Improving Answer Set Based Planning by Bidirectional Search

Peter Baumgartner
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken
Germany
Email: *baumgart@mpi-sb.mpg.de*

Anupam Mediratta *
IBM India Research Lab
Block 1, Indian Institute of Technology
Hauz khas, Delhi 110016
India
Email: *anupamme@in.ibm.com*

Abstract

Solving AI planning problems by transformation into (normal) logic programs and computing answer sets (stable models) has gained considerable interest over the last years. We investigate in this context a classical AI search technique, bidirectional search, where search is performed both from the initial facts towards the goal and vice versa. Our contribution is to show how bidirectional search can be realized in the logic programming/answer set paradigm to planning. This seems not having been investigated so far. We report on practical experiments on planning problems from an AIPS competition and show how our approach helps speeding up the planning process. We perceive our contribution mainly as a *technique* that is compatible with and complementary to existing extensions and improvements, rather than as a concrete planning system.

Keywords:

Planning and Scheduling, Search Techniques, Theorem Proving

*This work was done when the author was a student of IIT Guwahati and visited University of Koblenz for his internship.

Improving Answer Set Based Planning by Bidirectional Search

Abstract

Solving AI planning problems by transformation into (normal) logic programs and computing answer sets (stable models) has gained considerable interest over the last years. We investigate in this context a classical AI search technique, bidirectional search, where search is performed both from the initial facts towards the goal and vice versa. Our contribution is to show how bidirectional search can be realized in the logic programming/answer set paradigm to planning. This seems not having been investigated so far. We report on practical experiments on planning problems from an AIPS competition and show how our approach helps speeding up the planning process. We perceive our contribution mainly as a *technique* that is compatible with and complementary to existing extensions and improvements, rather than as a concrete planning system.

Keywords:

Planning and Scheduling, Search Techniques, Theorem Proving

1 Introduction

Planning is among the most prominent search space exploration problems in AI. The search space of a problem instance is determined by an initial state, a (possibly partially specified) goal state, and operators that transform one state to another. Solving a planning problem means to determine a sequence of operator applications that take the initial state to one that contains the goal state. The practical side of solving planning problems is to cope with the (usually) huge search space. In order to make a planning system fast, techniques like decomposing a problem into independent subproblems and heuristics to guide the search are needed. The heuristics used in leading systems like FF [Hoffmann and Nebel, 2001] even narrow down the search to such a high degree that completeness may be lost[1], and also there is no guarantee that a *shortest* plan will be computed. Nethertheless, systems like FF show impressive overall performance.

Our approach is on a different line: it is complete and also finds a shortest plan. The price to be paid is that it cannot solve many difficult planning problems solved by the fastest systems. However, our goal in this paper is not to report on a competitive planner. Merely we intend to contribute new *techniques* for search space pruning, which may be used in addition to existing techniques. We investigate our technique within the paradigm of "planning as answer set programming". This paradigm has gained considerably renewed interest over the last years [Eiter *et al.*, 2004; 2003; Brogi *et al.*, 2003; Niemelä, 1999; Lifschitz, 2002; Dimopolous *et al.*, 1997]. A planning problem then is translated to a normal logic program, and the stable models of the logic program encode the solutions (the plans). It was not only demonstrated that this way efficient planning

[1] FF includes a complete backup strategy.

systems can be built. In addition, it is easy to incorporate knowledge about the domain under consideration or heuristical knowledge by just adding appropriate rules to the logic program. These approaches often gain their speed through parallelizing sequences of operator applications, which also leads to non-minimal plans in general. With our contribution we suggest new, *complementary* improvements to the "planning as logic programming". Although we study these in isolation, there is no reason not to expect they could be combined with the known techniques and heuristics.

1.1 Idea of Our Contribution

In the main part of this paper we will describe a novel translation scheme from STRIPS-like planning problems to logic programs. Before proceeding to that part, we will explain the main ideas on a rather abstract level; they are illustrated in Figure 1.

