Planning for Biochemical Pathways: A Case Study of Answer Set Planning in Large Planning Problem Instances

Tran Cao Son and Enrico Pontelli

Department of Computer Science
New Mexico State University
tson,epontell@cs.nmsu.edu

Abstract. The paper describes an experiment of answer set planning in biochemical pathway planning. The focus is on large planning problem instances. It is shown that well-known planning techniques, such as planning graph analysis, landmarks recognition, and planning using landmarks are useful in answer set planning and can be easily incorporated in an answer set planning system.

1 Introduction

Over the past decade, answer set planning [6, 17, 26] has become a viable planning approach. It has been successfully applied in conformant planning [7, 25], conditional planning with sensing actions and incomplete information [28], planning with domain-specific knowledge [22], or dealing with user's preferences [23]. It has also been applied successfully in several real-world problems [1, 2]. Answer set planning builds on the idea of using answer set programming [20, 19] to support the process of reasoning about actions. The success of answer set planning rests on two factors. The first one is the availability of efficient answer set solvers, such as `smodels` [21], `dlv` [8], `cmodels` [16], and `ASSAT` [18]. The second factor is the combination of the simplicity and expressiveness of logic programming, which allows a simple representation and reasoning about action and change.

Despite its success and its elegance, and despite the development of excellent inference engines for answer set programming, answer set planning is not capable of handling large problem instances. In our experiments, answer set planners perform well in problem instances that admit short solutions, while it encounters difficulties in instances with long solutions—e.g., typically, when the length of the minimal solution is more than 20, the computation time grows beyond acceptable levels. One of the main reasons behind this problem is that answer set planning researchers did not concentrate on the development of special purpose planners. Rather, the focus has been on the development of methodologies for using answer set programming in planning. It is expected that large problem instances will be solvable by more efficient answer set solvers. While this is certainly true, it raises the question of whether the currently available technologies have more to offer.

Another reason leading to the fact that answer set planning cannot cope with large planning instances lies in the way solutions are computed in answer set programming.

Most inference engines rely on a two-phase computation. During the first phase, the program is grounded, and possibly simplified. The `lparse` is a typical program used for this phase. The actual solution (expressed by a collection of answer sets) will be computed by one of the answer set solvers in the second phase. This computing style does not allow for a direct application of well-known planning techniques to answer set planning (e.g., the use of the planning graph to simplify the domain, the use of heuristic in deciding which actions should be chosen, etc.) as many of these techniques require the ability to affect the way the computation search develops—inference engines for answer set programming typically do not expose the search process to the programmer. Furthermore, answer set planning puts a huge burden on the grounder, `lparse`, as the size of the grounded program for large problem instances is often too large to be produced or too large to be acquired by the answer set solver.

The limitation of the grounder has an important consequence on the representation of planning domains and instances, which sometimes requires a careful analysis of the domain and instances. For instance, if we wish to define an action $p(X, Y)$ where X and Y are variables with domain D_x and D_y respectively, a typical representation would lead to a clause of the form

$$action(p(X, Y)) \;{:}{-}\; d_x(X), d_y(Y).$$

Depending on the instance (D_x and D_y), `lparse` will simplify this clause and generate the correct set of actions—described by ground facts of the form $action(p(x, y))$. From the knowledge representation perspective, this is certainly a good practice, since it allows a simple specification of the problem instances (only facts need to be specified). This representation can, however, quickly increase the number of rules that the grounder has to deal with, as

(a) the number of parameters increases; and/or
(b) the size of the domain of the parameters increases.

As we will see later, this representation does increase the size of the grounding programs significantly.

In this work, we investigate the use of well-known planning techniques in the context of answer set planning. The planning techniques discussed in this paper involve a simplification of a planning problem based on *reachability analysis* [13] and *landmark* recognition, and the use of landmarks in planning [14].

We choose the *Biochemical Pathway* domain, one of the planning domains used in the recent International Planning Competition [12] as an example for our case study. The main reason behind this selection is the conceptual simplicity of the domain, and the need to deal with large instances. The following is an excerpt from the domain description available at [12]:

> *This domain is inspired by the field of molecular biology, and specifically biochemical pathways. "A pathway is a sequence of chemical reactions in a biological organism. Such pathways specify mechanisms that explain how cells carry out their major functions by means of molecules and reactions that produce regular changes. Many diseases can be explained by defects in pathways, and new treatments often involve finding drugs that correct those defects" [27].*

> *We can model parts of the functioning of a pathway as a planning problem by simply representing chemical reactions as actions. The biochemical pathway domain of the competition is based on the pathway of the Mammalian Cell Cycle Control as it described in [15] and modeled in [3].*

