Integrating Constraint Models for Sequential and Partial-Order Planning

Roman Barták*, Daniel Toropila*†

{roman.bartak, daniel.toropila}@mff.cuni.cz

*Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic

†Charles University, Computer Science Center
Ovocný trh 5, 116 36 Praha 1, Czech Republic

Abstract

Classical planning deals with finding a (shortest) sequence of actions transferring the world from its initial state to a state satisfying the goal condition. Traditional planning systems explore either paths in the state space (state-space planning) or partial plans (plan-space planning). In this paper we show how the ideas from plan-space (partial order) planning can be integrated into state-space (sequential) planning by combining constraint models describing both types of planning. In particular, we extend our existing constraint model for sequential planning by constraints describing satisfaction of open goals. We demonstrate experimentally that this extension pays-off especially when the planning problems become harder.

Introduction

Constraint satisfaction is a traditional technology for solving scheduling problems, but it is less frequently applied when solving planning problems completely. CPlan (van Beek and Chen, 1999) was one of the first attempts to formulate a planning problem as a constraint satisfaction problem (CSP). However, this approach was based on manually formulating the constraint model. Do and Kambhampati (2000) showed that constraint satisfaction techniques can be applied to plan extraction from the planning graph (Blum and Furst, 1997). Their system GP-CSP automatically encoded the planning graph as a CSP. Lopez and Bacchus (2003) suggested a more efficient constraint formulation of the planning graph – CSP-PLAN – that used Boolean variables and successor-state constraints (Reiter, 2001). Barták and Toropila (2008) reformulated these models for sequential planning with multi-valued state variables and suggested using tabular (combinatorial, extensionally defined) constraints instead of logical constraints in the form of disjunction used in GP-CSP and CSP-PLAN. The efficiency of constraint models can be further improved by including techniques such as symmetry breaking, singleton consistency, lifting, and nogoods learning (Barták and Toropila, 2009).

The above models assume a sequential view of planning where the system produces plans in the form of a sequence of actions (GP-CSP and CSP-PLAN use parallel plans consisting of a sequence of sets of actions, where the actions in the set can be ordered in any way). Opposite to sequential planning, partial-order (plan-space) planning generates plans as a partially ordered structure of actions so the sequential plan can be obtained by linear ordering of the actions respecting the partial order. This has the advantage that artificial plan-permutation symmetries that appear during sequential planning are not explored during partial-order planning. CPT (Vidal and Geffner 2004) is probably the most successful (in terms of International Planning Competition) constraint-based planner that does partial-order planning.

In this paper we suggest to extend the sequential constraint-based planner from (Barták and Toropila, 2009) by some ideas originated from partial-order planning (and in some sense from GP-CSP). Namely, we propose adding redundant variables and constraints that describe actions giving particular goals and the position of these actions in the sequential plan. This redundant part of the model increases the propagation power of constraints so more inconsistencies are filtered out from the variables' domains and hence the search space to be explored is smaller. Despite the overhead of propagating through the redundant constraints, the overall efficiency increases especially for more complex planning problems.

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 from (Barták and Toropila, 2008) and its extensions from (Barták and Toropila, 2009). After that, we will introduce the variables and constraints describing the redundant part of the model together with the channelling constraints connecting the redundant part with the original model. Finally, we compare all these models experimentally using several problems from the International Planning Competitions.

Copyright © 2009, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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 *et al.*, 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 the actions to be instantaneous so we only deal with the correct sequencing of actions respecting the causal relations (an action that achieves a precondition of another action must be ordered before that action and no action destroying the precondition can be ordered in-between).

We use the SAS+ formalism to formalize the planning problem. This formalism 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. World *state* is specified by values of all state variables at the given state, for 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 the evolution of values of certain state variable. *Actions* are then the entities changing the values of the state variables. Each action consists of preconditions specifying the required values of certain state variables and effects of setting the values of certain state variables. The left part of Figure 2 gives an example of how the actions are represented when using the multi-valued state variables (note that the state is not explicitly present in the identification of the variable). We implicitly assume the *frame axiom*, that is, other state variables than those mentioned among the effects of the action are not changed by applying the action.

The set of state variables together with the set of actions is called a *planning domain*. We assume both sets to be finite. The *initial state* is specified by values of all state variables (it is known completely); the *goal* is specified by values of certain state variables (any state where the state variables have these required values is a *goal state*). The *classical planning problem* is defined by the planning domain, the initial state and the goal. The planning task is to find a shortest sequence of actions called a *plan* that transfers the initial state into one of the goal states.

