Evaluating Answer Set Clause Learning for General Game Playing

Timothy Cerexhe[1], Orkunt Sabuncu[2], and Michael Thielscher[1]

[1] University of New South Wales
{timothyc,mit}@cse.unsw.edu.au
[2] Universität Potsdam
orkunt@cs.uni-potsdam.de

Abstract. In games with imperfect information, the 'information set' is a collection of all possible game histories that are consistent with, or explain, a player's observations. Current game playing systems rely on these best guesses of the true, partially-observable game as the foundation of their decision making, yet finding these information sets is expensive.

We apply reactive Answer Set Programming (ASP) to the problem of sampling information sets in the field of General Game Playing. Furthermore, we use this domain as a test bed for evaluating the effectiveness of oClingo, a reactive answer set solver, in avoiding redundant search by keeping learnt clauses during incremental solving.

1 Introduction

General Game Playing (GGP) research seeks to design systems able to understand the rules of new games and use such descriptions to play those games effectively. These systems must reason their way from the unadorned rules to a strategy capable of defeating adverse opponents under tight time constraints. The recent extension to stochastic games with imperfect information makes this process even harder by requiring players to also reason about knowledge and plan under uncertainty.

In game theory, the *information set* for a specific player is a collection of models (possible histories) of the current state of the game, that are each consistent with all observations made so far, and by extension are indistinguishable for that player [7]. Consider a simple game of 'number guessing' where a player must guess a (random) hidden number by asking a series of 'is the number $< n$?' questions. Clearly the best strategy is a binary search—by partitioning the search space in half each time we can be guaranteed a logarithmic worst-case. Further, this discovery can be detected in a game-general way by explicitly maintaining every model in the information set. This can be seen as the possible worlds approach. However the size of a typical game is so enormous that maintaining *every* world is impossible.

One response to the limits of a possible worlds approach is to accept a subset of all worlds. Traditional perfect information tree search can then be employed; this is an efficient (and sometimes admissible) substitute for genuinely reasoning about imperfect information [5,10]. In this scenario, a model is 'sampled' from the full set, either by progressing (and pruning) all possible worlds up to a fixed size [2], or by re-generating models from the rules. There is evidence that this latter case can be a sufficient approximation in a competition setting [10]. However this generation is expensive.

With this motivation, we seek to expand the current bounds on information set sampling in GGP through a conventional technology—Answer Set Programming (ASP). Specifically, we will benchmark the set sampling problem on Clingo and then compare against the newer oClingo to assess its claims of avoiding redundant search via learnt clauses. We test this problem on three games that have been unplayable at international GGP competitions.[1]

The rest of the paper is organised as follows: first, we formally introduce the Game Description Language, the *gringo* syntax for a logic program, and the *oClingo* extension. In section 4 we explain how to translate GDL to a logic program. Next we describe our experimental setup and present our findings. We conclude with a short discussion.

2 Game Description Language

The science of General Game Playing requires a formal language that allows an arbitrary game to be specified by a complete set of rules. The declarative Game Description Language (GDL) serves this purpose [4]. It uses a logic programming-like syntax and is characterised by the special keywords listed in Table 1.

Table 1. GDL-II keywords

`role(?r)`	`?r` is a player
`init(?f)`	`?f` holds in the initial position
`true(?f)`	`?f` holds in the current position
`legal(?r,?m)`	`?r` can do `?m` in the current position
`does(?r,?m)`	player `?r` does move `?m`
`next(?f)`	`?f` holds in the next position
`terminal`	the current position is terminal
`goal(?r,?v)`	`?r` gets payoff `?v`
`sees(?r,?p)`	`?r` perceives `?p` in the next position
`random`	the random player (aka. Nature)

Originally designed for games with complete information [4], GDL has recently been extended to GDL-II (for: *GDL with incomplete/imperfect information*) by the last two keywords (`sees`, `random`) to describe arbitrary (finite) games with randomised moves and imperfect information [13].

Example 1. The GDL-II rules in Fig. 1 formalise a simple game in which a player, whose role name is "`guesser`", must guess a randomly chosen number

[1] 1st Australian Open 2012, see `https://wiki.cse.unsw.edu.au/ai2012/GGP`

from 1 to 16. The player can ask a series of 'is the number $< n$?' questions before announcing that it is ready to guess.