There are different kinds of basic actions corresponding to the different kinds of reactions that appear in the pathway. For example, one of the actions, called *associate*, is encoded in PDDL as follows[1].

```
(:action associate
 :parameters (?x1 ?x2 - molecule ?x3 - complex)
 :precondition (and (association-reaction ?x1  ?x2  ?x3)
    (available ?x1) (available ?x2))
 :effect (and  (not (available ?x1))
               (not (available ?x2)) (available ?x3)))
```

Fig. 1. Action `associate`

In the above specification, `?x1`, `?x2`, and `?x3` denote variables; the condition `(available ?x)` states that `?x` is available; `(association-reaction ?x1 ?x2 ?x3)` says that there is an association reaction between `?x1` and `?x2` to create `?x3`. This action creates the complex molecule `?x3`, by associating the two molecules `?x1` and `?x2`. This action is executable only if the two molecules `?x1` and `?x2` are available and it is known that the two molecules `?x1` and `?x2` can combine in a reaction to produce `?x3`.

A planning instance, in this domain, is given by a set of available molecules and the information encoding the knowledge about the possibility of creating new molecules by association, syntheses, and other types of interactions.

This paper discusses different ways to introduce current planning techniques, taken from advanced planning systems, in answer ser planning. The paper also presents some preliminary experimental results; these provide encouraging indication that answer set planning can be used to tackle large planning instances. We start the presentation with the basics of answer set planning, and a brief description of the $\mathbb{ASP}-\mathbb{PROLOG}$ system. We then discuss the problems faced by answer set planners in the biochemical pathway domains, discuss a preliminary implementation of the planning graph analysis and landmark recognition techniques, and their use in answer set planning.

2 Preliminaries

2.1 Answer Set Planning

We will use a variation of the high-level action description language $\mathcal{A}$ of [11] to represent action theories. We assume the presence of two finite, disjoint sets of names called *actions* and *fluents*. A *fluent literal* is either a fluent f or its negation $\neg f$. We will also say that f and $\neg f$ are complement of each other. For a fluent literal l, $\neg l$ denotes its

[1] A complete description of the domain is included in [24].

complement. A fluent formula is a propositional formula constructed from fluent literals. For a set of fluent literals γ, $\neg\gamma = \{\neg l \mid l \in \gamma\}$. For a set of fluent literal γ, l holds in γ if $l \in \gamma$. In such a language, an action domain D is a set of propositions of the following form:

$$a \textbf{ causes } f \textbf{ if } \psi \tag{1}$$

$$a \textbf{ executable } \psi \tag{2}$$

where f and ψ's are fluent literal and fluent formula, respectively, and a is an action. The axiom (1) represents a *conditional effect* of a, while axiom (2) states an executability condition of a.

A set of fluent literals is consistent if it does not contain two complementary fluent literals. A state (of D) is a maximal and consistent set of fluent literals. An action a is executable in a state s if there exists an executability condition (2) such that $\psi \subseteq s$. The effects of an action a in a state s is denoted by $e(a, s)$ and is given by

$$e(a, s) = \{f \mid a \textbf{ causes } f \textbf{ if } \psi \in D, \psi \subseteq s\}.$$

Given a state s and an action a executable in s, the state resulting from the execution of a in s, denoted by $Res(a, s)$, is defined by

$$Res(a, s) = s \cup e(a, s) \setminus \neg e(a, s).$$

Let $\alpha = [a_1; \ldots; a_n]$ be a sequence of actions; we will denote with $\alpha[i]$ the sequence of actions $\alpha[i] = [a_1; \ldots; a_i]$, where, by convention, $\alpha[0]$ denotes the empty sequence. The Res function can be easily extended to describe the effects of a sequence of actions. Given a domain description D, a state s and a sequence $\alpha = [a_1; \ldots; a_n]$ of actions, the final state after α is executed in s, $\Phi(\alpha, s)$, is defined as follows:

$$\Phi(\alpha, s) = \begin{cases} s & \text{if } n = 0 \\ \bot & \text{if } s' = \bot \text{ or } a_n \text{ is not executable in } s' \\ Res(a_n, \Phi(\alpha[n-1], s)) & \text{otherwise} \end{cases}$$

For an action sequence α and a state s, if $\Phi(\alpha, s) \neq \bot$ then we say that α is executable in s. α is executable in a set of states S if it is executable in every state $s \in S$.

A *planning problem* is specified by a triple $\langle D, s_0, \Delta \rangle$, where D is an action domain, s_0 is a state describing the initial state of the world, and Δ is a fluent formula (or *goal*), representing the goal state.[2] A sequence of actions $\alpha = [a_1; \ldots; a_m]$ is a *plan for* Δ if $\Phi(\alpha, s_0) \neq \bot$ and Δ holds in $\Phi(\alpha, s_0)$.