Constraint Satisfaction

Many combinatorial optimization problems including the planning problems can be encoded and solved as constraint satisfaction problems (Dechter, 2003). 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. 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.

A typical constraint satisfaction approach is a combination of inference and depth-first search. The most frequently used inference technique for CSPs is called arc consistency. We say that a constraint 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. The CSP is *arc consistent* (AC) if all the constraints are arc consistent and no domain is empty. A stronger consistency technique derived from AC is singleton arc consistency. 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.

As consistency techniques usually do not remove all inconsistencies they need to be combined with search that resolves the remaining alternatives. The search procedure splits the problem into disjoint sub-problems that are solved separately. Typically, the sub-problems differ in the value assigned to a selected variable; however 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. Note that the consistency technique is typically applied after each search decision to see the effect of the decision – this is a form of *look ahead*. Either it fails to make the problem consistent, then the search backtracks (the whole sub-tree is pruned), or it prunes parts of the search space. Because the consistency procedure is called in each node of the search tree, 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, 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.

The efficiency of constraint solving is highly influenced by the choice of decision variables and constraints – a so called *constraint modeling*. For example, arc consistency cannot infer a lot for Boolean variables until some variable is instantiated (filtering out a value from the binary domain also corresponds to variable instantiation). Similarly, most constraint solvers do not achieve full arc consistency for the disjunctive constraints unless a more computationally expensive constructive disjunction is used. The constraint models can be enhanced by several ways. For example adding *symmetry breaking constraints* removes the symmetrical solutions from the search space and adding *redundant constraints* (the constraints whose satisfaction is guaranteed by the existing constraints) strengthens domain filtering and hence decreases the search space. We shall show later how these techniques can improve the constraint models for planning problems.

Base Constraint Model for Planning

One of the difficulties of planning is that the length of the plan, that is, the sequence of used actions, is unknown in advance so some dynamic technique which can produce plans of "unrestricted" length is required. Usually, the shortest plan is sought, which is a form of *optimal planning*. As it has been shown by 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. The whole process is repeated until the plan is found or the computation runs out of time or another termination condition applies. The same idea can be applied to modeling the problem as a series of CSPs.

The base constraint model is a version of CSP-PLAN (Lopez and Bacchus, 2003) reformulated by Barták and Toropila (2008) to sequential planning with multi-valued state variables. 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 the above-mentioned multi-valued variables, having 1st set denoting the initial state and kth 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 1).

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. There are two types of constraints in the model: precondition constraints and successor state constraints. For a given layer s the *precondition constraint* connects state variable layer V_i^s, $i \in \langle 0, v\text{-}1\rangle$ with action variable A^s:

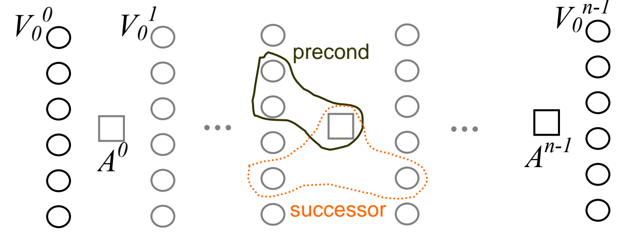


Fig. 1 Base decision variables and constraints.

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

The *successor state constraints* merge the effect constraints and the frame axioms together as follows: for each possible assignment of state variable $V_i^s = val$, $val \in \text{Dom}(V_i^s)$, we have a constraint between it and the same state variable assignment $V_i^{s\text{-}1} = val$ 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 variable V_i^s, or equation $V_i^{s\text{-}1} = val$ held in the previous layer and no action changed the assignment of variable V_i:

$$V_i^s = val \leftrightarrow A^{s\text{-}1} \in C(i,val) \vee (V_i^{s\text{-}1} = val \wedge A^{s\text{-}1} \in N(i)), \quad (2)$$

where $C(i,val)$ denotes the set of actions containing $V_i = val$ among their effects, and $N(i)$ denotes the set of actions that do not affect V_i.

As already noted, disjunctive constraints do not propagate well and hence all precondition constraints for a given layer are encoded as a single tabular constraint (sometimes also called a *combinatorial constraint*) where the set of admissible tuples is given in a table-like structure. Similarly, all successor state constraints for a given state variable (and a layer) are encoded as a single tabular constraint (Figure 2, right).