The intuition behind the rules is as follows.[2] Line 1 introduces the players' names. Lines 3–6 define some basic arithmetic relations as background knowledge. Line 8 defines the two features that comprise the initial game state. The possible moves are specified by the rules for `legal`: in the first round, the `random` player chooses a number (lines 10–11); then the guesser can repeatedly ask "`lessthan`" questions (line 14) until it decides that it is ready to guess (line 15), followed by making a guess (lines 16). The guesser's only percepts are true answers to its yes-no question (lines 18–21). The remaining rules specify the state update (rules for `next`); the conditions for the game to end (rule for `terminal`); and the payoff, which in case of the guesser depends on whether it got the number right and how long it took (rules for `goal`).

GDL-II comes with some syntactic restrictions—for details we must refer to [6,13] due to lack of space—that ensure that every valid game description has a unique interpretation as a state transition system as follows. The **players** in a game are determined by the derivable instances of `role(?r)`. The **initial state** is the set of derivable instances of `init(?f)`. For any state S, the **legal moves** of a player `?r` are determined by the instances of `legal(?r,?m)` that follow from the game rules *augmented by an encoding of the facts in S* using the keyword `true`. Since game play is synchronous in the Game Description Language,[3] states are updated by *joint* moves (containing one move by each player). The **next position** after joint move $\boldsymbol{m}$ is taken in state S is determined by the instances of `next(?f)` that follow from the game rules *augmented by an encoding of* $\boldsymbol{m}$ *and* S using the keywords `does` and `true`, respectively. The **percepts** (aka. information) a player `?r` gets as a result of joint move $\boldsymbol{m}$ being taken in state S is likewise determined by the derivable instances of `sees(?r,?p)` after encoding $\boldsymbol{m}$ and S using `true` and `does`. Finally, the rules for `terminal` and `goal` determine whether a given state is **terminal** and what the players' **goal values** are in this case.

On this basis, game play in GDL-II follows this protocol:

1. Starting with the initial state, which is completely known to all players, in each state each player selects one of their legal moves. By definition `random` must choose a legal move with uniform probability.
2. The next state is obtained by (synchronously) applying the joint move to the current state. Each role receives their individual percepts resulting from this update.
3. This continues until a terminal state is reached, and then the goal relation determines the result for all players.

[2] A word on the syntax: We use infix notation for GDL-II rules as we find this more readable than the usual prefix notation.

[3] Synchronous means that all players move simultaneously. Turn-taking games are modelled by allowing players only one legal move without effect (such as `noop`) if it is not their turn.

```
role(guesser).    role(random).

succ(0,1).  succ(1,2).  ...   succ(15,16).
number(?n) <= succ(?m,?n).
less(?m,?n) <= succ(?m,?n).
less(?m,?n) <= succ(?m,?k), less(?k,?n).

init(step(0)).    init(starttime).

legal(random,choosenumber(?n)) <= number(?n), true(starttime).
legal(random,noop) <= not true(starttime).

legal(guesser,noop)          <= true(starttime).
legal(guesser,lessthan(?n)) <= number(?n), true(questiontime).
legal(guesser,readytoguess) <=  true(questiontime).
legal(guesser,guess(?n))    <= number(?n), true(guesstime).

sees(guesser,yes) <= does(guesser,lessthan(?n)),
                     true(secretnumber(?m)), less(?m,?n).
sees(guesser, no) <= does(guesser,lessthan(?n)),
                     true(secretnumber(?m)), not less(?m,?n).

next(secretnumber(?n)) <= does(random,choosenumber(?n)).
next(secretnumber(?n)) <= true(secretnumber(?n)).

next(questiontime) <= true(starttime).
next(questiontime) <= true(questiontime), not does(guesser,readytoguess).
next(guesstime)    <= does(guesser,readytoguess).
next(right)        <= does(guesser,guess(?n)), true(secretnumber(?n)).
next(end)          <= does(guesser,guess(?n)).
next(step(?n))     <= true(step(?m)), succ(?m,?n).

terminal <= true(end).
terminal <= true(step(16)).

goal(guesser,100) <= true(right), true(step(?n)), less(?n,8).
goal(guesser, 90) <= true(right), true(step(?n)), less(?n,9).
...
goal(guesser, 10) <= true(right), true(step(?n)), less(?n,16).
goal(guesser,  0) <= not true(right).
goal(random,   0).
```