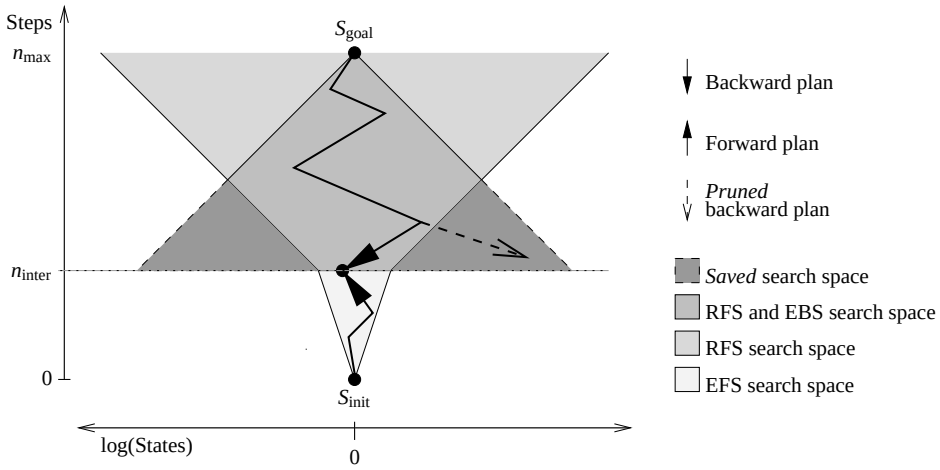


Figure 1: Dividing the search space.

The programs resulting from the translation encode bidirectional search space exploration: forward search (or "exact forward search", EFS, as called in Figure 1) from a given initial situation S_{init} towards a given goal situation S_{goal} is combined with backward search (or "exact backward search", EBS, in Figure 1), from S_{goal} towards S_{init}.[2] The motivation for bidirectional search can be explained as follows: assume as given a planning problem that can be solved with $n_{\max}$ operator applications to get from S_{init} to S_{goal}. Suppose further, for simplicity, that in every situation m different operator applications are possible. The search space thus consists of $m^{n_{\max}}$ situations when performing forward search alone (or backward search alone). Now let n_{inter} be some "intermediate timepoint" $0 \leq n_{\text{inter}} \leq n_{\max}$. It shall be used to separate forward from backward search in the following way: in a first phase, forward search is performed to compute and collect *all* situations reachable from S_{init} in n_{inter} steps. In the second phase then backward search is performed, which starts from S_{goal} and is limited to perform $n_{\max} - n_{\text{inter}}$ steps. The search stops as soon as one situation is derived that has been computed as a

[2] Instead of *a* goal situation, planning problems often are underspecified and specify a *set* of goal situations instead. While our approach works in this more general setting, we ignore it in this section because it is not essential to describe the main idea.

result of the forward search before. It is easy to see that the search space now consists of $m^{n_{\text{inter}}} + m^{n_{\text{max}} - n_{\text{inter}}}$ situations. This expression is minimal when $n_{\text{inter}} = \frac{n_{\text{max}}}{2}$ (to be rounded if n_{max} is an odd number). Taking $n_{\text{inter}} = \frac{n_{\text{max}}}{2}$ thus yields a search space size $2 \cdot m^{\frac{n_{\text{max}}}{2}}$, which, in general, is much smaller than $m^{n_{\text{max}}}$. This advantage is contrasted by the obvious drawback of possibly prohibitive memory requirements. However, as our experiments show, this is not always the case, and sometimes, depending from the domain, even values close to n_{max} can be used. In the past, there has been some work done in bidirectional planning. For instance, Prodigy [Veloso *et al.*, 1995] has been one of the successful bidirectional planner but had a disadvantage of incompleteness, which is not present in our planner.

Bidirectional search supports another improvement on top of it, which is also realized in our approach. It works by inserting another phase we call "relaxed forward search", RFS, between the EFS and EBS phases. It derives certain information from the situations derived by EFS so that EBS can take advantage of for search space pruning. The "relaxed" forward search phase computes an *approximation* from above of all possible situations reachable within $n_{\text{max}} - n_{\text{inter}}$ steps from the situations derived by EFS. As said, the purpose is to prune the search space of the subsequent EBS phase. Because, during backward search, any (partial) plan derived then can be ignored for further extension if it leads to a situation *outside* of this approximation. In order to take advantage of this idea, the cost of computing the approximation should be much lower than the cost of exact forward search from timepoint n_{inter} to n_{max}. Our approach follows the one realized in the FF system [Hoffmann and Nebel, 2001] by ignoring the delete list of the operators: on moving from one state to the next, fluents get added, but never deleted.[3] Computing such approximations can be done efficiently, in quadratic time.

To our knowledge, the idea of separating the search space and thereby taking advantage of an approximation as described above has not been considered before in planning. In particular we believe it is a novel contribution to the answer set approach to planning.

2 Translating Strips Planning Problems

We work in a basic Strips planning setting. In the following, we only recast usual definitions in a way suitable to describe our approach. We assume disjoint signatures of *fluent symbols*, *operator names*, *objects* and *variables*. Each fluent symbol is equipped with a fixed arity. A *fluent* is an expression of the form $f(s_1, \ldots, s_n)$, where f is an n-ary fluent symbol and each s_i ($i = 1, \ldots, n$) is either an object or a variable. When each s_i is an object, the fluent is called a *ground fluent*. Unless stated differently, in the sequel the letter f denotes a fluent symbol, N denotes an operator name, o denotes an object and x, y, z denote variables. Indices are used as needed.

