Answer Set Programming for Single-Player Games in General Game Playing

Michael Thielscher

Department of Computer Science
Dresden University of Technology
`mit@inf.tu-dresden.de`

Abstract. As a novel, grand AI challenge, General Game Playing is concerned with the development of systems that understand the rules of unknown games and play these games well without human intervention. In this paper, we show how Answer Set Programming can assist a general game player with the special class of single-player games. To this end, we present a translation from the general *Game Description Language* (GDL) into answer set programs (ASP). Correctness of this mapping is established by proving that the stable models of the resulting ASP coincide with the possible developments of the original GDL game. We report on experiments with single-player games from past AAAI General Game Playing Competitions which substantiate the claim that Answer Set Programming can provide valuable support to general game playing systems for this type of games.

1 Introduction

General Game Playing is concerned with the development of systems that understand the rules of previously unknown games and play these games well without human intervention. Identified as a new grand AI challenge, this endeavor requires to combine methods from a variety of a sub-disciplines, such as Knowledge Representation and Reasoning, Search, Game Playing, Planning, and Learning [1,2,3,4,5]. The annual AAAI General Game Playing contest has been established in 2005 to foster research in this area and to evaluate general game playing systems in a competitive setting [6]. During the competition, participating systems receive the rules of hitherto unknown games. The contestants get some time to "contemplate" about the game (typically 5 to 20 minutes) and then start playing against each other with a further time limit for each move (typically 20 to 60 seconds). All this takes place without human interference.

General game playing requires to formalize the rules of arbitrary games in such a way that they can be processed by machines. The *Game Description Language* (GDL) [7] serves this purpose by allowing one to describe any finite and information-symmetric n-player game. GDL uses the syntax of normal logic programs, and its semantics is given by a formal game model [8] on the basis of the *standard model* of stratified programs as defined in [9]. Due to the closeness of GDL rules to Answer Set Programs, in both syntax and semantics, the question naturally arises whether this programming paradigm can provide valuable support to a general game playing system.

In this paper, we give a positive answer to this question by showing that Answer Set Programming can assist general game players with the special class of single-player games. This type of games provides for an indirect competition: independent of the others, each player tries to achieve the best possible outcome according to the rules. Examples from previous AAAI competitions are the well-known game of Peg Jumping, where the goal is to end up with as few pegs as possible on a given board, or Knight's Tour, where the goal is to visit as many squares as possible on a checkerboard of a given size.

Successful general game playing systems, such as [2,3,4], use automatically generated heuristics in combination with search. A well-known technique in game playing is *endgame search* (see, e.g., [10]), which means to perform a depth-restricted, complete forward search from the current position in order to see whether a winning position has been reached. In this paper, we show how Answer Set Programming can be used for this purpose in General Game Playing. Inspired by existing approaches of using satisfiability techniques for planning problems [11,12,13], we first map any (single- or multi-player) GDL specification onto an ASP in such a way that the stable models coincide with the possible developments of the original game. We then show how Answer Set Programming can be used to perform a complete, depth-restricted forward search during game play in case of single-player games. Experiments with a variety of single-player games from past AAAI General Game Playing Competitions [6] show that for most games, Answer Set Programming clearly outperforms the techniques for complete forward search that are built into the currently best general game players and thus provides a valuable addition to any such system.

The rest of the paper is organized as follows. In the next section, we recapitulate the basic syntax and semantics of GDL. In the section that follows, we map GDL descriptions onto "temporally extended" answer set programs and prove that the stable models for the resulting program coincides with the possible developments of the original game. In Section 4, we present a provably correct method of applying this result to perform endgame search in single-player games using Answer Set Programming. In Section 5, we give an overview of successful experiments with an off-the-shelf ASP system [14] for a variety of single-player games taken from the past AAAI General Game Playing Competitions. We conclude in Section 6. For the rest of the paper, we assume that the reader is familiar with the basic concepts of answer sets, as can be found, e.g., in [15].

2 Game Description Language

The Game Description Language (GDL) has been developed to formalize the rules of any finite game with complete information in such a way that the description can be automatically processed by a general game player. Due to lack

`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)`	player `R` has legal move `M`
`does(R,M)`	player `R` does move `M`
`next(F)`	`F` holds in the next position
`terminal`	the current position is terminal
`goal(R,N)`	player `R` gets goal value `N`

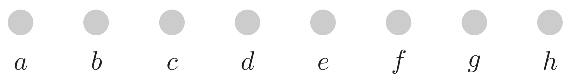
Table 1. The GDL keywords.

of space, we can give just a very brief introduction to GDL and have to refer to [7] for details.

