td>{\text{coins}(H), \text{coins}(T)}</td>
</tr>
<tr>
<td>coin</td>
<td>{H, T, U}</td>
</tr>
<tr>
<td>coins</td>
<td>{HH, HT, TH, TT, UU}</td>
</tr>
<tr>
<td>control</td>
<td>{alice, random}</td>
</tr>
<tr>
<td>choose</td>
<td>{HH, HT, TH, TT, noop}</td>
</tr>
<tr>
<td>play</td>
<td>{H, T, noop}</td>
</tr>
<tr>
<td>coin_{t+1}</td>
<td>{H, T, U}</td>
</tr>
<tr>
<td>coins_{t+1}</td>
<td>{HH, HT, TH, TT, UU}</td>
</tr>
<tr>
<td>control_{t+1}</td>
<td>{alice, random}</td>
</tr>
</tbody>
</table>

terminal constraint

<table>
<thead>
<tr>
<th>coin</th>
<th>coins</th>
<th>terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>HH</td>
<td>true</td>
</tr>
<tr>
<td>H</td>
<td>HT</td>
<td>true</td>
</tr>
<tr>
<td>T</td>
<td>TT</td>
<td>true</td>
</tr>
<tr>
<td>H</td>
<td>HU</td>
<td>false</td>
</tr>
<tr>
<td>H</td>
<td>UH</td>
<td>false</td>
</tr>
<tr>
<td>U</td>
<td>UU</td>
<td>false</td>
</tr>
</tbody>
</table>

goal constraint

<table>
<thead>
<tr>
<th>coin</th>
<th>coins</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>HH</td>
<td>100</td>
</tr>
<tr>
<td>T</td>
<td>TT</td>
<td>100</td>
</tr>
<tr>
<td>H</td>
<td>HT</td>
<td>50</td>
</tr>
<tr>
<td>H</td>
<td>TH</td>
<td>50</td>
</tr>
<tr>
<td>T</td>
<td>HT</td>
<td>50</td>
</tr>
<tr>
<td>T</td>
<td>TH</td>
<td>50</td>
</tr>
<tr>
<td>H</td>
<td>TT</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>HH</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2: A $\mu$SCSP encoding the GDL program of Hidden Matching Pennies. ($H = \text{heads}, T = \text{tails}, U = \text{unset}$)

Figure 3: The tree search of MAC-UCB-II for the hidden matching pennies.

Figure 6: Experimental Results

6 Experimental Results

Based on our framework and algorithm, we now present a series of experimental results conducted on a cluster of Intel Xeon E5-2643 CPU 3.3 GHz with 64 GB of RAM and four threads under Linux. Our framework was implemented in C++.

We have selected 10 deterministic games described in GDL-I from the Tiltyard server\(^1\), and 15 stochastic games in GDL-II including 5 with no sees rules. A majority of GDL-II games

\(^1\text{http://tiltyard.ggp.org/}\)
have been selected from the Dresden server\textsuperscript{2}. Experiments have been performed on a large variety of games for a total amount of 9,500 matches. More detailed information about specific game can be found on \url{boardgamegeek.com}.

Setup. Game competitions have been organized between different general game players. The first player is the multi-player version of the \textit{UCT} algorithm [Sturtevant, 2008], which is the state-of-the-art algorithm for the deterministic games in GGP. The second player is \textit{Sancho} version 1.61c\textsuperscript{3}, a Monte Carlo Tree Search player elaborated by S. Draper and A. Rose, which has won the 2014 International General Game Playing Competition. The third player is [Cazenave, 2015]\textquotesingle s \textit{GRAVE} algorithm, which implements the Generalized Rapid Action Value Estimation technique, a generalization of the RAVE method [Gelly and Silver, 2007] adapted for GGP. Finally, we also compare our player to \textit{CFR} [Shafiei \textit{et al.}, 2009], a GGP implementation of the well-known CounterFactual Regret technique used in partially observable games. We did not use HyperPlayer-II algorithm [Schofield and Thielscher, 2015], because it was not available online during experiments and we can not adapt it to our model.

For all matches, we used the 2015 Tiltyard Open (last international GGP competition) setup: 180 seconds for the start clock and 15 seconds for the play clock.

Our algorithm \textit{MAC-UCB-II} needs to split the play clock time into two parts: exploitation (the \textit{MAC} part) and exploration (the \textit{UCT} part). The same parameters as MAC-UCB in [Koriche \textit{et al.}, 2016] was used (90 % of the time dedicated to exploitation and 10 % to exploration).

For all the GDL games, we realized 100 matches between MAC-UCB-II and each other player. For the sake of fairness, the role of players were exchanged during each match.

Our results are summarized in Table 2. The rows are grouped into 4 parts, respectively capturing GDL-I games, GDL-II games with a random player (but no \texti{sees} rules), GDL-II games with information sharing (all players have the same partial observation of the game state) and one-player GDL-II games with \textit{sees} rules. Each column reports the average percent of wins for MAC-UCB-II against the selected adversary. For example, the entry of the \textit{UCT} column for the Chess game (3rd row) indicates that, on average, MAC-UCB-II wins 58\% of contests against \textit{UCT}. Since the fourth group only involves one-player games, the columns (excluding MAC-UCB-II) report the average number of times the one-player game is solved, divided by the total number of matches.

Results on GDL-I. For deterministic games, \textit{UCT} and \textit{CFR} were defeated by MAC-UCB-II, with sometimes a score greater than 80 \% (e.g. Reversi Suicide). \textit{Sancho}, the IGPG\textsuperscript{C}14 leader, and \textit{GRAVE} are on average less competitive than MAC-UCB-II. Notable examples are Hex, Connect Four 20x20 against \textit{Sancho} and Chess against GRAVE. The only exceptions are the Amazons torus 10x10 against GRAVE with a score on average of 46 \% and for the Breakthrough suicide against Sancho with 49 \%.

Results on GDL-II. For the GDL-II games with no \textit{sees} rules but a random player, the four opponents are beaten with a score higher than 60\% for MAC-UCB-II against Sancho, UCT, CFR, and higher than 50\% against GRAVE.

For games with partial observations, Sancho does not participate because it is dedicated to GDL-I (modulo a possible simulation of the chance player). For the puzzle games with imperfect information, MAC-UCB-II is the best player for those games except for the Wumpus or GuessDice. However, the GuessDice is not significant because there is no strategy to win, but just chance to guess the Dice. We can note that on the Mastermind, MAC-UCB-II and CFR obtain an equivalent score.

Finally, the last five games are partially observable games with information sharing. For instance, the TicTacToe Latent Random 10x10 involves 3 players, including the chance player. The chance player randomly places a cross or a round in the free cases.

<table>
<thead>
<tr>
<th>Game</th>
<th>UCT</th>
<th>CFR</th>
<th>GRAVE</th>
<th>Sancho</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazons torus 10x10</td>
<td>62</td>
<td>73</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>Breakthrough suicide</td>
<td>71</td>
<td>80</td>
<td>54</td>
<td>49</td>
</tr>
<tr>
<td>Chess</td>
<td>58</td>
<td>82</td>
<td>71</td>
<td>53</td>
</tr>
<tr>
<td>Connect Four 20x20</td>
<td>76</td>
<td>83</td>
<td>51</td>
<td>72</td>
</tr>
<tr>
<td>English Draughts</td>
<td>69</td>
<td>78</td>
<td>55</td>
<td>51</td>
</tr>
<tr>
<td>Hex</td>
<td>81</td>
<td>76</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Shmup</td>
<td>75</td>
<td>66</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>Skirmish zero-sum</td>
<td>63</td>
<td>77</td>
<td>63</td>
<td>59</td>
</tr>
<tr>
<td>TTCC4 2P</td>
<td>79</td>
<td>83</td>
<td>50</td>
<td>54</td>
</tr>
<tr>
<td>Reversi Suicide</td>
<td>86</td>
<td>86</td>
<td>57</td>
<td>53</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game</th>
<th>UCT</th>
<th>CFR</th>
<th>GRAVE</th>
<th>Sancho</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backgammon</td>
<td>70.0</td>
<td>66.7</td>
<td>54.6</td>
<td>97.5</td>
</tr>
<tr>
<td>Can\textsuperscript{'}s Stop</td>
<td>73.1</td>
<td>67.5</td>
<td>62.1</td>
<td>94.3</td>
</tr>
<tr>
<td>Kaseklu</td>
<td>73.6</td>
<td>71.5</td>
<td>56.2</td>
<td>80.3</td>
</tr>
<tr>
<td>Pickomino</td>
<td>65.2</td>
<td>60.6</td>
<td>60.2</td>
<td>74.2</td>
</tr>
<tr>
<td>Yahtzee</td>
<td>72.1</td>
<td>72.3</td>
<td>53.9</td>
<td>72.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game</th>
<th>UCT</th>
<th>CFR</th>
<th>GRAVE</th>
<th>Sancho</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacman</td>
<td>57.2</td>
<td>58.9</td>
<td>55.1</td>
<td></td>
</tr>
<tr>
<td>Schnappt Hubi</td>
<td>71.3</td>
<td>57.5</td>
<td>56.2</td>
<td></td>
</tr>
<tr>
<td>Sheep &amp; Wolf</td>
<td>68.2</td>
<td>56.2</td>
<td>55.0</td>
<td></td>
</tr>
<tr>
<td>TicTacToe Latent Random 10x10</td>
<td>82.6</td>
<td>78.7</td>
<td>69.1</td>
<td></td>
</tr>
<tr>
<td>War (card game)</td>
<td>72.0</td>
<td>69.1</td>
<td>66.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game</th>
<th>MAC-UCB-II</th>
<th>UCT</th>
<th>CFR</th>
<th>GRAVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GuessDice</td>
<td>15</td>
<td>15.7</td>
<td>16</td>
<td>16.5</td>
</tr>
<tr>
<td>MasterMind</td>
<td>67.8</td>
<td>53.8</td>
<td>68.1</td>
<td>60.1</td>
</tr>
<tr>
<td>Monty Hall</td>
<td>65.2</td>
<td>62.2</td>
<td>63.1</td>
<td>64.3</td>
</tr>
<tr>
<td>Vacuum Cleaner Random</td>
<td>61.5</td>
<td>34</td>
<td>46</td>
<td>58.8</td>
</tr>
<tr>
<td>Wumpus</td>
<td>32.1</td>
<td>40.1</td>
<td>44.1</td>
<td>51.2</td>
</tr>
</tbody>
</table>
and the other two players observe the squares marked by chance, only when they try to mark it. Schnappt Hubi and Sheep & Wolf are cooperative games, where all agent share their observation in order to beat the chance player. MAC-UCB-II wins with an average score higher than 55% against each opponent, with an average number of wins of about 69% for the TicTacToe Latent Random 10x10 game.