Given a planning problem $\langle D, s_0, \Delta \rangle$, answer set planning solves it by translating it into a logic program $\Pi(D, s_0, \Delta)$, whose answer sets correspond to plans for Δ. The signature of $\Pi(D, s_0, \Delta)$ includes terms corresponding to fluent literals and actions of D, as well as non-negative integers used to represent time steps. We often write $\Pi(D, n)$ to denote the restriction of $\Pi(D, s_0, \Delta)$ to time steps between 0 and n (i.e., plans of length at most n). Atoms of $\Pi(D, s_0, \Delta)$ are formed using the following (sorted) predicate symbols:

[2] For simplicity of our discussion, we will assume that Δ is a set of fluent literals. Encoding the goal can be done as in [22].

- $fluent(F)$ is true if F is a fluent;
- $literal(L)$ is true if L is a fluent literal;
- $contrary(L, L')$ is true if L is the complement of literal L';
- $h(L, T)$ is true if the fluent literal L holds at time step T;
- $occ(A, T)$ is true if the action A occurs at time step T;
- $poss(A, T)$ is true if the action A is executable at time step T.

In our representation, letters T, F, L, and A (possibly indexed) (resp. t, f, l, and a) are used to represent variables (resp. constants) of sorts time, fluent, fluent literal, and action correspondingly. For a set of fluent literals γ, we define:

$$h(\gamma, T) = \{h(l, T) \mid l \in \gamma\} \quad not\ h(\gamma, T) = \{not\ h(l, T) \mid l \in \gamma\} \quad \neg\gamma = \{\neg l \mid l \in \gamma\}$$

The set of rules of Π is divided into the following five subsets:

- *Dynamic causal laws:* for each statement of the form (1) in D, the rule:[3]

$$h(f, T{+}1) \leftarrow occ(a, T), h(\psi, T) \tag{3}$$

  belongs to $\Pi(D, s_0, \Delta)$. This rule states that if the action a occurs at time step T and the precondition ψ holds at that time step then f holds afterward.
- *Executability conditions:* for each statement of the form (2) in D, $\Pi(D, s_0, \Delta)$ contains the following rule:

$$poss(a, T) \leftarrow h(\psi, T) \tag{4}$$

$$\leftarrow occ(a, T), not\ poss(a, T) \tag{5}$$

  This rules state that a is executable at the time step T iff there exists one of the executability conditions of the form (2) such that ψ holds at time step T.
- *Initial state:* $\Pi(D, s_0, \Delta)$ contains the rule

$$h(s_0, 0) \leftarrow$$

- *Action generation:* $\Pi(D, s_0, \Delta)$ contains the rule

$$1\ \{occ(A, T) : action(A)\}\ 1 \leftarrow$$

  which states that, at every time step, exactly one action must occur.
- *Goal:* $\Pi(D, s_0, \Delta)$ contains the constraint

$$\leftarrow not\ h(\Delta, n)$$

- *Inertia:* $\Pi(D, s_0, \Delta)$ contains the following rule for the inertial law:

$$h(L, T) \leftarrow h(L, T-1), not\ h(\neg L, T), T > 0 \tag{6}$$

  This rule says that a literal L holds at time step T if it holds at the previous time step and its negation does not hold at T.

[3] In practice, the atom $h(\psi, T)$ has to be replaced by a conjunction of atoms for each literal in ψ.

- *Auxiliary rules:* $\Pi(D, s_0, \Delta)$ also contains the following rules:

$$literal(F) \leftarrow fluent(F) \quad (7)$$

$$literal(\neg F) \leftarrow fluent(F) \quad (8)$$

$$contrary(F, \neg F) \leftarrow fluent(F) \quad (9)$$

$$contrary(\neg F, F) \leftarrow fluent(F) \quad (10)$$

The first constraint stops two complementary fluent literals from holding at the same time. The last four rules are used to define fluent literals and complementary literals.

The next theorem states that the program $\Pi(D, s_0, \Delta)$ correctly solves the planning problem $\langle D, s_0, \Delta\rangle$ (see, e.g., [22, 29]).

Theorem 1. *Given a planning problem* $\langle D, s_0, \Delta\rangle$*,*
- ∘ *for each plan* $a_1, \ldots, a_n$ *for* Δ*, the program* $\Pi(D, n) \cup \{occ(a_i, i-1) \mid i = 1, \ldots, n\}$ *is consistent;*
- ∘ *if* A *is an answer set of* $\Pi(D, n)$ *then* $a_1, \ldots, a_n$ *is a plan for* Δ *where* $occ(a_i, i-1) \in A$ *for* $i = 1, \ldots, n$.

2.2 $\mathbb{ASP} - \mathbb{PROLOG}$

In order to support our development activities, we need a framework with the following characteristics:

- It provides access to an inference engine for answer set programming—to allow answer set planning;
- It provides access to a general purpose, declarative programming framework, which allows arbitrary forms of reasoning and transformation of an action theory.

