Enhancing Constraint Models for Planning Problems

Roman Barták*, Daniel Toropila*†

*Charles University in Prague, Faculty of Mathematics and Physics
Department of Theoretical Computer Science and Mathematical Logic
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
roman.bartak@mff.cuni.cz

†Charles University in Prague, Computer Science Center
Ovocný trh 5, 116 36 Praha 1, Czech Republic
daniel.toropila@ruk.cuni.cz

Abstract

Planning problems deal with finding a (shortest) sequence of actions that transfer the initial state of the world into a desired state. Frequently such problems are solved by dedicated algorithms but there exist planners based on translating the planning problem into a different formalism such as constraint satisfaction or Boolean satisfiability and using a general solver for this formalism. The paper describes how to enhance existing constraint models of planning problems by using techniques such as symmetry breaking (dominance rules), singleton consistency, and lifting.

Introduction

Planning is an important aspect of rational behavior and it is a fundamental topic of artificial intelligence since its beginning. Planning capabilities are necessary for autonomous controlling of vehicles of many types including space ships (Muscettola et al. 1998) and submarines (McGann et al. 2008), but we can also find planning problems in areas such as manufacturing, games or even printing machines (Ruml, Do and Fromherz 2005). Many approaches exist for solving planning problems; one of them is translation of the problem to a different formalism such as constraint satisfaction and using solving techniques developed for this formalism. In this paper we propose several improvements of the constraint model for solving classical AI (artificial intelligence) planning problems. The original base model was proposed in (Barták and Toropila 2008) and it used a straightforward encoding of the planning problem. In this paper we study three improvements of the base model. Two of these improvements are based on techniques used in constraint satisfaction. In particular, we use singleton consistency to prune more the search space and we use dominance rules to break symmetries in the problem (only one plan among the set of symmetrical plans is explored). The last improvement called lifting has been introduced in planning to decrease the branching factor during regression (backward) search.

The paper is organized as follows. We will first introduce the necessary concepts both from planning and from constraint satisfaction. Then we will describe the base constraint model used in our research. This is the best model from (Barták and Toropila 2008) that was obtained by reformulating constraints from the original model proposed in (Lopez and Bacchus 2003). After that, three new enhancements of this model will be proposed and finally these enhancements will be experimentally compared using several problems from International Planning Competition.

Classical AI Planning

Classical AI planning deals with finding a sequence of actions that transfer the world from some initial state to a desired state (Ghallab, Nau and Traverso 2004). The state space is large but finite. It is also *fully observable* (we know precisely the state of the world), *deterministic* (the state after performing the action is known), and *static* (only the entity for which we plan changes the world). Moreover, we assume actions to be instantaneous so we only deal with action sequencing.

Typically, the world *state* is described as a set of predicates that hold in the state, such as location(robot,loc1) saying that robot is located at loc1. In other words, for each predicate and for each state we describe whether the predicate holds in the state or not. This is called a classical representation of planning problems. *Actions* are described using a triple (Prec, Eff^+, Eff^-), where Prec is a set of predicates that must hold for the action to be applicable (preconditions), Eff^+ is a set of predicates that will hold after performing the action (positive effects), and finally Eff^- is a set of predicates that will not hold after performing the action (negative effects). For example, action move(robot,loc1,loc2) describing that robot moves from loc1 to loc2 is specified as triple ({location(robot,loc1)}, {location(robot,loc2)}, {location(robot,loc1)}). Formally, action a is applicable to state s if $\text{Prec}(a) \subseteq s$. The result of applying action a to state s is a new state $\gamma(s, a) = (s - \text{Eff}^-(a)) \cup \text{Eff}^+(a)$.

Notice that this description assumes a *frame axiom*, that is, other predicates than those mentioned among the effects of the action are not changed by applying the action. The set of predicates together with the set of actions is called a *planning domain*. We assume both sets of predicates and actions to be finite. The *goal* is specified as a set of predicates that must hold in the goal state, that is, if g is a goal then any state s such that $g \subseteq s$ is a goal state. The *classical planning problem* is defined by the planning domain, the initial state s_0 and the goal g and the task of planning is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ called a *plan* such that a_1 is applicable to the initial state s_0, a_2 is applicable to state $\gamma(s_0, a_1)$ etc. and $g \subseteq \gamma(\ldots\gamma(\gamma(s_0,a_1),a_2)\ldots,a_n)$.

