Faculty of Mathematics, Physics, and Informatics
Comenius University, Bratislava

Enhancing ASP-based Planning by Heuristic Graph-search Techniques

Michal Čertický

TR-2011-032

Technical Reports in Informatics

Enhancing ASP-based Planning by Heuristic Graph-search Techniques

Michal Čertický

Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava, Slovakia
`certicky@fmph.uniba.sk`

Abstract. In recent years, we have seen several attempts to use the ASP (Answer Set Programming) for planning. Typically, the whole planning domain is encoded into a large ASP meta-program, where time-action occurrence predicate $occurs(A, T)$ expresses when the actions are executed in a resulting plan. Since the complexity of answer set computation is exponential in the number of input atoms, we propose a technique of dividing this meta-program into several smaller ones - one for every state transition. Actual plan finding can then be outsourced to a graph-search algorithm. This method, called *"decomposition trick"*, increases the performance of planning with ASP semantics dramatically and makes it possible to produce much longer plans. We introduce the prototype of ASP planner $GRASP$ using this technique and provide the experimental comparison with an alternative planning system DLV^K.

1 Background

Since the year 1995, when the paper called Relating Stable Models and AI Planning Domains was published by Subrahmanian and Zaniolo, we have seen several attempts to use ASP solvers for planning tasks.

The ASP semantics enables us to elegantly deal with incompleteness of knowledge, and makes the representation of action's properties easy, due to nonmonotonic character of negation as failure [Lifschitz, 2002].

As an example of using ASP for planning we can mention, besides the original paper [Subrahmanian-Zaniolo, 1995], also the theoretic proposals in [Lifschitz, 2002], [Niemelä, 1998] and [Baral, 2003], or more implementation-oriented system descriptions published in [Eiter et al., 2002], [Gebser-Grote-Schaub, 2010] or the most recent [Gebser et al., 2011].

2 Typical Approach

These systems or approaches, despite varying in some details, have one thing in common. They encode the planning domain into a single logic meta-program composed of the following:

(1) Representation of initial world state, goal, and (optionally) a background knowledge.
(2) Set of universal axioms representing domain-independent causal dynamics.
(3) Collection of facts enumerating all the valid time steps:

$$time(1).\ time(2).\ \ldots\ time(maxTime).$$

(4) Description of preconditions and effects of every action of our domain.
(5) Action-time occurrence choice rules like these ones[1]:

$$occurs(A, T) \leftarrow action(A), time(T), not\ \ -occurs(A, T).$$

$$-occurs(A, T) \leftarrow action(A), action(A'), time(T), occurs(A', T), A \neq A'.$$

Then in the end, they use one of available ASP solvers (DLV [Leone et al., 2006], SMODELS [Syrjänen, 2001], iClingo [Gebser et al., 2008], etc.) to compute the answer sets of this meta-program. Each of these answer sets then represents one resulting plan.

Choice rules (type (5)) cause solver to generate several answer sets, where different actions occur in different time steps. Rules of a type (2) and (4) then generate the facts representing successive world states at individual time steps, based on the initial world configuration (1). Finally, rules of a type (3) and (4) filter out all the answer set candidates representing invalid plans: plans where action occurred even though its preconditions were not met, plans not leading to a goal, etc.

3 Problem

According to [Simons, 2002], finding the answer sets of a logic program is an NP-complete problem. The answer set finding algorithms (specifically the one used in Smodels solver[2]) are exponential in a number of atoms of a ground input program.

Current ASP planners generate the answer sets of a single meta-program, which represents the whole planning task. The problem is, that since the input meta-program represents the whole panning task, it is too big, making this technique unusable in larger domains.

Splitting the problem into several smaller logic programs could allow us to solve the task per partes. Since the ASP solving algorithms are **exponential in the number of input atoms**, solving even a large number of small inputs is potentially faster than solving one big input program.

[1] This example is a simplification of action-time occurrence choice rules from [Baral, 2003]. Slightly different encodings can be found in different papers, but the idea behind this remains the same.