7 Conclusion
This paper has presented a consequent forward step on using constraint-based formalisms for GGP by considering a large subclass of GDL-II games. We have identified an important fragment of imperfect information games that can be cast as SCSPs, and which can be solved using general-purpose Constraint Programming techniques. Our sequential decision algorithm for GDL-II games exploits constraint propagation and Monte Carlo sampling. Based on extensive experiments involving various types of games and computer opponents, we showed that general-purpose CP techniques are paying off.

A work in progress is to focus on symmetries in order to strongly decrease the search space. The idea is to exploit symmetries to avoid the useless resolution of symmetrical μSCSPs. For that, we need to unify and extend the constraint approaches [Cohen et al., 2006] and the ones in GGP [Schiffel, 2010].

References
Generating Sokoban Puzzle Game Levels with Monte Carlo Tree Search

Bilal Kartal, Nick Sohre, and Stephen J. Guy
Department of Computer Science and Engineering
University of Minnesota
(bilal,sohre,sjguy)@cs.umn.edu

Abstract
In this work, we develop a Monte Carlo Tree Search based approach to procedurally generate Sokoban puzzles. To this end, we propose heuristic metrics based on surrounding box path congestion and level tile layout to guide the search towards interesting puzzles. Our method generates puzzles through simulated game play, guaranteeing solvability in all generated puzzles. Our algorithm is efficient, capable of generating challenging puzzles very quickly (generally in under a minute) for varying board sizes. The ability to generate puzzles quickly allows our method to be applied in a variety of applications such as procedurally generated mini-games and other puzzle-driven game elements.

1 Introduction
Understanding and exploring the inner workings of puzzles has exciting implications in both industry and academia. Many games have puzzles either at their core (e.g. Zelda: Ocarina of Time, God of War) or as a mini-game (e.g. lock picking and terminal hacking in Fallout 4 and Mass Effect 3). Generating these puzzles automatically can reduce bottlenecks in design phase, and help keep games new, varied, and exciting.

In this paper, we focus on the puzzle game of Sokoban. Developed for the Japanese game company Thinking Rabbit in 1982, Sokoban involves organizing boxes by pushing them with an agent across a discrete grid board. Sokoban is well suited for consideration for several reasons. A well-known game, Sokoban exhibits many interesting challenges inherent in the general field of puzzle generation. For example, the state space of possible configurations is very large (exponential in the size of the representation), and thus intractable for search algorithms to traverse. Consequently, ensuring generated levels are solvable can be difficult to do quickly. Furthermore, it is unclear how to characterize what makes an initial puzzle state lead to an interesting or non-trivial solution. While Sokoban has relatively straightforward rules, even small sized puzzles can present a challenge for human solvers.

In this paper, we propose a method to procedurally generate Sokoban puzzles. Our method produces a wide range of

Figure 1: One of the highest scoring 5x5 puzzles generated by our method. The goal of the puzzle is to have the agent push boxes (brown squares) such that all goals (yellow discs) are covered by the boxes. Yellow filled boxes represent covered goals. Obstacles (gray squares) block both agent and box movement (Sprites from The Open Bundle1).

GIGA’16 Proceedings 47

1http://open.commonly.cc
To that end, we propose the use of Monte Carlo Tree Search (MCTS) for this puzzle generation. We show that the generation of Sokoban puzzles can be formulated as an optimization problem, and apply MCTS guided by an evaluation metric to estimate puzzle difficulty. Furthermore, we model the MCTS search as an act of simulated gameplay. This alleviates current bottlenecks by eliminating the need to verify the solvability of candidate puzzles post-hoc. Overall, the contributions of this work are three-fold:

- We formulate the generation of Sokoban puzzles as an MCTS optimization problem.
- We propose a heuristic metric to govern the evaluation for the MCTS board generation and show that it produces puzzles of varying difficulty.
- Our method eliminates the need to check post-hoc for board solvability, while maintaining the guarantee that all of our levels are solvable.

2 Background

There have been many applications of Procedural Content Generation (PCG) methods to puzzle games, such as genetic algorithms for Spelunky [Baghdadi et al., 2015], MCTS based Super Mario Bros [Summerville et al., 2015], and map generation for Physical TSP problem [Perez et al., 2014b] and video games [Snodgrass and Ontanon, 2015]. Other approaches proposed search as a general tool for puzzle generation [Sturtevant, 2013], and generation of different start configurations for board games to tune difficulty [Ahmed et al., 2015]. Recent work has looked at dynamically adapting games to player actions [Stammer et al., 2015]. Smith and Mateas (2011) proposed an answer set programming based paradigm for PCGs for games and beyond. A recent approach parses game play videos to generate game levels [Guzdial and Riedl, 2015]. Closely related to our work, Shaker et al. (2015) proposed a method for the game of Cut the Rope where the simulated game play is used to verify level playability. We refer readers to the survey [Togelius et al., 2011] and the book [Shaker et al., 2014] for a more comprehensive and thorough discussion of the PCG field, and to the survey particularly for PCG puzzles [Khalifa and Fayek, 2015].

2.1 Sokoban Puzzle

A Sokoban game board is composed of a two-dimensional array of contiguous tiles, each of which can be an obstacle, an empty space, or a goal. Each goal or space tile may contain at most one box or the agent. The agent may move horizontally or vertically, one space at a time. Boxes may be pushed by the agent, at most one at a time, and neither boxes nor the agent may enter any obstacle tile. The puzzle is solved once the agent has arranged the board such that every tile that contains a goal also contains a box. We present an example solution to a Sokoban puzzle level in Figure 2.

Previous work has investigated various aspects of computational Sokoban including automated level solving, level generation, and assessment of level quality.

Sokoban Solvers

Previously proposed frameworks for Sokoban PCG involve creating many random levels and analyzing the characteristics of feasible solutions. However, solving Sokoban puzzles has been shown to be PSPACE-complete [Culberson, 1999]. Several approaches have focused on proposing approximate solutions to reduce the effective search domain [Botea et al., 2002; Junghanns and Schaeffer, 2001; Cazenave and Jouandeau, 2010].

Recently, Pereira et al. (2015) have proposed an approach that uses pattern databases [Edelkamp, 2014] for solving Sokoban levels optimally, finding the minimum necessary number of box pushes (regardless of agent moves). The authors in [Perez et al., 2014a] applied MCTS for solving Sokoban levels, but concluded that pure MCTS performs poorly.

Level Generation

While there have been many attempts for solving Sokoban puzzles, the methods for their procedural generation are less explored. To the best of our knowledge, Murase et al. (1996) proposed the first Sokoban puzzle generation method which firstly creates a level by using templates, and proceeds with an exponential time solvability check. More recently, Taylor and Parberry (2011) proposed a similar approach, using templates for empty rooms and enumerating box locations in a brute-force manner. Their method can generate compelling levels that are guaranteed to be solvable. However, the run-time is exponential, and the method does not scale to puzzles with more than a few boxes.

Level Assessment

There have been several efforts to assess the difficulty of puzzle games. One example is the very recent work by [van Kreveld et al., 2015], which combines features common to puzzle games into a difficulty function, which is then tuned.
using user study data. Others consider Sokoban levels specifically, comparing heuristic based problem decomposition metrics with user study data [Jarůšek and Pelánek, 2010], and using genetic algorithm solvers to estimate difficulty [Ashlock and Schonfeld, 2010]. More qualitatively, Taylor et al. (2015) have conducted a user-study and concluded that computer generated Sokoban levels can be as engaging as those designed by human experts.

2.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a best-first search algorithm that has been successfully applied to many games [Cazenave and Saffidine, 2010; Pepels et al., 2014; Jacobsen et al., 2014; Frydenberg et al., 2015; Mirsoleimani et al., 2015; Sturtevant, 2015; Steinmetz and Gini, 2015] and a variety planning domains such as multi-agent narrative generation [Kartal et al., 2014], multi-robot patrolling [Kartal et al., 2015] and task allocation [Kartal et al., 2016], and others [Williams et al., 2015; Sabar and Kendall, 2015; Hennes and Izzo, 2015]. Bauters et al. (2016) show how MCTS can be used for general MDP problems. More recently, Zook et al. (2015) adapted MCTS such that it simulates different skilled humans for games enabling faster gameplay data collection to automate game design process. We refer the reader to the survey on MCTS [Browne et al., 2012].

MCTS proceeds in four phases of selection, expansion, rollout, and backpropagation. Each node in the tree represents a complete state of the domain. Each link in the tree represents one possible action from the set of valid actions in the current state, leading to a child node representing the resulting state after applying that action. The root of the tree is the initial state, which is the initial configuration of the Sokoban puzzle board including the agent location. The MCTS algorithm proceeds by repeatedly adding one node at a time to the current tree. Given that actions from the root to the expanded node is unlikely to find a complete solution, i.e., a Sokoban puzzle for our purposes, MCTS uses random actions, a.k.a. rollouts. Then the full action sequence, which results in a candidate puzzle for our domain, obtained from both tree actions and random actions is evaluated. For each potential action, we keep track of how many times we have tried that action, and what the average evaluation score was.

Exploration vs. Exploitation Dilemma

Choosing which child node to expand (i.e., choosing which action to take) becomes an exploration/exploitation problem. We want to primarily choose actions that had good scores, but we also need to explore other possible actions in case the observed empirical average scores don’t represent the true reward mean of that action. This exploration/exploitation dilemma has been well studied in other areas.