GDL is based on the standard syntax of normal logic programs. We adopt the Prolog convention according to which variables are denoted by uppercase letters and predicate and function symbols start with a lowercase letter. As a tailor-made specification language, GDL uses a few pre-defined predicate symbols shown in Table 1. A further standard predicate is `distinct(X,Y)`, which means syntactic inequality of the two arguments.
GDL imposes the following restrictions on the use of these keywords in a set of clauses describing a game.

– `role` only appears in facts;
– `init` and `next` only appear as head of clauses, and `init` is not connected (in the dependency graph for the set of clauses) to any of `true`, `legal`, `does`, `next`, `terminal`, or `goal`;
– `true` and `does` only appear in clause bodies, and `does` is not connected to any of `legal`, `terminal`, or `goal`.

As an example, Figure 1 shows a complete set of GDL rules for the following, simple single-player game. Starting with eight coins in a row,



jump with any coin forming a singleton stack over two coins onto another single coin. Repeat until you end up with as few as possible (ideally, zero) single coins.[1]

GDL imposes some further, general restrictions on a set of clauses with the intention to ensure finiteness of the set of derivable predicate instances. Specifically, the program must be *stratified* [9,16] and *allowed* [17]. Stratified logic programs are known to admit a specific *standard model* as defined in [9]. Based

[1] For instance, you may first jump with the coin in a over the coins in b and c onto the coin in d. Next, you can take the single coin in c and jump over the two coins which are now in position d, landing on the coin in e. But then no further move will be possible, which according to the rules in Figure 1 means goal value 0.

```
role(player).
succ(a,b).
...
succ(g,h).
init(cell(a,single)).
init(cell(Y,single)) :- succ(X,Y).
legal(P,jump(X,Y))   :- true(cell(X,single)), true(cell(Y,single)),
                        twobetween(X,Y).
legal(P,jump(X,Y))   :- true(cell(X,single)), true(cell(Y,single)),
                        twobetween(Y,X).
next(cell(X,nocoin)) :- does(player,jump(X,Y)).
next(cell(X,double)) :- does(player,jump(Y,X)).
next(cell(X,Number)) :- true(cell(X,Number)), does(player,jump(Y,Z)),
                        distinct(X,Y), distinct(X,Z).

terminal         :- not continuable
continuable      :- legal(player,Move).
goal(player,100) :- not lonelycoin.
goal(player, 50) :- lonelycoin, not threelonelycoins.
goal(player,  0) :- threelonelycoins.
lonelycoin       :- true(cell(X,single)).
threelonelycoins :- true(cell(X,single)), true(cell(Y,single)),
                    true(cell(Z,single)), distinct(X,Y),
                    distinct(X,Z), distinct(Y,Z).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), twobetween(Z,Y).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,single)), onebetween(Z,Y).
twobetween(X,Y)   :- succ(X,Z), true(cell(Z,double)), nilbetween(Z,Y).
onebetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), onebetween(Z,Y).
onebetween(X,Y)   :- succ(X,Z), true(cell(Z,single)), nilbetween(Z,Y).
nilbetween(X,Y)   :- succ(X,Z), true(cell(Z,nocoin)), nilbetween(Z,Y).
nilbetween(X,Y)   :- succ(X,Y).
```

Fig. 1. A complete GDL description for the coin game. The positions are encoded using the feature $\mathit{cell}(X,Y)$, where $X \in \{a,\ldots,h\}$ and $Y \in \{\mathit{nocoin}, \mathit{single}, \mathit{double}\}$.

on this concept of a standard model, a set of GDL rules can be understood as a description of a formal game model—a state transition system—as follows [8].

To begin with, any valid game description G in GDL contains a finite set of function symbols, including constants, which implicitly determines a set of ground terms Σ. This set constitutes the symbol base Σ in the formal semantics for G. The players and the initial position of a game can be directly determined from the clauses for, respectively, `role` and `init` in G. In order to determine the legal moves, update, termination, and goalhood for any given position, this position has to be encoded first, using the keyword `true`. To this end, for any *finite* subset $S = \{f_1,\ldots,f_n\} \subseteq \Sigma$ of a set of ground terms, the following set of logic program facts encodes S as the current position.

$$S^{\texttt{true}} \stackrel{\text{def}}{=} \{\texttt{true}(f_1)\texttt{.},\ \ldots,\ \texttt{true}(f_n)\texttt{.}\}$$

The legal moves for each player r in position S can then be determined as the derivable instances of $\texttt{legal}(r,\texttt{M})$. Similarly, the fact whether S is terminal is determined by whether `terminal` is derivable, in which case the derivable instances of $\texttt{goal}(r,\texttt{N})$ determine the goal values for the individual players.