Fig. 1. The GDL-II description of the Number Guessing game at the AI2012 GGP Competition

3 Logic Programming, *gringo*, and reactive ASP

First we recapitulate standard logic programming and answer set programming terminology. Rules are of the form $h_r \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$. where each a_i is an atom of the form $p(t_1, \ldots, t_k)$ and each t_i is a term (constant, variable, or function). The *head* h_r of rule r is either an atom, a *cardinality constraint* of the form $l\{h_1, \ldots, h_k\}u$ in which l, u are integers and $h1, \ldots, h_k$ are atoms, or the special symbol $\perp$. If h_r is a cardinality constraint, we call r a *choice rule*, and an *integrity constraint* if $h_r = \perp$. We denote the atoms occurring in h_r by *head*(r), ie. *head*$(r) = \{h_r\}$ if h_r is an atom, *head*$(r) = \{h_1, \ldots, h_k\}$ if $h_r = l\{h_1, \ldots, h_k\}u$, and *head*$(r) = \emptyset$ if $h_r = \perp$. The atoms occurring positively and negatively in the body are denoted by *body*$(r)^+ = \{a_1, \ldots, a_m\}$ and *body*$(r)^- = \{a_{m+1}, \ldots, a_n\}$. A logic program R is a set of rules; *atom*(R) denotes the set of atoms occurring in R. $head(R) = \cup_{r \in R} head(r)$ is the collection of all head atoms. The ground program $grd(R)$ is the set of all ground rules constructable from rules $r \in R$ by substituting every variable in r with some element of the Herbrand Universe of R. For further details we recommend [1,11,3].

We now examine Incremental Logic Programs, an extension of logic programming as described above. Incremental programs are constructed from modules, which for the purposes of this paper are effectively subprograms. An Incremental Logic Program $(B, P[t], Q[t])$ is composed of a base module B of time-independent ('rigid') rules, and two parameterised modules: a 'cumulative' module $P[t]$ (instantiated at each successive timestep t and which is accumulated) and a 'volatile' module $Q[t]$ (which is forgotten after each timestep; only one instantiation exists at a time). This is further extended by oClingo to produce an Online Incremental Logic Program. These programs are accompanied by an 'online progression'—a sequence of input atoms for each timestep t. oClingo programs rely on `#external` directives as domain predicates for grounding rules that rely on these input atoms.

As a final note, a great strength of the Potassco suite of ASP solvers is that clause learning is 'baked in'.

4 Translation

An ASP system is a natural platform for the Game Description Language, due to the finiteness guarantee, uninterpreted functions[4], and the presence of negation-as-failure. Indeed GDL is an extension of Datalog$^{\neg}$ with function symbols, so a syntactic translation is fairly direct [12]. We will now briefly summarise this process, which converts GDL rules to the gringo input language. After this, we will present a modification that produces rules suitable for oClingo as well.

The key aspect of this translation is the 'temporal extension' of the GDL features—GDL has implicit timepoints (initial, current, and next) which must be made explicit for an ASP system. That is, `init` rules initialise fluents for time zero. Rules for `legal` or the value of derived fluents are functions of the current time (relative to a state). Fluent update needs to reference the fluent's value at the 'next' timepoint (relative to the current time). This extension is largely achieved by wrapping fluents in binary `holds(`F`,`T`)` relations that tie the fluent F to a given timepoint T. Fluent update is handled by rules for `holds` with a timepoint one step ahead of the timepoints in the body ($T + 1$ vs T). Derived fluents have the same timepoint in the head and the body.