For this project, we selected a recently developed framework called $\mathbb{ASP} - \mathbb{PROLOG}$ [10]. $\mathbb{ASP} - \mathbb{PROLOG}$ is a fully modular system, which allows the integration of modules written in Prolog with modules written in the SMODELS flavor of answer set programming. Each $\mathbb{ASP} - \mathbb{PROLOG}$ program is a composition of modules. It allows programmers to compose modules expressed using different flavors of logic programming, including Prolog, Constraint Logic Programming, and answer set programming. Each program is composed of a main module—at this time restricted to be a Prolog or CLP module and encoded in CIAO Prolog[4]—and a collection of modules organized according to an acyclic graph structure (e.g., see Fig. 2).

Each Prolog module is allowed to import predicates defined in other modules, through an import declaration, and to export predicates defined within the module (all solutions to the given predicates are exported). Similarly, each ASP module is allowed to import and export predicates.

Importing from a Prolog module m will effectively achieve the effect of enriching the local module with the least Herbrand model of m projected over its exported predicates. Importing from an ASP module will allow to either perform skeptical reasoning—e.g., in

[4] http://www.clip.dia.fi.upm.es/Software/Ciao

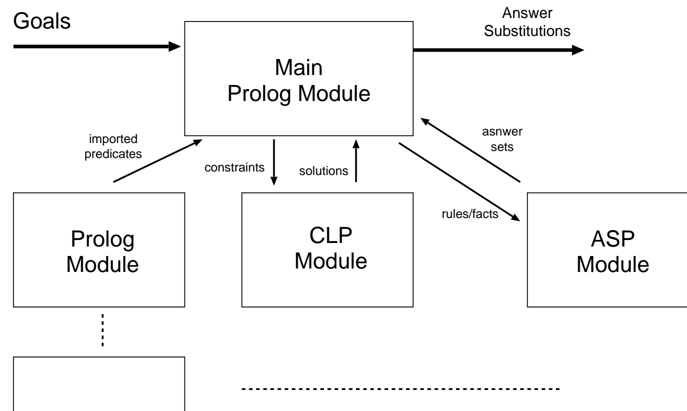

Fig. 2. Program Organization in $\mathbb{ASP} - \mathbb{PROLOG}$

```
:- import(aspmodule1, 'aspmodule1.lp').
...
... :- ... aspmodule1:p(5) ...
```

`aspmodule1:p(5)` will succeed only if `p(5)` holds in each answer set of `aspmodule1`—or to access each individual answer set—e.g., in

```
:- import(aspmodule1, 'aspmodule1.lp').
...
... :- ... aspmodule1:model(Q), Q:p(5) ...
```

the conjunction `aspmodule1:model(Q), Q:p(5)` will succeed if `p(5)` holds in at least one answer set of `aspmodule1`.

Prolog modules are also allowed to perform meta-operations on other modules—e.g., they can use `clause` to read the clauses of a module, and they can use `assert` and `retract` to add or remove rules.

In the context of this project, answer set programming modules are employed to encode the answer set planners, while the Prolog modules are used to perform analysis of action theories and to drive the planning process (e.g., implement heuristics). Prolog is particularly advantageous, thanks to its ability to easily manipulate the syntax of action theories and its flexible search and backtracking mechanisms.

3 Describing Biochemical Pathway in Answer Set Planning

The problem of finding a biochemical pathway can be represented as a planning problem. The properties $level$, $simple$, and $complex$ (representing, correspondingly, the substrate level of a molecule, a simple molecule, and a complex molecule) can be specified as domain predicates, and the two rules

$$molecule(X) \leftarrow simple(X)$$
$$molecule(X) \leftarrow complex(X)$$

encode the fact that every molecule is either simple or complex. There are five actions:

- ◦ $choose(X, L_1, L_2)$—the action requires that X is a simple molecule and L_1 is a higher substrate level than L_2; the effects of this action are that the simple molecule is chosen and L_1 indicates the substrate level considered.
- ◦ $initialize(X)$—creates the simple molecule X if it has been chosen;
- ◦ $associate(X_1, X_2, X_3)$—this is an action if the association reaction between X_1, X_2, and X_3 exists; the effect of this action is to create the molecule X_3 if the two molecules X_1 and X_2 are available;
- ◦ $associate_with_catalyze(X_1, X_2, X_3)$—creates the molecule X_3 if the two molecules X_1 and X_2 are available and a catalyzed association reaction between X_1, X_2, and X_3 exists;
- ◦ $synthesize(X_1, X_2)$—creates the molecule X_2 from the molecule X_1 if it is available and there is a synthesis reaction between X_1 and X_2.