Finally, for any function $A : \{r_1, \ldots, r_n\} \mapsto \Sigma$ that assigns a move to each player $r_1 \in \Sigma, \ldots, r_n \in \Sigma$, let the following set of facts encode A as joint move.

$$A^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_1, A(r_1))\texttt{.},\ \ldots,\ \texttt{does}(r_n, A(r_n))\texttt{.}\}$$

The derivable instances of $\texttt{next(F)}$ then determine the position resulting from joint move A in the current position encoded by $S^{\texttt{true}}$. All this is summarized in the following definition.

Definition 1. [8] *Let G be a GDL specification whose signature determines the set of ground terms Σ. The* semantics *of G is the state transition system (R, S_1, T, l, u, g) where*[2]

- $R = \{r \in \Sigma : G \models \texttt{role}(r)\}$ *(the players);*
- $S_1 = \{f \in \Sigma : G \models \texttt{init}(f)\}$ *(the initial position);*
- $T = \{S \in 2^{\Sigma} : G \cup S^{\texttt{true}} \models \texttt{terminal}\}$ *(the terminal positions);*
- $l = \{(r, a, S) : G \cup S^{\texttt{true}} \models \texttt{legal}(r, a)\}$*, where $r \in R$, $a \in \Sigma$, and $S \in 2^{\Sigma}$ (the legality relation);*
- $u(A, S) = \{f \in \Sigma : G \cup S^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$*, for all $A : (R \mapsto \Sigma)$ and $S \in 2^{\Sigma}$ (the update function);*
- $g = \{(r, n, S) : G \cup S^{\texttt{true}} \models \texttt{goal}(r, n)\}$*, where $r \in R$, $n \in \mathbb{N}$, and $S \in 2^{\Sigma}$ (the goal relation).*

For example, given the game rules in Figure 1 it is easy to see that the initial state is

$$S_1 = \{\mathit{cell}(a, \mathit{single}), \ldots, \mathit{cell}(d, \mathit{single}), \ldots, \mathit{cell}(h, \mathit{single})\}$$

The addition of $S_1^{\texttt{true}}$ to the game rules shows that $(\mathit{player}, \mathit{jump}(a, d), S_1) \in l$, and the resulting position (with $A = \{\mathit{player} \mapsto \mathit{jump}(a, d)\}$) is

$$u(A, S_1) = \{\mathit{cell}(a, \mathit{nocoin}), \ldots, \mathit{cell}(d, \mathit{double}), \ldots, \mathit{cell}(h, \mathit{single})\}$$

Definition 1 provides a formal semantics by which a GDL description is interpreted as an abstract n-player game: in every position S, starting with S_1, each player r chooses a move a that satisfies $l(r, a, S)$. As a consequence the game state changes to $u(A, S)$, where A is the joint move. We introduce the following

[2] Below, entailment $\models$ is via the aforementioned standard mode as defined in [9], and 2^{Σ} denotes the *finite* subsets of Σ.

notation for possible developments in a game. Consider two finite sequences of, respectively, joint moves $A_1, \ldots, A_k$ and states $S_2, \ldots, S_{k+1}$ $(k \geq 0)$. Then

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots \xrightarrow{A_{k-1}} S_k \xrightarrow{A_k} S_{k+1} \tag{1}$$

if for each $i = 1, \ldots, k$ we have

– $S_i \notin T$,
– $(r, A_i(r), S_i) \in l$ for each $r \in R$, and
– $S_{i+1} = u(A_i, S_i)$.

The game ends when a position in T is reached, and then g determines the outcome for each player. The syntactic restrictions in GDL (see [7] for details) ensure that entailment wrt. the standard model is decidable and that only finitely many instances of each predicate are entailed. This guarantees that the definition of the semantics is effective.

3 Mapping GDL to a Logic Program with Time

The semantics for GDL according to Definition 1 shows that the plain game rules need to be repeatedly applied when determining the legal moves and their effects for different positions. Therefore, in order to be able apply logic programming techniques to directly reason about the evolution of a game position, the rules need to be "temporalized" in a way that is common for encodings of temporal domains as logic programs (see, e.g., [11,12,18]). In the following, we use the new predicate `holds(F,T)` to denote that feature `F` holds in the game position *at time* `T`. Time shall be encoded by natural numbers starting with `1`.

