

# Encoding Domain Transitions for Constraint-Based Planning

Nina Ghanbari Ghooshchi<sup>a,b\*</sup>

NINA.GHANABRI@GRIFFITHUNI.EDU.AU

Majid Namazi<sup>a\*</sup>

M.NAMAZI@GRIFFITH.EDU.AU

M.A.Hakim Newton<sup>a</sup>

MAHAKIM.NEWTON@GRIFFITH.EDU.AU

Abdul Sattar<sup>a,b</sup>

A.SATTAR@GRIFFITH.EDU.AU

<sup>a</sup>*Institute for Integrated and Intelligent Systems (IIIS),*

*Griffith University, Brisbane, Australia*

<sup>b</sup>*Data61, CSIRO, Australia*

## Abstract

We describe a constraint-based automated planner named Transition Constraints for Parallel Planning (TCPP). TCPP constructs its constraint model from a redefined version of the domain transition graphs (DTG) of a given planning problem. TCPP encodes *state transitions* in the redefined DTGs by using table constraints with cells containing *don't cares* or wild cards. TCPP uses Minion the constraint solver to solve the constraint model and returns a parallel plan. We empirically compare TCPP with the other state-of-the-art constraint-based parallel planner PaP2. PaP2 encodes *action successions* in the finite state automata (FSA) as table constraints with cells containing sets of values. PaP2 uses SICStus Prolog as its constraint solver. We also improve PaP2 by using *don't cares* and *mutex* constraints. Our experiments on a number of standard classical planning benchmark domains demonstrate TCPP's efficiency over the original PaP2 running on SICStus Prolog and our reconstructed and enhanced versions of PaP2 running on Minion.

## 1. Introduction

The use of inference-based pruning techniques and path heuristic-based choice guidances are key to solve a combinatorial optimisation problem efficiently (Hooker, 2005). Automated planning is a combinatorial optimisation problem that has achieved significant progress over the last decade. However, the progress has been achieved mostly through the use of effective but cheap relaxation-based path heuristics within the traditional informed search frameworks. Overall, inferences are somewhat neglected within the heuristic search for planning. Constraint satisfaction problems (CSP) are another kind of combinatorial optimisation problems where inference-based pruning techniques play pivotal role in their solution techniques. Path heuristics are in general not developed in CSP because path to the solution is not of concern. In this paper, we take the approach of using CSP techniques to solve planning problems and describe our recently developed constraint-based planner.

Constraint-based planners translate a given planning problem into a series of constraint satisfaction problems. The translated problems are then solved by using a typical CSP solver. These planners attempt to take the advantage of enhanced propagation machineries and better pruning mechanisms available for typical CSPs. Overall, constraint-based planners do not yet obtain the performance level of the state-of-the-art heuristic search

---

\*. These authors have contributed equally to this paper

planners. In a constraint satisfaction problem, the constraint model, the search algorithm and the selection heuristics interact with each other (Beacham, Chen, Sillito, & van Beek, 2001) and their choices should not be made independently. In planning, a problem model is already given in the form of a domain description using a planning language such as the planning domain definition language (PDDL) (Ghallab, Knoblock, Barrett, Christianson, Friedman, Kwok, Golden, Penberthy, Smith, Sun, & Weld, 1998). However, there could be various ways in which to model a given planning problem as CSPs and then exactly which CSP solver is to use to solve the constraint model. To improve constraint-based planning, in this paper, we investigate this direction and describe our constraint-based planner.

Considerably little work has been done in solving planning problems by using CSP techniques. Constraint models for planning problems have been designed for temporal planning in (Ghallab & Laruelle, 1994). Later, manually designed constraint models were used in classical planning (van Beek & Chen, 1999). Planners in (Do & Kambhampati, 2001) and (Lopez & Bacchus, 2003) translate the so called *planning graph* structures (Blum & Furst, 1995) into CSPs with a view to generating parallel plans. These planners mostly use constraints with logical formulas; which follows the propositional nature of the PDDL language (Ghallab et al., 1998). Constraints for planning problems are later represented extensionally by table constraints and are constructed from the multi-valued SAS<sup>+</sup> representation, which is a member of the Simplified Action Structure family (Bäckström & Nebel, 1995). The extensional representation has showed a great improvement in the efficiency, which has later been further improved by inclusion of symmetry breaking, singleton arc consistency, and *no-good* learning (Barták & Toropila, 2009a, 2009b). Few constraint-based planners (Cesta & Fratini, 2008; Verfaillie, Pralet, & Lemaître, 2010) are based on time lines. The planner in (Gregory, Long, & Fox, 2010) uses dominance constraints, and another in (Judge & Long, 2011) applies goal and variable/value heuristics and uses meta-CSP variables. In (Barták, 2011a, 2011b), the finite state automata (FSA) which are similar to domain transition graphs (DTG) (Helmert, 2006) are used to build the constraint model.

In this paper, we describe a constraint-based automated planner named Transition Constraints for Parallel Planning (TCPP)<sup>1</sup>. TCPP constructs its constraint model from a redefined version of DTGs of a given planning problem. In the redefined DTGs, loops are used to model *no-ops* at the vertexes. Moreover, a *don't care* value is used in the redefined DTGs for each variable that is not in the preconditions of an action but is in the effects of the same action. The use of DTGs is to exploit the structural information of the SAS<sup>+</sup> formalism and is inspired by a similar use by the SAT-based planner SASE (Huang, Chen, & Zhang, 2010). TCPP encodes *state transitions* in the DTGs by using table constraints with cells containing *don't cares* or wild cards. Table constraints are efficient when the number of valid assignments is small with respect to the total number of assignments. However, we additionally and more importantly use *don't cares* or wild cards in table cells to allow a compact representation for many constraints that would otherwise need consideration of all possible combinations of certain column values. To ensure parallel actions in a plan are not conflicting with each other, TCPP encode parallelism constraints by using parallelism variables. TCPP optionally also use negative table constraints to encode *mutex* constraints. TCPP uses Minion the constraint solver (Gent, Jefferson, & Miguel, 2006) to solve the constraint

---

1. A preliminary report of this work has been published in the Proceedings of the Twenty-Ninth National Conference on Artificial intelligence (AAAI) (Ghooshchi, Namazi, Newton, & Sattar, 2015)

model and returns the parallel plan. Minion can efficiently handle table constraints with *don't cares* through algorithms for short support enabled arc consistency propagations. This combination improves the efficiency of our planner over the state-of-the-art constraint-based planner PaP2 (Barták, 2011b) on a set of standard classical benchmark domains.

Constraint-based planner PaP2 encodes the *action successions* in FSA as table constraints with cells containing *sets of values* and uses SICStus Prolog as its constraint solver. The FSA are similar to the original DTGs but additionally there are loops for *no-ops* in the FSA. FSA are different from our redefined DTGs, which additionally might have *don't care vertexes*. Moreover, there are differences in the labelling of the edges in the FSA and our redefined DTGs. Nevertheless, action successions in FSA are pairs of actions such that in the FSA, the latter action follows the former action satisfying their causal dependency. Besides using the original PaP2, we also have reconstructed it to obtain a version PaPR that runs on Minion and thus allows us to observe the effect of different constraint models running on the same solver platform. While in TCPP, we use table constraints with *don't cares* to encode state transitions in DTGs, in this paper, we attempt to do the same to encode PaP2 model. Moreover, we optionally use mutex constraints. These results in an enhanced version of PaPR. On the same benchmark domains as mentioned before, the enhanced PaPR demonstrates its efficiency over PaP2. Overall, Minion as a CSP solver along with its table constraints allowing *don't cares* are empirically found to be effective in exploiting both the state transitions and actions successions in the DTGs.