[2] Note that the computational engines of ASP systems like DLV or Smodels use very similar procedures [Faber-Leone-Pfeifer, 2001].

4 Suggestion for Improvement

We suggest using more traditional methods (preferably a heuristic graph-search algorithm like A^* [Hart-Nilsson-Raphael, 1968]) for planning itself, and using the ASP solvers only for calculating the **transitions between world states** (small sub-problems). We will call this technique a **"decomposition trick"**.

Decomposition trick allows us to:

- Reduce the size of logic programs passed to the ASP solver.
- Solve a number of state transitions in parallel.
- Outsource part of planning task to a faster graph-search algorithm.
- Still use the ASP semantics for reasoning and knowledge representation.

Describing graph-search algorithms in detail is not necessary for the purposes of this paper. It is sufficient to explain the relation between our planning problem and an oriented edge-labeled graph $G = (N, E)$:

Any **node** $s \in N$ of a graph corresponds to a single **world state**. A **labeled edge** $(s_1, s_2) \in E$ with a label a then means, that the execution of **action** a in a world state s_1 transitions the world into a state s_2. Existence of such a node also means, that the action a is executable in s_1. Any **path** within such graph leading from an initial node s_0 to one of the nodes corresponding to a goal state represents a resulting plan.

During the search, we traverse the graph one node at a time. We start at a node representing an initial state - s_0. This node contains the set of all literals that hold in the initial state. Then we use the function **expandNode(s_0)** to generate the children of this node - they represent all the states that we can reach from s_0 by applying a single action. Depending on the search algorithm, we then chose one of the children and check whether the goal is satisfied. If it isn't, we expand it. See figure 1 for the pseudocode of **expandNode(S)** function.

Function called **getPreConstraints(A)** prepares the constraint-based representation of all the preconditions of action A. Suppose, that we have two kinds of preconditions: positive $Pre^+(A)$ and negative $Pre^-(A)$, expressing what *must* hold, and what *must not* hold in a given world state. For every literal $p \in Pre^+(A)$ we generate a constraint "$\leftarrow$ *not holds*$(p, 0)$." and for every $p \in Pre^-(A)$ we generate "$\leftarrow$ *holds*$(p, 0)$.". These constraints will prevent the creation of children nodes by applying inexecutable actions.

S: Single graph node, containing one **world state** (represented by a set of literals).
BK: A logic program representing the **background knowledge** about our domain.
INST: The set of all possible **action instances** (where an *action instance* is any action with its parameters substituted by *constants*).

```
def expandNode(S):
        children = $\emptyset$
        for $\forall$ action instances $A \in$ INST:

                # get the constraint representation
                # of all the preconditions of A
                PC = getPreConstraints(A)

                # encode BK to a meta-program
                # form required for our reasoning
                BK = getMetaBK(BK)

                # compute new world states
                # using ASP solver
                NewS = applyEffects(S,A,PC,BK)

                # add these new states to a set of results
                children = children $\cup$ $NewS$

        return children
```

Fig. 1. Function **expandNode(S)** generates the children of the node S within a search graph.

getMetaBK(BK) returns the background knowledge encoded into a time-aware meta-program, that will be compatible with the representations used in the other functions. It simply replaces every rule "$h \leftarrow b_1, \ldots, b_k$, *not* b_{k+1}, $\ldots$ *not* b_n." by two new rules:

$$\begin{aligned} holds(h,0) \leftarrow\ & holds(b_1,0),\ldots,holds(b_k,0),\\ & not\ holds(b_{k+1},0),\ldots\ not\ holds(b_n,0).\\ holds(h,1) \leftarrow\ & holds(b_1,1),\ldots,holds(b_k,1),\\ & not\ holds(b_{k+1},1),\ldots\ not\ holds(b_n,1). \end{aligned}$$

Note 1. By now we should be able to see, that we will be reasoning in context of two consecutive time steps denoted by constants 0 and 1.

applyEffects(S,A,PC,BK) is the main reasoning function. It returns a new world state resulting from applying a *single action instance* A to a state S (see figure 2). To do so, it constructs a small meta-program *LP* containing appropriately encoded *background knowledge*, *world state* S, *preconditions* and *effects* of A, and the rules of *inertia*. Then it uses an external ASP solver to compute the answer sets, and translates them into resulting world states.