There exists an alternative to the above logical formalism that is based on so called *multi-valued state variables*, as mentioned in (Bäckström and Nebel 1995) or (Helmert 2006). For each feature of the world, there is a variable describing this feature, for example rloc(robot,S) describes the position of robot at state S. Now instead of specifying validity of the predicate in some state S, say location(robot,loc1), we can specify the value of the state variable in a given state, in our example rloc(robot, S) = loc1. Hence the evolution of the world can be described as a set of state-variable functions where each function specifies evolution of values of certain state variable. Now, the actions are described as entities changing the values of state variables. We can still use preconditions specifying required values of certain state variables, but the positive and negative effects are merged to effects of setting the values of certain state variables. Notice that this multi-valued formulation is more compact than the logical formulation, where, for example, one needs to express explicitly that if robot is at loc1 then it is not present at another location. In the classical representation the action of moving robot from loc1 to loc2 needs to explicitly describe (in negative effects) that after performing the action, the predicate location(robot,loc1) is no more valid. In the multi-valued representation assigning value loc2 to state variable rloc(robot,S) implicitly means that robot is not at a different location at state S. The left part of Figure 3 gives an example how actions are represented when using the multi-valued state variables (note that the state is not explicitly present in the identification of the variable)

One of the difficulties of planning is that the length of the plan, that is, the set of used actions, is unknown in advance so some dynamic technique which can produce plans of "unrestricted" length is required. Frequantly, the shortest plan is being looked for, which is a form of *optimal planning*. As it has been shown in (Kautz and Selman 1992), the problem of shortest-plan planning can be translated to a series of SAT problems, where each SAT instance encodes the problem of finding a plan of a given length. First, we start with finding a plan of length 1 and if it does not exist then we continue with a plan of length 2 etc. until the plan is found. There exist criteria to stop these extensions if the plan does not exist, but for simplicity reasons in this paper we assume that a plan always exists. Now, the problem of finding a plan of length n can be encoded as a constraint satisfaction problem (Dechter 2003).

Constraint Satisfaction

A *constraint satisfaction problem* (CSP) P is a triple (X, D, C), where X is a finite set of decision variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints. A constraint is a relation over a subset of variables that restricts possible combinations of values to be assigned to the variables. Formally, a constraint is a subset of the Cartesian product of the domains of the constrained variables. A *solution to a CSP* is a complete assignment of values to the variables such that the values are taken from respective domains and all the constraints are satisfied. We say that a constraint C is (generalized) *arc consistent* if for any value in the domain of any constrained variable, there exist values in the domains of the remaining constrained variables in such a way that the value tuple satisfies the constraint. This value tuple is called a support for the value. Note that the notion of arc consistency is usually used for binary constraints only, while generalized arc consistency is used for n-ary constraints. For simplicity, we will use the term arc consistency independently of constraint's arity. The CSP is *arc consistent* (AC) if all the constraints are arc consistent and no domain is empty. To make the problem arc consistent, it is enough to remove values that have no support (in some constraint) until only values with a support (in each constraint) remain in the domains. If any domain becomes empty then the problem has no solution. We say that a value a in the domain of some variable x_i is *singleton arc consistent* if the problem $P|x_i=a$ can be made arc consistent, where $P|x_i=a$ is a CSP derived from P by reducing the domain of variable x_i to $\{a\}$. The CSP is *singleton arc consistent* (SAC) if all values in variables' domains are singleton arc consistent. Again, the problem can be made SAC by removing all SAC inconsistent values from the domains. Figure 1 shows an example of a CSP and its AC and SAC forms.

Consistency techniques such as AC or SAC can remove many infeasible values from domains of variables, but these techniques cannot guarantee finding the solution (removing all infeasible values) in a general case. Hence,

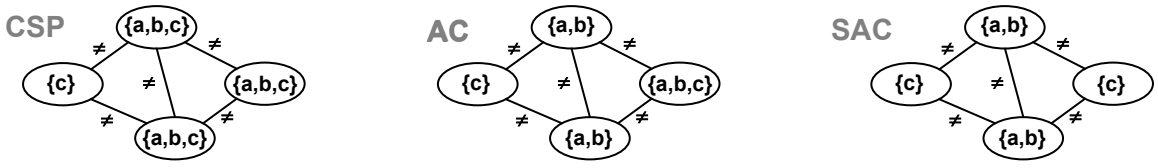


Fig. 1. A graph representation of a CSP, an arc consistent problem, and a singleton arc consistent problem (from left to right).

they are usually combined with a search algorithm in the following way. If the problem is consistent (for example AC or SAC) and there is a variable whose domain contains more than one value then one of such un-instantiated variables is selected (typically using some variable ordering heuristic) and a value from its domain (typically selected by some value ordering heuristic) is assigned to the variable. Then the problem is made consistent again and the procedure is repeated until all variables are instantiated. It may happen that after some instantiation the problem is found to be inconsistent. Then another value is tried and if there are no more values in the domain then the algorithm backtracks to the previous variable and tries an alternative value there. Briefly speaking, the search procedure splits the problem into disjoint sub-problems and these sub-problems are solved separately. In the above procedure, the sub-problems differed in the value assigned to the selected variable. Naturally, it is possible to use different branching schemes, for example splitting the domain of the variable into disjoint sets. This leads to a smaller branching factor during search and we will use this technique to improve efficiency of our constraint-based planner.