A *planning problem* consists of the following information: (1) *initial situation:* a set of ground fluents S_{init} – (2) *goal situation:* a set of ground fluents S_{goal} – (3) *maximum time steps* required for achieving the goal situation, n_{max}: a natural number – (4) *intermediate time point* n_{inter}: a natural number – (5) A finite set of objects O containing at least the objects mentioned in the operators below – (6) A finite set of *operators*, each consisting of the following:

- An operator name N and a list of variables $x_1, \ldots, x_n$, for some $n \geq 0$ (standing for the objects the operator

[3] Notice that "situations" may come up that are not derivable otherwise. In the blocksworld domain, for instance, several blocks may then be stapled on top of a *single* block. The only requirement to preserve completeness is that every derivable situation is contained in some such "situation".

works on). We assume this list of variables comprises at least the variables mentioned in the following:

- Preconditions: a set of fluents *Pre* (the fluents that have to be true in order to apply the operator).
- Add list: a set of fluents *Add* (the fluents which become true on the application of the operator).
- Delete list: a set of fluents *Del* (the fluents which become false on the application of the operator).

Each operator Op thus is a tuple of the form $\langle N(x_1,\ldots,x_n), Pre, Add, Del\rangle$.

We state only informally that solving a planning problems means to determine a sequence of operator instances that take the initial situation to any situation satisfying what is specified by the goal situation.

Example 2.1 (Blocksworld Problem) Let $S_{\text{init}} = \{\text{On}(\text{b},\text{table}), \text{On}(\text{a},\text{b}), \text{Clear}(\text{a})\}$, $S_{\text{goal}} = \{\text{On}(\text{b},\text{a})\}$, $n_{\max} = 2$, $n_{\text{inter}} = 1$, $O = \{\text{a},\text{b},\text{table}\}$ and the single operator $\text{Move}(x,y,z)$ be as follows[4] ("move block x from y on top of z"): Pre: $\text{On}(x,y), \text{Clear}(x), \text{Clear}(z)$ Add: $\text{On}(x,z), \text{Clear}(y)$ Del: $\text{Clear}(z), \text{On}(x,y)$ □

As stated above, a planning problem contains the two parameters $n_{\max}$ and n_{inter}. The parameter $n_{\max}$ limits the search for plans consisting of at most $n_{\max}$ steps, and the parameter n_{inter} acts as timepoint that separates EFS from RFS and EBS. Notice that both n_{inter} and $n_{\max}$ are parameters to the translation scheme below. A complete plan search procedure would iterate $n_{\max}$ over the natural numbers in a, say, incremental way, chose some value for n_{inter} on each iteration, apply the transformation and execute the result until success. Such a scheme is easy to realize and is not described in the sequel. However, not so trivial is the choice of n_{inter} for a given value of $n_{\max}$: while $n_{\max}$ may be chosen arbitrarily in principle, a "good" choice may heavily impact performance (see Section 3 below).

Next, we are going to describe the announced translation of planning problems. They translate into a single logic program, which, when executed in a bottom-up way, realizes the EFS, RFS and EBS phases in this order. Of course, certain information has to be passed from EFS to RFS, and from RFS to EBS to realize the idea explained in Section 1.1. To this end, additional rules come into play.

2.1 Exact Forward Search

Initialization. For each fluent $f(o_1,\ldots,o_n) \in S_{\text{init}}$ include a fact $f\_0(o_1,\ldots,o_n,\text{init})$. They express that the initial situation, written as a collection of facts, holds at timepoint 0. Notice the additional last argument, which, in general, encodes a partial plan up to the timepoint mentioned in the index to the fluent. In the example we thus get the facts $\text{On\_0}(\text{b},\text{table},\text{init})$, $\text{On\_0}(\text{a},\text{b},\text{init})$ and $\text{Clear\_0}(\text{a},\text{init})$.

Operator application. An operator can be applied if all the fluents in its preconditions are satisfied and a specific additional condition is satisfied (to be explained below). More precisely, consider an operator $\langle N,(x_1,\ldots,x_n), Pre, Add, Del\rangle$, where *Pre* consists of m fluents written as $pre^j(s^j_1,\ldots,s^j_{k_j})$, for $j = 1,\ldots,m$ and some $k_j \geq 0$. Each operator of a given planning problem is transformed to the following set of rules (left

[4] Actually, this formalization is not quite correct, as it does not reflect that Clear(table) should hold in every situation. In our experiments we therefore use a *slightly* different, correct formulation. For space reasons we neglect this and some other details that are irrelevant for our results.