Definition 2. *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.*

1. *Each occurrence of* `init`(φ) *is replaced by* `holds`$(\varphi$`,1)`*, and each atom $p(t_1, \ldots, t_n)$ in the body of a clause for* `init` *is replaced by $p(t_1, \ldots, t_n, \texttt{1})$, provided that $p \neq$* `distinct`*.*
2. *Each occurrence of* `true`(φ) *is replaced by* `holds`$(\varphi$`,T)`*, and each* `next`(φ) *by* `holds`$(\varphi$`,T+1)`*.*
3. *Each occurrence of each atom $p(t_1, \ldots, t_n)$ is replaced by $p(t_1, \ldots, t_n, \texttt{T})$, provided that $p \notin \{\texttt{init}, \texttt{true}, \texttt{next}, \texttt{role}, \texttt{distinct}\}$, and* `distinct`$(t_1, t_2)$ *is replaced by* `not` t_1`=`t_2*.*

As an example, consider the temporal extension of the game rules in Figure 1.

```
role(player).
succ(a,b,T).
...
succ(g,h,T).
holds(cell(a,single),1).
holds(cell(Y,single),1) :- succ(X,Y,1).
```

```
legal(P,jump(X,Y),T) :- holds(cell(X,single),T),
                        holds(cell(Y,single),T),
                        twobetween(X,Y,T).
legal(P,jump(X,Y),T) :- holds(cell(X,single),T),
                        holds(cell(Y,single),T),
                        twobetween(Y,X,T).

holds(cell(X,nocoin),T+1) :- does(player,jump(X,Y),T).
holds(cell(X,double),T+1) :- does(player,jump(Y,X),T).
holds(cell(X,Number),T+1) :- holds(cell(X,Number),T),
                             does(player,jump(Y,Z),T),
                             not X=Y, not X=Z.

terminal(T)         :- not continuable(T).
continuable(T)      :- legal(player,Move,T).
goal(player,100,T)  :- not lonelycoin(T).
goal(player, 50,T)  :- lonelycoin(T), not threelonelycoins(T).
goal(player,  0,T)  :- threelonelycoins(T).
lonelycoin(T)       :- holds(cell(X,single),T).
threelonelycoins(T) :- holds(cell(X,single),T),
                       holds(cell(Y,single),T),
                       holds(cell(Z,single),T),
                       not X=Y, not X=Z, not Y=Z.

twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     twobetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,single),T),
                     onebetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,double),T),
                     nilbetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     onebetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,single),T),
                     nilbetween(Z,Y,T).
nilbetween(X,Y,T) :- succ(X,Z,T), true(cell(Z,nocoin),T),
                     nilbetween(Z,Y,T).
nilbetween(X,Y,T) :- succ(X,Y,T).
```

It is easy to see that the resulting program can be made more efficient by omitting the time argument in any predicate that

1. is not among the GDL keywords, and
2. does not depend on `true` in the original game description.

Thus, for instance, predicate $succ(x, y)$ in our example need not carry the time argument because its extension does not depend on the current game position.

This independence can be easily computed from the dependency graph for a set of GDL rules.

The ease with which GDL descriptions can be mapped onto a logic program with explicit time is the major reason for the expectation that Answer Set Programming can be a valuable addition to a general game playing system. The temporalized GDL rules allow us to encode the fact that *at time* t the players choose a joint legal move $A : R \mapsto \Sigma$ (where $R = \{r_1, \ldots, r_n\}$ are the roles in the game and Σ the symbol base) by the following additional facts.

$$A^{\texttt{does}}(t) \overset{\text{def}}{=} \{\texttt{does}(r_1, A(r_1), t)\texttt{.},\ \ldots,\ \texttt{does}(r_n, A(r_n), t)\texttt{.}\}$$

With this, the mapping of a GDL description to a logic program with time can be proved correct with regard to the semantics of GDL according to Definition 1.

Theorem 1. *Let* G *be a valid GDL description and* (R, S_1, T, l, u, g) *its semantics. For any finite sequence of legal joint moves and states, we have that*

$$S_1 \xrightarrow{A_1} S_2 \xrightarrow{A_2} \ldots S_k \xrightarrow{A_k} S_{k+1}$$

if and only if the standard model $\mathcal{M}$ *of* $ext(G) \cup A_1^{\texttt{does}}(1) \cup \ldots \cup A_k^{\texttt{does}}(k)$ *satisfies the following.*

- $S_i = \{f \in \Sigma : \mathcal{M} \models \texttt{holds}(f, i)\}$ *for each* $1 \leq i \leq k+1$ *and*
- $\mathcal{M} \models \texttt{legal}(r, A_i(r), i)$ *for each* $r \in R$ *and each* $1 \leq i \leq k$.