The rest of the paper is organised as follows: Section 2 gives an overview of classical planning, DTGs, CSPs, and constraint-based planners; Section 3 outlines the architecture of our constraint-based planner TCPP; Section 4 discusses DTG-based planning model; Section 5 describes our encoding of planning problems into CSPs by using table constraints with *don't cares* and decoding of CSP solutions back to plans; Section 6 presents our reconstructed planner that models PaP2-style action successions from DTGs; Section 7 presents our experimental results and analyses; and finally, Section 8 presents the conclusions.

## 2. Preliminary Knowledge

We give an overview of classical planning and domain transition graphs. We also give an overview of constraint satisfaction problems and constraint-based planning. While we explain the preliminary concepts using somewhat detailed examples, an experienced reader could just go through the definitions to know our notations.

### 2.1 Classical Planning

Given an *initial state* and a desired *goal state*, the *planning problem* is to find a sequence of *actions*, called a *plan*, that transforms the initial state into the goal state. In planning, actions are responsible for changing the world and are described by their preconditions and effects. Preconditions specify the conditions the world needs to have before applying an action and effects specify the changes that an action makes to the world.

To illustrate a planning problem, we use a simplified driverlog domain (see Figure 1 top). In this domain, we have drivers and trucks, but no packages. We thus restrict this domain only to the transportation of drivers by trucks. A driver can change his location by walking or by driving a truck. There are roads (solid lines) for driving trucks and footpaths

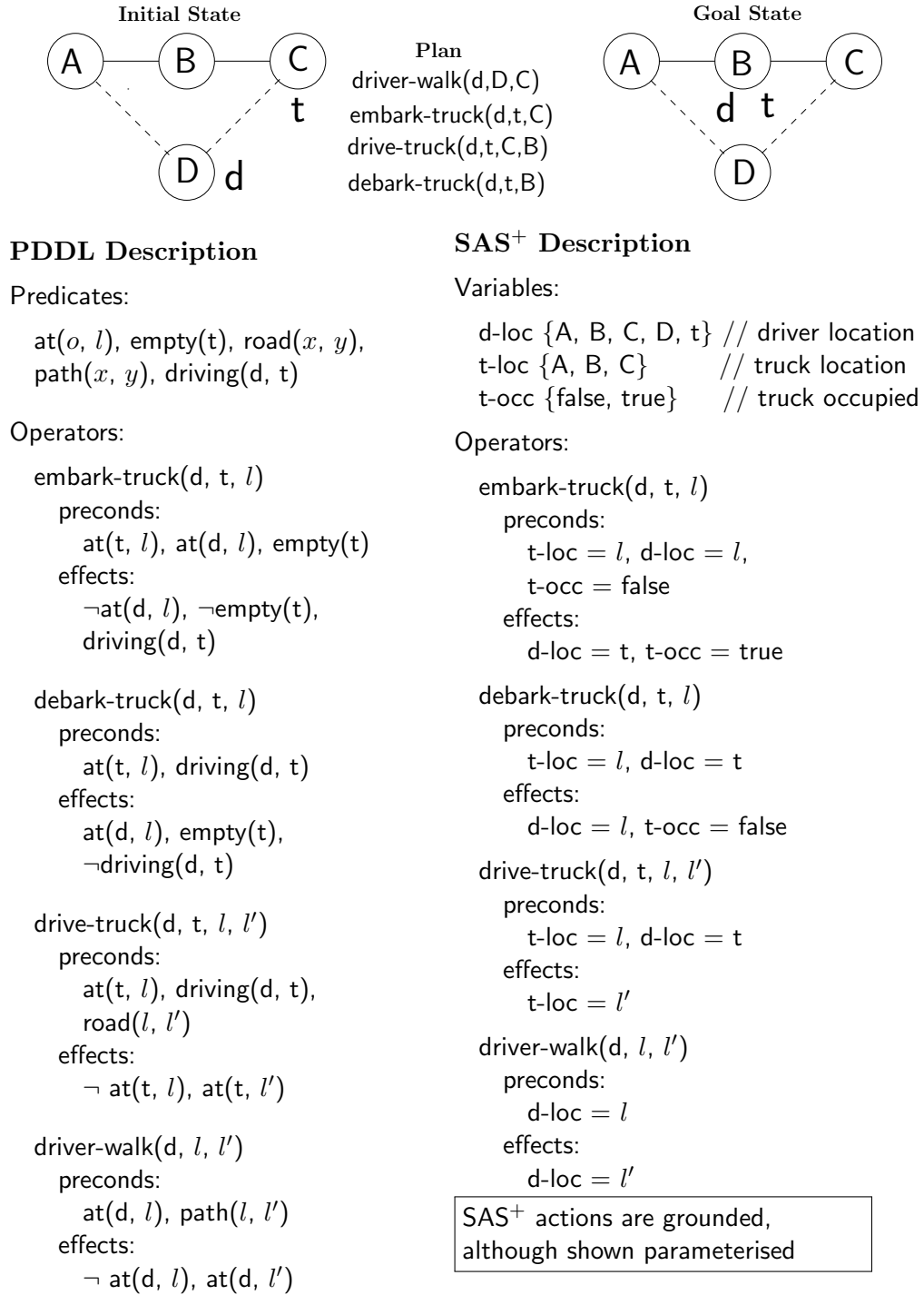


Figure 1: A driverlog problem instance: initial state (top-left), goal-state(top-right), a plan (top-middle), representation in PDDL (bottom-left) and SAS<sup>+</sup> (bottom-right).

(dotted lines) for drivers to walk. In Figure 1, there are four locations A, B, C, and D. In the initial state, the driver  $d$  is at location D and the truck  $t$  is at location C. In the goal state,

both the driver and the truck are at location B. To get to the goal state from the initial state, the driver needs to execute a plan that comprises walking to location C, embarking on the truck, driving it from location C to location B, and debarking from the truck.

The most common language used in describing planning problems is PDDL (Ghallab et al., 1998). In this paper, we use the STRIPS-style planning (Fikes & Nilsson, 1971) described by the PDDL version in (Fox & Long, 2003). The PDDL representation of the driverlog problem is shown in bottom-left of Figure 1. Below we provide our formal definition of a classical planning task. In this definition, we consider each *operator* in PDDL to be grounded or instantiated, thus we only reason about *actions*. Table 1 lists the actions in the driverlog domain described in Figure 1. Throughout the paper, these actions will be referred to by their identification numbers written before them.

Table 1: Actions i.e. the instantiated or grounded operators in the driverlog domain. The action identification numbers are used henceforth to refer to these actions.

1 embark-truck(d,t,A)	4 debark-truck(d,t,A)	7 drive-truck(d,t,A,B)	11 driver-walk(d,A,D)
2 embark-truck(d,t,B)	5 debark-truck(d,t,B)	8 drive-truck(d,t,B,A)	12 driver-walk(d,D,A)
3 embark-truck(d,t,C)	6 debark-truck(d,t,C)	9 drive-truck(d,t,B,C)	13 driver-walk(d,D,C)
		10 drive-truck(d,t,C,B)	14 driver-walk(d,C,D)

Now, we formally define the classical planning tasks, that are supported by our planner.