As noted in the original translation paper [12], this method temporalises all user (derived) rules, even if they are time-independent 'rigids'. This introduces a substantial increase in redundant grounding. As such, we will first formally define the notion of a rigid rule in terms of the dependency graph of the GDL

[4] That is, functions have no fixed interpretation and must be specified by other axioms. ASP in contrast typically interprets + as addition (and similarly for other simple arithmetic operators). This means no additional logic needs to be ported along with the GDL when translating to ASP.

rules. Then we present our augmented translation that ensures rigids are left unadorned.

Definition 1. *Construct the dependency graph $D = (V, E)$ of a set of GDL rules G as follows:*

– *The vertex set V contains all predicate symbols found in G.*
– *If predicate symbol a appears in the head of some rule $r \in G$ and predicate symbol b appears in the body of r, then D has an edge from b to a, ie. $(b, a) \in E$.*

With the dependency graph, we can now formally define the common notion of rigid rules:

Definition 2. *A rule* `h(a1,...,am) <= b1,...,bn` *is* rigid *wrt a set of GDL rules G iff there is no path from h to* `true` *or* `does` *in the dependency graph for G.*

We now present the main translation:[5]

Definition 3. *Let G be a set of GDL rules, then the* temporal extension *of G, written $ext(G)$, is the set of logic program clauses obtained from G as follows. Each occurrence of:*

– `init(`ϕ`)` *is replaced by* `holds(`ϕ`,0)`.
– `true(`ϕ`)` *is replaced by* `holds(`ϕ, T`)`, *and each* `next(`ϕ`)` *by* `holds(`$\phi, T+1$`)`.
– `sees(`R, ϕ`)` *is replaced by* `sees(`$R, \phi, T+1$`)`.
– `distinct(`t_1, t_2`)` *is replaced by* `not` $t_1 = t_2$.
– $p(t_1, \ldots, t_n)$ *where p is keyword* `does`, `legal`, `terminal`, *or* `goal` *is replaced by $p(t_1, \ldots, t_n, T)$.*
– $p(t_1, \ldots, t_n)$ *where p is* rigid *(by Definition 2) is left unadorned.*[6]

All other atoms $p(t_1, \ldots, t_n)$ are replaced by `derived(`$p(t_1, \ldots, t_n), T$`)` *(or by* `derived(`$p(t_1, \ldots, t_n), 0$`)` *if they are in the body of an* `init` *rule).*

In order to produce a valid program, these rules must also be augmented with information about the moves and percepts seen to date, constraints on move selection, and a domain predicate for timepoint variables:

Definition 4. *Given a set of GDL rules G, a role name N, a round number $R \geq 1$, the move history H of player N (a set of R* `does` *rules, one for each timepoint) and a set of percepts P (of form* `observed(`S, T`)` *where S is a ground percept and $T \in [0, R)$ is the timepoint), construct a logic program L containing:*

– *the temporal extension of G (by Definition 3).*
– *a time domain predicate* `time(0..R-1).` *(or* `time(0).` *if $R = 1$).*
– *our move history H.*

[5] Due to space constraints we cannot present a full translation and instead refer to [12].
[6] Note that this includes keyword `role` due to restrictions in the GDL specification [6].

– *an action 'generator' (choice rule)*
  `{ does(R,A,T) } :- role(R), time(T), legal(R,A,T).`
– *a unique action constraint*
  `:- not 1 { does(R,A,T) : input(R,A) } 1, role(R), time(T).`
– *constraints to guarantee correct percepts are generated*
  `:- sees(N,P,T+1), not observed(P,T+1), time(T).` *and*
  `:- not sees(N,P,T+1), observed(P,T+1), time(T).`

The logic program produced by Definition 4 is now sufficient to produce a sample of the information set and is the basis for our experiments. Note that we also intend to apply this program to GGP competitions where we only want Clingo to report back the latest game *state*, ie. `holds` statements (since the state, not the history, is the foundation for move selection). This can be achieved with the directives `#hide. #show holds/2.` appended to the rules. Note that our introduction of a `derived` keyword (not present in the original translation) allows us to easily retrieve the complete state if this is preferred.