Notice, that we compute resulting world states very similarly to current ASP planners. Important difference is, that we always use input programs representing only *one action instance* and *two* subsequent *time steps*.

S:	Single graph node, containing one **world state** (represented by a set of literals).
A:	**Action instance** we want to apply to S.
PC:	**Preconditions** of A in a form of pre-prepared constraints.
BK:	The **background knowledge** represented in a meta-program form.

```
def applyEffects(S,A,PC,BK):
    NewS = $\emptyset$
    LP = BK + PC

    # add rules representing inertia
    LP += "$holds(F,1) \leftarrow holds(F,0),\ not\ holds(-F,1).$"
    LP += "$holds(-F,1) \leftarrow holds(-F,0),\ not\ holds(F,1).$"

    # add facts representing state S
    for $\forall$ literals $p \in$ S:
        LP += "$holds(p,0).$"

    # add the representation of the effects of A
    for $\forall$ effects $E \in eff(A)$ with conditions $\{c_1,\ldots,c_n\}$:
        LP += "$holds(E,1) \leftarrow holds(c_1,0),\ldots,holds(c_n,0).$"

    # generate the answer sets of LP: if there are
    # any, A was applicable in S
    # we just need to decode them and return new states
    for AS in getAnswerSets(LP):
        NewS = NewS $\cup$ decode(AS)

    return NewS
```

Fig. 2. Function **applyEffects(S,A,PC,BK)** uses an ASP solver to compute the world state resulting from application of action instance A to a state S.

The **getAnswerSets(LP)** function simply uses an external ASP solver (DLV, SMODELS, etc.) and returns the collection of answer sets of a logic program LP.

We can conclude the explanation of pseudo-code with the function called **decode(AS)**. It simply takes the set of literals of a type "$holds(F, T)$" and returns all those F, where $T = 1$. In other words, it decodes the answer set of a meta-program to its original form, while keeping only the information about what holds after the execution of action.

4.1 Advantages and Disadvantages

Let us now take a look at main advantages and disadvantages that result from using the *decomposition trick*.

- We can use **heuristics** while searching for the plans. In addition to heuristic methods employed by ASP solvers (speeding up the answer set generation), we can use some of the heuristics specifically designed for planning[3].
- **Parallelism** can be used during the planning. Decomposing the problem allows us to solve its individual parts in parallel. Specifically, we have two levels of parallelism:
  (a) At higher level, we can call the **expandNode** function for several nodes at once.
  (b) Al lower level, every time we expand the node, we can try applying all the action instances to it at once (**applyEffects** function).

- Planning with **concurrent actions** is not possible. Typical graph search algorithms are not capable of producing the plans with concurrent actions. Alternative search methods would have to be used for that.
- We need to have an actual **implementation** of graph search algorithm in addition to ASP solver. Planning without the *decomposition trick* doesn't in fact require anything besides the ASP solver, even though ASP planning systems often come with additional pieces of software or frontends (for translation from planning language into a meta-program).

5 Complexity

In this section, we will try to compare the time complexity of typical ASP-based planning, and planning with *decomposition trick*.

Both methods will use one of the *answer set computing algorithms*, which in general (given a ground logic program) work in two steps: they *generate interpretations* of a program that are candidates for answer sets, and then they submit them to a *model checker* for *verification* [Faber-Leone-Pfeifer, 1999]. Despite

[3] See the chapter 11 of [Russel-Norvig, 2003] for examples of planning heuristics.

slight differences (see [Baral, 2003] for comprehensive comparison), all these algorithms solve the NP-complete problem with worst-case complexity *exponential in the number of atoms* of input program. Let us therefore denote the complexity of answer set computation for a program Π simply $O(a^{\mathcal{AC}})$, where $\mathcal{AC}$ ("atom count") is the number of atoms contained in Π.