A planning problem in this domain is characterized by the following parameters:

- The number of simple molecules;
- The number of complex molecules;
- The number of substrate levels;
- The number of association reaction combinations;
- The number of catalyzed association reaction combinations; and
- The number of synthesis reaction combinations.

The number of actions in this domain grows very fast. The next table describes some of the biochemical planning problems, used in the recent planning competition[5], in terms of the parameters listed above. The last two columns indicate the number of potentially useful actions and the length of a known plan in each problem.

Problem	# Simple Molecules	# Complex Molecules	# Number Subs	# Asso. Combi.	# Cata. Combi.	# Syn. Comb.	# Actions	Plan length
1	16	9	4	7	5	0	75	5
2	12	26	4	14	0	14	75	10
3	19	24	4	21	5	10	111	14
4	22	46	4	33	2	22	145	14
5	22	66	7	53	0	25	254	26
10	39	117	14	99	9	102	795	84
15	45	143	18	120	12	149	1135	?

Table 1. Biochemical Pathways as Planning — Problem and Parameters

3.1 Using Answer Set Planning: Some Problems

The first problem we have to deal with when using answer set planning to tackle this planning domain is the size of the ground instances. Besides the set of laws describing the actions' effects and executability conditions, the set of action generation rules is very large. The current parser `lparse` is effective only for problems with short solutions. This led us to search for ways to reduce the size of the ground instances.

[5] See `http://zeus.ing.unibs.it/ipc-5/`

One of the commonly used techniques in planning is to examine the planning graph [4]. Intuitively, a planning graph is a structure consisting of alternative sets of fluents and actions, $F_0, A_0, \ldots, F_n, A_n, \ldots$. F_i is the set of fluents that can be reached by every possible action sequences whose length is less than or equal to i, and A_i is the set of possible actions that can be executed after i actions. The planning graph has been useful in analyzing planning problems and extracting heuristics [5]. Given a planning problem, a planning graph can be easily computed in Prolog, using the following rules:[6]

```
forward_closure(0, Fluents, Actions) :-
   findall(G,(fluent(G), initially(G)), Fluents),
   findall(A,(action(A), executable(A,[])), Actions).
forward_closure(Time, Fluents, Actions) :-
   Time1 is Time-1,
   forward_closure(Time1, PrevFluents, PrevActions),
   collect_applicable(PrevFluents,NewActions),
   collect_consequence(NewActions,NewFluents),
   union(PrevFluents, NewFluents, Fluents),
   union(PrevAction,NewActions,Actions).
```

where `collect_applicable` determines the actions whose (positive) executability conditions are met by `PrevFluents`, and `collect_consequences` collects all the positive consequence of the actions in `NewActions`. The collection of actions and consequences can be easily realized using appropriate instances of the `findall` predicate—e.g., for the consequences:

```
collect_consequences([],[]).
collect_consequences([Action|Rest],Fluents) :-
   findall(Res,(causes(Action,Res1,_),
                member(Res,Res1),
                \+(Res=neg(_)
               ),              List1),
   collect_consequences(Rest,List2),
   append(List1,List2,Fluents).
```

A planning graph can provide us with the set of actions that can be possibly executed given the initial state of the world, and the set of fluents that can be possibly changed their value from $false$ to $true$. This information allows us to *(1)* remove actions that can never be executed, *(2)* remove fluents that never change value, and *(3)* simplify the remaining actions. The above can be repeated until every action can be possibly executed and every fluent might change its value from $false$ to $true$. The planning graph can also be used in a backward fashion, to eliminate actions that are irrelevant to the goal. This can be done using the following Prolog rules:

```
back_closure(0,Fluents,Actions) :-
   findall(G,goal(G),Fluents), Actions=[].
back_closure(Time, Fluents, Actions) :-
```

[6] Simplified to enhance readability.

```
Time1 is Time-1, back_closure(Time1,RFluents,RActions),
findall(A, (action(A), causes(A,Cons),
            intersect(Cons,RFluents)),NewActions),
findall(F1, (member(A,NewActions),
             executable(A,Cons), member(F1,Cons),
             fluent(F1)), Set1),
findall(F2, (member(A,NewActions), causes(A,Cons),
             member(F2,Cons),fluent(F2)), Set2),
union(RActions,NewActions, Actions),
union(RFluents, Set1, Set2, Fluents).
```

The result of the execution of this module are described in Table 2.