Note finally that maintaining consistency during search is a form of inference. After any choice during search, the decision is propagated via given consistency procedure to the rest of the problem so we can see some consequences of the decision (it is also called look ahead). Because the consistency procedure removes inconsistent values from domains, it actively prunes the search space. Naturally, the more inconsistencies are removed, the smaller search space we need to explore. On the other hand, stronger consistency techniques are computationally expensive and hence it is necessary to find a balance between the strength of the consistency technique (determined by the number of removed inconsistent values) and its efficiency. AC is the most frequently used consistency level maintained during search (the problem is made AC in each node of the search tree), while SAC is computationally too expensive to be applied in every node of the search tree. Nevertheless, SAC can be applied before search to remove some global inconsistencies as we shall show later for our planner.

Constraint Models for Planning Problems

As noted above, the problem of finding a plan of length n can be encoded as a constraint satisfaction problem. Then the following iterative technique from (Kautz and Selman 1992) can be used to find the shortest plan. A specific constraint model is first used to encode the problem of finding a plan of length n (starting with n=1). Then the search for plan is performed. In case of success, the plan found is returned, otherwise encoding for the problem of finding a plan of length n+1 is constructed (by extending the previous encoding with the new layer of variables and constraints, not building it from scratch). Whole process is repeated until the plan is found or computation runs out of time or another termination condition applies.

Existing Approaches

There exist several constraint models for the problem of finding a plan of length n. All these models share the idea of using a set of variables describing the states of the world and a set of variables describing the actions. We will present here the version with multi-valued variables introduced in (Barták and Toropila 2008). The world state is described using v multi-valued variables, instantiation of which exactly specifies a particular state. A CSP denoting the problem of finding a plan of length n consists of n+1 sets of above mentioned multi-valued variables, having 1st set denoting the initial state and k^{th} set denoting the state after performing k-1 actions, for $k \in \langle 2, n+1 \rangle$, and of n variables indicating the selected actions. Hence, we have $v(n+1)$ state variables V_i^s and n action variables A^j, where i ranges from 0 to v-1, j ranges from 0 to n-1, and s ranges from 0 to n (Figure 2).

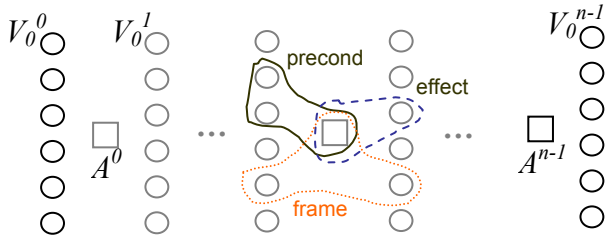


Fig. 2. Base decision variables and constraints modeling sequential plans.

Particular constraint models differ mainly in the set of constraints specifying the relations between the variables. These constraints connect two adjacent sets of state variables through the corresponding action variable between them. In other words, the constraints describe how the state variables change between the states if a particular action is selected.

A straightforward constraint model has been described in (Ghallab, Nau and Traverso 2004). It uses separate constraints to model action preconditions and effects. For given s we connect state variable layers V_i^s and V_i^{s+1}, $i \in \langle 0, v\text{-}1 \rangle$, through the action variable A^s:

$$A^s = act \rightarrow \text{Pre}(act)^s, \forall act \in \text{Dom}(A^s), \quad (1)$$

$$A^s = act \rightarrow \text{Eff}(act)^{s+1}, \forall act \in \text{Dom}(A^s), \quad (2)$$

where $\text{Pre}(act)^s$ and $\text{Eff}(act)^{s+1}$ are conjunctions of equalities setting the values for required state variables corresponding to preconditions of action act in layer s, and its effects in layer s+1 respectively. We also need constraints representing the *frame axioms*, i.e. constraints that would enforce equalities between those state variables V_i^s and V_i^{s+1}, which are not affected by selected action A^s (the frame assumption is implicit in classical planning representations):

$$A^s \in \text{NonAffAct}(V_i) \rightarrow V_i^s = V_i^{s+1}, \forall i \in \langle 0, v\text{-}1 \rangle, \quad (3)$$

where $\text{NonAffAct}(V_i)$ is a set of actions that do not have state variable V_i among its effects. Figure 2 shows which variables are connected by constraints (1) – (3).