This translation scheme was conceived for standard ASP systems, but we also wish to employ the newer, reactive *oClingo*—we want to measure the benefit of an incremental logic program to this domain. This introduces two new subtleties: first, the latest timepoint is `t`, so 'next' rules must occupy this time (ie t instead of $T+1$), and 'now' rules must be `t-1` (instead of T)—timepoints will need to be shuffled. A further complexity is that oClingo—for reactive, *incremental* logic programs—has a program that must adhere to module theory, and in particular a firm modularity condition[7].

We first present the alternate temporal extension for an oClingo-compatible domain, and then the game-independent rules that tell oClingo what problem to solve.

Definition 5. *Let G be a set of GDL rules, then the* reactive temporal extension *of G, written oExt(G), is the set of logic program clauses obtained from G as follows. For each rule, adorn the head:*

head	*replaced by*	*time variable in body*
init(ϕ)	*holds*(ϕ, 0)	0
next(ϕ)	*holds*(ϕ, t)	$t-1$
legal(R, A)	*legal*($R, A, t-1$)	$t-1$
sees(R, P)	*sees*($R, P, t-1$)	$t-1$
terminal	*terminal*(t)	t
goal(R, V)	*goal*(R, V, t)	t
$p(a_1, \ldots, a_n)$; *p is not rigid*	*derived*($p(a_1, \ldots, a_n), t-1$)	$t-1$
otherwise the head is unmodified (it and its body are rigid)		

Now update the atoms in the bodies with the appropriate time variable (as determined by the head of the rule):

[7] Due to space constraints we must defer this technical detail to [3].

GDL	*time variable X (determined by head)*
`true(`ϕ`)`	`holds(`ϕ, X`)`
`does(`R, A`)`	`does(`R, A, X`)`
`distinct(`t_1, t_2`)`	`not` $t_1 = t_2$
$p(a_1, \ldots, a_n)$; *p is not rigid*	`derived(`$p(a_1, \ldots, a_n), X$`)`
otherwise the atom is unmodified (it is rigid)	

Definition 6. *Given a set of GDL rules G, a role name N, the move history H of player N (a set of R* `does` *rules, one for each timepoint) and a set of percepts P (of form* `observed(`S, T`)` *where S is a ground percept and T is the timepoint), construct an reactive, incremental logic program L containing:*

- *the reactive temporal extension of G (by Definition 5). Note that the rigid rules go in the base module, all other rules go in the cumulative section.*
- *domain predicates* `input(R,A)` *and* `percepts(P)` *for actions A and percepts P.*
- `#external` *declarations:* `#external exec/2. #external observed/2.`
- *an action 'generator' (choice rule)*
  `{ does(R,A,t-1) } :- role(R), legal(R,A,t-1).`
- *a combined uniqueness+liveness constraint*
  `:- not 1 { does(R,A,t-1) : input(R,A) } 1, role(R).`
- *correct action constraint*
  `:- not does(N,A,t-1), exec(A,t-1), input(N,A).`
- *constraints to guarantee correct percepts are generated*
  `:- sees(N,P,t-1), not observed(P,t-1).` *and*
  `:- not sees(N,P,t-1), observed(P,t-1), percepts(P).`

And construct an online progression O, as a contiguous sequence of steps of the form:

```
#step X.
exec(A,X-1).
observed(P,X-1).
#endstep.
```

For each round $X \geq 1$*. Note that each step will contain exactly one* `exec` *statement (player N executed action A at time* $X - 1$*) and zero or more* `observed` *statements for the percepts that resulted from that action (as per Definition 4).*

These straight-forward procedures have two additional problems that we have not yet discussed: domain predicates are not always present, and GDL permits a large class of symbols for its identifiers[8]. Obviously the naming issue can be addressed with a simple symbol table. The problem of domain predicates is starting to be mitigated by a growing convention in the GGP community to supply

[8] For example hyphens, which ASP systems tend to interpret as a subtraction operator (or classical negation, based on context).

these domains with ‘input’ and ‘base’ keywords (for actions and fluents, respectively). However no such keyword has been proposed for percepts. Finding the minimal model of the negation-free program is a reasonably efficient method for grounding these domains on the back-catalog of games without these predicates. Alternatively, more efficient GDL-centric methods have been proposed [12,9], though these are beyond the scope of this paper.