side), for every $t = 0, \ldots, n_{\text{inter}} - 1$, where $t' = t + 1$. The right side shows the transformation for the example (the time parameters t and t' are left uninstantiated; arguments to predicates with capital letters are variables).

$$
\begin{array}{l}
N\_t(x_1, \ldots, x_n, \text{Plan}) \leftarrow \\
\quad pre^1\_t(s^1_1, \ldots, s^1_{k_1}, \text{Plan}), \ldots, \\
\quad pre^m\_t(s^m_1, \ldots, s^m_{k_m}, \text{Plan}).
\end{array}
\qquad\qquad
\begin{array}{l}
\text{Move\_}t(\text{X}, \text{Y}, \text{Z}, \text{Plan}) \leftarrow \\
\quad \text{On\_}t(\text{X}, \text{Y}, \text{Plan}), \\
\quad \text{Clear\_}t(\text{X}, \text{Plan}),\ \text{Clear\_}t(\text{Z}, \text{Plan}).
\end{array}
$$

Successor State Axioms. Every member of the add list of an operator becomes true when the operator is applied. Let $\langle N, (x_1, \ldots, x_n), Pre, Add, Del \rangle$ be an operator as above, and assume *Add* consist of m fluents written as $add^j(s^j_1, \ldots, s^j_{k_j})$, for $j = 1, \ldots, m$ and some $k_j \geq 0$. Then, *Add* is transformed into the following set of rules (left side), for every $t = 0, \ldots, n_{\text{inter}} - 1$ and every $j = 1, \ldots, m$, where $t' = t + 1$. The right side shows the transformation for the On fluent in the add list of the Move operator (the transformation of the Clear fluent is omitted). As above, this transformation is to be done for every operator in the given planning problem.

$$
\begin{array}{l}
add^j\_t'(s^j_1, \ldots, s^j_{k_j}, N(x_1, \ldots, x_n, \text{Plan})) \leftarrow \\
\quad N\_t(x_1, \ldots, x_n, \text{Plan}).
\end{array}
\qquad\qquad
\begin{array}{l}
\text{On\_}t'(\text{X}, \text{Z}, \text{Move}(\text{X}, \text{Y}, \text{Z}, \text{Plan})) \leftarrow \\
\quad \text{Move\_}t(\text{X}, \text{Y}, \text{Z}, \text{Plan}).
\end{array}
$$

Frame Axioms. The problem of formalizing frame axioms has a long tradition in the planning literature. In the context of model-based planning, different solutions have been suggested. The main issue is at every time point to retain the truth value of those fluents that are not present in neither the add list nor delete list of the operator applied at this time point. If a fluent is in the add list of the operator[5] then it must become true after the operator is applied. Because a state is represented as the set of fluents being "true" in it (and those being "false" are absent), a fluent in the operator's add list is just inserted into the representation of the successor state, as prescribed by the successor state axioms above. But if a fluent is not there, rules for moving fluents from a state to a successor state are needed. The rules are determined according to two distinguished cases: for every pair of a fluent and an operator, the fluent lies in the delete list of this fluent – or it does not.

To capture the first case suppose $f(x'_1, \ldots, x'_k)$ is in the delete list of an operator named N, for some fluent symbol f and variables $x'_1, \ldots, x'_k$. Then, the following rules are emitted (left side), for every $t = 0, \ldots, n_{\text{inter}} - 1$, where $t' = t + 1$ and $y_1, \ldots, y_k$ are fresh variables. As above, the right side shows an example.

$$
\begin{array}{l}
f\_t'(y_1, \ldots, y_k, N(x_1, \ldots, x_n, \text{Plan})) \leftarrow \\
\quad N\_t(x_1, \ldots, x_n, \text{Plan}), \\
\quad f\_t(y_1, \ldots, y_k, \text{Plan}), \\
\quad \text{not equal}([x'_1, \ldots, x'_k], [y_1, \ldots, y_k]).
\end{array}
\qquad\qquad
\begin{array}{l}
\text{On\_}t'(\text{X1}, \text{Y1}, \text{Move}(\text{X}, \text{Y}, \text{Z}, \text{Plan}) \leftarrow \\
\quad \text{Move\_}t(\text{X}, \text{Y}, \text{Z}, \text{Plan}), \\
\quad \text{On\_}t(\text{X1}, \text{Y1}, \text{Plan}), \\
\quad \text{not equal}([\text{X1}, \text{Y1}], [\text{X}, \text{Y}]).
\end{array}
$$