Base Search Strategy

Modeling is an important step, but as already noted, it is also necessary to specify the search strategy for instantiating the variables. One can use generic labeling

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

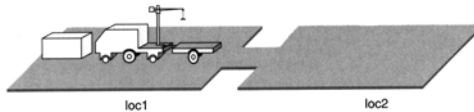
rloc ∈ {loc1,loc2} ;; robot's location
cpos ∈ {loc1,loc2,r} ;; container's position

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	{loc1,loc2,r}
2	loc2	{loc1,loc2,r}
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,loc2}	loc1
1	{loc1,loc2}	loc2
{3,4,5,6}	loc1	loc1
{3,4,5,6}	loc2	loc2

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

Fig. 2. Example of constraint model using combinatorial constraints; the domain is taken from (Ghallab *et al*., 2004).

techniques, for example based on dom heuristics (select the variable with the smallest domain first), but our first experiments showed that this is not efficient for the proposed constraint model. 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 *et al.*, 2004). The actions (values) in the action variable are explored in the order of appearance in the plan – the action that appeared later in the plan is tried first for instantiation.

Model Enhancements

It is rare that a direct constraint model is enough to solve complex problems and frequently some extensions are necessary. In (Barták and Toropila, 2009) we proposed four improvements of the base model. In particular, we suggested using lifting (called domain splitting in constraint satisfaction) to decrease the branching factor, dominance rules to break plan-permutation symmetries in the problem, singleton consistency to prune more of the search space by eliminating certain unreachable actions, and, finally, nogoods recording to learn unsatisfied goals.

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 (regression/backward planning is used). As noted in (Ghallab *et al.*, 2004) this strategy may be overcommitted, for example, when requiring a robot to be at certain location by selecting the move action we are also deciding from which location the robot will go. There might be many such actions (depending on the number of locations) so it seems more appropriate to postpone some particular decision to later. This is called *lifting* as instead of selecting a particular action we lift the decision by assuming a set of "similar" actions reaching the same goal. In terms of constraint satisfaction, this is realized by splitting the domain of the action variable rather than instantiating the variable.

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 the 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 select 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 the sequential plans. Assume that we have two actions a_1 and a_2 such that these actions do not interfere, for example, a move action of certain robot and load action of a different robot. If we have a valid plan where a_1 is right before a_2 then the plan where we swap both actions is also valid. This feature, called *plan-permutation symmetry* (Long and Fox, 2003), can be exploited during search in the following way.

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: $\gamma(\gamma(s,a_1),a_2) = \gamma(\gamma(s,a_2),a_1)$, where $\gamma(s,a)$ is a state obtained by applying action a to state s. Such situation happens if actions a_1 and a_2 are independent (Ghallab *et al.*, 2004), but a weaker *allowance condition* can be applied:

EffVars(a_1) $\cap$ PrecVars(a_2) = $\emptyset$,
Effects(a_2) don't clash with Preconditions(a_1),
Effects(a_1) don't override Effects(a_2).

We say that effects of action *a clash* with preconditions of action b if action a assigns value *val* to some state variable V_i such that the assignment $V_i = val$ is inconsistent with the preconditions of action b. Also, we say that effects of action a do not *override* effects of action b if both actions a and b set the same value for each state variable $V \in$ (EffVars(a) $\cap$ EffVars(b)).

The allowance relation between a_1 and a_2 guarantees that if a_1 appears before a_2 in some plan then we can swap the actions without changing the resulting state. However, the allowance relation is not symmetrical (in contrast to the independence relation) and it does not guarantee a sound swap of actions if a_2 appears before a_1.

Now it is possible to include the following dominance rule to the search procedure. We choose an 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 (the action at position s). Then, when selecting the action for state variable A^{s-1}, we only consider actions a_j for which at least one of the following conditions holds: either a_j and a_i violate the allowance condition or $j > i$. This way, we prevent the solver from exploring permutations of actions leading to the same resulting state.

Singleton Consistency

So far, we discussed improvements of the search strategy. Another way to improve the efficiency of constraint solving is incorporating a stronger consistency technique. Singleton arc consistency (SAC) would be a good candidate because it is easy to implement on top of arc consistency. However, it is computationally expensive to make the problem SAC so we suggest applying SAC in a restricted form.