Proof. By induction. For $k = 1$ the claim follows from

- the replacement of $\texttt{init}(\varphi)$ by $\texttt{holds}(\varphi, 1)$ in $ext(G)$, and by the construction of S_1 in Definition 1;
- the replacement of $\texttt{legal}(\varrho, \alpha)$ and $\texttt{true}(\varphi)$ by, respectively, $\texttt{legal}(\varrho, \alpha, \texttt{T})$ and $\texttt{holds}(\varphi, \texttt{T})$ in $ext(G)$, and by the construction of l in Definition 1.

The induction step follows from

- the replacement of $\texttt{next}(\varphi)$, $\texttt{true}(\varphi)$, and $\texttt{does}(\varrho, \alpha)$ by $\texttt{holds}(\varphi, \texttt{T}+1)$, $\texttt{holds}(\varphi, \texttt{T})$, and $\texttt{does}(\varrho, \alpha, \texttt{T})$, respectively, in $ext(G)$, and by the construction of u in Definition 1;
- the replacement of $\texttt{legal}(\varrho, \alpha)$ and $\texttt{true}(\varphi)$ by, respectively, $\texttt{legal}(\varrho, \alpha, \texttt{T})$ and $\texttt{holds}(\varphi, \texttt{T})$ in $ext(G)$, and by the construction of l in Definition 1.

This result shows that the temporally extended logic program can be used to infer the evolution of the game position given a sequence of joint moves. As an example, consider the addition of the sequence of moves (cf. Footnote 1)

```
does(player,jump(a,d),1).
does(player,jump(c,e),2).
```

to the temporalized extension of the game description in Figure 1. It is easy to verify that the standard model for the resulting logic program includes each of the following.

```
legal(player,jump(a,d),1)       legal(player,jump(c,e),2)
holds(cell(a,nocoin),3)         holds(cell(b,single),3)
holds(cell(c,nocoin),3)         holds(cell(d,double),3)
holds(cell(e,double),3)         holds(cell(f,single),3)
holds(cell(g,single),3)         holds(cell(h,single),3)
terminal(3)                     goal(player,0,3)
```

4 Using ASP for Single-Player Games

Theorem 1 lays the foundation for the use of Answer Set Programming to perform endgame search for single-player games, that is, a complete, depth-restricted search starting in the current game position with the aim to find a winning sequence of moves within the given horizon. Because the standard model of a stratified program coincides with its only answer set (see, e.g., [15]), ASP can be used directly to determine the legality of a sequence of moves and the result from any current position. The basic idea, then, is to take the temporal extension of the GDL rules for a single-player game and to search for a sequence of moves that satisfies the following conditions.

1. Exactly one move is made at every point in time unless a terminal position has been reached.
2. Each move is legal when being played.
3. A terminal position is eventually reached.
4. The terminal position determines the intended goal value.

Any answer set that satisfies these constraints provides a solution to the game, that is, a sequence of moves that leads from the current position to a terminal position with the intended goal value.

In order to implement this search in a general game player, we need two common additions that have been defined for ASP [19]: a *weight atom*

$$m \; \{ \; p \; : \; d \; \} \; n$$

means that an answer set contains at least m and at most n different instances of atom p for which condition d holds (in the answer set). A *constraint* is a rule of the form `:-` $b_1, \ldots, b_k$ and excludes any answer set that satisfies all literals $b_1, \ldots, b_k$.

With the help of weight atoms and constraints, endgame search is performed by augmenting a temporally extended GDL specification for a single-player game by the clauses

```
1:  1{does(r,M,T) : move_domain(M)}1 :- not terminated(T).
    terminated(T)   :- terminal(T).
    terminated(T+1) :- terminated(T).
2:  :- does(r,M,T), not legal(r,M,T).                              (2)
3:  :- 0{terminated(T) : time_domain(T)}0.
4:  :- terminated(T), not terminated(T-1), not goal(r,g_max,T).
    :- terminated(1), not goal(r,g_max,1).
```

Here, r is assumed to be the (only) constant for which the game rules include the clause `role`(r), and natural number g_{max} stands for the goal value the game player is aiming for. These additional clauses provide a formal encoding of the four aforementioned conditions on the answer sets to provide a solution to the single-player game.

1: Predicate *terminated*(t) is used to indicate that a terminal position has been reached at (or before) time t. This auxiliary predicate is used to restrict the requirement for a legal move to every position before reaching a terminal one.
2: No answer set can stipulate a move that is not legal.
3: No answer set can have zero instances of *terminated*(t).
4: The state at the exact time of termination must have the desired goal value. (The last clause deals with the very special case that the game is already terminated at time 1.)