Recall the requirement that the list of variables $x_1, \ldots, x_n$ in an operator description must include all variables mentioned, in particular, in the fluents in its delete lists. Therefore, each variable $x'_1, \ldots, x'_k$ is equal to one of $x_1, \ldots, x_n$. The informal meaning of the rule now can be explained as follows: suppose that the rule body is satisfied. The variables $x_1, \ldots, x_n$ are instantiated with concrete objects then, and the corresponding instance of the subgoal $N\_t(x_1, \ldots, x_n, \text{Plan})$ means that the operator N is to be applied to those objects. Because $y_1, \ldots, y_k$ are fresh variables, the subgoal $f\_t(y_1, \ldots, y_k, \text{Plan})$ applies to every fluent of this form holding at the current

[5] To be correct, it is an operator *instance* but not an operator (definition). In the sequel, we will often simply speak of an "operator" when an operator instance is meant; this should not give rise to confusion.

state. Now, if an instance of $f\_t(y_1,\ldots,y_k,\text{Plan})$ is such that the equality as expressed in the last subgoal does not hold, i.e. the subgoal itself is satisfied, then this instance is not in the instantiated delete list of the operator[6]. Therefore, it must be retained in the successor state, as expressed by the rule head. Conversely, if the equality mentioned does hold, then the fluent instance is in the instantiated delete list, and therefore the fluent instance is not retained in the successor state.

To capture the second case, suppose a k-ary fluent symbol f such that the delete list of an operator named N does not contain a fluent with the symbol f. That the truth value of such fluents is preserved under application of N is expressed by the following rule, where $t' = t+1$ and $y_1,\ldots,y_k$ are fresh variables.

$$f\_t'(y_1,\ldots,y_k,N(x_1,\ldots,x_n,\text{Plan})) \leftarrow N\_t(x_1,\ldots,x_n,\text{Plan}),\ f\_t(y_1,\ldots,y_k,\text{Plan}).$$

Notice this does transformation does not apply in the example problem because the delete list of the operator Move mentions all fluents, On and Clear. These transformations are to be applied for any pair of operator and fluent symbol in the given planning problem. We would like to emphasize that thanks to having default negation at disposal in our language, the representation of the frame axioms is very compact. Notice that *all* states reachable in $n_{\max}$ steps, together with all plans leading to them are computed.[7] It is therefore natural to expect memory problems as $n_{\max}$ grows. Recall, however, that forward planning is not employed to search a plan with $n_{\max}$ steps, but only to compute plans with n_{inter} steps, and n_{inter} may be chosen much smaller.

2.2 Rough Forward Search

As explained in Section 1.1, the RFS phase is a kind of forward planning, however where the delete list of the operators are ignored. EFS starts from the timepoint n_{inter}. Technically, this translates into rules of the following form (for all n-ary fluent symbols f): $\text{Rough\_}f\_n_{\text{inter}}(x_1,\ldots,x_n) \leftarrow f\_n_{\text{inter}}(x_1,\ldots,x_n)$. In the example, we thus get for instance (if $n_{\text{inter}} = 1$) the rule $\text{Rough\_On\_1}(\text{X},\text{Y}) \leftarrow \text{On\_1}(\text{X},\text{Y})$. (This example indicates another notational convention, namely, that the "fluents" computed by RFS are prefixed by "Rough_".) In comparison to the EFS transformation in Section 2.1, the frame axioms are rather trivial now. They now just express that all fluents are preserved. As a further simplification, plans need not be remembered, because only the *situation* is needed for the purpose of pruning, not the plan leading to it. Because the transformation for RFS is similar as for EFS, we do not include it here for space reasons.

2.3 Exact Backward Search

Initialization. For each fluent $f(o_1,\ldots,o_n) \in S_{\text{goal}}$ include a fact $\text{Goal\_}f\_n_{\max}(o_1,\ldots,o_n)$. They just express the goal specification to be reached at timepoint $n_{\max}$. In the example we thus get the fact $\text{Goal\_On\_2}(\text{b},\text{a})$. Notice that this time no additional argument position holding a plan is added. It is not needed, because plans are encoded directly as models (different plans are encoded as different models). As a general notational convention, the predicate symbols in the EBS phase are prefixed with "Goal_".

[6] The equality predicate is specified as $\text{equal}(x,x) \leftarrow$ and thus means syntactic equality. The translation scheme is such that no floundering problem can occur under a bottom-up evaluation scheme.

[7] By using a built-in comparable to Prolog's findall, which is available in the system we use , it is possible to store only *one* plan for each situation derivable.