Regarding timepoints. You may note that the choice of actions (`does`) occurs at time T in Clingo and time $t-1$ in oClingo. Similarly percepts (`sees`) occur at time $T+1$ compared with $t-1$ between the two versions. The reason for this is historical: the constraints on oClingo are firm[9], but the translation for Clingo was done first (and follows the original translation from [12]). Other variations are possible, however these translations are the ones we tested, and so these are the ones we present.

5 Method

In order to reason about the rules of a game we must first convert them from GDL to an ASP encoding, as presented in Section 4. Next we generate a random play through of the game for each role. This yields a collection of legal (reachable) states, the joint moves that led to those points, and the percepts that each role would see at each step. By replaying one set of moves and percepts for a select player, Clingo (or oClingo) can recreate the state (or find equivalent states subject to its imperfect information). That is, it can sample the information set.

In our experiments we generate 100 random plays for each game for each role[10]. We then ask (o)Clingo to solve for a sample of the game’s information set at each round. All times are averaged over three duplicate runs. Experiments were performed on the UNSW cluster to satisfy the time and RAM constraints. Note that individual runs used a single 2.20 GHz Opteron core, but were allocated a complete node (48 processor cores) to eliminate interference from other processes.

We explicitly point out here that our results only measure the time to achieve the *first* model, since we did not have time to repeat our experiments for larger sample sizes. However this is still a useful metric: a single model is enough to start evaluating moves in a game player. Further, the process can be dynamically improved as more models are reported (as in [10]). From this perspective, the time-to-first model is the *most* useful measure of the value of our (ASP) set-sampler, since this is the ‘dead time’ before the GGP system can start making decisions.

We also ran oClingo with the `--ilearnt=forget` flag, which disables clause learning between timesteps (ie. clauses learnt in timestep n are thrown away before timestep $n+1$ begins). Comparing oClingo’s performance with and without

[9] Facts added ‘to the future’ are prone to either violating oClingo’s modularity condition, or being ignored by the target module parameterisation.

[10] This number was reduced for the larger games due to time constraints.

this feature should demonstrate the value of Incremental Logic Programs for this type of search problem, as well as validate claims regarding the effectiveness of oClingo's clause learning. Finally, by measuring precisely the effect of clause learning between timesteps we can account for how significant its impact is, while controlling for other (smaller) differences between the Clingo and oClingo systems.

Due to the youth of GDL-II and the complexity of games it describes, there is a distinct lack of rules that tax an ASP system under our use-case. Early tests revealed that most games are slow to ground, but their game trees are then fairly simple. For most rounds of most games, both Clingo and oClingo consistently solved the search problem presented in fractions of a second. As such our experiments focus on the role of grounding, and we have chosen three of the hardest domains for the task. These games, taken from past international competitions, are:

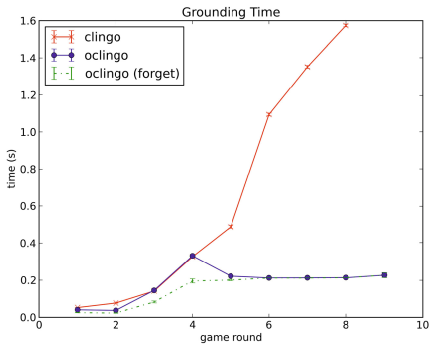
Blind Breakthrough. A two-player, zero-sum, turn-taking game played on a chess board. Each player has two rows of pieces against their side, but all pieces are pawns. The winner is the first player to reach the other side of the board ('break through' the opponent's ranks). The 'blind' aspect indicates that a player cannot see the opponent's pieces and is instead informed of the success/failure of attempted moves and the existence of a capturing move. We vary the board size between 6x6, 7x7, and 8x8 squares.

Battleships in Fog. Two navies (on separate, 4x4 grid oceans) can fire at their opponent and are informed of hit/miss. In this variant, players may also sail their single, two-by-one cell ship to an adjacent square, or perform a 'noisy sensing' action that returns three possible opponent locations (one correct, the other two not).