The ASP clauses in (2) require the definition of the domain of moves (using the predicate `move_domain`) according to the underlying game description. A suitable definition can be easily computed on the basis of the dependency graph for the given game description; see Section 5 for details. A similar definition is required for the domain of time, which is assumed to be given as $\{1, \ldots, n+1\}$ where n is the intended horizon for the endgame search.

If clauses (2) are added to the temporal extension of a GDL game according to Definition 2, then this amounts to a complete, depth-restricted search right from the initial position. Aiming instead at endgame search from the current position during game play, this can be easily achieved by substituting the collection of `holds`$(f, 1)$ facts, which result from the given `init`(f) clauses, by a collection of similar facts using the features that constitute the current position.

As an example, recall from the preceding section the temporal extension of the GDL rules of Figure 1. Let these be augmented by

```
coordinate(a).
...
coordinate(h).
move_domain(jump(X,Y)) :- coordinate(X), coordinate(Y).

1 { does(player,M,T) : move_domain(M) } 1 :- not terminated(T).
terminated(T)   :- terminal(T).
terminated(T+1) :- terminated(T).
:- does(player,M,T), not legal(player,M,T).
:- 0 { terminated(T) : time_domain(T) } 0.
:- terminated(T), not terminated(T-1), not goal(player,100,T).
:- terminated(1), not goal(player,100,1).
```

The answer sets for this program coincide with the solutions to the original game; an example is the answer that includes the following atoms.

```
does(player,jump(d,g),1)        does(player,jump(f,b),2)
does(player,jump(c,a),3)        does(player,jump(e,h),4)
terminal(5)                     goal(player,100,5)
```

The correctness of the method to solve single-player games with the help of Answer Set Programming is given by the following two theorems.

Theorem 2. *Consider a GDL description G with semantics (R, S_1, T, l, u, g) such that $R = \{r\}$ for some r. Let $\alpha_1, \ldots, \alpha_n$ and $S_2, \ldots, S_{n+1}$ be two sequences ($n \geq 0$) such that*

– $S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1}$,
– $S_{n+1} \in T$, *and*
– $(r, g_{max}, S_{n+1}) \in g$.

Then $ext(G) \cup (2)$ admits an answer set in which

$$\mathtt{does}(r, \alpha_1, 1) \quad \ldots \quad \mathtt{does}(r, \alpha_n, n) \tag{3}$$

are exactly the positive instances of predicate `does`.

Proof. From Theorem 1 and the fact that the only answer set for $ext(G)$ coincides with its standard model, and given that none of $S_1, \ldots, S_n$ is terminal,[3] it follows that there is an answer set for $ext(G)$ augmented by the first three clauses in (2) that includes (3) as the only instances of predicate `does`. This is also an answer set for the entire program $ext(G) \cup (2)$ since

– $(r, \alpha_i, S_i) \in l$ for each $i = 1, \ldots, n$,
– $S_{n+1} \in T$, and
– $(r, g_{max}, S_{n+1}) \in g$ and either $S_n \notin T$ or $n = 0$.

Theorem 3. *Consider a GDL description G with semantics (R, S_1, T, l, u, g) such that $R = \{r\}$ for some r. If $\mathcal{A}$ is an answer set for $ext(G) \cup (2)$, then there exists $n \geq 0$ such that*

$$\mathtt{does}(r, \alpha_1, 1) \quad \ldots \quad \mathtt{does}(r, \alpha_n, n) \tag{4}$$

are exactly the positive instances of predicate `does` *in $\mathcal{A}$, and there are states $S_2, \ldots, S_{n+1}$ such that*

– $S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1}$,
– $S_{n+1} \in T$, *and*
– $(r, g_{max}, S_{n+1}) \in g$.

Proof. Since $ext(G) \cup (2)$ contains only one clause with `does` in the head, the clauses labeled '1:' and '3:' in (2) ensure the existence of a finite sequence $\alpha_1, \ldots, \alpha_n$ (where $n \geq 0$) such that (4) are the only positive instances of `does` in $\mathcal{A}$. From Theorem 1 and the clauses labeled '2:' and '3:' in (2) it follows that there are states $S_2, \ldots, S_{n+1}$ such that

$$S_1 \xrightarrow{\{\alpha_1\}} \ldots \xrightarrow{\{\alpha_n\}} S_{n+1} \text{ and } S_{n+1} \in T$$

Finally, the constraints labeled '4:' in (2) ensure that $(r, g_{max}, S_{n+1}) \in g$.