Upper Confidence Bounds (UCB) [Auer et al., 2002] is a selection algorithm that seeks to balance the exploration/exploitation dilemma. Using UCB with MCTS is also referred to as Upper Confidence bounds applied to Trees (UCT). Applied to our framework, each parent node \( p \) chooses its child \( s \) with the largest \( UCB(s) \) value according to Eqn. 1. Here, \( w(\cdot) \) denotes the average evaluation score obtained by Eqn. 2, \( \hat{\pi}_s \) is the parent’s updated policy that includes child node \( s \), \( p_v \) is visit count of parent node \( p \), and \( s_v \) is visit count of child node \( s \) respectively. The value of \( C \) determines the rate of exploration, where smaller \( C \) implies less exploration. \( C = \sqrt{2} \) is necessary for asymptotic convergence of MCTS [Kocsis and Szepesvári, 2006].

\[
UCB(s) = w(\hat{\pi}_s) + C \times \sqrt{\frac{\ln p_v}{s_v}}
\]  

(1)

If a node with at least one unexplored child is reached \( (s_v = 0) \), a new node is created for one of the unexplored actions. After the rollout and back-propagation steps, the selection step is restarted from the root again. This way, the tree can grow in an uneven manner, biased towards better solutions.

Variations in Selection Methods. There are numerous other selection algorithms that can be integrated to MCTS. In this work, as a baseline, we employed UCB selection algorithm. However, considering a possible relationship between the variance of tree nodes and agent movement on the board, we experimented with UCB-Tuned [Auer et al., 2002], and UCB-V [Audibert et al., 2007] for Sokoban puzzle generation using our evaluation function shown in Eqn. 2.

UCB-Tuned and UCB-V both employ the empirical variance of nodes based on the rewards obtained from rollouts with the intuition that nodes with high variance need more exploration to better approximate their true reward mean. UCB-Tuned purely replaces the exploration constant of the UCB algorithm with an upper bound on the variance of nodes, and hence requires no tuning, whereas UCB-V has two additional parameters to control the rate of exploration.

3 Approach Overview

One of the challenges for generating Sokoban puzzles is ensuring solvability of the generated levels. Since solving Sokoban has been shown to be PSPACE-complete, directly checking whether a solution exists for a candidate puzzle becomes intractable with increasing puzzle size. To overcome this challenge, we exploit the fact that a puzzle can be generated through simulated gameplay itself.

To do so, we decompose the puzzle generation problem into two phases: puzzle initialization and puzzle shuffling. Puzzle initialization refers to assigning the box start locations, empty tiles, and obstacle tiles. Puzzle shuffling consists of performing sequences of Move agent actions (listed in section 3.1) to determine goal locations. In a forward fashion, as the agent moves around during the shuffling phase, it pushes boxes to different locations. The final snapshot of the board after shuffling defines goal locations for boxes.

We apply MCTS by formulating the puzzle creation problem as an optimization problem. As discussed above, the search tree is structured so that the game can be generated by simulated gameplay. The search is conducted over puzzle initializations and valid puzzle shuffles. Because puzzle shuffles are conducted via a simulation of the Sokoban game rules, invalid paths are never generated. In this way our method is guaranteed to generate only solvable levels.

The main reasons for using MCTS for Sokoban puzzle generation is its success in problems with large branching factors and its anytime property. MCTS has been applied to many
problems with large search spaces [Browne et al., 2012]. This is also the case for Sokoban puzzles as the branching factor is in the order of $O((mn)$ for an $m \times n$ puzzle.

Anytime algorithms return a valid solution (given a solution is found) even if it is interrupted at any time. Given that our problem formulation is completely deterministic, MCTS can store the best found puzzle after rollouts during the search and optionally halt the search with some quality threshold. This behavior also enables us to create many puzzle levels from a single MCTS run with varying increasing scores.

### 3.1 Action set

Our search tree starts with a board fully tiled with obstacles, except for an agent which is assumed to start at the center of the board. At any node, the following actions are possible in the search tree:

1. **Delete obstacles**: Initially, only the obstacle tiles surrounding the agent are available for deletion. Once there is an empty tile, its surrounding obstacle tiles can also be turned into an empty tile (this progressive obstacle deletion prevents boards from containing unreachable hole shaped regions).

2. **Place boxes**: A box may be placed in any empty tile.

3. **Freeze level**: This action takes a snapshot of the board and saves it as the start configuration of the board.

4. **Move agent**: This action moves the agent on game board. The agent cannot move diagonally. This action provides the *shuffling* mechanism of the initialized puzzle where the boxes are pushed around to determine box goal positions.

5. **Evaluate level**: This action is the terminal action for any action chain; it saves the shuffled board as the solved configuration of the puzzle (i.e. current box locations are saved as goal locations).

This action set separates the creation of initial puzzle configurations (actions taken until the level is frozen) from puzzle shuffling (agent movements to create goal positions). Before move actions are created as tree nodes during MCTS simulations, the agent moves randomly during rollouts. As search is deepened, agent moves become more non-random.

Once *Evaluate level* action is chosen, before we apply our evaluation function to the puzzle, we apply a simple post-processing to the board. We turn all boxes that are placed to the board but never pushed by the agent into obstacles as this doesn’t violate any agent movement actions. By applying evaluation function after post-processing, we make sure our heuristic metrics’ values are correctly computed.

The proposed action set has several key properties which make it well suited for MCTS-based puzzle generation. In contrast to our approach, interleaving puzzle creation and agent movement requires the MCTS method to re-simulate all actions from the root to ensure valid action sequences, which would render the problem much more computationally expensive and MCTS less efficient.

### 3.2 Evaluation Function

MCTS requires an evaluation function to optimize over during the search. For game playing AI, evaluation functions typically map to 0 for loss, 0.5 for tie, and 1 for winning sequence of actions. For Sokoban puzzle generation, this is not directly applicable as we do not have winning and losing states. Instead, we propose to use a combination of two metrics, i.e. terrain and congestion, to generate interesting Sokoban levels. Our approach seeks to maximize the geometric mean of these two metrics as shown in Eqn. 2. This provides a good balance between these two metrics without allowing either term dominate. Parameter $k$ is employed to normalize scores to the range of 0 to 1.

$$f(P) = \sqrt{\text{Terrain} \times \text{Congestion}}$$

These two components of our evaluation function are intended to guide the search towards boards that have congested landscapes with more complex box interactions.

#### Congestion Metric

The motivation for the congestion metric is to encourage puzzles that have some sort of congestion with respect to the paths from boxes to their goals. The intuition is that overlaps between box paths may encourage precedence constraints of box pushes. We compute this by counting the number of boxes, goals, and obstacles in between each box and its corresponding goal. Formally, given an $m \times n$ Sokoban puzzle $P$, and given $b$ boxes on the board, let $b^i$ denote the initial box locations, and let $b^f$ denote the final box locations. For each box $b_i$, we create a minimum area rectangle $r_i$ with corners of $b_i^*$ and $b_i^f$. Within each rectangle $r_i$, let $s_i$ denote the number of box locations, let $g_i$ denote the number of goal locations, and let $o_i$ denote the number of obstacles. Then

$$\text{Congestion} = \sum_{i=1}^{b} (\alpha s_i + \beta g_i + \gamma o_i).$$

where $\alpha$, $\beta$, and $\gamma$ are scaling weights.

#### Terrain Metric

The value of *Terrain* term is computed by summing the number of obstacle neighbors of all empty tiles. This is intended to reward boards that have heterogeneous landscapes, as boards having large clearings or obstacles only on the borders make for mostly uninteresting puzzles.

### 4 Results

In this section, we firstly present a comparison between different selection algorithms. Then, we report the anytime performance of our approach. Lastly, we present a set of levels generated by our approach.

#### 4.1 Experimental Design

All experiments are performed on a laptop using a single core of an Intel i7 2.2 GHz processor with 8GB memory. Search time is set to 200 seconds for all results and levels reported in
this work. Given the simple game mechanics of Sokoban puzzles, our algorithm on average can perform about 80K MCTS simulations per second for 5x5 tile puzzles.

For our experiments, we set $C = \sqrt{2}$ for UCB which is presented in Eqn. 1. For the two parameters involved with UCB-V, we use those suggested in [Audibert et al., 2007]. For the congestion metric (Eqn. 3), we used the following weights for all boards generated: $\alpha = 4$, $\beta = 4$, and $\gamma = 1$. The normalization constant of $k = 200$ is employed for all experiments. We ran our experiments for 5 runs of varying random seeds across each board size generated (5x5 to 10x10). Boards were saved each time the algorithm encountered an improvement in the evaluation function.

### 4.2 Computational Performance

Our method was able to produce approximately 700 candidate puzzles (about 100 per board size) on a single processor over a period of 6 hours. Because of the anytime nature of our approach, non-trivial puzzles are generated within a few seconds, and more interesting puzzles over the following minutes (see Figure 3). The anytime behavior of our algorithm for different puzzle board sizes is depicted in Figure 4. A puzzle set is shown in Figure 5.

We compare the results of different MCTS selection methods in Figure 6. While UCB-Tuned has been shown to outperform these selection algorithms for other applications [Perick et al., 2012], our experiments reveal a slight advantage for UCB-V. Although all selection methods perform statistically similarly, UCB-V slightly outperforms other techniques consistently across different tested board sizes.

Compared to other methods that are exponential in the number of boxes, our method is a significant improvement on the run time. Our algorithm is capable of quickly generating levels with a relatively large number of boxes. As can be seen in Figure 5, puzzles with 8 boxes or more are found within a few minute time span. Previous search-based methods may not be able to generate these puzzles in a reasonable amount of time given the number of boxes and board size.
Figure 5: A variety of puzzles with different board sizes. Our algorithm makes use of larger amounts of board space in optimizing the evaluation function, leading to interesting box interactions in all the various sizes. Score value is computed by our evaluation function, and time refers to how many seconds it takes for MCTS to generate the corresponding puzzle.

Figure 6: Different MCTS selection algorithms are compared. UCB-V slightly outperforms UCB and UCB-Tuned algorithms. All algorithms were provided the same search budget. The results are obtained by averaging 5 runs with different seeds.

4.3 Generated Levels

Our algorithm is capable of generating interesting levels of varying board sizes. Figure 5 showcases some examples. Our algorithm does not require any templates, generating the levels from an initially obstacle-filled board. Figure 3 shows how board quality evolves over time. Lower scoring boards are quickly ignored as the search progresses towards more interesting candidates. The leftmost board depicted in Figure 3, while only 5x5 tiles, presented a challenge for the authors, taking over a minute to solve. As MCTS is a stochastic search algorithm, we obtain different interesting boards when we initialize with different random seeds.