5.1 Typical ASP Planners

Common ASP-based planning systems work in three steps:

1. they **encode** the planning problem from a given action language into a meta-program Π,
2. **compute** the **answer sets** of Π,
3. and **decode** resulting answer sets back to original language.

Since the *encoding* and *decoding* run in linear time, we will ignore them. We are only interested in the $\mathcal{AC}$ - *atom count* of the meta-program Π. Problem is, that the encodings used in different methods vary slightly, which prevents us from expressing the *atom count* precisely in general. Instead, we will focus on two kinds of atoms that all these methods have in common: $holds(F, T)$ and $occurs(A, T)$. Set of atoms of any other kind will be denoted by $\mathcal{ADD}$ ("additional atoms"). In general, atoms from $\mathcal{ADD}$ represent axioms of inertia and domain dynamics, and should not contribute to overall atom count significantly.

Let $\mathcal{F}$ be a set of all the **literals** of our planning domain, $\mathcal{T}$ a set of **time steps**, and $\mathcal{INST}$ a set of all the **action instances**. Worst-case time complexity of typical ASP-based solution of a planning problem is then:

$$O\left(a^{|\mathcal{T}||\mathcal{F}|+|\mathcal{T}||\mathcal{INST}|+\mathcal{ADD}}\right) = O\left(a^{|\mathcal{T}|(|\mathcal{F}|+|\mathcal{INST}|)+\mathcal{ADD}}\right)$$

where the $|\mathcal{T}||\mathcal{F}|$ expression on the left specifies the number of $holds(F, T)$ atoms where variables F and T are substituted by items from sets $\mathcal{F}$ and $\mathcal{T}$. Similarly, the $|\mathcal{T}||\mathcal{INST}|$ expression correspond to a number of $occurs(A, T)$ atoms with A and T are substituted by items from $\mathcal{INST}$ and $\mathcal{T}$.

5.2 Planning with decomposition trick

With *decomposition trick*, actual planning can be implemented as any graph-search algorithm, where the ASP solver is only used to expand the nodes. Even though we suggest using one of heuristic search algorithms (like A*), we leave the final choice for the reader/programmer. Either way, to compute the time complexity, we must take into account two of its components:

1. The **graph search algorithm** itself, that explores the state space by repeatedly expanding the nodes,

2. and the actual **expansion of the nodes**, by **computing answer sets** of small meta-programs representing state transitions.

For the sake of this comparison, let us simply denote the complexity of our selected search algorithm $O(S)$.

Now, if we take a look at three of our algorithms (specifically **getPreConstraints(A)**, **getMetaBK(BK)**, and **applyEffects(S,A,PC,BK)**), we can see, that we only use one kind of atoms in our meta-programs: $holds(F, T)$. Here we still substitute F by domain literals from $\mathcal{F}$, but for T we now only choose one of two possible constants: 0 or 1. Number of atoms that we use is then $O(2|\mathcal{F}|)$.

Note 2. Notice also, that we don't generate any choice rules in our meta-programs. We cannot guarantee, that in conjunction with BK there won't be any, but in typical case our meta-programs are choice-free.

Overall worst-case performance of our method, when using *breadth-first search* algorithm, is therefore the product of complexities of *answer set generation* and an actual *search*:

$$O\left(a^{2|\mathcal{F}|}\right) \cdot O(S)$$

We can easily see, that we managed to *lower the exponent* significantly, for the cost of *repeating* the computation of *answer sets* several times. More specifically, we have removed the number of time steps $|\mathcal{T}|$ from it, which suggests that using the *decomposition trick* will prove useful for computing increasingly *long plans*. Let us now take a look at the experiments.