**Definition 1** (Classical Planning Task). A classical planning task  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$  comprises a set of logical atoms  $\mathcal{V}$ , an initial state  $\mathcal{I} \subseteq \mathcal{V}$ , a set of goals  $\mathcal{G} \subseteq \mathcal{V}$ , and a set of actions  $\mathcal{A}$ . Each action  $\alpha$  in  $\mathcal{A}$  is represented by  $\langle \text{pre}, \text{add}, \text{del} \rangle$ , where  $\text{pre} \subseteq \mathcal{V}$  is the set of preconditions of action  $\alpha$ ,  $\text{add} \subseteq \mathcal{V}$  is the add set of action  $\alpha$ , and  $\text{del} \subseteq \mathcal{V}$  is the delete set of action  $\alpha$ . Given the current state  $s \subseteq \mathcal{V}$ , an action  $\alpha$  is applicable on the state  $s$  if the preconditions are satisfied i.e.  $\text{pre} \subseteq s$  and resultant state  $s'$  produced by the application of  $\alpha$  on the state  $s$  is denoted by  $s' = \text{app}(s, \alpha) = (s \setminus \text{del}) \cup \text{add}$ .

An alternative to PDDL is to use the SAS<sup>+</sup> formalism (Bäckström & Nebel, 1995). The SAS<sup>+</sup> formalism has become popular after being used in Fast-Downward planner (Helmert, 2006). In the SAS<sup>+</sup> formalism, planning problems are represented by multi-valued state variables. Moreover, mutually exclusive predicates do not appear in the state description. Furthermore, extracting structural information such as DTGs (Helmert, 2006) is also straightforward. The SAS<sup>+</sup> representation of our driverlog example is also shown in Figure 1 (bottom-right). In this representation of the problem, we have two state variables d-loc and t-loc for locations of the driver and the truck respectively, and a state variable t-occ to denote whether the truck is occupied by a driver. The domain for each of the variables is shown in the figure as well. Note that operators are all instantiated or grounded in SAS<sup>+</sup> (i.e. they are actions) although in the figure, they are shown as parameterised. Since action prototypes in SAS<sup>+</sup> are the same as in PDDL, Table 1 lists the SAS<sup>+</sup> actions in the driverlog domain as well as the PDDL actions. We use a translator that comes with the Fast-Downward planner (Helmert, 2006) to translate a given PDDL planning task into a SAS<sup>+</sup> planning task. Below we provide a formal definition of a multi-valued planning task (Helmert, 2006).

**Definition 2** (Multi-Valued Planning Task). A multi-valued planning task  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$  comprises a set of multi-valued state variables  $\mathcal{V}$ , an initial state  $\mathcal{I}$ , a set of goals  $\mathcal{G}$ , and a set of actions  $\mathcal{A}$ . Each state variable  $v \in \mathcal{V}$  can be assigned a value  $k$  (denoted by  $v = k$ ) from its finite domain  $\mathcal{D}(v)$ . An assignment  $s$  is a set  $\{(v = k) : v \in \mathcal{V} \wedge k \in \mathcal{D}(v)\}$  denoting which variables are assigned which values; no variable is assigned twice in an assignment. We use  $v \Leftarrow s$  to denote  $v$  appears or is assigned a value in assignment  $s$ . We also use  $s(v)$  to denote the value assigned to  $v$  by  $s$  when  $v \Leftarrow s$ . A partial assignment assigns values to a subset of variables in  $\mathcal{V}$  while a complete assignment assigns a value to each variable. The initial state  $\mathcal{I}$  is a complete assignment while the set of goals  $\mathcal{G}$  is a partial assignment. Each action  $\alpha \in \mathcal{A}$  has two partial assignments  $p_\alpha$  and  $e_\alpha$  that respectively denote the preconditions and the effects of  $\alpha$ . Given the current state  $s$ , which is a complete assignment, an action  $\alpha$  is applicable on the state  $s$  if  $s(v) = p_\alpha(v)$  whenever  $v \Leftarrow p_\alpha$ . The resultant state  $s'$  produced by the application of  $\alpha$  on the state  $s$  is denoted by  $s' = \text{app}(s, \alpha)$  where  $s'(v) = e_\alpha(v)$  whenever  $v \Leftarrow e_\alpha$ ; otherwise  $s'(v) = s(v)$  when  $v \not\Leftarrow e_\alpha$ . For any action  $\alpha$ , if  $v \Leftarrow p_\alpha$  and  $v \Leftarrow e_\alpha$  then  $p_\alpha(v) \neq e_\alpha(v)$ , otherwise only  $v \Leftarrow p_\alpha$  should hold, not  $v \Leftarrow e_\alpha$ .

So far for both classical and multi-valued planning, we have provided the definitions of the preconditions of an action  $\alpha$  being applicable on a state  $s$ . For both types of planning, we also have provided the definitions of the effects of the action  $\alpha$  in terms of the computation of  $\text{app}(s, \alpha)$  to produce the resultant state  $s'$  from a given state  $s$  where the preconditions of  $\alpha$  hold. Below we partition the variables involved in an action based on whether they appear only in the preconditions or only in the effects or in both.

**Definition 3** (Appearance Partitioning). Given an action  $\alpha$ , we define the following mutually disjoint sets of variables that appear in the preconditions and/or effects of  $\alpha$ :

1.  $\text{PnotE}(\alpha) = \{v | v \Leftarrow p_\alpha \wedge v \not\Leftarrow e_\alpha\}$ : The set of variables that appear only in the preconditions  $p_\alpha$  but not in the effects  $e_\alpha$  of action  $\alpha$ .
2.  $\text{EnotP}(\alpha) = \{v | v \Leftarrow e_\alpha \wedge v \not\Leftarrow p_\alpha\}$ : The set of variables that appear only in the effects  $e_\alpha$  but not in the preconditions  $p_\alpha$  of action  $\alpha$ .
3.  $\text{PandE}(\alpha) : \{v | v \Leftarrow p_\alpha \wedge v \Leftarrow e_\alpha\}$ : The set of variables that appear both in the preconditions  $p_\alpha$  and in the effects  $e_\alpha$  of action  $\alpha$ .

Below we define a *sequential*, and a *parallel plan*. These definitions are the same as those of the sequential plan and  $\forall$ -Step parallel plan in (Rintanen, Heljanko, & Niemelä, 2006).

**Definition 4** (Action Sequence). The result of application of a sequence  $\langle \alpha_1, \dots, \alpha_n \rangle$  of  $n$  actions is defined by  $\text{app}(s, \langle \alpha_1, \dots, \alpha_n \rangle) = \text{app}(\text{app}(s, \langle \alpha_1, \dots, \alpha_{n-1} \rangle), \alpha_n)$  where as a basis  $\text{app}(s, \langle \alpha \rangle) = \text{app}(s, \alpha)$  and  $\text{app}(s, \alpha)$  is defined either in Definition 1 or in Definition 2.