A very similar model has been used in GP-CSP (Do and Kambhampati 2000). This planner exploits the structure of a planning graph (Blum and Furst 1997) and it allows parallel non-conflicting actions in each layer. Briefly speaking, there is an action variable for each state variable. This action variable specifies the action that sets the value of the state variable. Special *no-op* actions are necessary there to model that the state variable is not changed.

A different constraint model is used in CSP-PLAN (Lopez and Bacchus 2003). Rather than encoding the effects of actions, this model uses *successor state constraints* originally described in (Reiter 2001). From the perspective of previous models, the successor state constraints can be viewed as expressions that merge effect constraints and frame axioms together. The encoding of action preconditions is again using constraints of type (1). However, in contrary to the model above we use the successor state axioms as follows: for each possible assignment of state variable $V_i^s = val$, $val \in \mathrm{Dom}(V_i^s)$, we have a constraint between it and the same state variable assignment $V_i^{s-1} = val$ in the previous layer. The constraint says that state variable V_i^s takes value *val* if and only if some action assigned this value to the variable V_i^s, or equation $V_i^{s-1} = val$ held in the previous layer and no action changed the assignment of variable V_i. Formally:

$$V_i^s = val \leftrightarrow A^{s-1} \in C(i,val) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i)), \quad (4)$$

where $C(i,val)$ denotes the set of actions containing $V_i = val$ among their effects, and $N(i)$ denotes the set NonAffAct(V_i) as described within the previous models.

Base Model

In (Barták and Toropila 2008) we identified several problems of above models, namely the large number of constraints and weak domain filtering. The large number of constraints means that the consistency procedure runs longer as it needs to check consistency of all constraints. Weak domain filtering is mainly due to a disjunctive character of the above logical constraints (implication is a syntactic sugar for disjunction) – due to efficiency issues, most existing constraint solvers do not achieve full arc consistency for disjunctive constraints. Moreover, there are many constraints with the same scope (a set of constrained variables), which also contributes to weak domain filtering (each constraint is processed separately in AC).

To overcome these problems, in (Barták and Toropila 2008) we adopted the approach of substituting some of the above mentioned propositional formulae using the constraints with extensionally defined set of admissible tuples (sometimes also called *combinatorial constraints*). The idea was to union the scope of "similar" constraints and to define the admissible tuples in extension rather than using a formula. Such types of constraints are frequently called *table constraints*, because the set of admissible tuples is given in a table-like structure. There also exist compressed formats for describing admissible tuples, for example using a DAG (directed acyclic graph) that resembles decision trees. In SICStus Prolog, such a constraint is called *case constraint* (Carlsson, Ottosson and Carlson 1997). Our experiments showed that the reformulated version of CSP-PLAN leads to the best efficiency (Barták and Toropila 2008) so we will describe here only this reformulation which will serve as the base model for further enhancements.

As mentioned above, the main idea of the reformulation is substituting a set of "similar" logical constraints using a combinatorial constraint. The reformulated model of CSP-PLAN uses two types of combinatorial constraints. One constraint models the preconditions (1) and another constraint models the successor state axioms (4). The set of proposition formulae (1) in a single layer can be replaced by one combinatorial constraint in the following way. The scope of the constraint consists of action variable A^s and state variables V_i^s, that is, v+1 variables in total. Each action *act* is represented by a single row in the table of admissible tuples, which basically describes a restriction imposed by constraints (1) for $A^s = act$. The upper right table in Figure 3 gives an example of compatible tuples according to precondition constraints. Values of preconditions are filled there, while the remaining cells contain sets of values $\{0,\ldots,d_i\text{-}1\}$, where d_i is the domain size of variable V_i^s (each row describes several admissible

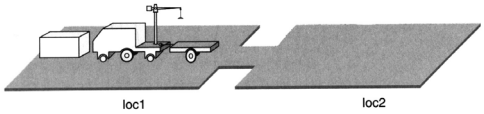
Domain
DWR domain with two locations (loc1,loc2), a robot capable of loading and unloading containers by itself (r), and one container (c)

State Variables
rloc ∈ {loc1,loc2} ;; robot's location
cpos ∈ {loc1,loc2,r} ;; container's position

Actions
1 : move(r, loc1, loc2)
;; robot r at location loc1 moves to location loc2
Precond: *rloc* = loc1
Effects: *rloc* ← loc2