Problem	Forward		Forward + Backward		Plan Found
	# Fluents	# Actions	# Fluents	# Actions	by `smodels`
1	61	75	45	37	Yes
2	67	75	55	34	Yes
3	95	111	76	51	Yes
4	115	145	93	63	Yes
5	142	254	120	163	No
10	250	795	211	638	No
15	297	1135	252	953	No

Table 2. Simplifications due to forward and backward planning graph analysis

It should be noted that the application of this method allows for a domain representation which is less susceptible to the specification of actions and fluents. For example, we examine the PDDL representation of the domain and define the `associate` action by the rule

```
action(associate(X,Y,Z)):-
       molecule(X), molecule(Y), complex(Z),
       association_reaction(X,Y,Z).                    (*)
```

In doing so, `association_reaction(X,Y,Z)` becomes a static property of the domain. This is slightly different than the encoding of [12] where the representation

```
action(associate(X,Y,Z)):-
       molecule(X), molecule(Y), complex(Z).           (**)
```

is used. In this case `association_reaction(X,Y,Z)` is viewed as a fluent. The second representation (**) will be better than the first one (*) if the information on whether or not `association_reaction(X,Y,Z)` holds is not a static relation. This encoding, will, however, increase the size of the grounded program tremendously comparing to the first encoding as the number of `associate(X,Y,Z)` actions is now the product of the square of the number of molecules and the number of complex molecules. As an example, consider the first instance of the problem (Table 1). In this instance, there are 25 molecules, 9 complex molecules, and 7 possible association reactions among the molecules. Thus, the second encoding will yield 25*25*9=5625 possible `associate` actions while the first encoding records only 7 possible actions.

Planning graph analysis allows us to remove the actions that might be defined but are not possible in the domain. We experimented with both representations and found that the number of actions that are retained for the plan computation step is the same. For this reason, there is little change in the number of actions between the two tables if only `forward` analysis is used as we used the first encoding in our experiment.

3.2 Landmarks Recognition

The size of the ground program does matter in the sense that if the grounder `lparse` cannot finish its work, our quest of computing a plan using answer set programming cannot even begin. The second problem that answer set planning needs to face is the size of the search space. To this end, we investigate another technique, called *ordered landmarks*, that has been developed in [14] and is currently implemented in various planners, such as FF [13]. Let us recall some of the definitions.

Definition 1. *Given a planning problem $\mathcal{P} = \langle D, s_0, \Delta\rangle$, a fluent literal l is called a* landmark *of $\mathcal{P}$ iff for every solution $\alpha = [a_1; \ldots; a_k]$ of $\mathcal{P}$, there exists an integer i, $1 \leq i \leq k$, such that $l \in \Phi(\alpha[i], s_0)$.*

Intuitively, a landmark l represents a "necessary" precondition that needs to be satisfied before (or at the same time) the goal can be achieved.

Example 1. Let $D = \{a$ **causes** f **if** h b **causes** f **if** $h, \neg f$ c **causes** $h\}$ It is easy to see that h is a landmark of the problem $\langle D, \{\neg f, \neg h\}, \{f\}\rangle$.

Definition 2. *Given a planning problem $\mathcal{P} = \langle D, s_0, \Delta\rangle$ and two fluent literals l and l'. There is a* necessary order *between l and l', denoted by $l \rightarrow_n l'$, iff $l' \notin s_0$ and for every action sequence $\alpha = [a_1; \ldots; a_k]$, if $l' \in \Phi(\alpha, s_0)$ then $l \in \Phi(\alpha[n-1], s_0)$.*

The ordering between l and l' states that l is necessary for achieving l'. In Example 1, there is a necessary order between h and f.

Definition 3. *Let $\mathcal{P} = \langle D, s_0, \Delta\rangle$ be a planning problem and l, l' two fluent literals.*

1. *Let $S_{(l,\neg l)}$ be the set of states s such that there exists an action sequence $\alpha = [a_1; \ldots; a_k]$, $s = \Phi(\alpha, s_0)$, $l' \in e(a_k, s)$, and $l \notin \Phi(\alpha[i], s_0)$ for $0 \leq i \leq k$.*
2. *l' is* in the aftermath of *l if, for all states $s \in S_{(l,\neg l')}$, and all solutions $\alpha = [a_1; \ldots; a_k]$ of the planning problem $\langle D, s, \Delta\rangle$, there are $1 \leq i \leq j \leq k$ such that $l \in \Phi(\alpha[i], s)$ and $l' \in \Phi(\alpha[j], s)$.*
3. *There is a* reasonable order *between l and l', denoted by $l \rightarrow_r l'$, if l' is in the aftermath of l and*

$$\forall s \in S_{(l,\neg l')} : \forall \alpha = [a_1, \ldots, a_k] : l \in \Phi(\alpha, s) \rightarrow \exists i : a_i \textbf{ causes } \neg l' \textbf{ if } \psi \in D.$$

Intuitively, $S_{(l,\neg l')}$ is the set of states in which l' is just added to the state and l has not been achieved yet. The aftermath relation states that for every solution starting from $S_{(l,\neg l')}$, l' must be achieved simultaneously with l or at some later point. $l \rightarrow_r l'$ states that for every $s \in S_{(l,\neg l')}$, every action sequence achieving l deletes l' at some point. This implies that a planner can try to achieve a state $\neg l'$ before try to achieve the goal l.