**Definition 5** (Sequential Plan). A sequential plan  $\Sigma = \langle \alpha_1, \dots, \alpha_n \rangle$  of plan length  $n$  for a (classical or multi-valued) planning problem  $\mathcal{P}$  is a sequence of actions such that there is a sequence of states  $\langle s_0, s_1, \dots, s_n \rangle$  for which (i)  $s_0 = \mathcal{I}$ , (ii)  $\forall_{\tau \in [1, n]} \text{app}(s_{\tau-1}, \alpha_\tau) = s_\tau$ , and (iii)  $\mathcal{G} \subseteq s_n$  for a classical planning task  $\mathcal{P}$  or  $\mathcal{G}(v) = s_n(v)$  for each variable  $v \Leftarrow \mathcal{G}$  of a multi-valued planning task  $\mathcal{P}$ .

**Definition 6** (Action Serialisability). A set of actions  $A$  is serialisable if for each permutation  $\vec{A} = \langle \alpha_1, \dots, \alpha_{|A|} \rangle$  of the actions in  $A$ ,  $\text{app}(s, \vec{A})$  produces the same resulting state  $s'$  from a given state  $s$ . For a serialisable set of actions  $A$ , we define  $\text{app}(s, A) = \text{app}(s, \vec{A})$  where  $\vec{A}$  is an arbitrary permutation of the actions in  $A$ .

**Definition 7** (Parallel Plan). A parallel plan  $\Pi = \langle A^1, \dots, A^m \rangle$  of makespan  $m$  for a (classical or multi-valued) planning problem  $\mathcal{P}$  is a sequence of sets of serialisable actions  $A^\tau$  such that there exists a sequence of states  $\langle s_0, s_1, \dots, s_m \rangle$  for which (i)  $s_0 = \mathcal{I}$ , (ii) for all  $\tau \in [1, m]$   $\text{app}(s_{\tau-1}, A^\tau) = s_\tau$ , and (iii)  $\mathcal{G} \subseteq s_m$  for a classical planning task  $\mathcal{P}$  or  $\mathcal{G}(v) = s_m(v)$  for each variable  $v \Leftarrow \mathcal{G}$  of a multi-valued planning task  $\mathcal{P}$ . Assuming arbitrary permutations  $\langle \alpha_1^\tau, \dots, \alpha_{|A^\tau|}^\tau \rangle$  of the actions in  $A^\tau$ , for a parallel plan  $\Pi$ , we have a serialised plan  $\vec{\Pi} = \langle \alpha_1^1, \dots, \alpha_{|A^1|}^1, \dots, \alpha_1^\tau, \dots, \alpha_{|A^\tau|}^\tau, \alpha_1^{\tau+1}, \dots, \alpha_{|A^{\tau+1}|}^{\tau+1}, \dots, \alpha_1^m, \dots, \alpha_{|A^m|}^m \rangle$ . A serialised plan is thus a sequential plan of length  $n = |A^1| + \dots + |A^m|$ .

Henceforth, we discuss multi-valued planning tasks and parallel plans as defined above.

## 2.2 Domain Transition Graphs

From the SAS<sup>+</sup> representation of a planning problem, a domain transition graph (Helmert, 2006) can be extracted for every state variable to show how these variables can change their values. Below we provide the formal definition of a domain transition graph but only considering the multi-valued planning task formally described above.

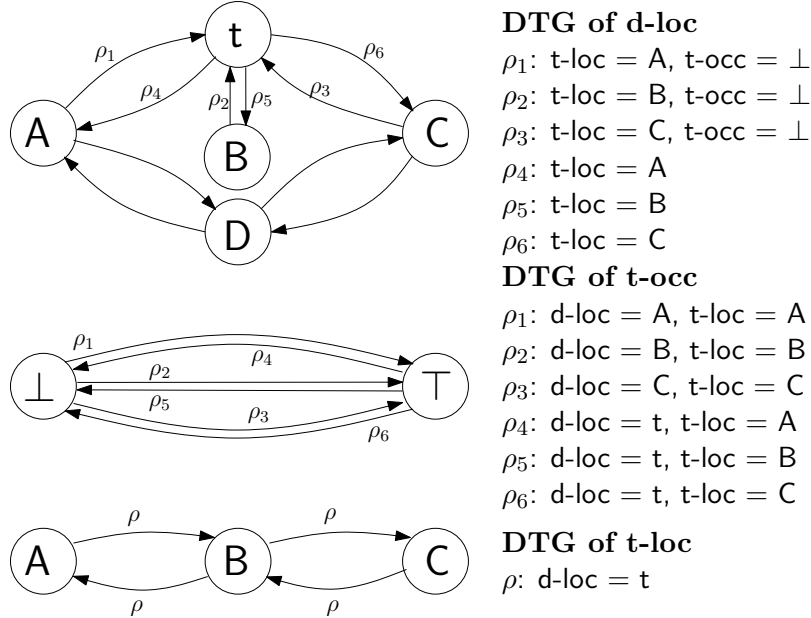


Figure 2: Domain transition graphs for the driverlog problem in Figure 1. The labels  $\rho_j$ s on the edges are the conditions listed at the right column of the respective DTG at the left column. Any occurrence of the transition needs the condition  $\rho_j$  written on it to hold before the transition. Boolean values are true  $\top$  and false  $\perp$ .

**Definition 8** (Domain Transition Graph). Assume  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$  be a multi-valued planning task. The domain transition graph  $\text{DTG}(v)$  of  $v \in \mathcal{V}$  is an edge-labelled directed graph with the vertexes  $\mathcal{D}(v)$  and the following edges:

1. For each action  $\alpha \in \mathcal{A}$ , there will be an edge  $\langle k, k' \rangle$  originated at  $k$  and terminated at  $k'$  with label  $\rho = p_\alpha \setminus \{(v = k)\}$ , if  $v \in \text{PandE}(\alpha)$ , and  $p_\alpha(v) = k$  and  $e_\alpha(v) = k'$ .
2. For each action  $\alpha \in \mathcal{A}$ , there will be an edge  $\langle k, k' \rangle$  originated at each  $k \in (\mathcal{D}(v) \setminus \{k'\})$  and terminated at  $k'$  with label  $\rho = p_\alpha$ , if  $v \in \text{EnotP}(\alpha)$  and  $e_\alpha(v) = k'$ .

DTGs for the example in Figure 1 are shown in Figure 2. In the DTG of variable **d-loc**, we have an edge from **A** to **t** labelled with  $\rho_1$  which is for action **embark-truck(d, t, A)**. This action changes the value of **d-loc** from **A** to **t**. Since variables **t-loc** and **t-occ** appear in the preconditions of this action, they all appear on the condition  $\rho_1$  of the corresponding edge meaning that to change the value of **d-loc**, these variables should have values **A** and  $\perp$  (false) respectively. Similarly, all edges in the same DTG and other DTGs could be explained.

### 2.3 Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) has a number of *variables* each having its *domain* of *values*. One has to find one value for each variable from its domain such that a given number of *constraints* are all satisfied. To illustrate a CSP problem, consider 3 variables  $x_1, x_2$ , and  $x_3$  that all take their values from domain  $\{0, 1, 2\}$ . We have three constraints:  $x_1 + x_2 + x_3 < 3$ ,  $x_1 \leq x_2$  and  $x_2 + 2x_3 \geq 3$ . Clearly, values 0, 1, 1 when assigned to variables  $x_1, x_2, x_3$  respectively solve the problem. Below we formally define CSPs.