Operator application. In a backward search strategy, an operator can be applied (backwards) when any one of the fluents in its add list is true in the current situation (because, conceptually, applying it forwards from the previous timepoint achieves that fluent, which might be necessary to find the plan), and none of the fluents in its delete list hold in the current situation (because, conceptually, having applying it forwards from the previous timepoint cannot achieve the current situation then). More precisely, consider an operator $\langle N, (x_1, \ldots, x_n), Pre, Add, Del \rangle$, where *Pre* consists of p fluents written as $pre^q(v_1^q, \ldots, v_{r_q}^q)$, for $q = 1, \ldots, p$ and some $r_q \geq 0$, and *Add* consists of m fluents written as $add^j(s_1^j, \ldots, s_{k_j}^j)$, for $j = 1, \ldots, m$ and some $k_j \geq 0$, and *Del* consists of l fluents written as $del^i(u_1^i, \ldots, u_{o_i}^i)$, for $i = 1, \ldots, l$ and some $o_i \geq 0$. Each operator of a given planning problem is transformed to the following set of rules (left side), for every $j = 1, \ldots, m$ and every $t = n_{\text{inter}}, \ldots, n_{\max} - 1$, where $t' = t + 1$. The right side shows one transformation result for the example (the time parameters t and t' are left uninstantiated).

$\text{Goal\_}N\text{\_}t(x_1, \ldots, x_n) \vee \text{No\_Goal\_}N\text{\_}t(x_1, \ldots, x_n) \leftarrow$
$\quad add^j\text{\_}t'(s_1^1, \ldots, s_{k_j}^j),$
$\quad \text{not } del^1\text{\_}t'(u_1^1, \ldots, u_{o_1}^1), \ldots,$
$\quad \text{not } del^l\text{\_}t'(u_1^l, \ldots, u_{o_l}^l),$
$\quad \text{Rough\_}pre^1\text{\_}t(v_1^1, \ldots, v_{r_1}^1), \ldots,$
$\quad \text{Rough\_}pre^p\text{\_}t(v_1^p, \ldots, v_{r_p}^p).$

$\text{Goal\_Move\_}t(\text{X}, \text{Y}, \text{Z}) \vee \text{No\_Goal\_Move\_}t(\text{X}, \text{Y}, \text{Z}) \leftarrow$
$\quad \text{On\_}t'(\text{X}, \text{Z}),$
$\quad \text{not Clear\_}t'(\text{Z}),$
$\quad \text{not On\_}t'(\text{X}, \text{Y}),$
$\quad \text{Rough\_On\_}t(\text{X}, \text{Y}),$
$\quad \text{Rough\_Clear\_}t(\text{X}),$
$\quad \text{Rough\_Clear\_}t(\text{Z}).$

Notice the disjunction in the head of the rule. It specifies whether the operator named N shall be applied – or not (the "No_" case). Additional rules are emitted, saying that at each timepoint (except n_{inter}) *exactly* one operator instance is applied (again, omitted for space reasons). The use of disjunction is not essential and the nondeterminism it stands for can easily be expressed in normal logic programs as well (as mainly used in answer set programming). We have chosen this formulation, because our interpreter, the KRHyper system [Wernhard, 2003] supports it natively.[8] KRHyper's models *for the rules generated* by the translation coincides with the possible model semantics, which in turn would coincide with the stable model semantics when having used normal logic programs as the target language. Thus, we do not leave "essentially" the standard paradigm of using normal logic programs for planning purposes.

Notice also the subgoals in the rules prefixed with "Rough_". They realize the search space pruning as explained in Section 1.1: in order for the operator to be applicable, the new situation it derives, as determined by its precondition list (cf. "Successor state axioms" just below) must be approximated from above in the RFS phase.

Successor state axioms. Whenever the left disjunction $\text{Goal\_}N\text{\_}t(x_1, \ldots, x_n)$ of a disjunctive rule is chosen, this means we operator is to be applied (backwards), and hence the preconditions of this operator must be added as new goals to the current situation. This translates into rules of the form $\text{Goal\_}pre\text{\_}t(s_1, \ldots, s_k) \leftarrow \text{Goal\_}N\text{\_}t(x_1, \ldots, x_n)$., for each fluent $pre\text{\_}t(s_1, \ldots, s_k)$ in the precondition list, for every operator.

Further rules. The transformations given above form the core of our approach. Beyond, additional rules are emitted: notably, rules to take care of the frame axioms (somewhat dual to the rules for the frame axioms as for EFS above), rules to determine if EBS generates a plan leading to some situation as derived by EFS, and some rules for bookkeeping purposes. We omit the corresponding transformation for space reasons.

[8] Additionally, KRHyper requires *stratified* (disjunctive) programs, a property which is satisfied by our transformation. The main point to achieve this is to include the timepoints within the names of the predicates.