2 : move(r, loc2, loc1)
;; robot r at location loc2 moves to location loc1
Precond: *rloc* = loc2
Effects: *rloc* ← loc1

3 : load(r, c, loc1)
;; robot r loads container c at location loc1
Precond: *rloc* = loc1, *cpos* = loc1
Effects: *cpos* ← r



4 : load(r, c, loc2)
;; robot r loads container c at location loc2
Precond: *rloc* = loc2, *cpos* = loc2
Effects: *cpos* ← r

5 : unload(r, c, loc1)
;; robot r unloads container c at location loc1
Precond: *rloc* = loc1, *cpos* = r
Effects: *cpos* ← loc1

6 : unload(r, c, loc2)
;; robot r unloads container c at location loc2
Precond: *rloc* = loc2, *cpos* = r
Effects: *cpos* ← loc2

Table for precondition constraint

A^s	$rloc^s$	$cpos^s$
1	loc1	{…}
2	loc2	{…}
3	loc1	loc1
4	loc2	loc2
5	loc1	r
6	loc2	r

Tables for successor state constraint

A^s	$rloc^s$	$rloc^{s+1}$
2	{…}	loc1
1	{…}	loc2
{3,4,5,6}	loc1	loc1
{3,4,5,6}	loc2	loc2

A^s	$cpos^s$	$cpos^{s+1}$
5	{…}	loc1
6	{…}	loc2
{3,4}	{…}	r
{1,2}	loc1	loc1
{1,2}	loc2	loc2
{1.2}	r	r

Fig. 3. Example of constraint model using combinatorial constraints.

tuples in a compact form). Clearly, the table can be computed using just the description of planning domain. In summary, all preconditions at a certain layer are now modeled using a single combinatorial constraint.

Successor state axioms can be represented using combinatorial constraints in a similar way. The scope of the combinatorial constraint consists of action variable A^s and state variables V_i^s and V_i^{s+1} so it is a ternary constraint. Briefly speaking, each such constraint describes how to set a value of variable V_i^{s+1} depending on action A^s and value of V_i^s (hence there are v such ternary constraints in each layer). The rows in the table are now determined by the values of variable V_i^{s+1}. There are at most $2d_i$ rows in the table: one row per value represents the situation when the value is not changed and one row per value represents the situation when the value is set by a certain action. This corresponds to disjunction in formula (4). The two lower right tables in Figure 3 give an example of admissible triples. The upper rows describe actions with effects related to variable V_i. The lower rows correspond to frame axioms for the same variable (all values are listed there). Note, finally, that information about preconditions can be used to reduce further the table for successor states. For example, actions 3 and 5 require *rloc* = loc1, so they can be removed from the last row of the table describing the change of rloc in Figure 3.

Base Search Strategy

So far, we only paid attention to constraint models of planning problems. Modeling is an important step, but as already noted, it is also necessary to specify a search strategy for instantiating the variables. One can use generic labeling techniques, for example based on first-fail principle (select the variable with the smallest domain first), however, it is usually better to utilize a labeling strategy derived from the nature of the problem. First, one should realize that it is enough to instantiate just the action variables A^s because when their values are known, then the values of remaining variables, in particular the state variables, are set by means of constraint propagation. Of course, we assume that the values for state variables V_i^0 modeling the initial state were set and similarly the state variables V_i^n in the final layer were set according to the goal (the final state is just partially specified so some state variables in the final layer remain un-instantiated).

We utilized a regression planning approach in the search strategy meaning that we instantiate the action variables in the decreasing order from A^{n-1} to A^0. This is called a fixed variable ordering in constraint satisfaction. For each action variable we assume only actions that contribute to (sub)goal in the next state layer – these actions are called relevant in (Ghallab, Nau and Traverso 2004). The values in each state variable were ordered decreasingly by the index i of the action variable A^i where the action appeared first. In other words, actions that appeared later in state variables were tried first for instantiation.

Model Enhancements

As shown in (Barták and Toropila 2008), the reformulation of constraints from the logical form to the combinatorial form reduced significantly the runtime for solving planning problems. The question, which we are answering in this paper, is whether it is possible to further improve efficiency by including more advanced solving techniques. In this section we describe three enhancements motivated by existing techniques from planning and constraint satisfaction.