**Definition 9** (Constraint Satisfaction Problem). A constraint satisfaction problem  $P = \langle X, C \rangle$  comprises a set of variables  $X$  and a set of constraints  $C$ . Each variable  $x \in X$  can be assigned a value  $\nu$  (denoted by  $x = \nu$ ) from its finite domain  $D(x)$ . An assignment  $\sigma$  is a set  $\{(x = \nu) : x \in X \wedge \nu \in D(x)\}$  denoting which variables are assigned which values; no variable is assigned twice in an assignment. A partial assignment assigns values to a subset of variables in  $X$  while a complete assignment assigns a value to each variable. Each constraint  $c \in C$  is defined on a set  $X(c) \subseteq X$  of variables called the *scope* of  $c$  and specifies the combinations of allowed values of the variables in  $X(c)$ . A solution to a given  $P$  is a complete assignment such that all the constraints in  $C$  are satisfied. If there exists at least one solution for a given  $P$ , we say  $P$  is *satisfiable*, otherwise it is *unsatisfiable*.

In our constraint model for planning, we use an extensional constraint representation called (*positive*) *table constraints*. The table constraints for the CSP described above are shown in Figure 3. Look at the table for the constraint  $x_1 + x_2 + x_3 < 3$ . Each row of the table is an assignment of values to the variables  $x_1, x_2$ , and  $x_3$  from the domains of the respective variables such that the inequality constraint is satisfied. Similar statements hold for other constraints and their corresponding table constraints. As we see later, due to various *consistency checking* and related *constraint propagation*, certain values could be deleted from a variable's domain because those values do not lead to a solution. Consequently, certain rows are also deleted from the positive table constraints. Besides positive table constraints, we also use *negative table constraints* in our constraint modelling for planning. The columns of a negative table constraint are the variables in  $X(c)$  of a constraint  $c$  and the



rows list the assignment of values to the variables such that the constraint is not satisfied. While one uses some rows from a positive table to satisfy the constraint, one takes into account all rows of a negative table to avoid the constraint being not satisfied.

An example CSP:

$$x_1 + x_2 + x_3 < 3$$

$$x_1 \leq x_2$$

$$x_2 + 2x_3 \geq 3$$

$x_1$	$x_2$	$x_3$
0	0	0
0	0	1
0	0	2
0	1	0
0	1	1
0	2	0
1	0	0
1	0	1
1	1	0
2	0	0

$x_1$	$x_2$
0	0
0	1
0	2
1	1
1	2
2	2

$x_2$	$x_3$
0	2
1	1
1	2
2	1
2	2

Figure 3: Table constraints for a CSP example

**Definition 10** (Table Constraints). *Given a constraint  $c \in C$ , a (positive) table constraint  $T$  in its rows lists the allowed combinations of values of the variables in  $X(c)$  while a negative table constraint  $\tilde{T}$  lists the forbidden combinations. The tables have a column for each variable in  $X(c)$ . If  $t$  is a row in  $T$ , then  $t[x]$  denotes the value of variable  $x$  in that row. Each row  $t$  in a positive table is called a support for value  $t[x]$  of the variable  $x$ .*

We briefly explain *generalised arc consistency* for table constraints. In Figure 3, consider the table constraint for  $x_1 + x_2 + x_3 < 3$ . For every variable in this constraint and for every value in the variable's domain, we have a row in the table. For example, rows 1–6 are the supports for  $x_1 = 0$ , rows 7–9 are the supports for  $x_1 = 1$ , and row 10 is the support for  $x_1 = 2$ . Also, rows 1–3 and rows 7, 8, 10 are the supports for the  $x_2 = 0$ , rows 4, 5, 9 are the supports for the  $x_2 = 1$  and row 6 is the support for the  $x_2 = 2$ . Similarly, we have supports for all values of variable  $x_3$ . Because of this property,  $x_1 + x_2 + x_3 < 3$  is called generalised arc consistent. We see that constraint  $x_1 \leq x_2$  is also generalised arc-consistent but this is not the case for constraint  $x_2 + 2x_3 \geq 3$ . In constraint  $x_2 + 2x_3 \geq 3$ , we do not have any support for  $x_3 = 0$  because we do not have any row having value 0 for variable  $x_3$ . If we maintain arc-consistency, we should delete this value from the domain of variable  $x_3$ . The current domain of variable  $x_3$  is now  $\{1, 2\}$ . We need to delete all rows of other tables having value 0 for variable  $x_3$ . New tables are represented in Figure 4.

The new tables still are not arc-consistent and we need to follow the same procedure. Looking at the first table, we see that there are no supports for  $x_1 = 2$  and  $x_2 = 2$ . Therefore, value 2 should be deleted from the domains of variables  $x_1$  and  $x_2$  and consequently all tables should be updated. The final tables are shown in figure 5. Note that the domains of the variables  $x_1$ ,  $x_2$ , and  $x_3$  are now  $\{0, 1\}$ ,  $\{0, 1\}$  and  $\{1, 2\}$  respectively and these tables are all generalised arc-consistent with respect to the new domains.

We define generalised and singleton arc consistency for table constraints below.

$x_1 + x_2 + x_3 < 3$ $x_1 \leq x_2$ $x_2 + 2x_3 \geq 3$	$x_1 + x_2 + x_3 < 3$			$x_1 \leq x_2$		$x_2 + 2x_3 \geq 3$	
	$x_1$	$x_2$	$x_3$	$x_1$	$x_2$	$x_2$	$x_3$
	0	0	1	0	0	0	2
	0	0	2	0	1	1	1
	0	1	1	1	1	1	2
	1	0	1	1	2	2	1
				2	2	2	2

Figure 4: Table constraints after first stage of maintaining generalised arc-consistency

$x_1 + x_2 + x_3 < 3$ $x_1 \leq x_2$ $x_2 + 2x_3 \geq 3$	$x_1 + x_2 + x_3 < 3$			$x_1 \leq x_2$		$x_2 + 2x_3 \geq 3$	
	$x_1$	$x_2$	$x_3$	$x_1$	$x_2$	$x_2$	$x_3$
	0	0	1	0	0	0	2
	0	0	2	0	1	1	1
	0	1	1	1	1	1	2
	1	0	1				

Figure 5: Final table constraints after maintaining generalised arc-consistency

**Definition 11** (Arc-Consistency). *Value  $\nu$  for variable  $x$ , denoted by  $x = \nu$ , is generalised arc-consistent if for every constraint  $c$  such that  $x \in X(c)$ , there exists a row  $t$  with  $t[x] = \nu$  in the table constraint for  $c$ . Any value  $\nu$  that is not generalised arc-consistent could be removed from  $x$ 's domain to obtain its current domain. This removes rows  $t$  with  $t[x] = \nu$  from tables and in a cascaded fashion, could remove values of other variables in  $t$  from their domains. A table constraint is generalised arc consistent if for every variable  $x \in X(c)$  and for every value  $\nu$  in  $x$ 's current domain, there exists a support for  $x = \nu$ , that is there is a row  $t$  with  $t[x] = \nu$ . A CSP is generalised arc-consistent if all of its constraints are generalised arc-consistent. Value  $\nu$  for variable  $x$ , denoted by  $x = \nu$ , is singleton arc-consistent if after assigning the value  $\nu$  to variable  $x$ , the CSP can be made generalised arc-consistent. A CSP is singleton arc-consistent if for each variable  $x$  and for each value  $\nu$  in its current domain,  $x = \nu$  is singleton arc-consistent (Bessiere, 2006).*

We briefly explain singleton arc-consistency. In our example in Figure 5,  $x_1 = 1$  is not singleton arc-consistent because if we set value 1 to the variable  $x_1$ , our first constraint imposes that variable  $x_2$  should take value 0. This is because the last row of the table is the only row with value 1 for variable  $x_1$  and in this row, we have  $x_2 = 0$ . However, with this assignment, we will not have any support in the middle table in Figure 5 meaning the constraint  $x_1 \leq x_2$  could not be satisfied if  $x_1 = 1$  and  $x_2 = 0$ . So to maintain singleton arc-consistency, we need to remove value 1 from the domain of variable  $x_1$  and continue this process until for each  $x$  and for each  $\nu$ ,  $x = \nu$  is singleton arc-consistent.