[3] which follows from $S_1 \xrightarrow{\{\alpha_1\}} \ldots S_n \xrightarrow{\{\alpha_n\}} S_{n+1}$; cf. the conditions stated below (1)

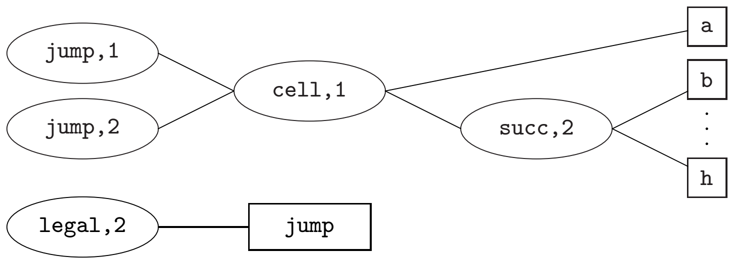


Fig. 2. An excerpt of a dependency graph for calculating domains of functions and predicates. Ellipses denote argument *positions* of functions or predicates, respectively, while rectangles denote function symbols themselves (including constants).

5 Experimental Results

We have implemented the ASP-based endgame search for single-player games using Clingo [14] as an off-the-shelf answer set solver in combination with the general game playing system described in [4]. The answer set programs are automatically generated from the game description. This requires to first compute the domain of moves as used in predicate `move_domain(M)` in clauses (2). The domains, or more precisely supersets thereof, of predicates and functions in a given game description can, in general, be computed by generating a dependency graph from the rules. The graph contains one node for every argument position of every function and predicate symbol, and one node for each function symbol itself (including each constant). An edge is added between an argument node and a function symbol node if the latter appears in the respective argument of a function or predicate in a rule of the game. An edge between two argument position nodes is added if there is a rule in the game in which the same variable appears in both arguments. Argument positions in each connected component of the graph share a domain, and the constants and function symbols in the connected components are the domain elements. Specifically, we take as the overall domain of the moves that of the second argument of `legal`.

Figure 2 shows a small excerpt of the dependency graph for our running example game. The first argument of *jump* and the first argument of *cell* are connected because they share variable X in the game rule

```
next(cell(X,nocoin)) :- does(player,jump(X,Y)).
```

Similarly, the second argument of *jump* and the first argument of *cell* are connected because they share variable X in the game rule

```
next(cell(X,double)) :- does(player,jump(Y,X)).
```

The `init` rules along with the *succ* facts then imply that the first and second argument of *jump* and the first argument of *cell* all share the domain

Game	ASP search time	FLUXPLAYER	Solution length
asteroids	0.11	2.60	10
asteroidsparallel	0.72	97.19	10
blocksworldparallel	9.78	1.80	3
duplicatestatelarge	0.16	17.54	14
eightpuzzle	77.11	$*$	30
factoringaperturescience	0.28	3.86	4
factoringdeterminate	0.05	2.14	5
knightmove (8×8-board)	$*$	$*$	(64)
knightstour (6×5-board)	7.33	117.70	30
peg	$*$	9.53	31
ruledepthlinear	0.19	9.60	49
ruledepthquadratic	$*$	12.70	44
statespacelarge	0.27	343.38	14
wargame01	$*$	39.53	48

Table 2. Results of a complete search for a variety of single-player games, with times given in seconds. Experiments were run on a 1.66GHz processor. Symbol $*$ indicates that the ASP system was aborted because it used more than 1 GB RAM, while Fluxplayer was aborted after not finding a complete solution within 30 minutes. (We enforced these rather strict limits in view of practical play, as endgame search is only one of several tasks during the contemplation phase or when deciding on the next move.)

$\{a, \ldots, h\}$. The dependency graph also contains a link from the second argument of `legal` to function *jump*. This is a consequence of the clause

```
legal(P,jump(X,Y)) :- true(cell(X,single)), ... .
```

Hence, the domain of moves in this game contains every possible instance of $jump(X,Y)$ with $X, Y \in \{a, \ldots, h\}$.

In addition to the domain of moves, the intended maximal depth for an endgame search defines the domain for the additional time variable which is used in the temporally extended program as well as in the clauses (2). Given domain restrictions for all variables, any existing answer set programming system can be employed to carry out the endgame search for single-player games.