Problem	Time	n_{inter}	n_{max}
4-2	0.079	1	3
4-2	0.087	2	3
4-2	0.089	2	3
5-2	0.401	0	8
5-2	0.399	1	8
5-2	0.393	2	8
5-2	0.378	3	8
5-2	0.382	4	8
5-2	0.374	5	8
5-2	0.446	6	8
5-2	0.514	7	8
5-2	0.569	8	8
6-2	0.184	0	10
6-2	0.163	1	10
6-2	0.141	2	10
6-2	0.61	3	10
6-2	0.71	4	10
6-2	0.69	5	10
(continued)			
6-2	0.841	6	10
6-2	1.364	7	10
6-2	2.669	8	10
6-2	3.282	9	10
6-2	4.2	10	10
7-2	52.264	0	10
7-2	18.048	1	10
7-2	6.665	2	10
7-2	1.954	3	10
7-2	1.223	4	10
7-2	1.276	5	10
7-2	2.889	6	10
8-2	1.248	1	8
8-2	1.004	2	8
8-2	1.045	3	8
8-2	1.014	4	8
9-2	20.97	6	13

(a) Blocksworld domain.

Problem	Time	n_{inter}	n_{max}
1-1	0.089	0	3
1-1	0.095	1	3
1-1	0.093	2	3
1-1	0.091	3	3
2-4	0.352	0	7
2-4	0.552	1	7
2-4	0.266	2	7
2-4	0.229	3	7
2-4	0.217	4	7
2-4	0.221	5	7
2-4	0.207	6	7
2-4	0.207	7	7
3-4	18.34	3	10
3-4	1.975	4	10
3-4	0.561	5	10
3-4	0.438	6	10
3-4	0.424	7	10
(continued)			
3-4	0.397	8	10
3-4	0.405	9	10
3-4	0.401	10	10
4-3	73.969	8	15
4-3	5.702	9	15
4-3	1.735	10	15
4-3	1.626	11	15
4-3	1.548	12	15
4-3	1.579	13	15
4-3	1.581	14	15
4-3	1.568	15	15
5-2	7.714	11	15
5-2	5.072	12	15
5-2	4.929	13	15
5-2	4.804	14	15
5-2	4.919	15	15

(b) Elevator domain.

Table 1: Results for our planner.

3 Experimental Results and Analysis

In order to assess the practical relevance of our approach, we have implemented (in Prolog) the transformation scheme of Section 2. We have tried our system on various problems from different domains picked up from the competition at AIPS'00.[9] As the actual system to run the resulting logic programs we have chosen the KRHyper deduction system [Wernhard, 2003], which can be used as a reasonably fast interpreter for normal logic programs. We report here on the results for the blocksworld and the elevator domains of the AIPS'00 competition, as here our systems performs best. We have compared the results of our planner with some of the benchmark planners which participated in AIPS'00.

Blocksworld Domain. The results for the blocksworld domain are displayed in Table 1(a). Here, "Problem $N-M$" means the M-th variant of a problem set with N blocks (taken from AIPS'00). Because the length of a minimal plan is not known in advance, we employed an incremental scheme of planning problems with $n_{\text{max}} = 1, 2, 3, \ldots$, until a plan is found. The "Time" entries in Table 1 denotes the *accumulated* times over these runs. We experimented with different settings for n_{inter}, dependent from the (current) value of n_{max}. Results were generally best when setting n_{inter} to half of n_{max}, sometimes +1 or -1. This is also shown in Table 1(a).[10] For the problem $9-2$ our planner could only find a solution within reasonable time when setting $n_{\text{inter}} = 6$, as shown. From the results we conclude that for this domain our bidirectional search technique yields much better results than pure forward or pure backward search alone.

[9] See `http://www.cs.toronto.edu/aips2000/`.

[10] Due to the lack of space we do not display the value of n_{inter} for each n_{max}.

Problem	Our Planner	Yochan	Hoffmann	Propplan	Stoerr
4-2	0.11	0.081	0.01	0.3	0.16
5-2	0.522	0.189	0.01	1.6	0.63
6-2	0.851	0.789	0.03	8.5	3.00
7-2	1.706	0.136	0.05	40.7	18.66
8-2	1.401	0.087	0.02	178	-

(a) Blocksworld domain.

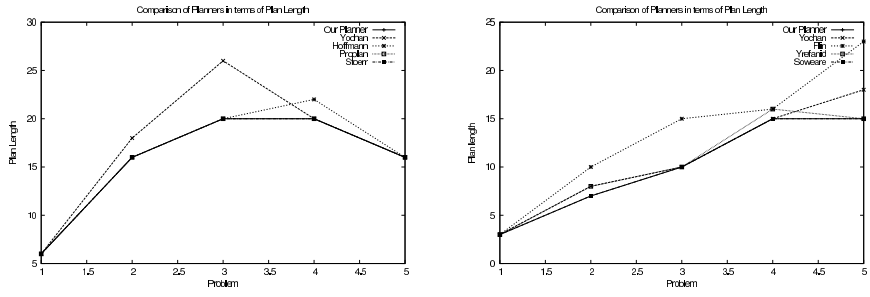
Problem	Our Planner	Yochan	Flin	Yrefanid	Soweare
1-1	0.124	0.038	0.02	0.40	0.00
2-4	0.289	0.047	0.04	0.40	0.01
3-4	0.559	0.015	0.06	0.40	0.13
4-3	2.16	0.097	0.12	0.43	1.68
5-2	6.702	0.126	0.15	0.44	1.27

(b) Elevator domain.