Many algorithms have been developed to maintain generalised arc-consistency during search. These algorithms prune domains and reduce the branching factor of the search. One of the algorithms for maintaining generalised arc-consistency is STR2+, optimised Simple Tabular Reduction algorithm (Lecoutre, 2011). This algorithm dynamically maintains the

table of supports while applying the generalised arc-consistency. Since the table constraints we use in our model contain *don't care* values, we use an extension of this algorithm named Short-STR2 (Jefferson & Nightingale, 2013). In short-STR2, there can be short supports in the tables meaning that some values of the variables in a row can be missing.

## 2.4 Constraint-Based Planners

One of the approaches developed for planning is based on translation of the given planning problem into a different formalism such as satisfiability (SAT) or CSP and then solving it using respective solvers. One key issue with this approach is that to be able to transform the problem to SAT or CSP, we need to know the makespan in advance. Since in planning the makespan is not known beforehand, a fixed bound  $n$  is therefore imposed on the makespan (also called horizon) and the problem of finding a plan of makespan  $n$  is translated to a SAT/CSP problem. If the translated problem does not have any solution, the bound is increased and this process continues until a solution is found. Then, the plan is extracted from the solution to the translated SAT/CSP problem.

CSP-based planners can be categorised into two groups depending on the type of the plan they generate: those generating sequential plans (Barták & Toropila, 2008; Gregory et al., 2010; Judge & Long, 2011) and those generating partial order (Vidal, 2004) and parallel plans (Barták, 2011a, 2011b; Do & Kambhampati, 2001; Lopez & Bacchus, 2003). In sequential plans, each time only one action can take place while in parallel plans several actions can take place simultaneously if they don't conflict with each other. In partial-order planning, only a partial-order is defined between actions. A sequential plan can be generated by totally ordering the actions in partial-order plans or parallel plans. In partial-order and parallel planning, symmetry checking is avoided while this is not the case in sequential planning. By performing symmetry checking, we try to find the plans that remain the same after changing the order of their actions. This process is very time-consuming. Recent CSP-based planners compute parallel plans.

CSP techniques were applied in temporal planning in (Ghallab & Laruelle, 1994). Later, manually designed constraint models were used in classical planning (van Beek & Chen, 1999). While this planner is a sequential planner, the other planners (Do & Kambhampati, 2001; Lopez & Bacchus, 2003) transform the planning graphs (Blum & Furst, 1995) into CSPs such that they can deal with parallel plans. In the dynamic CSP model in (Do & Kambhampati, 2001), variables are used to represent the propositions at each layer of the planning graph. The domain of these variables are the actions supporting these propositions. To encode the relationships between propositions, action *mutex*, fact *mutex* and subgoal *mutex* constraints are used. The constraint model in (Lopez & Bacchus, 2003) tries to transform the planning graph in a different manner by using variables to represent the facts and actions at each layer. Transitions are logical formulas between variables that encode the initial state, the goals, preconditions and effects, and frame axioms. Most of the above mentioned planners use Boolean variables to encode the planning problem as CSP. Another constraint model that uses logical formulas with different types of constraints is proposed in (Ghallab, Nau, & Traverso, 2004).

Using a multi-valued representation of a planning task, one normally has fewer variables with larger domains where domain filtering normally pays off. Based on this representation,

the planner in (Barták & Toropila, 2008) reformulates the constraint models and summarises the set of logical formulas from original models in table constraints that extensionally list the valid tuples for the constraints. By exploiting table constraints, more inconsistencies are filtered out than could be done by using the original models and the time needed for constraint propagation is also reduced significantly. Later, these ideas are improved by using lifting, symmetry breaking, singleton arc consistency and **nogood** learning techniques (Barták & Toropila, 2009a, 2009b). Also in (Gregory et al., 2010), dominance constraints are used for further inference and in (Judge & Long, 2011), a goal-centric heuristic is proposed for variable/value selection to guide the search towards a solution.

Similar to domain transition graphs (Helmert, 2006), which are based on the multi-valued representation of planning domains, in (Barták, 2011a), an automaton is considered for each state variable. By synchronising the state transitions in all automata, the CSP model supports parallel planning. In the proposed model in (Barták, 2011a), CSP variables are considered for state variables and actions at each time step and constraints are used for encoding the edges in each automaton and also for synchronising the transitions of different automata. A slightly different encoding of automata is proposed in (Barták, 2011b) in which, rather than encoding state variables and actions responsible of changing them, only the actions are encoded as CSP variables and state variables are omitted completely.

To summarise the state of the art of constraint-based planning, planner PaP2 (Barták, 2011b) is faster than PaP1 and SeP (Barták, 2011a). SeP performs better than GP-CSP (Do & Kambhampati, 2001) and CSP-plan (Lopez & Bacchus, 2003). Constance (Gregory et al., 2010) is a sequential planner that performs better than CPT (Vidal & Geffner, 2006) and SeP (Barták & Toropila, 2008) but is much worse than PaP2 (Ghooshchi et al., 2015).

### 3. Our Planner Architecture

Based on the DTGs extracted from the SAS<sup>+</sup> representation of the planning problem, we have developed a new constraint model for parallel planning. In our model, we directly encode the DTGs of the problem into table constraints and use a general-purpose CSP solver to solve the corresponding constraint satisfaction problem. In contrast to the state of the art constraint-based planners, we do not have any CSP-variables for actions taking place at each time and we can extract the final plan by taking into account the transitions on the DTGs of state variables. The overall approach of our planner is shown in Figure 6.

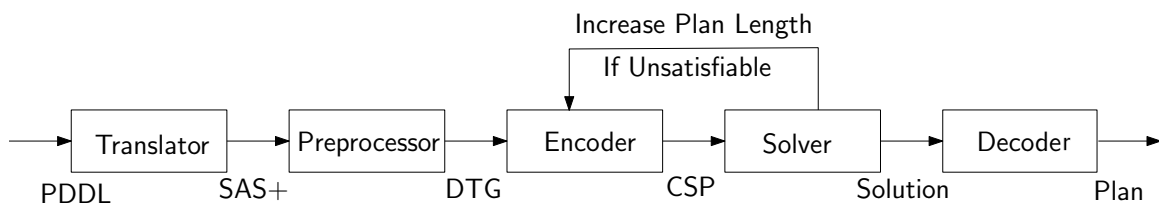


Figure 6: Architecture of our constraint-based planner

**Translation.** The input to the translator part is the PDDL representation of the given classical planning problem. PDDL is the standard language to describe planning problems

and is used as a common language in the international planning competitions. Our planner supports the STRIPS-style classical planning described by the PDDL version in (Fox & Long, 2003). We use the translator used in (Helmert, 2006) to translate the PDDL description into SAS<sup>+</sup> formalism with multi-valued variables and corresponding instantiated actions. The classical and multi-valued planning tasks are formally defined in Definitions 1 and 2. Nevertheless, the PDDL to SAS<sup>+</sup> translator, along with the SAS<sup>+</sup> representation, also produces the **mutex-groups**. The **mutex groups** are sets of variable-value pairs that cannot occur at the same time. In our driverlog example, we have a **mutex-group**  $\{d\text{-loc} = t, t\text{-occ} = \perp\}$ , which denotes that when the variable *d-loc* takes the value *t*, the variable *t-occ* can not take the value false and vice versa. A **mutex-group** that involves only one variable could actually be ignored because such a mutual exclusion condition is already captured by the characteristics of a multi-valued variable. Henceforth, by **mutex groups**, we will mean **mutex-groups** that involve more than one variable.