When a new layer is added to the constraint model we check whether the newly assumed actions have a support in the previous layer. 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). For any precondition $V_i=v$ of a, if there is no action b such that $V_i=v$ is among its effects, 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 (the n-th layer is the first layer where action a appeared so the precondition cannot be provided fully by actions before layer n-1).

Nogood Learning

The last (but not least) improvement we have incorporated is the use of so-called *nogoods*. This well-known technique, often mentioned in the connection with dependency-directed backtracking or backjumping, helps the planner to leverage from the failures it has encountered during the search, and uses them to avoid the same failures later, saving thus valuable time the search procedure would have spent otherwise for exploring the same failure again. In order to beware that case, we have to do a single thing: memorize the reason of the failure – a nogood.

As described above, the search starts with the goals to be satisfied and continues backwards by selecting an action A^n that satisfies some of the goals. Preconditions of the selected action are then merged with unsatisfied goals in order to create new goals for the next step of the search (note that the next step of the search moves to the layer n-1). Thus, encountering a failure in fact means that the set of goals at a given layer is unsatisfiable (there is no other action that we could try to apply in order to satisfy the required goals) and that the next time we can avoid trying to satisfy the same set of goals at that layer – a nogood is recorded for future reference as a set of unsatisfiable goals at certain layer. Formally, the goal is a set of acceptable values for a given state variable:

$$Goals \approx V_1 \in Vals_1 \wedge V_2 \in Vals_2 \wedge \ldots \wedge V_k \in Vals_k,$$

where $Vals_i$ represents the set of allowed values for variable V_i.

For each layer we record the sets of goals that are proved to be unsatisfiable for that layer – a nogood. Next time, when we reach the same layer with a different set of goals G, we claim this set of goals to be unsatisfiable (without search) if it is more demanding than some stored nogood H. The set of goals G is more demanding than set of goals H, if for each state variable $V_i \in H$ also $V_i \in G$ and $Vals_i(G) \subseteq Vals_i(H)$, where $Vals_i(X)$ denotes the set of allowed values for variable V_i within the set of goals X. Currently, we store only the "direct" nogoods without trying to generalize the set of stored nogoods.

Dual Model

There are several problems of sequential planning that decrease the overall efficiency of planners. One of them is exploration of symmetrical plans that do not solve the planning problem (Long and Fox, 2003). We addressed (partially) this problem by introducing the symmetry breaking constraints. Another problem is hidden in the incremental extensions of the plan length as proposed by Kautz and Selman (1992). Our experiments showed that the sequential planner spends most of the time by proving that plans of shorter length does not exist and as soon as the planner reaches the particular layer where the goal can be satisfied, finding the plan is frequently fast. Hence, it would be a big advantage if the planner can easily (in a fast way) detect that the plan of a given length does not exist and move to the next layer.

The particular problem that we are trying to resolve is as follows. Assume that there are k different state variables in the goal and the value of each of these state variables must be set by a different action. Clearly, it means that we need the plan with at least k actions. Unfortunately, the sequential planner cannot detect such a situation and it explores by search many plans of shorter lengths. However, if we know that k different actions are necessary and there are less than k available slots (layers), we can deduce immediately that no plan exists.

The resolve the above problems we suggest exploiting partial-order planning also known as plan-space planning (Ghallab *et al.*, 2004). This type of planning focuses purely on causal relations and does not explicitly assume states. This is not appropriate for the planning problems where, for example, the values of some state variables are derived from the values of other state variables. Hence, we tried to include some of the principles of partial-order planning in the sequential planner where we still have the explicit representation of world states.

We suggest the following extension of the constraint model that we call a *dual model*. The motivation goes from partial-order planning where one of the tasks is closing open goals, that is, finding an action that provides the value of the state variable in the goal or in the precondition of another action. For each state variable V_i in the goal condition we introduce a *support variable* S_i describing which action is setting the requested value. Assume that the requested value of the state variable is b then the domain of the support variable is a set of actions (supports) that have $V_i \leftarrow b$ among their effects. If there are more possible values for the state variable (this may happen due to lifting) then we use a binary constraint between the support variable and the state variable that ensures that actions in the domain of the support variable provide the