5 Discussion

The difficulty of a Sokoban puzzle is not easily quantified (even knowing the optimal solution would not make determining relative difficulty easy). Some properties of the scores produced by our evaluation function are worth noting. In general, higher scores led to higher congestion (more box path interactions) and a larger number of required moves. This does lead to more challenging and engaging puzzles as the score values increase. However, there are other aspects of difficulty that are not explicitly captured by our metrics. A level that requires more moves than another doesn’t necessarily mean it is more difficult to recognize the solution (e.g., having to move 3 boxes 1 space each vs 10 boxes 1 space each). Examples exist in which a very challenging level has a much lower score than a “busier” but otherwise menial one. Additionally, the box path interactions in our challenging puzzles usually correspond to determining a small number key moves, after which the puzzle is easily solved. In contrast, human-designed puzzles can require the player to move boxes carefully together throughout the entire solution.

A key motivation of our work is the ability to use our generated levels for games that have puzzle mechanics as an integral part of gameplay. Here, the ability to generate the puzzles behind these elements could make the entire game experience different on every play through. Additionally, levels of varying difficulties could be generated to represent mini-games corresponding to various skill levels. For example, one could imagine the reskinning of Sokoban puzzles as a “hacking” mini-game where the difficulty can be tuned to the desired challenge based on contextual information.

The properties of our algorithm in particular make it well suited to the aforementioned applications. Since the speed of our method could allow new puzzle to be generated for every instance, players could be prevented from searching for solutions on game forums (if that property is desired). Additionally, an evaluation metric that corresponds well to puzzle difficulty could allow the game to dynamically present puzzles of various difficulty to the player based on their character’s profile (e.g., “hacking skill”) or other contextual information. Finally, the anytime nature of the algorithm allows for the generation of multiple levels of varying difficulty in a single run, which could be stored for later use in the game.

Our method has the potential to generalize to other puzzle games or games of other genres containing analogous mechanics. First we must assert that the game to be generated has some finite environment whose elements can be represented discretely. Then the initial phase of our method may be applied simply by changing the action set to include actions that manipulate the elements of the environment. In many puzzle games, the game is won by manipulating the environment through player actions such that the elements match some goal state conditions. The efficiency of our method comes from exploiting this property. To apply the second phase of our method, one need simply change the action set of the Sokoban agent to the action set available to the player.
during game play. Given these and a simulation framework, our method will generate solvable puzzles. Finally, an evaluation function must be carefully designed to produce the desired puzzle qualities. This is typically the most difficult part of applying our method to new puzzle types. The evaluation function needs to be specific to the game being generated, and must balance between being efficient to compute but still predictive of the desired difficulty of the puzzle.

6 Conclusion

In this work, we have proposed and implemented an MCTS approach for the fast generation of Sokoban puzzles. Our algorithm requires no human designed input, and is guaranteed to produce solvable levels. We developed and utilized heuristic evaluation function that separates trivial and uninteresting levels from non-trivial ones, enabling the generation of challenging puzzles. We have shown, with examples, that our method can be used to generate puzzles of varying sizes.

Limitations: While our method can be used as is to generate interesting levels, there are areas in which the method could potentially be improved. No structured evaluation of our proposed metrics’ ability to select interesting levels has been performed. While we have presented levels that were challenging to us, we have not tested the validity of these metrics statistically.

Future Work: We plan to pursue the extension and improvement of this work in two main directions. One is to deepen our knowledge of puzzle features that can be efficiently computed and reused within the puzzle generation phase while the other is improving the performance of the algorithm. To validate our evaluation function, we plan to conduct a user study to collect empirical annotations of level difficulty, as well as other characteristics. Having humans assess our puzzles would allow the testing of how well our evaluation function corresponds to the puzzle characteristics as they are perceived. To begin overcoming the challenge of better understanding puzzle difficulty, we must expand our notion of what makes a puzzle engaging or challenging. This could potentially be achieved by using machine learning methods to identify useful features on both human designed and generated puzzles with annotated difficulty. To increase the scalability and performance of our approach, we plan to parallelize MCTS potentially with the use of GPUs.

References


An Investigation into the Effectiveness of Heavy Rollouts in UCT

Steven James\textsuperscript{1}, Benjamin Rosman\textsuperscript{1,2} & George Konidaris\textsuperscript{3}
\textsuperscript{1}University of the Witwatersrand, Johannesburg, South Africa
\textsuperscript{2}Council for Scientific and Industrial Research, Pretoria, South Africa
\textsuperscript{3}Duke University, Durham NC 27708, USA

steven.james@students.wits.ac.za, brosman@csir.co.za, gdk@cs.duke.edu

Abstract
Monte Carlo Tree Search (MCTS) is a family of directed search algorithms that has gained widespread attention in recent years, with its domain-independent nature making it particularly attractive to fields such as General Game Playing. Despite the vast amount of research into MCTS, the dynamics of the algorithm are still not yet fully understood. In particular, the effect of using knowledge-heavy or biased rollouts in MCTS still remains largely unknown, with surprising results demonstrating that better-informed rollouts do not necessarily result in stronger agents. We show that MCTS is well-suited to a class of domains possessing a smoothness property, and that any error due to incorrect bias is compounded in non-smooth domains, particularly for low-variance simulations.

1 Introduction
Monte Carlo Tree Search (MCTS) has found great success in a number of seemingly unrelated applications, ranging from Bayesian reinforcement learning [Guez et al., 2013] to Ms Pac-Man [Pepels et al., 2014]. Originally developed to tackle the game of Go [Coulom, 2007], it is often applied to domains for which only low-quality heuristics exist. MCTS combines a traditional tree search with Monte Carlo simulations (also known as rollouts), and uses the outcome of these simulations to evaluate states in a lookahead tree. That MCTS requires neither expert knowledge nor heuristics makes it a powerful general-purpose approach, particularly relevant to tasks such as General Game Playing [Genesereth et al., 2005]. It has also shown itself to be a flexible planner, recently combining with deep neural networks to achieve super-human performance in the game of Go [Silver et al., 2016].

While many variants of MCTS exist, the UCT (Upper Confidence bound applied to Trees) algorithm [Kocsis and Szepesvári, 2006] is widely used in practice, despite its shortcomings [Domshlak and Feldman, 2013]. A great deal of analysis on UCT revolves around the tree-building phase of the algorithm, which provides theoretical convergence guarantees and upper-bounds on the regret [Coquelin and Munos, 2007]. Less is known about the simulation phase.

UCT calls for this phase to be performed by randomly selecting actions until a terminal state is reached. The outcome of the simulation is then propagated to the root of the tree. Averaging these results over many iterations provides a fairly accurate measure of the strength of the initial state, despite the fact that the simulation is completely random. As the outcome of the simulations directly affects the entire algorithm, one might expect that the manner in which they are performed has a major effect on the overall strength of the algorithm.

A natural assumption to make is that completely random simulations are not ideal, since they do not map to realistic actions. A different approach is that of so-called heavy rollouts, where moves are intelligently selected using domain-specific rules or knowledge. Counterintuitively, some results indicate that using these stronger rollouts can actually result in a decrease in overall performance [Gelly and Silver, 2007].

While UCT is indeed domain-independent, it cannot be simply seen as a panacea—Ramanujan et al. [2011] demonstrates its poor performance in chess, for example. Coupled with the above results regarding heavy playouts, this raises two questions: when is UCT a good choice of algorithm and what is the effect of non-uniformly random rollouts?

There are a number of conflating factors that make analysing UCT in the context of games difficult, especially in the multi-agent case. These include the strength of our opponents and whether they adapt their policies in response to our own, as well as the additional complexity of the rollout phase, which now requires policies for multiple players. Aside from Silver and Tesauro [2009] who propose the concept of simulation balancing to learn a Go rollout policy that is weak but “fair” to both players, there is little to indicate how best to simulate our opponents. Furthermore, the vagaries of the domain itself can often add to the complexity—Nau [1983] demonstrates how making better decisions throughout a game does not necessarily result in the winning rate that should be expected. Given all of the above, we choose to simplify matters by restricting our investigation to the single-agent case.

We propose that a key characteristic of a domain is its smoothness, and then demonstrate that UCT is well-suited to domains possessing this property. We provide results which show that biased rollouts can indeed improve performance, but identify high-bias, low-variance rollout policies as potentially dangerous choices that can lead to worse performance. This is further compounded in non-smooth domains.
2 Background
2.1 Markov Decision Process
A Markov Decision Process (MDP) is defined by the tuple $(S,A,T,R,\gamma)$ over states $S$, actions $A$, transition function $T : S \times A \times S \rightarrow [0,1]$, reward function $R : S \times A \times S \rightarrow \mathbb{R}$ and discount factor $\gamma \in (0,1)$ [Sutton and Barto, 1998].

Suppose that after time step $t$ we observe the sequence of rewards $r_{t+1},r_{t+2},r_{t+3},\ldots$. For episodic tasks, a finite number of rewards will be observed. In general, we wish to maximise our expected return $\mathbb{E}[R_t]$, where $R_t = \sum_{i=1}^N \gamma^{i-1}r_{t+i}$ represents the discounted sum of the rewards.

A policy $\pi : S \times A \rightarrow [0,1]$ is a mapping that specifies the probability of executing an action in a given state. For a policy $\pi$, the value of a state is the expected reward gained by following $\pi$:

$$V^\pi(s) = \mathbb{E}_\pi[R_t | s_t = s].$$

A policy $\pi^*$ is optimal if $\forall s \in S$, $V^\pi(s) = \max_\pi V^\pi(s)$.

2.2 Monte Carlo Tree Search
MCTS iteratively builds a search tree by executing four phases (Figure 1). Each node in the tree represents a single state, while the tree’s edges correspond to actions. In the selection phase, a child-selection policy is recursively applied until a leaf node is reached.

![Figure 1: Phases of the Monte Carlo tree search algorithm.](image)

UCT uses a policy known as UCB1, a well-known solution to the multi-armed bandit problem [Auer et al., 2002]. At each state $s$, we store the visitation count $n_s$ and average return $X_s$. For a given node $s$, the policy then selects child $i$ that maximises the upper confidence bound

$$X_i + C_p \sqrt{\frac{2\ln(n_s)}{n_i}},$$

where $C_p$ is an exploration parameter.

Once a leaf node is reached, the expansion phase adds a new node to the tree. A simulation is then run from this node according to the rollout policy, with the outcome being backpropagated up through the tree, updating the nodes’ average scores and visitation counts.