Table 2: Comparison of our planner with others with respect to time to generate the plan.

These results were compared with different planners participating in the AIPS'00 competition. Since the architecture on which our system was run is different to the one used in AIPS'00 we had to normalize the results. By running, as a sample, Hoffman's planner (FF) on our hardware we determined a machine-factor of 1.4, which we used to normalize the results reported at AIPS'00 (Table 2 takes care of this).

For comparing, we have used mainly two principles: the time taken to find a plan, whether the plan length is optimal (i.e. shortest) or not. Table 2(a) and Figure 2(a) show the comparison of our planner with some others (names as mentioned in the AIPS'00 competition). Of the other planners, Yochan and Hoffman were faster, but their plan length is not optimal whereas our planner always produces shortest length plans. This can be seen in Figure 2, which compares the plan length of different planners. Here, since our planner, Propplan and Stoerr compute optimal length plan, their graphs overlap. So, only three lines are visible instead of the five expected ones. In this figure, 1,2,...5 on the X axis correspond to the five problems, for which results are compared in Table 2. Propplan and Stoerr were taken into account because both of them are logic based and hence somewhat similar to our methodology. From Table 2(a) one can see that our planner is significantly faster.



(a) Plan length (Blocks World). (b) Plan length (Elevator Domain)

Figure 2: Comparison of our planner with some others.

Elevator Domain. The elevator domain is about planning servicing passengers for transport from their

origin to their destination floors. We used the operator definitions exactly as in the AIPS'00 competition. The results are displayed in Table 1(b) (for an explanation of the table entries see above the blocksworld domain). It can be observed that for this domain an almost pure forward search strategy is the best. More precisely, best results were obtained when n_{inter} is between $0.7*n_{\text{max}}$ and n_{max} (depending on the problem). So, an almost forward search strategy works relatively best in this domain.

We evaluated our planner and compared it with others in the elevator domain, using the same criteria as for the blocksworld domain. Table 2(b) and Figure 2(b) show the comparison of our planner with some others. Relativization of results was done in this domain also by the same factor as in Blocks World. Among the other planners, Flin and Yochan were faster but their plan length was not optimal. This was not the case with our planner, as can be seen in Figure 2(b), which compares the plan length found by different planners. Soweare is better than our system in time and computes optimal length plans. Yrefanid takes approximately the same time for different levels of problems: so, it takes more time than our planner when the problem level is easy, but takes less time when the problem level is difficult. But Yrefanid does not compute optimal length plans.

4 Conclusion

We described a novel technique to improve answer-set based planning systems. It is based on bidirectional search, splitting the search space at an intermediate time point "n_{inter}" and approximative forward planning. From the results we conclude that the technique is effective and has its merits in terms of completeness and calculating shortest length plans. The parameter n_{inter} opens up options for a planner to be tuned towards specific domains. Apparently, it will play an equally important role in domains other than described here.

References

[Brogi *et al.*, 2003] A. Brogi, V. S. Subrahmanian, and C. Zaniolo. A Deductive Database Approach to A.I. Planning. *Journal of Intelligent Information Systems*, 20(3):215–253, 2003.

[Dimopolous *et al.*, 1997] Y. Dimopolous, B. Nebel, and J. Köhler. Encoding Planning Problems in Nonmonotonic Logic Programs. In *European Conference on Planning (ECP-97)*. Springer, 1997.

[Veloso *et al.*, 1995] M. Veloso, J. Carbonell, A. Perez, D. Borrajo, E. Fink, J. Blythe. Integrating Planning and Learning: The PRODIGY Architecture. In *Journal of Theoretical and Experimental AI, 7(1), 1995*.

[Eiter *et al.*, 2003] Th. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: the DLV system. *Artif. Intell.*, 144(1-2):157–211, 2003.

[Eiter *et al.*, 2004] Th. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Trans. Comput. Logic*, 5(2):206–263, 2004.

[Hoffmann and Nebel, 2001] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Lifschitz, 2002] V. Lifschitz. Answer set programming and plan generation. *AI*, 138:39–54, 2002.

[Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[Wernhard, 2003] C. Wernhard. System Description: KRHyper. Fachbericht 14–2003, Univ. Koblenz, 2003.