Lifting

The search strategy used for the base model resembles the labeling technique in constraint satisfaction. At each step, we select an action that contributes to the current goal. Assume that there are 100 locations and part of the goal is that the robot must be at a specific location. When selecting the move action to satisfy this goal, we actually determined also the location from which the robot goes. This might be too restrictive, because we are building the plan from the end (backward planning), so we may find later that it is not possible to get the robot to the required start location. Hence, we need to backtrack to the choice point, where the move action was selected, and try a different one. Notice that we have 100 options at that choice point and the decision must be done without information which start location is the best one.

We propose to postpone the decision to the point when more information is available. This is called lifting in the planning community (Ghallab, Nau and Traverso 2004). This idea can be easily realized by modification of the search strategy. Rather than selecting a particular move action, we reduce the domain of the action variable to all move actions that lead to a given location. In terms of constraint satisfaction, we split the domain into two parts: one with the compatible move actions and one with the remaining actions that can be used at that step of the plan. As the search proceeds, the domain of the action variable may be further reduced via maintaining arc consistency (the later chosen actions may determine possible start locations for the move action). Still, it may happen that when reaching the initial state, some actions are not decided yet. In such a situation, we apply the standard labeling procedure to instantiate the action variables.

Let us now describe the process of lifting more formally. Let PrecVars(a) be the state variables appearing in the precondition of action a and EffVars(a) be the state variables changed by action a (these variables appear in effects of a). We say that actions a and b have the same scope if and only if PrecVars(a) = PrecVars(b) and EffVars(a) = EffVars(b). Let the base search procedure selects action a to be assigned to variable A^s; in other words we split the search space by resolving the disjunction $A^s = a \vee A^s \neq a$. In the lifted version, we are resolving the following disjunction:

$$A^s \in \text{SameScope}(a) \vee A^s \notin \text{SameScope}(a),$$

where SameScope(a) = { b | b has the same scope as a }.

Dominance Rules (a.k.a. Symmetry Breaking)

Recall, that we are looking for sequential plans, that is, for a sequence of actions. Assume that we have two actions a_1 and a_2 such that these actions do not interfere, for example, move action of a robot and load action of a different robot. If we have a valid plan where a_1 is right before a_2 then a plan where we swap both actions is also valid. This feature, sometimes called symmetry, can be exploited during search in the following way. Assume that at some stage of search we selected a_1 to be right before a_2 at some position of the plan. If this decision leads to a failure (no complete plan was found) then it is not necessary to explore plans where a_2 is right before a_1 at the same position because such plans will also be invalid. This feature can be used to prune the search space by omitting exploration of symmetrical plans. Another way to re-solve the very same problem is allowing parallel actions like in Graphplan (Blum and Furst 1997) or using partial order plans.

First, we need to define formally what it means that two actions do not interfere. Recall, that our motivation is that two actions a_1 and a_2 can be swapped without influencing validity of the plan. Swapping of actions a_1 and a_2 can be realized if for any state s the following condition holds: $\gamma(\gamma(s,a_1),a_2) = \gamma(\gamma(s,a_2),a_1)$. Such situation happens if actions a_1 and a_2 are independent which holds if sets $\mathrm{Eff}^-(a_2) \cap (\mathrm{Prec}(a_1) \cup \mathrm{Eff}^+(a_1))$ and $\mathrm{Eff}^-(a_1) \cap (\mathrm{Prec}(a_2) \cup \mathrm{Eff}^+(a_2))$ are empty (Ghallab, Nau and Traverso 2004) This definition for classical state representation can be transformed to the multi-valued state representation by requiring sets $\mathrm{EffVars}(a_2) \cap (\mathrm{PrecVars}(a_1) \cup \mathrm{EffVars}(a_1))$ and $\mathrm{EffVars}(a_1) \cap (\mathrm{PrecVars}(a_2) \cup \mathrm{EffVars}(a_2))$ to be empty.

Once we know how to recognize independent actions, we propose to include the following dominance rule to the search procedure. We choose arbitrary ordering of actions such that action a_i is before action a_{i+1} in the ordering (this ordering has nothing in common with the ordering of actions in the plan). Assume that action a_i has been assigned to state variable A^s. Then, when selecting action for state variable A^{s-1} (recall, that we are building the plan from the end), we only consider actions a_j for which at least one of the following conditions holds: either a_j and a_i are not independent, or $j > i$. In other words, if actions a_i and a_j are independent and $i < j$ then we explore only the plans where a_j can be right before a_i but not vice versa. This way, we prevent the solver from exploring permutations of mutually independent actions which decreases the size of the search space.

Note that the dominance rule can be combined with lifting presented in the previous section thanks to the definition of action independence. When using lifting, we do not assign a particular action to the action variable but we restrict the domain of the action variable to a set of actions with the same scope. Hence, even if a particular action is not yet selected but a set of actions is used for A^s, we can still take action a_i with the smallest i in the domain of A^s (after domain splitting) to participate in the dominance rule.