This cycle of selection, expansion, simulation and back-propagation is repeated until some halting criteria is met, at which point the best action (usually that which leads to the most visited state) is selected.

2.3 Smoothness
There is some evidence to suggest that the key property of a domain is the smoothness of its underlying value function. The phenomenon of game tree pathology [Nau, 1982], as well as work by Ramanujan et al. [2011], advance the notion of trap states, which occur when the value of two sibling nodes differs greatly. It is thought that UCT is unsuited to domains possessing many such states. Furthermore, in the context of $X$-armed bandits (where $X$ is some measurable space), UCT can be seen as a specific instance of the Hierarchical Optimistic Optimisation algorithm, which attempts to find the global maximum of the expected payoff function using MCTS. Its selection policy is similar to UCB1, but contains an additional term that depends on the smoothness of the function. For an infinitely smooth function, this term goes to 0 and the algorithm becomes UCT [Bubeck et al., 2011].

In defining what is meant by smoothness, one notion that can be employed is that of Lipschitz continuity, which limits the rate of change of a function. Formally, a value function $V$ is $M$-Lipschitz continuous if $\forall s,t \in S$,

$$|V(s)-V(t)| \leq Md(s,t),$$

where $M \geq 0$ is a constant, $d(s,t) = ||k(s)-k(t)||$ and $k$ is a mapping from state space to some vector space.

3 Function Optimisation
Reasoning about the smoothness (or lack thereof) of an MDP is difficult for all but the vaguest of statements. To develop some method of controlling and visualising the smoothness, we consider the task of finding the global maximum of a function. Simple, monotonic functions can be seen as representing smooth environments, while complicated ones represent non-smooth domains.

For simplicity, we constrain the domain and range of the functions to be in the interval $[0,1]$. Each state represents some interval $[a,b]$ within this unit square, with the starting state representing $[0,1]$. We assume that there are two available actions at each state: the first results in a transition to a new state $[a,\frac{a+b}{2}]$, while the second transitions to $[\frac{a+b}{2},b]$. This approach forms a binary tree that covers the entire state-space. As this partitioning could continue ad infinitum, we truncate the tree by considering a state to be terminal when $b-a \leq 10^{-7}$.

In the simulation phase, actions are executed uniformly randomly until a leaf is encountered, at which point some reward is received. Let $f$ be the function and $c$ be the midpoint of the leaf reached by the rollout. At iteration $t$, a binary reward $r_t$, drawn from a Bernoulli distribution $r_t \sim \text{Bern}(f(c))$, is generated.

At the completion of the algorithm, we calculate the score by descending the game tree from root to leaf (where a leaf node is a node that has not yet been fully expanded), selecting at each state its most visited child. The centre of the leaf node’s interval represents UCT’s belief of the location of the global maximum.

To illustrate UCT’s response to smoothness, consider two functions $f(x) = 4x(1-x)$ and $g(x) = \max(3.6x(1-x),1-10|0.9-x|)$. $f$ has a single global
maximum at $x = 0.5$, while $g$ has a local maximum at the same point, and a global maximum at $x = 0.9$. Despite the fact that both functions are relatively simple, UCT occasionally fails to find the true optimal value, as illustrated by the first two rows of Table 1.

To see why this occurs, consider an additional two functions which are far more complex: $h(x) = |\sin \frac{1}{x}|$ and $j(x) = \begin{cases} \frac{1}{2} + \frac{1}{2} |\sin \frac{x}{x^2}| & \text{if } x < \frac{1}{2} \\ \frac{20}{2} + \frac{1}{2} |\sin \frac{x}{x^2}| & \text{if } x \geq \frac{1}{2} \end{cases}$.

Notice that the frequency of the function $h$ decreases as $x$ increases. Since the function attains a maximum at many points, we can expect UCT to return the correct answer frequently. Visiting an incorrect region of the domain here is not too detrimental, since there is most likely still a state that attains the maximum in the interval.

With that said, there is clearly a smoother region of the space that can be searched. In some sense, this is the more conservative space, since a small perturbation does not result in too great a change in value. Indeed, UCT prefers this region (Figure 2a), with the leaf nodes concentrating around this smooth area despite there being many optima at $x \leq 0.5$.

On the other hand, the function $j$ is a tougher proposition, despite having the same number of critical points as $h$. Here, the “safer” interval of the function’s domain (at $x > 0.5$) preferred by UCT is now suboptimal. In this case, UCT finds it difficult to make the transition to the true optimal value, since it prefers to exploit the smoother, incorrect region.

After a sufficient number of simulations, however, UCT does indeed start to visit the optimal region of the graph (Figure 2b). Since the value of nearby states in this region changes rapidly, robust estimates are required to find the true optimum. For function $j$, UCT achieves an average score lower than even that of the local maxima. This suggests that the search spends time at the suboptimal maxima before switching to the region $x < 0.5$. However, because most of the search had not focused on this space previously, its estimates are inadequate, which results in very poor returns.

### 4 Bias in the Simulation Phase

Having demonstrated the effect of the domain’s smoothness on UCT, we now turn our attention to heavy rollouts. Oftentimes rollouts that are not uniformly random are referred to as biased rollouts. Since the simulation phase is a substitute for the value function, almost all policies suffer from some bias, even uniformly random ones. As this applies to both deterministic and random rollouts—policies at opposite ends of the spectrum—it is important to differentiate between the two.

To draw an analogy, consider Bayesian inference. Here a prior distribution, which represents the knowledge injected into the search had not focused on this space previously, its estimates are inadequate, which results in very poor returns.

### Table 1: Average maximum value found by UCT for the functions $f$, $g$, $h$ and $j$, averaged over 100 runs.

<table>
<thead>
<tr>
<th>Function</th>
<th>500</th>
<th>1000</th>
<th>5000</th>
<th>10000</th>
<th>20000</th>
<th>50000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$1 \pm 0.001$</td>
<td>$1 \pm 0.001$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
</tr>
<tr>
<td>$g$</td>
<td>$0.89 \pm 0.002$</td>
<td>$0.90 \pm 0.002$</td>
<td>$0.94 \pm 0.002$</td>
<td>$0.95 \pm 0.002$</td>
<td>$0.96 \pm 0.001$</td>
<td>$0.96 \pm 0.001$</td>
<td>$0.97 \pm 0.001$</td>
</tr>
<tr>
<td>$h$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
<td>$1 \pm 0.0$</td>
</tr>
<tr>
<td>$j$</td>
<td>$0.60 \pm 0.012$</td>
<td>$0.62 \pm 0.007$</td>
<td>$0.65 \pm 0.003$</td>
<td>$0.66 \pm 0.003$</td>
<td>$0.67 \pm 0.002$</td>
<td>$0.67 \pm 0.002$</td>
<td>$0.68 \pm 0.002$</td>
</tr>
</tbody>
</table>

(a) Percentage of visits to leaves after 50 000 iterations of UCT for the function $h$.

(b) Percentage of visits to leaves after 50 000 iterations of UCT for the function $j$.

Figure 2: The percentage of total visits assigned to each leaf node, averaged over 100 runs. A scaled version of $h$ and $j$ is overlaid for reference. For illustration purposes, leaves are grouped into 1000 buckets, with the sum of the visits to leaves in each bucket plotted.
into the system, is modified by the evidence received from the environment to produce a posterior distribution. Arguments can be made for selecting a maximal entropy prior—that is, a prior that encodes the minimum amount of information. Based on this principle of indifference, the posterior that is produced is directly proportional to the likelihood of the data.

Selecting a prior distribution that has small variance, for instance, has the opposite effect. In this case, far more data will need to be observed to change it significantly. Thus, a prior with low entropy can effectively overwhelm the evidence received from the environment. If such a prior is incorrect, this can result in a posterior with a large degree of bias.

Uncertainty in a domain arises from the fact that we are unaware of the true policy being used by an agent. This is especially true beyond the search tree’s horizon, where there exist no value estimates. The simulation phase is thus responsible for managing this extreme uncertainty. The choice of rollout policy can therefore be viewed as a kind of prior distribution over the policy space—one which encodes the user’s knowledge of the domain, with uniformly random rollouts representing maximal entropy priors, and deterministic rollouts minimal ones.

To illustrate the advantage of selecting a high-entropy simulation policy, we consider biasing simulations for the function optimisation task by performing a one-step lookahead and selecting an action proportional to the value of the next state. We also consider an inversely-biased policy which selects an action in inverse proportion to its value.

The choice of rollout policy affects the initial view MCTS has of the function to be optimised. The figures in Figure 3 demonstrate this phenomenon for the random, biased and inversely-biased policies when optimising the function $y(x) = \frac{1}{2} \left( \sin(5\pi x) + \cos(x) \right)$.

Random rollouts perfectly represent the function, since their expected values depend only on the function’s value itself, while the biased policy assigns greater importance to the region about the true maximum, but does not accurately represent the underlying function. This serves to focus the search in the correct region of the space, as well as effectively prune some of the suboptimal regions. This is not detrimental here since the underestimated regions do not contain the global maximum. Were the optimal value to exist as an extreme outlier in the range $[0.5, 1]$, then the policy would hinder the ability of MCTS to find the true answer, as it would require a large number of iterations to correct this error. A sufficiently smooth domain would preclude this event from occurring.

Finally, the last figure demonstrates how an incorrectly biased policy can cause MCTS to focus initially on a completely suboptimal region. Many iterations would thus be required to redress the serious bias injected into the system.

5 Risky Simulation Policies

To illustrate the possible risk in selecting the incorrect simulation policy, consider a perfect $k$-ary tree which represents a generic extensive-form game of perfect information. Vertices represent the state-space, and edges the action-space, so that $\mathcal{A}(s) = \{0, 1, \ldots, k-1\}$. Rewards in the range $[0, 1]$ are assigned to each leaf node such that $\forall s, \pi^*(s, \frac{k}{2}) = 1$. For non-optimal actions, rewards are distributed randomly. A $k$-ary tree of height $h$ is referred to as a $[k, h]$ tree henceforth.

A uniformly random rollout policy $\pi_{\text{rand}}$ acts as a baseline with which to compare the performance of other simulation policies. These policies sample an action from normal distributions with varying mean and standard deviation—that is, policies are parameterised by $\beta \sim N(\mu, \sigma)$ such that

$$\pi_\beta(s, a) = \begin{cases} 1 & \text{if } a = \lfloor \beta \mod k \rfloor \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Figure 4 presents the results of an experiment conducted on a $[5, 5]$ instance of the domain. We limit the MCTS algorithm to 30 iterations per move to simulate an environment in which the state-space is far greater than what would be computable given the available resources. Both the mean and standard deviation are incrementally varied from 0 to 4, and are used to parameterise a UCT agent. The agent is then tested on 10 000 different instances of the tree.