values of the state variable and vice versa. Moreover, we also introduce *layer variable* L_i identifying the layer where the support action is located. Assume that the goal appears at layer n, then the initial domain of the layer variable is $\{0,\ldots,n\text{-}1\}$ (recall that the action layers are numbered from 0). Though it is possible to introduce the above support and layer variables statically based on the maximal number of possible sub-goals during planning, we have found it to be more efficient to introduce these variables dynamically on-demand. It means that at the beginning we introduce these variables just for the state variables in the planning goal and when we move during search to the previous layer we introduce these variables for the preconditions of the selected action (and remove the variables upon backtracking). This dual model mimics the behavior of partial-order planners where the planner is finding a supporting action for the open goal. Note also that similar support variables were used in GP-CSP planner (Do and Kambhampati, 2000) with the difference that the support variables in GP-CSP are defined for each state variable and the support action is assumed to be exactly in the previous layer (hence no-op actions are used in case the action setting the value of the state variables is not in the layer directly preceding the sub-goal). In our model, the support actions are introduced just for the open goals (preconditions) and the support action can be in any previous layer (defined in the layer variable).

Clearly the dual model must be connected via so called *channelling constraints* to the original model to ensure that the support action appears exactly in the layer specified by the layer variable. There is already a channelling constraint between the support variable and the state variable, but we can do more by connecting the support variables with the action variables (slots). This is where we use the layer variables. Assume that we are at layer k so we are still looking for actions in action variables $A^0,\ldots, A^{k\text{-}1}$ (recall that we are doing backward planning). For each support variable S and its corresponding layer variable L we post the constraint *element*(L,[$A^0,\ldots, A^{k\text{-}1}$],$S$). The semantics of *element*(X,List,Y) is as follows: Y is the X-th element in List (the first element has index 0). This constraint connects support actions with actions that can be used in the plan in particular positions (reachable actions). It helps with pruning actions that cannot be used as a support for a particular goal because they are not reachable.

Let us recall the original motivation for introducing the dual model – we would like to detect situations where k different support actions are necessary but less than k slots are available. This can be implemented by using constraints ensuring that there are enough slots (layers) for the support activities. Assume that we are finding supports for m goals in the k-th layer, that is, there are k available slots for actions. If $k < m$ then it may happen that it is not possible to place all supports to the slots unless the supports are shared (one action supports several goals). In such a situation, we post a constraint *nvalue* (Bessiere *et al.*, 2005) over the set of support variables in this layer and some integer variable N whose domain is $\{1,\ldots,k\}$. The constraint *nvalue*(S,N) ensures that there are exactly N different values in the set S. In our setting, the constraint ensures that there are at most k different support actions for a given layer. The combination of *nvalue* and *element* constraints may deduce that actions cannot be at given position in the plan because otherwise there are not enough slots for other supports.

In summary, the dual constraint model is introduced dynamically during the search. Each time we introduce a new goal (this is the precondition of just selected action in the plan) we generate the support and layer variables and post the corresponding constraints that use these variables.