Singleton Consistency

So far, we discussed improvements of the search strategy. As we already mentioned, another way to improve efficiency of constraint solving is incorporating a stronger consistency technique. Singleton arc consistency would be a good candidate because it is easy to implement on top of arc consistency. However, as we already mentioned, it is computationally expensive to maintain SAC during search or even to make the problem SAC before search. Nevertheless, we can apply the idea of SAC in a restricted form. Recall, that in SAC we assign a value to some variable and propagate this information using AC to other variables. If a failure is detected then the respective value can be removed from the domain of the variable.

Assume that we failed to find a plan of length n and hence a new layer is added to the constraint model. New action variable A^n (recall, that A^0 is the first action) is introduced and connected with the state variables (new state variables V_i^n are also introduced). After posting new constraints involving this action variable, the problem is made AC. This removes some actions from the domain of variable A^n but according to our observations many actions that cannot be assigned to A^n still remain in the domain. To remove some of them, we propose to exploit the idea of SAC in the following way. We take all actions from the domain of A^n that do not appear in the domain of A^{n-1}. These are the newly introduced actions and we want to validate if these actions can be used at $(n+1)$-th step of the plan. Let a be such a newly introduced action. We assign a to variable A^n and try to find action b for A^{n-1} that supports $a - b$ provides certain precondition of a (in other words, we instantiate A^{n-1} by b). If this is not possible, we can remove a from the domain of A^n, because a can never be assigned to A^n. This process is performed for every newly introduced action. The hope is that we can eliminate actions that would otherwise be tried during search.

Let us describe the above process more formally. Let P be a constraint model describing the problem of finding a plan of length $n+1$ and a be an action that appears in the domain of A^n but not in the domain of A^{n-1} (a newly introduced action). If there is no action b such that $V_i = v$ is among its effects, $V_i = v$ is among preconditions of a, and $P|A^n=a, A^{n-1}=b$ is arc consistent then a can be removed from the domain of A^n. The reason for filtering out action a is that there is no plan of length n giving the preconditions of a. Hence this action cannot be used at the $(n+1)$-th position of any plan.

Experimental Comparison

To evaluate the proposed enhancements we implemented them in SICStus Prolog 4.0.2 (Carlsson, Ottosson and Carlson 1997) and compared them using selected planning problems from past International Planning Competitions (STRIPS versions). Namely, we used Gripper, Logistics, Mystery (IPC1), Blocks, Elevator (IPC2), Depots, Zenotravel, DriverLog (IPC3), Airport, PSR (IPC4), and

Pipesworld, Rovers, TPP (IPC5). The experiments ran on Pentium M 730 1.6 GHz processor with 1GB RAM under Windows XP.

Figure 4 shows comparison of runtimes to find a plan for all three enhancements when they are used separately. We sorted the planning problems increasingly using the runtime of the base model. In comparison with the base model, all enhancements produce some speedup (note that the logarithmic scale is used in the graphs) though no method alone is a clear winner. Either singleton consistency or lifting seems to be the methods that achieve best performance.

We also combined the proposed methods with the goal to strengthen their power. The experimental results are presented in Figure 5. For some problems, the runtime is worse than for the base model. This is due the computational complexity of singleton consistency (compare with Figure 4) that outweighs the positive effect of search space reduction in these problems. Nevertheless, there is a clear evidence that with the increasing hardness of the problems, the proposed enhancements pay off.

Conclusions

The paper proposed three enhancements of the base constraint model for solving sequential planning problems. Common feature of these enhancements is an attempt to reduce search space whose large size is a major obstacle when solving planning problems. The preliminary experiments showed that these enhancements contribute to better efficiency of planning though their individual contribution is not fully comparable. When all enhancements are used together, they significantly outperform (orders of magnitude) the base model, especially when the problems become hard.