The results demonstrate that there is room for bettering random rollouts. Quite naturally, the performance of the UCT agent is best when the distribution from which rollout policies are sampled are peaked about the optimal action. However, the worst performance occurs when the rollouts have incorrect bias and are over-confident in their estimation (that

![Figure 3](https://example.com/figure3.png)

(a) Expected value under a uniformly random policy.

(b) Expected value under a biased policy.

(c) Expected value under an inversely-biased policy.

Figure 3: The view of the overlaid function under the different policies. The expected value is calculated by multiplying the probability of reaching each leaf by its value.
Figure 4: Results of rollout policies averaged over 10 000 [5, 5] games. The x and y axes represent the mean and standard deviation of the rollout policy used by the UCT agent, while the z-axis denotes the percentage of times the correct action was returned. The performance of a uniformly random rollout policy (which returned the correct move 39.4% of the time) is represented by the plane, while the red region indicates policies whose means are more than one standard deviation from the optimal policy.

is, with small standard deviations), their performance dropping below even that of random. When the rollouts have too great a variance, their performance degenerates to that of random. There is thus only a small window for improvement, which requires the correct bias and low variance. One should be certain of the correct bias, however, as the major risk of failure occurs for low-variance, high-bias distributions.

6 Heavy Rollouts in Non-Smooth Domains

One interesting question is the manner in which the rollouts allow UCT to handle noise or unexpected encounters in a domain. To investigate this, we consider a maze domain with deterministic transition dynamics in which an agent navigates a grid (the start and end squares are randomly assigned). A number of obstacles may be placed on the grid according to two strategies. The first is to simply place obstacles randomly, while the second is to form a cluster of obstacles. Informally, clustered obstacles only affect an isolated region of the state-space, while the randomly placed ones affect the entire space. The clustered obstacles therefore create a localised non-smooth region, whereas the randomly placed obstacles make the entire space non-smooth.

The agent has four available actions at each state: UP, DOWN, LEFT and RIGHT. If the agent executes an action that would cause it to collide with an obstacle or leave the grid, it then remains in the same state and receives a reward of −10. An agent receives a reward of 0 if it enters the goal state, and −1 in all other cases. The value returned by a rollout is calculated by adding the rewards it receives at each simulated step, until either the goal state is encountered or the sum becomes less than −1000. The final sum of rewards is then linearly scaled to the range [0, 1].

In order to create a policy for this domain, each action is first assigned a base value of 1. Actions that lead to states closer to the goal (ignoring obstacles) have some additional weight added to them. An action is then selected proportionally to its assigned value.

We parameterise the policies by the value that is added to these actions. For instance, π0 represents a uniformly random policy, while π∞ is a deterministic greedy policy. We test the performances of four policies (π0, π1, π5, π∞) in a 10 × 10 grid with no obstacles, 15 obstacles and 15 clustered obstacles, with results presented in Figures 5, 6 and 7 respectively.

With no obstacles, the results are fairly straightforward. The greedy policy, which in this case is also the optimal policy, is the most successful, the random the least and the others in between. When obstacles are added randomly, the situation changes completely. Since the rollout policies were constructed to head towards the goal without knowledge of any obstacles, their presence damages the performance of UCT. The worst performing agent in this case is the deterministic policy, while the more conservatively biased policy is the best choice. Random also remains unaffected by the obstacles, with little difference between it and the biased policies.
We have also demonstrated that selecting a low-variance policy can markedly improve the performance of UCT in [38x244]Correct Move%
0.92 0.94 0.96 0.98 1.00
60 GIGA’16 Proceedings

uct is required to explore for a longer period of time, re-

One additional point of interest is the poor initial performance of $\pi_\infty$, even when it is indeed the optimal policy. This occurs because the optimal policy actually provides less information than the random rollout, which better disambiguates the action values. To illustrate, consider a $1 \times 5$ grid, where the goal state is the rightmost square and the only available actions are LEFT and RIGHT. Since the values of states are so close to one another under the optimal policy, UCT is required to explore for a longer period of time, re-

sulting in poor performance at the beginning. The random policy, on the other hand, clearly differentiates between adjacent states, allowing UCT to begin exploiting much earlier. Figure 9 demonstrates this phenomenon.

As a final experiment, consider the Taxi domain. Here the agent has two additional actions (PICKUP and DROPOFF), which need to be executed at the appropriate state (the agent incurs a penalty of $-10$ otherwise). The agent's aim is to navigate to some state and execute the PICKUP action, before proceeding to a final state and executing DROPOFF.

Initially, it may seem as if we can expect similar results, since this domain can be seen as two sequential instances of the maze task. However, the key difference is the critical requirement of executing a single action in a single state. Thus while the obstacles provide some additional difficulty, they pale in comparison with the bottleneck of having to execute these critical actions at the correct time.

Adopting the previous approach, Figure 10 illustrates that the number of randomly placed obstacles has minimal effect on the lower variance policies when compared with Figure 8—executing PICKUP and DROPOFF at the proper time is evidently far more important than the presence of the obstacles.

The domain is therefore indicative of many games where selecting the correct action at critical times is more important than playing well at all other times. This also supports the approach of rollout policies such as that of the Go agent MoGò, which is hardcoded to make critical moves when necessary, but plays randomly otherwise [Wang and Gelly, 2007].

7 Conclusion and Future Work

We have shown that the smoothness of a domain is key to deciding whether to apply the UCT algorithm to the problem. This supports prevailing theories regarding the reason for its poor performance in chess, despite its strong showing in Go.

We have also demonstrated that selecting a low-variance policy can markedly improve the performance of UCT in
the correct situation, especially in those situations that require a critical action to be selected at the correct state, but can also result in extremely poor performance exacerbated by non-smooth domains. In situations of uncertainty, a higher-variance rollout policy may thus be the better, less-risky choice. When it comes to learning simulation policies, this suggests that learning a distribution over policies may be superior to any kind of pointwise approach.

Future work should also investigate methods and heuristics for quantifying the smoothness of a particular domain. One approach may be first to construct a game tree using uniformly random rollouts. The tree could then be analysed by estimating the smoothness of sibling nodes’ values throughout. This can be achieved by assigning a smoothness score to a node, calculated as a linear combination of the lag-one autocorrelation of its children’s values and respective smoothness scores. This value would indicate the smoothness of the tree (or subtree) and provide information as to UCT’s likely performance beforehand.

Acknowledgements

The authors wish to thank the anonymous reviewers for their helpful comments and feedback.

References


C. Domshlak and Z. Feldman. To UCT, or not to UCT? (position paper). In *Sixth Annual Symposium on Combinatorial Search*, 2013.


Evolving UCT Alternatives for General Video Game Playing

Ivan Bravi, Ahmed Khalifa, Christoffer Holmgård, Julian Togelius
New York University, Tandon School of Engineering
ivan.bravi@nyu.edu, ahmed.khalifa@nyu.edu, holmgard@nyu.edu, julian@togelius.com

Abstract
We use genetic programming to evolve alternatives to the UCB1 heuristic used in the standard UCB formulation of Monte Carlo Tree Search. The fitness function is the performance of MCTS based on the evolved equation on playing particular games from the General Video Game AI framework. Thus, the evolutionary process aims to create MCTS variants that perform well on particular games; such variants could later be chosen among by a hyper-heuristic game-playing agent. The evolved solutions could also be analyzed to understand the games better. Our results show that the heuristic used for node selection matters greatly to performance, and the vast majority of heuristics perform very badly; furthermore, we can evolve heuristics that perform comparably to UCB1 in several games. The evolved heuristics differ greatly between games.

1 Introduction
Monte Carlo Tree Search (MCTS) is a popular and effective algorithm for planning and game playing, which has in particular seen successes in general game playing, i.e. playing unseen games where no a priori domain information is possible [Kocsis and Szepesvári, 2006; Browne et al., 2012]. In its most common formulation, the algorithm builds a search tree through exploring nodes in a best-first manner, and every time a new node is explored a random playout of the game/planning problem is performed to stochastically estimate the value of the node; values are also propagated up to all ancestors of a node. At the core of the MCTS algorithm is the UCB1 equation which allows the algorithm to balance between exploration and exploitation.

A large number of modifications to the basic MCTS algorithm have been advanced to deal with particular games and other problem domains. The modifications that make MCTS work better for a particular problem might very well make it worse for another problem, meaning that domain knowledge is necessary to invent and select the right modification for a given problem. The literature contains a large number of MCTS modifications, a handful of which are listed in a well-known survey paper [Browne et al., 2012]. While some of these modifications change the way the overall MCTS algorithm works or focuses on particular aspects such as the roll-out, others change the UCB1 equation itself [Jacobsen et al., 2014].

As exploring new variants of the MCTS algorithm is a currently fruitful area, one wonders whether it would be possible to automate some version of this research. In other words: automate the invention of such modifications. Furthermore, one wonders if it would be possible to automatically create MCTS variations that are specifically tailored to particular games or problems. This could be useful for example in agents based on hyper-heuristics, that would select appropriate MCTS variations for particular games/problems [Burke et al., 2013]. But it could also be useful for understanding the characteristics of a particular game/problem through finding which MCTS variation performs best at it; in other words analyzing the problem through finding strategies for solving it.

In this paper we propose using genetic programming to evolve replacement equations for the UCB1 equation. The idea is that different versions of, or alternatives to, UCB1 might make MCTS more suitable for particular problems, and that we can find such alternatives or versions automatically. For our testbed problems, we use the games in the General Video Game Playing framework. We show that for several games, we can find replacements for UCB1 that makes MCTS play the game as well.

2 Background
This section reviews the background on Monte Carlo Tree Search, how it has been combined with evolution, genetic programming and general video game playing.