The main problem in utilizing this knowledge is that the computation of the aftermath ordering or reasonable ordering among landmarks is PSPACE-complete. As such, in systems employing this technique, only an approximation of this ordering is computed and used in the search process. The key ideas in this task are:

- Compute a graph (called LGG), consisting of the landmark candidates with an approximated greedy necessary order between them;
- Remove from LGG the candidates that cannot be proved to be landmarks; and
- Use the landmarks as intermediate goals in the search for a solution.

The search starts with the goal as the disjunction of all leaf nodes of LGG. As soon as one disjunct is satisfied, the LGG is updated, by removing the node corresponding to the achieved landmark and the links to and from this node. The set of leaf nodes is then recomputed (as a disjunction) and set as the new goal. The planner continues until all landmarks have been achieved.

3.3 Implementation

The Prolog preprocessor described earlier has been extended to support the LGG computation. The graph is described by a list of nodes and a list of edges. The main predicate for the LGG computation is:

```
hoffmann(Fluents,Actions, Nodes, Edges) :-
   compute_goal_state(Goals,Fluents),
   compute_initial_state(Init),
   candidate(Goals,[],[],Nodes1,Edges1,Fluents,Actions),
   findall([neg(X),X],
              (member(neg(X),Nodes1), member(X, Nodes1)), CEdges),
   append(CEdges,Edges1,Edges2),
   verify_landmarks(Nodes1,Edges2,Fluents,Init,Actions,Goals,
                                                        Nodes,Edges).
```

The core of the computation is performed by the predicate `candidate`, which navigates the dependence graph, composed of executability conditions and effects of actions, to locate elements that represent potential landmarks. The `verify_landmarks` procedure is simply used to verify that the elements collected in the LGG graph are indeed reachable w.r.t. the given initial state of the action theory.

This recursive predicate `candidate` is defined as follows:[7]

```
candidate([],N,E,N,E,_,_).
candidate([A|B],N,E,FinalNodes,FinalEdges,Fluents,Actions) :-
   level(A,0),!,
   candidate(B,N,E,FinalNodes,FinalEdges,Fluents,Actions).
candidate([A|B],N,E,FinalNodes,FinalEdges,Fluents,Actions) :-
   level(A,L2),
   findall(X,(member(X,Actions),causes(X,List,_),
              member(A,List),level(X,L1),L1 =:= L2-1
             ),Actions),
```

[7] The definition as been simplified for readability.

```
findall(Y,appears_always(Y,Actions),Cback),
findall(Y1,appears_forward(Y1,Actions),Cforward),
append(Cback,B,B21), append(B21,Cforward,NewB),
findall([Z,A],(member(Z,Cback),level(Z,ZZ),ZZ>0),NewEdges1),
findall([N1,N2],(member(N1,Cback),level(N1,ZZ1), ZZ1>0,
                 member(N2,Cforward)), NewEdges3),
append(E,NewEdges1,NewEdges2),
append(NewEdges3,NewEdges2, Edges1),
candidate(NewB,[A|N],Edges1,FinalNodes,FinalEdges,Fluents,Actions).
```

The candidate procedure iterates until the set of items of interest (initialized to the set of goals) becomes empty. Candidate nodes are added to the set if they have a level greater than 0 (i.e., they are not part of the initial state) and they either

– appear in the preconditions of all the actions that in one step produce another LGG node (predicate appears_always), or
– appear in the consequence of all the actions that in one step produce another LGG node (predicate appears_forward).

This is intuitively illustrated in figure 3. The edges are created in the obvious manner to link fluents connected by the selected actions.

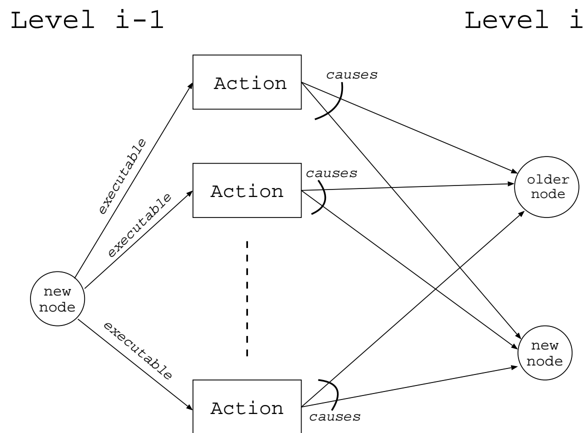

Fig. 3. Intuition behind the LGG construction

4 Experimentation

We implemented the planning graph and the LGG computation in Prolog. The simplified planning problem is then fed into smodels. In all, we were able to solve 5 problems from the set of problems given at the planning contest, a results comparable with most of the planning systems competing in the IPC'2006 (see [12]). The first four instances can be solved using a single call to smodels (as shown in Table 2). For the 5th instance,

we use $\mathbb{ASP} - \mathbb{PROLOG}$ in the following way. We have a Prolog module that performs the following activities:

- It takes *(a)* an answer set program representing the instance with the parameter `length`, and *(b)* a disjunctive goal consisting of the leaf nodes of the landmark graph, and calls `smodels` to find a plan for the disjunctive goal; the value of the parameter `length` is iteratively changed from 1 to 2, to 3, etc., until an answer set is returned (as described in [9].
- It analyzes the answer set, creates the new initial state and the new goal (by removing achieved goals from the landmark graph), and repeats the first step.

We observed that the system does not require backtracking on the choice of satisfied landmark. Analyzing the problem and the landmark graph, we found that the landmark graph does indeed provide an ordering that can be achieved one by one. Whether this is always the case (even for this domain) is an important question that is currently under investigation.

5 Conclusions

In this paper, we described our preliminary investigation of how to bring state-of-the-art techniques developed by the planning community to the realm of answer set planning. Our preliminary results shows that the adoption of logic programming technologies does not prevent the use of simplification techniques (such as reachability analysis and landmarks identification), and these techniques can be introduced in an elegant and declarative manner. In particular, the use of logic programming (specifically, Prolog) significantly simplifies the problem of implementing different forms of analysis of the action theories.

We demonstrated our approach on a challenging planning instance, dealing with a problem from systems biology and obtained from the most recent International Planning Competition.

References

1. M. Balduccini and M. Gelfond. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4,5):425–461, 2003.
2. M. Balduccini, M. Gelfond, and M. Nogueira. Answer Set Based Design of Knowledge Systems. *Annals of Mathematics and Artificial Intelligence*, 2006.
3. BIOCHAMP. `http://contraintes.inria.fr/BIOCHAM/EXAMPLES/cell_cycle/cell_cycle.bc`.
4. A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann Publishers, San Francisco, CA, 95.
5. D. Bryce, S. Kambhampati, and D. Smith. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research*, 26:35–99, 2006.
6. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of European conference on Planning*, pages 169–181, 1997.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1-2):157–211, 2003.

8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.
9. O. Elkhatib, E. Pontelli, and T. C. Son. $\mathbb{ASP} - \mathbb{PROLOG}$: A System for Reasoning about Answer Set Programs in Prolog. PADL-04, 148-162, 2004.
10. O. Elkhatib, E. Pontelli, T.C. Son. A Tool for Knowledge Base Integration and Querying. *AAAI Spring Symposium*, AAAI/MIT Press, 2006.
11. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.
12. A. Gerevini, Y. Dimopoulos, P. Haslum, and A. Saetti. 5th international planning competition — deterministic part. `http://zeus.ing.unibs.it/ipc-5/`.
13. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
14. J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278, 2004.
15. K. Kohn. Molecular interaction map of the mammalian cell cycle control and dna repair systems. *Mol Biol Cell*, 10(8), 1999.
16. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and NonMonotonic Reasoning Conference (LPNMR'04)*, volume 2923, pages 346–350. Springer Verlag, LNCS 2923, 2004.
17. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.
18. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, pages 112–117, 2002.
19. V.W. Marek and M. Truszczyński. Stable Models as an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*, Springer Verlag, 1999.
20. I. Niemelä. Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. *AMAI*, 25(3–4):241–273, 1999.
21. P. Simons, N. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
22. T.C. Son, C. Baral, N. Tran, and S. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Trans. Comput. Logic*, 7(4):613–657, 2006.
23. T.C. Son and E. Pontelli. Planning with Preferences using Logic Programming. *Theory and Practice of Logic Programming*, 6:559–607, 2006.
24. T.C. Son and E. Pontelli. Planning for Biochemical Pathways: A Case Study of Answer Set Planning in Large Planning Problem Instances. Technical Report. NMSU-CS-2007-004. 2007.
25. T.C. Son, P.H. Tu, M. Gelfond, and R. Morales. An Approximation of Action Theories of $\mathcal{AL}$ and its Application to Conformant Planning. In *Proceedings of the the 7th International Conference on Logic Programming and NonMonotonic Reasoning*, pages 172–184, 2005.
26. V.S. Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In *Proceedings of the International Conference on Logic Programming*, pages 233–247, 1995.
27. P. Thagard. Pathways to biomedical discovery. *Philosophy of Science*, 70, 2003.
28. P. H. Tu, T. C. Son, and C. Baral. Reasoning and Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Logic Programming. *Theory and Practice of Logic Programming*, 7:1–74, 2006.
29. H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.