The proposed constraint models are not yet competitive with the best current planners. Our goal is to provide an efficient framework for sequential planning that can be extended to cover more complex state transitions. In particular, constraints can describe any relation so we can go beyond logical formulas and use arithmetic formulas in preconditions and effects

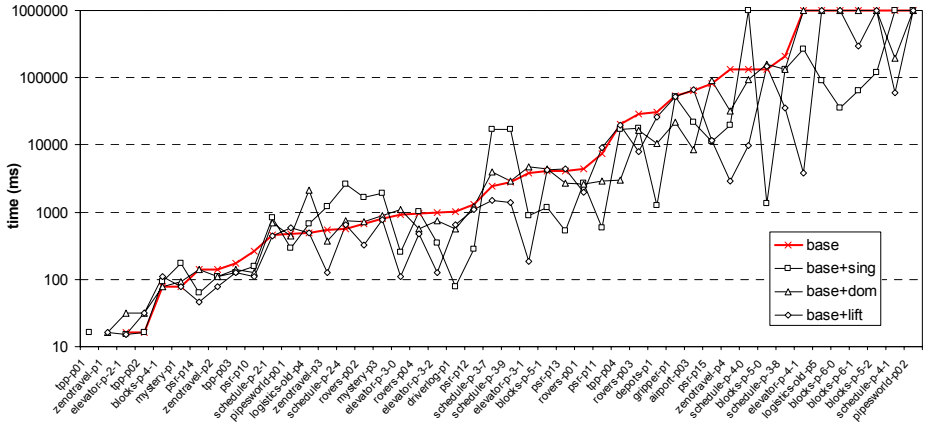


Fig. 4. Comparison of runtimes (logarithmic scale) for selected problems from IPC 1-5.

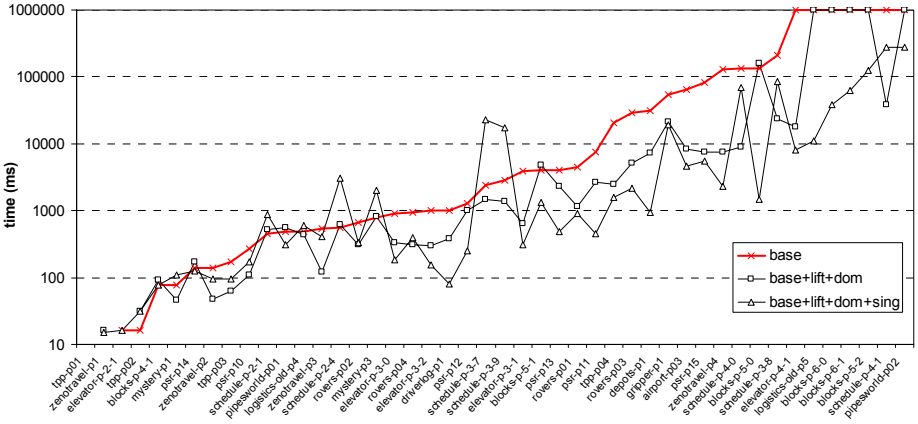


Fig. 5. Comparison of runtimes (logarithmic scale) for selected problems from IPC 1-5 when the methods are combined.

Acknowledgements

The research is supported by the Czech Science Foundation under the project 201/07/0205. We thank Malte Helmert for providing us the implementation of translator from PDDL to MPT planning problem specification.

References

Bäckström, Ch., Nebel, B. 1995. Complexity results for SAS^+ planning. *Computational Intelligence* 11(4), 625-655.

Barták, R., Toropila D. 2008. Reformulating Constraint Models for Classical Planning. *Proceedings of the 21st International Florida AI Research Society Conference (FLAIRS 2008)*. AAAI Press, pp. 525-530.

Blum, A. and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90, 281-300.

Carlsson M., Ottosson G., Carlson B. 1997. An Open-Ended Finite Domain Constraint Solver. *Programming Languages: Implementations, Logics, and Programs*.

Dechter, R 2003. *Constraint Processing*. Morgan Kaufmann.

Do, M.B. and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. *Proceedings of the Fifth International Conference on Artificial Planning and Scheduling* (AIPS-2000), AAAI Press, 82-91.

Ghallab, M., Nau, D., Traverso P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research 26*, 191-246.

Kautz, H. and Selman, B. 1992. Planning as satisfiability. *Proceedings of ECAI*, 359-363.

Lopez, A. and Bacchus, F. 2003. Generalizing GraphPlan by Formulating Planning as a CSP. *Proceedings of IJCAI*, 954-960.

McGann, C., Py, F., Rajan, K., Ryan, J., Henthorn, R. 2008. Adaptive Control for Autonomous Underwater Vehicles. *Proceedings of the Twenty-Third National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 1319-1324.

Muscettola, N., Nayak, P., Pell, B., Williams, B. 1998. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2), 5-47.

Reiter, R. 2001. *Knowledge in Action: Logical Foundation for Specifying and Implementing Dynamic Systems*. MIT Press.

Ruml, W., Do M.B., and Markus Fromherz, M. 2005. On-line planning and scheduling for high-speed manufacturing. In *Proceedings of the International Conference on Automated Planning and Scheduling* (ICAPS), AAAI Press, 30–39.