6 Experiments and Performance

In order to demonstrate the properties of planning with *decomposition trick*, we have created an experimental planner called *GRASP* (Graph-search based ASP Planner)[4].

It implements the A^* search algorithm with simple heuristics favouring those nodes, that have the lowest number of unsatisfied goal conditions.

For the description of domain and planning problems we use the $\mathcal{K}$ language, introduced in [Eiter et al., 2000]. We will be comparing the computation times of *GRASP* to those of DLV^K planner, since they both use the same planning language. The input files for both planners were exactly the same and can be found at http://grasp-planner.net23.net/experiments/.

Experiments were conducted on a machine with Intel(R) Core(TM) i5-661 CPU running a 64bit Linux Operating System.

[4] GRASP planner can be found and downloaded at http://grasp-planner.net23.net/ for now.

6.1 Modified Yale Shooting

First experimental scenario is loosely inspired by a well-known *Yale Shooting* problem [Hanks-McDermott, 1987]. Domain was however modified and made more complex, so that we could easily create problems with increasingly long solutions (plans).

Our task in this scenario is to kill some of the turkeys $tur(1), tur(2), \ldots, tur(n)$, by executing the $shoot(G,T)$ action, using one of our two guns $gun(g_1), gun(g_2)$. We need to load our gun using the action $load(G)$ before every shot. Domain contains two fluent predicates $alive(T)$ and $loaded(G)$.

Plan Length	$GRASP$ **Time**	DLV^K **Time**
1	0.0413	0.0134
2	0.0543	0.0111
3	0.0889	0.0262
4	0.1127	0.0311
5	0.1761	0.0428
6	0.1932	0.0602
7	0.3125	0.0979
8	0.3397	0.1349
9	0.5091	1.7684
10	0.5275	4.1648
11	0.7591	98.3018
12	0.8039	238.8315
13	1.1211	6543.4261
14	1.1673	-
15	1.5717	-
16	1.6029	-
17	2.1154	-
18	2.1554	-
19	2.8135	-
20	2.8520	-
21	3.6411	-
22	3.7024	-
23	4.6628	-
24	4.6906	-
25	5.8278	-
26	5.8930	-
27	7.1975	-
28	7.2949	-
29	8.8861	-
30	8.8824	-
31	10.6377	-
32	10.7968	-
33	12.8011	-
34	13.3014	-
35	15.1884	-
36	15.2869	-
37	17.9263	-
38	18.0586	-
39	20.9016	-
40	21.0281	-

Fig. 3. Computation times of increasingly complex instances of Modified Yale Shooting, using $GRASP$ and DLV^K planners. Times are in seconds.

In figure 4, we can see the exponential explosion in computation time using the DLV^K planner, which makes it unusable for plans longer than 10. On the other hand, *GRASP* planner using the *decomposition trick* seems to be capable of producing long plans in reasonable time.

6.2 Blocks World

Another scenario used in our experiments is probably the most common "toy domain" for reasoning about actions - Blocks World [Nilsson, 1980,Slaney-Thiebaux, 2001]. It consists of a set of blocks $block(1), block(2), \ldots, block(n)$ on the table. These blocks can be moved from position P_1 to P_2 using the action $move(B, P_1, P_2)$. Single fluent type $on(B, P)$ determines whether the block B is on the table or on one of the other blocks. The goal is stacking the blocks into a specific configuration (e.g. $on(1, table), on(2, 1), on(3, table)$).

In this case, we have deliberately hand-crafted the problem instances in such way, that *GRASP* heuristics were ineffective[5]. This resulted in worse performance at lower plan lengths, but still it was superior to DLV^K for longer plans.

Plan Length	*GRASP* **Time**	DLV^K **Time**
9	2.1954	0.4873
10	2.7959	0.5953
11	7.0475	0.8015
12	6.3209	1.2791
13	82.2836	4.6389
14	129.9655	39069.9096

Fig. 4. Computation times of increasingly complex problematic instances of Blocks World, using *GRASP* and DLV^K planners. Times are in seconds.