We conducted experiments with all single-player games that were available at the time of publication through the online repository `games.stanford.edu`. Most of these games were used in past AAAI General Game Playing Competitions [6]. For the sake of reproducibility, we only report on the experiments where ASP search was applied straight away to the initial position. The results are shown in Table 2. It turns out that most of the games can actually be solved completely in reasonable, often very short, time that would have allowed a general game player to pre-compute a winning strategy during the "contemplation" phase. The results also show that in most cases the ASP search clearly outperforms the previously used forward search in our FLUXPLAYER—which over the past competitions proved to be the overall best performing system on single-player games [4].

6 Conclusion

We have shown how any game specified in the general Game Description Language can be mapped onto a normal logic program with time, and we have proved the correctness of this mapping against the formal game semantics for GDL. On this basis, we have illustrated how Answer Set Programming can be successfully deployed as a search method to assist general game players with tackling single-player games. Due to the closeness between GDL and ASP—in both syntax and semantics—the latter is ideally suited for performing blind search as part of a general strategy to solve single-player games.

We have substantiated this claim by reporting on experimenting with single-player games from past AAAI General Game Playing Competitions. The results show that actually most of these games could have been solved right from the start by an off-the-shelf ASP system. As games become more complex, they cannot be expected to be tackled by blind search alone, but still an ASP-based component constitutes a valuable addition to any general game playing system when it comes to performing depth-limited endgame search during game play.

The main limitation for the deployment of current ASP systems is the required grounding of the program, which easily becomes too large to be of practical use. Fortunately, a general game playing system can use the sizes of the domains for each variable to give an estimate of the size of the fully grounded, temporally extended game rules. On this basis, the system can easily decide on the fly whether or not it should execute the ASP-based endgame search in the current position.

For future work, we intend to investigate ways to use ASP for endgame search in multi-player games on the basis of Theorem 1. This will not be a straightforward extension of the method presented in this paper, because a single answer set determines a winning joint strategy for all players rather than a winning strategy against one or more opponents. Another direction of future work consists in investigating whether the very recently developed method of first-order Answer Set Programming [20] can be used to help a general game playing system perform endgame search in cases where grounding is too expensive.

Acknowledgment

This research was partially supported by *Deutsche Forschungsgemeinschaft* under Contract TH 541/16-1.

References

1. Pell, B.: Strategy Generation and Evaluation for Meta-Game Playing. PhD thesis, Trinity College, University of Cambridge (1993)
2. Kuhlmann, G., Dresner, K., Stone, P.: Automatic heuristic construction in a complete general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Boston, AAAI Press (2006) 1457–1462

3. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, AAAI Press (2007) 1134–1139
4. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, AAAI Press (2007) 1191–1196
5. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Chicago, AAAI Press (2008) 259–264
6. Genesereth, M., Love, N., Pell, B.: General game playing: overview of the AAAI competition. AI Magazine **26** (2005) 62–72
7. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University (2006) Available at: `games.stanford.edu`.
8. Schiffel, S., Thielscher, M.: A multiagent semantics for the game description language. In Filipe, J., Fred, A., Sharp, B., eds.: Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART), Porto, Springer (2009)
9. Apt, K., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann (1987) 89–148
10. Müller, M., Gasser, R.: Experiments in computer Go endgames. In: Nowakowski, R., ed.: Games of No Chance. Cambridge University Press (1996) 273–284
11. Kautz, H., Selman, B.: Planning as Satisfiability. In Neumann, B., ed.: Proceedings of the European Conference on Artificial Intelligence (ECAI), John Wiley & Sons, Ltd (1992) 359–363
12. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In Pollack, M.E., ed.: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Nagoya, Japan, Morgan Kaufmann (1997) 1169–1176
13. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138** (2002) 39–54
14. Potsdam Answer Set Solving Collection (2008) `potassco.sourceforge.net`.
15. Gelfond, M.: Answer sets. In van Harmelen, F., Lifschitz, V., Porter, B., eds.: Handbook of Knowledge Representation, Elsevier (2008) 285–316
16. van Gelder, A.: The alternating fixpoint of logic programs with negation. In: Proceedings of the 8th Symposium on Principles of Database Systems, ACM SIGACT-SIGMOD (1989) 1–10
17. Lloyd, J., Topor, R.: A basis for deductive database systems II. Journal of Logic Programming **3** (1986) 55–67
18. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153** (2004) 49–104
19. Niemelä, I., Simons, P., Soininen, T.: Stable model semantics of weight constraint rules. In Gelfond, M., Leone, N., Pfeifer, G., eds.: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). Volume 1730 of LNCS, El Paso, Springer (1999) 317–331
20. Lee, J., Meng, Y.: On loop formulas with variables. In Brewka, G., Doherty, P., Lang, J., eds.: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), Sydney (2008) 444–453