2.1 Monte Carlo Tree Search
Monte Carlo Tree Search (MCTS) is a relatively recently proposed algorithm for planning and game playing. It is a tree search algorithm which selects which nodes to explore in a best-first manner, which means that unlike Minimax (for two-player games) and breadth-first search (for single-player games) Monte Carlo Tree Search focuses on promising parts of the search tree first, while still conducting targeted exploration of under-explored parts. This balance between exploitation and exploration is usually handled through the application of the Upper Confidence Bound for Trees (UCT) algorithm which applies UCB1 to the search tree.
The basic formulation of UCB1 is given in Equation 1, but many variations exist for different games [Auer et al., 2002; Browne et al., 2012; Park and Kim, 2015].

\[
UCB1 = \bar{x}_j + 2C\sqrt{\frac{\ln n}{n_j}}
\]  

(1)

These variations change UCB1 by e.g. optimizing it for single-player games or incorporating feature selection to name a few variations. However, when we use MCTS for general game playing it becomes impossible to know if we are better off using “plain UCB” or some specialized version, since we do not know which game we will be encountering.

Ideally, we need some way of searching through the different possible variations of tree selection policies to find one that is well suited for the particular game in question. We propose addressing this problem by evolving tree selection policies to find specific formulations that are well suited for specific games. If successful, this would allow us to automatically generate adapted versions of UCB for games we have never met, potentially leading to better general game playing performance.

2.2 Combinations of evolution and MCTS

Evolutionary computation is the use of algorithms inspired by Darwinian evolution for search, optimization, and/or design. Such algorithms have a very wide range of applications due to their domain-generality; with an appropriate fitness function and representation, evolutionary algorithms can be successfully applied to optimization tasks in a variety of fields.

There are several different ways in which evolutionary computation could be combined with MCTS for game playing. Perhaps the most obvious combination is to evolve game state evaluators. In many cases, it is not possible for the rollouts of MCTS to reach a terminal game state; in those cases, the search needs to “bottom out” in some kind of state evaluation heuristic. This state evaluator needs to correctly estimate the quality of a game state, which is a non-trivial task. Therefore the state evaluator can be evolved; the fitness function is how well the MCTS agent plays the game using the state evaluator. This is done routinely for Minimax search and has been done several times in the literature for MCTS [Pettit and Helmbold, 2012].

Of particular interest for the current investigation is Cazenave’s work on evolving UCB1 alternatives for Go [Cazenave, 2007]. It was found that it was possible to evolve heuristics that significantly outperformed standard UCB formulations; given the appropriate primitives, it could also outperform more sophisticated UCB variants specifically aimed at Go. While successful, Cazenave’s work only concerned a single game, and one which is very different from a video game.

At first sight, it would seem that evolutionary algorithms and MCTS are very different kinds of algorithms used for very different purposes. However, it has recently emerged that they can be used for very similar purposes. MCTS has been used for content generation [Browne, 2011; Browne et al., 2012] and continuous optimization [McGuinness, 2016]. Evolutionary algorithms have also been used for real-time planning in single-player [Perez et al., 2013] and two-player games [Justesen et al., 2016].

This points to the ability of both MCTS and evolutionary search to focus limited computational resources on the most promising parts of a large search space, given simple metrics of outcomes and, in the case of MCTS, indications of how to balance exploration and exploitation of promising areas found during the search. More fundamentally, it raises the question whether these two algorithms are similar on some deeper level—perhaps there could even be a framework defining a space with MCTS at one end and a genetic algorithm or evolution strategy on the other.

2.3 General Video Game Playing

The field of General Video Game Playing (GVGP) is an extension of General Game Playing (GGP) [Levine et al., 2013] which focuses on asking computational agents to play unseen games. Agents are evaluated on their performance on a number of games which the designer of the agent did not know about before submitting the agent. GVGP focuses on real time games compared to board games (turn based) in General Game Playing.

In this paper, we use the General Video Game AI framework (GVGAI), which is the software framework associated with the GVGAI competition [Perez et al., 2015; Perez-Liebana et al., 2016]. In the learning track of the GVGAI competition, competitors submit agents which are scored on playing ten unseen games which resemble (and in some cases are modeled on) classic arcade games from the seventies and eighties.

It has been shown in the past that for most of these games, simple modifications to the basic MCTS formulation can provide significant performance improvements. However, these modifications are non-transitive; a modification that increases the performance of MCTS on one game is just as likely to decrease its performance on another [Frydenberg et al., 2015]. This points to the need for finding the right modification for each individual game, either manually or automatically.

We selected five different games from the framework as testbeds for the tree selection policy evolution:

- **Boulderdash:** is a VGDL port of Boulderdash. The player’s goal is to collect at least ten diamonds then reach the goal while avoiding getting killed either by enemies or boulders.
- **Zelda:** is a VGDL port of The legend of Zelda dungeon system. The player’s goal is to reach the exit without getting killed by enemies. The player can kill enemies using its sword.
- **Missile Command:** is a VGDL port of Missile Command. The player’s goal is to protect at least one city building from being destroyed by the incoming missiles. The player can move around and destroy missiles by attacking them.
- **Solar Fox:** is a VGDL port of Solar Fox. The player’s goal is to collect all the diamonds and avoid hitting the side walls or the enemy bullets. The player is always moving like a missile which makes it harder to control.
**Butterflies**: is an arcade game developed for the framework. The player’s goal is to collect all the butterflies before they destroy all the flowers.

These games require very different strategies from agents for successful play and together provide varied testbeds for the approach. They also have in common that standard MCTS with the UCB1 equation does not play the game perfectly (or even very well) and that other agents have been shown to play the game better in the past.

### 2.4 Genetic Programming

Genetic Programming (GP) [Poli et al., 2008] is a branch of evolutionary algorithms [Bäck and Schwefel, 1993] which evolves computer programs as a solution to the current problem. GP is essentially the application of genetic algorithms (GA) [Whitley, 1994] to computer programs. Like GAs, GP evolves solutions based on Darwinian theory of evolution. A GP run starts with a population of possible solutions called chromosomes. Each chromosome is evaluated for its fitness (how well it solves the problem). New chromosomes are generated using genetic operators, such as crossover and mutation, from the current chromosomes, creating a new population. This process is repeated until a given termination condition is met.

In GP, chromosomes are most commonly represented as syntax trees where inner nodes are functions (e.g. addition, subtraction, if-condition, ...etc) while leaf nodes are terminals (e.g. constants, variables, ...etc). Fitness is calculated by running the current program and see how well it solves the problem. GP uses Crossover and Mutation to evolve the new chromosomes. Crossover in GP combines two different programs at a selected node by swapping the subtrees at these nodes. Mutation in GP alters the selected node value to a new suitable value.

### 3 Methods

The chromosome representation in our GP algorithm is a syntax tree where the nodes represent either unary or binary functions while the leaves are either constants values or variables. The binary functions are addition, subtraction, multiplication, division and power. The unary functions are square root, absolute value, multiplicative inverse and natural logarithm. The constant values come from a set of possible values ranging from -30 to 30. Namely: 0.1, 0.25, 0.5, 1, 2, 5, 10, 30, -0.1, -0.25, -0.5, -1, -2, -5, -10, -30. The formula can contain variables belonging to two sets: `TreeVariable set` and `AgentVariable set`. The variables regarding the state of tree built by MCTS belong to the Tree Variables set, namely: child depth, child value, child visits, parent visits and child max value. Instead the Agent Variables set contains the variables related to the agent’s behavior like: history reverse value, it represents the number of opposite actions taken w.r.t. the current; history repeating value, times the current action has been repeated; useless value, number of actions that don’t produce any effect; exploration value, number of times the current position has been visited before; exploration max value, it represents the number of times the most visited position has been visited.

The fitness function is based on two parameters derived from the simulation of 100 playthroughs of one level of the game: `win ratio` and `average score`. Equation 2 shows how these two parameters are combined. We followed the same competition rules as `win ratio` have higher priority than `aver-
age score.

\[ \text{Fitness} = 1000 \times \text{win\_ratio} + \text{avg\_score} \]  

(2)

We use a rank based selection to choose the chromosomes to generate the new chromosomes. The offsprings are created as follows. Two chromosomes are selected from the current generation, then a subtree crossover is performed and finally a mutation operation is applied between: point mutation, subtree mutation and constant mutation. Point mutation selects a node with 5% probability and swap it with a node of the same type, the types are unary node, binary node, variable and constants. The subtree mutation, instead, selects a subtree and substitutes it with a random tree of randomly distributed depth between 1 and 3. Finally, the constants mutation selects a constant node from the tree and selects a new value from the set of available constant values. Both trees derived from the two trees selected are put in the new generation.

The population of the first generation is composed of 1 UCB1 chromosome and 99 random chromosomes. We added the UCB1 equation in the initial population to push the GP to converge faster and find something better. We run the GP for 30 generation. In each generation, we use a 10% elitism to guarantee that the best chromosome is carried out to the next generation.

The number of repetitions per fitness evaluation is high enough to give a decent evaluation within a reasonable amount of time.

Once gathered the results from the genetic algorithm we pick the best tree evolved and we verify its validity by running a simulation over 2000 playthroughs.

For Solar Fox and Zelda we evolved a UCT formula using only the Tree Variable set. While for Boulderdash, Butterflies and Missile Command we evolved two new formulas: UCT, using only the Tree Variable set; and UCT++, using both the Tree Variable and Agent Variable sets.

4 Results

In this section, we explore the implications of replacing the UCB1 equation with alternative equations in five games from the publicly available training set in the GVGAI framework.

For each game, we first attempted replacing the UCB1 equation with random replacement equations. This was done to investigate the impact the UCB1 equation itself has, and the range of performance exhibited by the basic MCTS algorithm when its core equation is changed. Our experiments showed that the replacement resulted in agents with very low performance, as shown in Table 1. Figure 1 shows the distribution of the fitness of these random equations over all five games.

In the following we describe the results of evolving new UCB1 replacements following the process outlined above. For each game we describe the best equation found, compare its performance to that of UCB1, and discuss what this says about the role of UCB1 and about the AI problem posed by the specific game.