**Preprocessing.** After translation, the multi-valued representation of the planning problem is preprocessed to extract the DTGs for all state variables. This is done by using our own extractor program which is similar to the one in Fast-Downward planner (Helmert, 2006). In our DTGs, we use *don't cares*  $\times$  or wildcards to denote a value that actually could be any value in the domain of the respective variable. Moreover, our DTGs also have self-loops as edges to denote *no-ops*. Further details on these are described later.

**Encoding.** In this step, imposing a fixed bound on the makespan, our DTGs are used to encode the problem of finding a plan of makespan  $n$  as a CSP problem. A CSP solver is then used to solve the encoded problem and if it is not successful in solving it, the makespan is increased and the same process is repeated until a plan is found or a given time limit exceeds. To find the first makespan to start with, for those state variables that occur in the goals, we look in their DTGs for the shortest path from their values in the initial state to their values in the given goals. We select the maximum path length as the first makespan to start with. Further details on our encoding are explained later in the paper.

**Solving.** As mentioned before, we use the CSP solver named Minion (Gent et al., 2006) to solve the CSP problem generated by the encoder. Minion is a general-purpose constraint solver that supports table constraints with *don't care* values. The solver also has a special constraint propagation technique named shortSTR2 for this kind of tables. Minion has a complete search algorithm that guarantees finding a solution if it exists. This is critical for our approach because before increasing the makespan, we need to be sure that no plan with a smaller makespan exists and therefore we have to try a larger makespan.

**Decoding.** When the solver finds a solution for the encoded CSP, we can extract the final plan from the values of the CSP variables. To do this, we need to check the values of the state variables at times  $\tau$  and  $\tau + 1$ . If a state variable changes its value, the actions responsible for this change can be extracted from the edges of the corresponding DTG. For example, if the value of a state variable  $v$  changes from  $k$  to  $k'$ , we need to look at the edges from vertex  $k$  or *don't-care* to vertex  $k'$  in the DTG of the variable  $v$ . If the value of a state variable is not changed, still we need to check the edges from *don't-care* to its value at time  $\tau$ . There may be more than one edge with different actions on them, but we need the actions with preconditions satisfied at time  $\tau$  and effects matching at time  $\tau + 1$ . Since

there may be more than one action at each time step of a plan, our planner allows parallel plans. Later, we will describe the decoding procedure further.

#### 4. DTG-Based Planning

To formulate our constraint modelling for planning, we redefine the DTGs. The redefined DTGs are similar to the FSA in (Barták, 2011a) in that both have loops for *no-ops*. However, unlike FSA, our redefined DTGs could have a vertex  $\times$  that represents *don't care* value for the respective state variable. This *don't care* vertex is also new in our DTGs when compared to the original DTGs (Helmert, 2006). For an action  $\alpha$ , if  $v \in \text{EnotP}(\alpha)$  then in the original DTGs, there is an edge from each of the other vertex  $k \neq k'$  to the vertex  $k'$  where  $k' = e_\alpha(v)$ . In our redefined DTGs, we simplify this by using a vertex with value  $\times$  to denote a *don't care* value and draw an edge from this  $\times$  vertex to the vertex  $k'$ . Moreover, our labelling of the edges are conceptually more detailed than that in the original DTGs. In the label, we show the action, the other preconditions and other effects as well. For an efficient implementation, the information in the labels is however extracted from the actions as needed. In the FSA, the edge labels are just the respective actions.

**Definition 12** (Domain Transition Graph Redefined). *Assume  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$  be a multi-valued planning task. The redefined domain transition graph  $\text{DTG}(v)$  of  $v \in \mathcal{V}$  is an edge-labelled directed graph with the vertexes  $\mathcal{D}(v) \cup \{\times\}$ , where  $\times$  denotes a don't care or wild card meaning it represents any value in  $v$ 's domain. The edges that are in the redefined  $\text{DTG}(v)$  are described below. Note that if the degree of the don't care vertex  $\times$  is zero after all the following edges are created, for the efficiency of implementation, the don't care vertex, deeming unnecessary, is removed from the redefined DTG.*

1. *For each action  $\alpha \in \mathcal{A}$ , there is an edge  $\langle p_\alpha(v), e_\alpha(v) \rangle$  from  $p_\alpha(v)$  to  $e_\alpha(v)$  with label  $\langle \alpha, \rho, \epsilon \rangle$ , if  $v \in \text{PandE}(\alpha)$ . Here,  $\rho = (p_\alpha \setminus \{(v = p_\alpha(v))\})$  and  $\epsilon = (e_\alpha \setminus \{(v = e_\alpha(v))\})$ .*
2. *For each action  $\alpha \in \mathcal{A}$ , there is an edge  $\langle \times, e_\alpha(v) \rangle$  from don't care vertex  $\times$  to vertex  $e_\alpha(v)$  with label  $\langle \alpha, \rho, \epsilon \rangle$ , if  $v \in \text{EnotP}(\alpha)$ . Here,  $\rho = p_\alpha$  and  $\epsilon = (e_\alpha \setminus \{(v = e_\alpha(v))\})$ .*
3. *For each value  $k \in \mathcal{D}(v)$ , there will be a loop  $\langle k, k \rangle$  from vertex  $k$  to vertex  $k$  to represent a no-op with label  $\langle \alpha, \phi, \phi \rangle$  where for each loop,  $\alpha$  is a distinct no-op action with preconditions  $\{(v = k)\}$  and no effects, and  $\phi$  is an empty assignment.*

The original and the redefined domain transition graphs for variable **t-occ** in the driverlog problem in Figure 1 are shown in Figure 7. Action numbers 4, 5, 6 **debark-trucks** in Table 1 have an effect **t-occ** = **false**, but no precondition on **t-occ**. So in our redefined DTG, there is an edge from  $\times$  to  $\perp$  for each of the **debark-truck** actions at locations A, B, C. This is just an example to show how the *don't care* is represented in our DTGs. Since **t-occ** is just a boolean variable, the usefulness of *don't care* might not yet be clear. We need an actual multi-valued variable to see clearer distinctions. In Figure 8, we show the redefined DTGs for variables **d-loc** and **t-loc**, which we will use later to encode our table constraints.