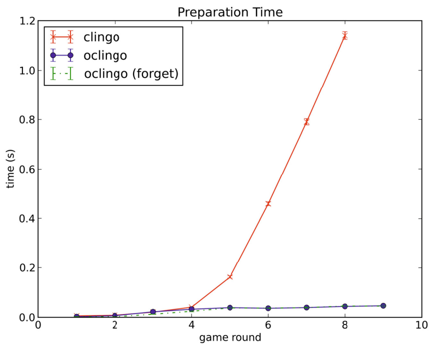
Small Dominion. Players each have a small hand of cards (either money or land) that is filled from a larger, face-down deck. Several low-value cards (eg. copper) can be used to buy a single higher-value card (eg. silver). Doing so allows a player to slowly increase the value of their hand and get more 'victory points' as a result. The game finishes when certain sets of cards are exhausted. These rules yield an interesting alternate strategy where a player buys low-value cards as quickly as possible in order to trigger an early game termination (before the opponent has won by high-value cards).

6 Results

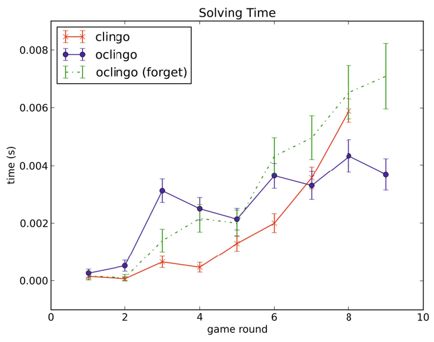
The first and most important observation is that these domains are hard because their search spaces are huge. But these are human-playable games, which suggests some structure must exist on their game trees. This is reflected in our results: grounding remains the most significant factor in the time to find a model, while actually 'solving' is lightning quick. An exception to this case is Blind



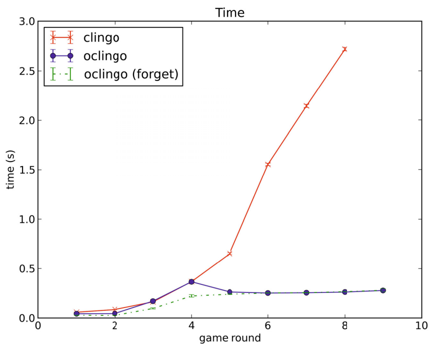
a. Grounding



b. Preparing



c. Solving



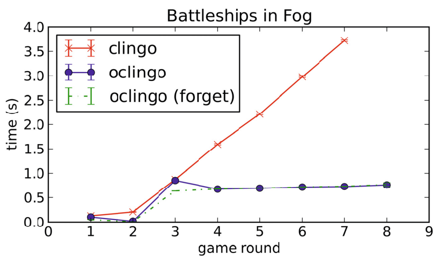
d. Total time

Fig. 2. Timing results for Clingo vs oClingo. Results are for Blind Breakthrough and averaged across all board sizes. Error bars indicate a 95% confidence interval.

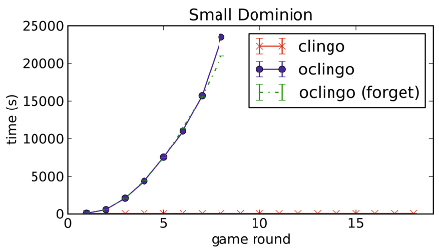
Breakthrough where solving time can be higher due to the myriad interleavings of moves explaining the same observation.

The second observation is that grounding can be prohibitive in this space and it is necessarily exacerbated by oClingo because it offers (potential) speedups later in a game tree by doing extra work[11] at each step. This was catastrophic for the game of Small Dominion, where the dealer (`random`) chooses three—mostly unused—random values in every round. Obviously this is a poor axiomatisation from an ASP perspective (all these unused values must be ground *before* they can be ignored), but this is the reality of the input GDL, where such encodings are fine for Prolog-based systems. It should be noted that this 'extra work' is clearly

[11] Eg. the grounding process needs to account for all the possible external inputs. In contrast, Clingo needs to ground only the inputs it actually receives—a liberty afforded to it since the 'externals' must be provided up-front with the program itself.



a. Total time in Battleships in Fog



b. Total time in Small Dominion

Fig. 3. Timing in Battleships and Small Dominion. Note that Clingo is dramatically more effective in Small Dominion.

at fault, since straight Clingo was still competitive in this domain. This indicates that there is a cross-over point: small domains are easy for both systems, oClingo has a strong advantage for medium-size domains, but then falls behind as the additional grounding increases and its rewards diminish. That is, after this point oClingo is swamped by its own optimisation.