Table 2 compares the UCB1 equation with the generated alternatives \( UCB_+ \) and \( UCB_{++} \) and only exploitation term \( X_j \) for each different game. The data is collected from 2000 runs using \( UCB_1, X_j, UCB_+, \) and \( UCB_{++} \) for all the five games. The performance of \( X_j \) is slightly worse or equal in all games except win ration in Solarfox. The performance of \( UCB_+ \) is nearly similar to the \( UCB_1 \) over score and wins. \( UCB_+ \) achieves higher win rate over Butterflies and Zelda while worse in Missile Command and Solarfox. \( UCB_+ \) achieves higher mean score over Butterflies and Solarfox while worse in the three remaining games. The performance of \( UCB_{++} \) is always better than \( UCB_1 \) in both wins and score except for score in Butterflies. In this table, we compare the mean scores and win ratio for \( UCB_1, X_j, UCB_+, \) and \( UCB_{++} \) for each game. The scores are compared using the Mann-Whitney U test for scores and the Chi Squared test applied to the absolute number of wins out of 2000. For ease of reading, double asterisk means \( p-value < 0.01 \) while single asterisk means \( p-value < 0.05 \). Figure 2 shows the increase of average fitness over generations during evolving \( UCB_+ \) for all five games. Figure 3, instead, shows the increase of average fitness over generations during evolving \( UCB_{++} \) for Boulderdash, Butterflies and Missile Command.

4.1 Boulderdash

\[ UCB_+ = \max \text{Value}(\max \text{Value}^d_i + 1) - 0.25 \]  

(3)

The evolved Equation 3 pushes MCTS to exploit more without having any exploration. The reason is Boulderdash map is huge compared to other games with a small amount of diamonds scattered throughout the map. GP finds exploiting the best path is far more better than wasting time steps in exploring the rest of the tree.

\[ UCB_+ = \max \text{Value} + d_i + 1 / E_j + 1.25 \]  

(4)

The evolved Equation 4 consists of two exploitation terms and one exploration term. The exploitation term tries to focus on the deepest
explored node with the highest value, while the exploration pushes
MCTS to explored nodes that are least visited in the game space.

4.2 Butterflies

$$UCB_B = \bar{X}_j + \frac{1}{n^2 \cdot d_j \cdot \sqrt{0.2 \cdot d_j \cdot maxValue + 1}}$$ (5)

The evolved Equation 5 is similar to MCTS with exploitation and
exploration terms. The exploitation term is similar to MCTS while
exploration term is more complex. The exploration term tries to
explore the shallowest least visited nodes in the tree with the least
maximum value. The huge map of the game with butterflies spread
all over it leads MCTS to explore the worst shallowest least visited
node. The value of the exploration is very small compared to the
exploitation term so it will only differentiate between similar valued
nodes.

$$UCB_{B+} = \sqrt{maxValue + 2\bar{X}_j} - \frac{X_j}{R_j} - \left(\frac{\ln X_j}{\sqrt{maxExp + X_j^{-0.25}}} + \sqrt{U_j}\right)^{maxExp}$$ (6)

The evolved Equation 6 is similar to MCTS with mixmax modifi-
cation [Frydenberg et al., 2015]. The first two terms resemble the
mixmax with different balancing between average child value and
maximum child value. The other two terms force the MCTS to search
for nodes with the least useless moves and with the most number of
reverse moves. The useless forces the agent to go deeper in branches
that have more moves, while the number of reverse moves in butterfly
force the agent to move similar to the butterflies in the game which
leads to capture more of them.

4.3 Missile Command

$$UCB_B = \bar{X}_j + \left(10 + \frac{X_j}{\ln n}\right)^{-1/\ln n}$$ (7)
<table>
<thead>
<tr>
<th>Game</th>
<th>Tree Policy</th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>SD</th>
<th>Win ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boulderdash</td>
<td>UCB1</td>
<td>5.30</td>
<td>4.00</td>
<td>0</td>
<td>186</td>
<td>5.22</td>
<td>0</td>
</tr>
<tr>
<td>Boulderdash</td>
<td>$X_j^+$</td>
<td>4.83</td>
<td>3.00</td>
<td>0</td>
<td>18</td>
<td>3.00</td>
<td>0</td>
</tr>
<tr>
<td>Boulderdash</td>
<td>UCB$_{++}$</td>
<td>5.05</td>
<td>4.00</td>
<td>0</td>
<td>18</td>
<td>2.85</td>
<td>0</td>
</tr>
<tr>
<td>Boulderdash</td>
<td>UCB$_{+++}$</td>
<td>19.51</td>
<td>3.00</td>
<td>0</td>
<td>1580</td>
<td>117.72</td>
<td>0.0182**</td>
</tr>
<tr>
<td>Butterflies</td>
<td>UCB1</td>
<td>37.39</td>
<td>32.00</td>
<td>8</td>
<td>86</td>
<td>18.92</td>
<td>0.902***</td>
</tr>
<tr>
<td>Butterflies</td>
<td>$X_j^+$</td>
<td>37.64</td>
<td>32.00</td>
<td>8</td>
<td>88</td>
<td>18.78</td>
<td>0.852**</td>
</tr>
<tr>
<td>Butterflies</td>
<td>UCB$_{++}$</td>
<td>36.34</td>
<td>30.00</td>
<td>8</td>
<td>88</td>
<td>18.68</td>
<td>0.89</td>
</tr>
<tr>
<td>Butterflies</td>
<td>UCB$_{+++}$</td>
<td>35.84</td>
<td>30.00</td>
<td>8</td>
<td>80</td>
<td>18.43</td>
<td>0.914</td>
</tr>
<tr>
<td>Missile Command</td>
<td>UCB1</td>
<td>2.88</td>
<td>2.00</td>
<td>2</td>
<td>8</td>
<td>1.37</td>
<td>0.641</td>
</tr>
<tr>
<td>Missile Command</td>
<td>$X_j^+$</td>
<td>2.57</td>
<td>2.00</td>
<td>2</td>
<td>5</td>
<td>1.18</td>
<td>0.409**</td>
</tr>
<tr>
<td>Missile Command</td>
<td>UCB$_{++}$</td>
<td>3.03</td>
<td>2.00</td>
<td>2</td>
<td>8</td>
<td>1.44</td>
<td>0.653</td>
</tr>
<tr>
<td>Missile Command</td>
<td>UCB$_{+++}$</td>
<td>4.95</td>
<td>5.00</td>
<td>2</td>
<td>8</td>
<td>2.13</td>
<td>0.785**</td>
</tr>
<tr>
<td>Solarfox</td>
<td>UCB1</td>
<td>6.31</td>
<td>5.00</td>
<td>0</td>
<td>32</td>
<td>6.06</td>
<td>0.00565</td>
</tr>
<tr>
<td>Solarfox</td>
<td>$X_j^+$</td>
<td>6.30</td>
<td>5.00</td>
<td>0</td>
<td>32</td>
<td>5.84</td>
<td>0.00633</td>
</tr>
<tr>
<td>Solarfox</td>
<td>UCB$_{++}$</td>
<td>6.49</td>
<td>5.00</td>
<td>0</td>
<td>32</td>
<td>5.81</td>
<td>0.0075</td>
</tr>
<tr>
<td>Zelda</td>
<td>UCB1</td>
<td>3.58</td>
<td>4.00</td>
<td>0</td>
<td>8</td>
<td>1.85</td>
<td>0.088</td>
</tr>
<tr>
<td>Zelda</td>
<td>$X_j^+$</td>
<td>3.58</td>
<td>4.00</td>
<td>0</td>
<td>8</td>
<td>1.85</td>
<td>0.064**</td>
</tr>
<tr>
<td>Zelda</td>
<td>UCB$_{++}$</td>
<td>6.32</td>
<td>6.00</td>
<td>0</td>
<td>8</td>
<td>1.26</td>
<td>0.155**</td>
</tr>
</tbody>
</table>

Table 2: Descriptive statistics for all the tested games. Mean, Median, Min, Max, and SD all relate to the score attained using $UCB1$, $X_j^+$, $UCB_{++}$, and $UCB_{+++}$, respectively. Wins simply indicates the number of wins out of 2000 possible obtained with either policy.

The evolved Equation 7 has the same exploitation term as $UCB1$. Although the second term is very complex, it forces MCTS to pick nodes with less value. This second term is very small compared to the first term so its only affecting when two nodes have nearly similar values.

$$UCB_{++} = \overline{X}_j + \frac{\maxValue}{n \cdot E_j \cdot (2X_j)^{0.2\minV}} \cdot \left( d_j - \frac{1}{2\maxValue + 2U_j/X_j + 2\ln X_j + 1/n} \right)^{-1} \right)$$

The evolved Equation 8 have the same exploitation term from $UCB1$. Although the second term is very complex, it forces MCTS to explore the least spatially visited node with the least depth. This solution is most likely evolved due to the simplicity of Missile Command which allows GP to generate an overfitted equation that suits this particular game.

4.4 Solarfox

$$UCB_+ = \overline{X}_j + \sqrt{\frac{d_j}{n}}$$

The evolved Equation 9 is a variant of the original UCB1 equation. The exploitation term is the same while the exploration term is simplified to select the deepest least selected node regardless of anything else.

4.5 Zelda

$$UCB_+ = (n + \maxValue)^{n + \maxValue}$$

The evolved Equation 10 is pure exploitation. This equation selects the node with maximum value. This new equation leads the player to be more courageous which leads to higher win rate and higher score than standard $UCB1$.  

5 Discussion and Conclusion

We have described an experiment in evolving node selection heuristics to replace UCB1 in individual GVGAI games. The goal has not been to find a better alternative to UCB1 in general, rather to find alternatives that can exploit the properties of individual games. Such alternatives could then be used in a hyper-heuristic approach to create a general agent, and also studied to understand the characteristics of the individual problem.

Our results show that changing the node selection heuristic has very large effects on the performance of MCTS-based agents, with most heuristics performing very poorly. Evolved heuristics perform on par with UCB1 for all game save one. Analyzing the evolved heuristics shows a large variety, though in almost all cases both an exploration and an exploitation term can be discerned. Almost all the evolved UCB equations kept the exploitation term in some way or another while the second term varied from being total exploration to endorse more exploitation. This makes sense, for if an agent would know the real score for each node, the best playing algorithm is a greedy algorithm which exploits the best path. For some games, the state evaluation heuristic is simply more accurate, suggesting that exploration can be downplayed. These variations reflect basic properties of the games, as expected; e.g. games with little need for exploration downplay this element. Using more variables in evolving the equation grants us better evolved equations than the original $UCB1$.

In the future, it would be interesting to explore the inclusion of additional variables in the evolved heuristics and apply on more games. For example, using parameters that is tailored for VGDL. It is likely that the advantage of evolving node selection heuristics will come out more clearly when provided with more primitives to build such heuristics from.

References

[Auer et al., 2002] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. Ma-