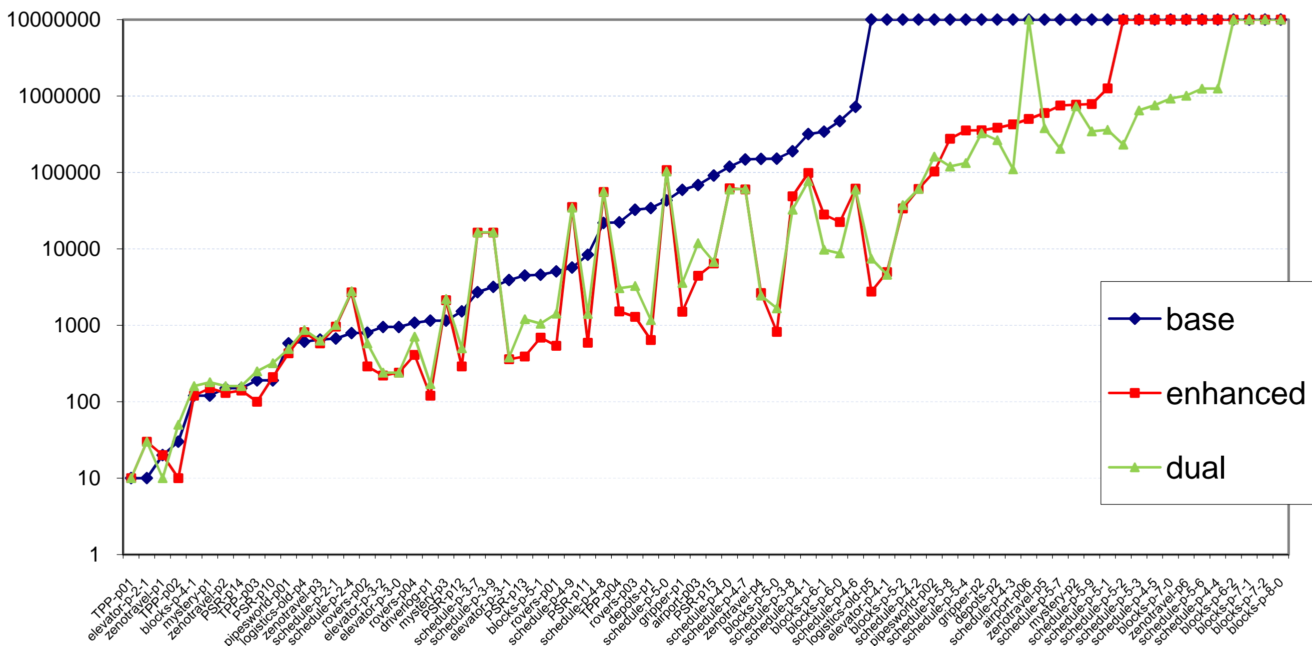


Fig. 3. Comparison of runtimes (logarithmic scale) for selected problems from IPC 1-5.

Experimental Comparison

We implemented the base model, its enhancements, and the model with redundant constraints (we call it dual) using clpfd library of SICStus Prolog 4.0.5 and we compared all three models using the 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 Intel Xeon CPU E5335 2.0 GHz processor with 8GB RAM under Ubuntu Linux 8.04.2 (Hardy Heron). The reported runtime includes the time to generate the constraint model (prepare the tables) and the time to find the shortest (optimal) plan.

Figure 3 shows the comparison of runtimes (in milliseconds; using 30 min. time limit) to find the shortest plan for all models. We sorted the planning problems increasingly using the runtime of the base model.

The results clearly demonstrate that the enhancements of the base model improved efficiency though for some simpler problems the runtime is worse due to the computational complexity of singleton consistency that outweighs the positive effect of search space reduction in these problems. The influence of particular enhancement to efficiency is studied in more details in (Barták and Toropila, 2009). Adding the redundant constraints in the style of partial-order planning further improved efficiency for the harder problems and we can solve more problems than with the original constraint model. Moreover, the overhead of propagating through the redundant constraints is not significant though for some simpler problems it makes the runtime worse than for the original model.

Conclusions

The paper studies the constraint models for traditional sequential planning. We presented the base model from (Barták and Toropila, 2008) that already significantly outperformed a sequential version of CSP-PLAN. This model was enhanced by some traditional techniques used in planning and constraint satisfaction (Barták and Toropila, 2009). A completely new contribution of this paper is the introduction of the redundant constraints in the style of partial-order planning. This dual model further improves the time efficiency. It is important to mention that all constraint models are generated fully automatically from the declarative description of the planning problem (we have a semi-automated translation from PDDL to this Prolog-like problem description). Moreover, thanks to the constraint satisfaction technology, the models are ready to go beyond logical reasoning and various numerical preconditions and effects can be naturally integrated.

As a future work, we plan to improve the filtering power of the channeling constraints as we believe that even stronger integration and hence pruning may be achieved. We will also focus on improving the search strategy; in particular, it might be interesting to do search (at least partially) in the dual part of the model.

Acknowledgments

The research is supported by the Czech Science Foundation under the projects 201/07/0205 and 201/09/H057.

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. In: *21st International Florida AI Research Society Conference* (FLAIRS 2008), AAAI Press, 525-530

Barták, R., Toropila D. 2009. Enhancing Constraint Models for Planning Problems. To appear in ISMIS 2009 (a preliminary version available from PlanSIG 2008).

Bessiere, Ch., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T. 2005. Filtering algorithms for the NValue constraint. In *Proceedings CPAIOR'05*, LNCS 3524, Springer Verlag.

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

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.

Long, D., Fox. M. 2003. Plan Permutation Symmetries as a Source of Planner Inefficiency. In: *22nd Workshop of UK Planning and Scheduling Special Interest Group* (PlanSIG-22)

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

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

van Beek, P. and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. *Proceedings of AAAI-99*, 585-590.

Vidal, V. and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. *Proceedings of AAAI-04*, 570-577.