7 Conclusion

It is clear that oClingo's ability to avoid redundant search and grounding offers an impressive speed-up of over its predecessor Clingo. However this gain is tempered by the size of the target domain; medium-size domains benefit most since they are complex enough to utilise the learnt clauses, but not so large as to grind to a halt whilest grounding. For the field of General Game Playing these features literally increase the horizon of 'solvable' domains. Further, this is achieved within the time constraints of a typical GGP competition—this system is ready for competition play. oClingo is not a silver bullet though, and the largest games are still well beyond the reach of game-general set sampling techniques.

Using an ASP system for a fixed-size sampling of a game's information set is not the only approach to the problem of imperfect-information game play; possible worlds systems store and incrementally update the complete information set. As an efficiency-oriented optimisation, 'particle filter' systems [2] maintain and progressively filter a large subset of the possible worlds—this helps mitigate the capacity demands of storing huge search spaces. Filtering has also been augmented with backtracking in order to avoid pruning all the possible worlds away [10]. This approach excels when successive information sets are local on the game tree, but complex games bring out its exponential complexity. Yet complex games are the interesting ones: games with high branching factors, long periods without percepts, or multiple but very different[12] explanations for the same observations. All of these properties are found in our harder test domains—Blind

[12] ie. distant on the game tree

Breakthrough, Battleships in Fog, and Small Dominion—and demonstrate that finding the information set under these constraints is fundamentally a search problem, where an efficient, domain-independent system like an ASP solver is well-suited. Of course other, more efficient methods are also possible when additional assumptions can be made about the domain [8]. A full side-by-side comparison of these methods remains as critical future work.

One notable shortcoming of our approach is the absence of model weights—we find unique histories that describe the current state, but some states are more likely than others. Essentially this means opponent modelling, which is beyond the scope of this paper. Applying soft constraints or gringo's `#maximize` statements to this problem would also be valuable future work.

Acknowledgements. This research was supported under the Go8-DAAD Australia-Germany Joint Research Cooperation scheme and DFG grant SCHA 550/9-1. The third author is the recipient of an ARC Future Fellowship (FT 0991348).

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, New York (2003)
2. Edelkamp, S., Federholzner, T., Kissmann, P.: Searching with partial belief states in general games with incomplete information. In: Glimm, B., Krüger, A. (eds.) KI 2012. LNCS, vol. 7526, pp. 25–36. Springer, Heidelberg (2012)
3. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 190–205. Springer, Heidelberg (2008)
4. Genesereth, M.R., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005), `http://games.stanford.edu/competition/misc/aaai.pdf`
5. Long, J.R., Sturtevant, N.R., Buro, M., Furtak, T.: Understanding the success of perfect information monte carlo sampling in game tree search. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press (2010)
6. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Tech. Rep. LG–2006–01, Stanford Logic Group (2006)
7. Rasmusen, E.: Games and Information: an Introduction to Game Theory, 4th edn. Blackwell Publishing (2007)
8. Richards, M., Amir, E.: Information set sampling for general imperfect information positional games. In: Proc. IJCAI 2009 Workshop on GGP, GIGA 2009, pp. 59–66 (2009)
9. Saffidine, A., Cazenave, T.: A forward chaining based game description language compiler. In: Proc. IJCAI 2011 Workshop on GGP, GIGA 2011 (July 2011)
10. Schofield, M., Cerexhe, T., Thielscher, M.: Hyperplay: A solution to general game playing with imperfect information. In: Proc. AAAI, Toronto (July 2012)

11. Simons, P., Niemelá, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
12. Thielscher, M.: Answer set programming for single-player games in general game playing. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 327–341. Springer, Heidelberg (2009)
13. Thielscher, M.: A general game description language for incomplete information games. In: Proc. AAAI, Atlanta, pp. 994–999 (July 2010)