7 Conclusion

We have pointed out the possibility for improvement in ASP based planning by dividing the usual meta-program representation of a domain into several smaller programs and using graph search algorithms for actual planning task. This *decomposition trick* improves the performance of long plan computation, while preserving and utilising the expressive capabilities of ASP.

The prototype planner *GRASP* was introduced and used for the performance comparison. Planner itself however is not optimally coded and should primarily serve as a demonstration of this technique. We encourage the authors and programmers of mainstream planning systems to use this *decomposition trick* in their future implementations.

[5] Blocks World is in general not an ideal domain for heuristics based purely on the number of satisfied goals, due to the fact, that satisfying one goal may easily prevent us from satisfying others. We have created our problem instances with this in mind.

References

[Baral, 2003] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, Cambridge, U.K. 2003.

[Hart-Nilsson-Raphael, 1968] P. E. Hart, N. J. Nilsson, B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics SSC4 4(2): 100–107. 1968.

[Faber-Leone-Pfeifer, 2001] W. Faber, N. Leone, G. Pfeifer. *Experimenting with heuristics for answer set programming.* In Proceedings of IJCAI'01: 635–640. 2001.

[Faber-Leone-Pfeifer, 1999] W. Faber, N. Leone, G. Pfeifer. *Pushing Goal Derivation in DLP Computations.* In Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). 1999.

[Niemelä, 1998] I. Niemelä. *Logic programs with stable model semantics as a constraint programming paradigm.* In Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning: 72–79. 1998.

[Russel-Norvig, 2003] S Russell, P Norvig. *Artificial Intelligence: A Modern Approach, 2nd edition.* Prentice Hall. 2003.

[Hanks-McDermott, 1987] S. Hanks, D. McDermott. *Nonmonotonic logic and temporal projection.* Artificial Intelligence, 33(3):379-412. 1987.

[Lifschitz, 2002] V. Lifschitz. *Answer Set Programming and Plan Generation.* Artificial Intelligence, vol. 138. 2002.

[Eiter et al., 2000] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. *Planning Under Incomplete Knowledge.* Lecture Notes in Computer Science, Volume 1861: 807-821. 2000.

[Eiter et al., 2002] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. *The DLV^K Planning System: Progress Report.* Lecture Notes in Computer Science, vol. 2424/2002: 541-544. 2002.

[Simons, 2002] P. Simons, I. Niemela, and T. Soininen. *Extending and Implementing the Stable Model Semantics.* Artificial Intelligence, 138(1-2):181–234. 2002.

[Slaney-Thiebaux, 2001] J. Slaney, S. Thiebaux. *Blocks World revisited.* Artificial Intelligence 125: 119-153. 2001.

[Nilsson, 1980] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto. 1980.

[Subrahmanian-Zaniolo, 1995] V. S. Subrahmanian, C. Zaniolo. *Relating Stable Models and AI Planning Domain.* In proceedings of ICLP-95. 1995.

[Gebser et al., 2011] M. Gebser, R. Kaminiski, M. Knecht, T. Schaub. *plasp: A Prototype for PDDL-Based Planning in ASP.* In Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11), LNAI 6645: 358-363, Springer Verlag. 2011.

[Gebser-Grote-Schaub, 2010] M. Gebser, T. Grote, T. Schaub. *Coala: A Compiler from Action Languages to ASP.* Lecture Notes in Computer Science, 2010, Volume 6341/2010: 360-364. 2010

[Leone et al., 2006] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S Perri, F. Scarcello. *The DLV system for knowledge representation and reasoning.* ACM Transactions on Computational Logic (TOCL), Volume 7, Issue 3. 2006.

[Syrjänen, 2001] T. Syrjänen. *The Smodels System.* Logic Programming and Nonmonotonic Reasoning. 2001.

[Gebser et al., 2008] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, S. Thiele. *Using gringo, clingo and iclingo.* 2008.