Algorithm 1 describes the procedure to construct the redefined DTGs in Definition 12. For each variable  $v \in \mathcal{V}$ , we have a DTG with vertexes for each  $k \in \mathcal{D}(v) \cup \{\times\}$ . Then, for each action  $\alpha \in \mathcal{A}$ , we look at the preconditions and effects of  $\alpha$ . If a variable  $v$  appears

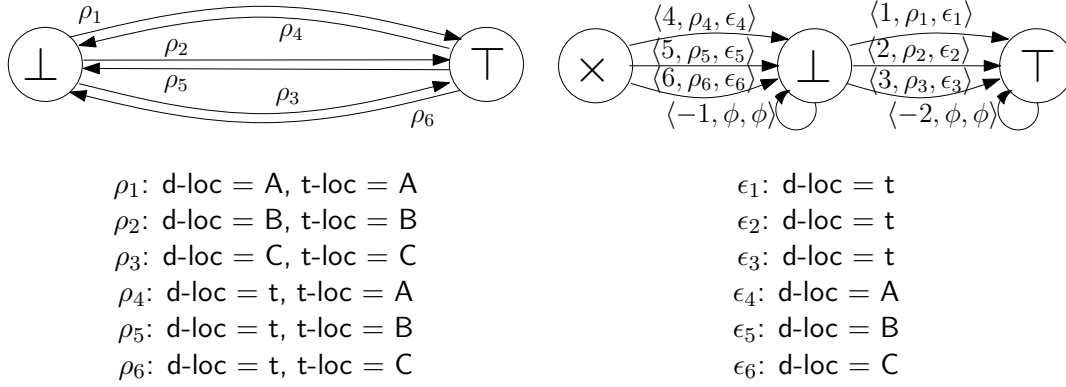


Figure 7: Original and redefined domain transition graphs for variable t-occ in the driverlog problem in Figure 1. Top-Left: original DTG as per Definition 8 and (Helmert, 2006); Top-Right: our redefined DTG. Positive numbers on the edges in our DTG denote the action identification numbers in Table 1. The  $\rho_j$ s on the edges of both DTGs are the conditions on other variables to hold before the transitions, and the  $\epsilon_j$ s on the edges in our DTG are the effects of the transitions on other variables. Boolean values: true is  $\top$  and false is  $\perp$ ; and *don't care* is  $\times$ . Loops at nodes are *no-ops* and each one is given a distinct negative identification number.

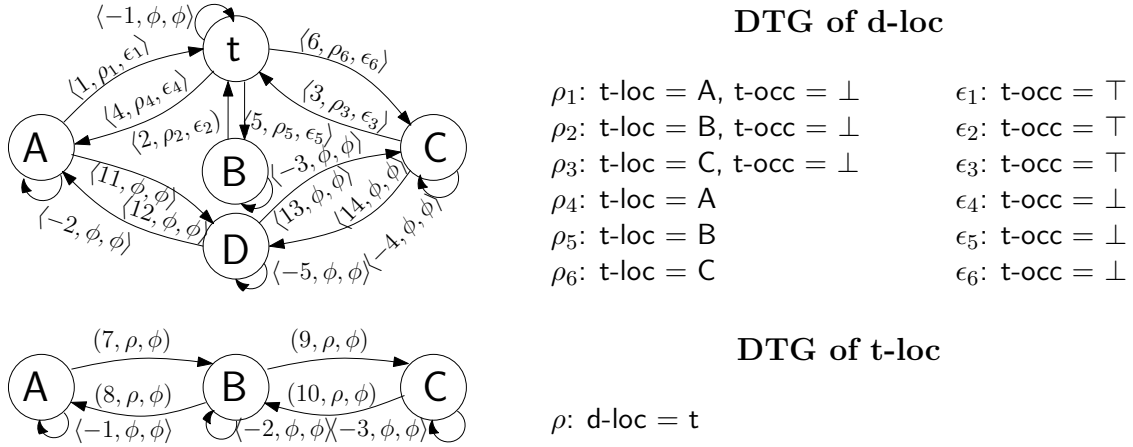


Figure 8: Redefined domain transition graphs for variables d-loc and t-loc of the driverlog problem in Figure 1. The  $\rho_j$ s in the labels of the edges are the conditions listed at the right column of the respective DTGs at the left column. Any occurrence of the transition needs the respective condition on other variables to hold before the transition. The  $\epsilon_j$ s in the labels are the effects of the transitions on other variables. Boolean values are true  $\top$  and false  $\perp$ ; and *don't care* is  $\times$ .

in both preconditions and effects of  $\alpha$ , then we add an edge  $\langle p_\alpha(v), e_\alpha(v) \rangle$  to  $\text{DTG}(v)$ . If a variable  $v$  appears only in the effects but not in the preconditions, then we add an edge  $\langle \times, e_\alpha(v) \rangle$  to  $\text{DTG}(v)$ . There is no edge for the variables that appear only in the

**Algorithm 1** Construction of Redefined DTGs

---

<i>//Add vertexes to DTGs</i>	<i>//Add edges to DTGs</i>	<i>//Delete useless don't cares</i>
<b>foreach</b> variable $v \in \mathcal{V}$	<b>foreach</b> action $\alpha \in \mathcal{A}$	<b>foreach</b> variable $v \in \mathcal{V}$
<b>foreach</b> value $k \in \mathcal{D}(v)$	<b>foreach</b> variable $v \in \text{PandE}(\alpha)$	<b>if</b> no edge from vertex $\times$
add vertex $k$ to $\text{DTG}(v)$	add edge $\langle p_\alpha(v), e_\alpha(v) \rangle$ to $\text{DTG}(v)$	remove $\times$ from $\text{DTG}(v)$
add vertex $\times$ to $\text{DTG}(v)$	<b>foreach</b> variable $v \in \text{EnotP}(\alpha)$	
	add edge $\langle \times, e_\alpha(v) \rangle$ to $\text{DTG}(v)$	

---

preconditions. The edge labels could be computed as defined in Definition 12, but we compute them later whenever needed. The *don't care* vertex is removed from a DTG if it has no incident edges.

**Lemma 1** (DTG Computation). *Computation of the DTGs as per Algorithm 1 requires  $O(|\mathcal{V}|(\widehat{\mathcal{D}}+1)+|\mathcal{A}|\widehat{V}_a+|\mathcal{V}|)$  time and  $O(|\mathcal{V}|(\widehat{\mathcal{D}}+1)+|\mathcal{A}|\widehat{V}_a)$  memory, where  $\widehat{\mathcal{D}}$  is the maximum domain size of a variable in  $\mathcal{V}$  and  $\widehat{V}_a$  is the maximum number of variables that appear in the preconditions and effects of an action (i.e. the maximum parameter size of an operator).*

*Proof.* The proof is straightforward from the pseudocode in Algorithm 1. Adding vertexes needs  $O(|\mathcal{V}|(\widehat{\mathcal{D}}+1))$  time and space, adding edges needs  $O(|\mathcal{A}|\widehat{V}_a)$  time and space and deleting useless *don't care* vertexes needs  $O(|\mathcal{V}|)$  time.  $\square$

**Algorithm 2** A Non-Deterministic Search for Parallel Planning

---

<b>func performSearch</b> ( $\mathcal{P}, m$ ) <b>returns</b> $\Pi$	<b>func selectParallelActions</b> ( $\mathcal{V}$ )
$\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ : multi-valued plan prob	$E$ : set of edges comprising at least one from
$m$ : makespan level to search the plan for	each $\text{DTG}(v)$ . Edges chosen from the same
<b>foreach</b> $v \in \mathcal{V}$ , $\text{curr}(v) = \mathcal{I}(v)$ <i>//curr val</i>	$\text{DTG}(v)$ must originate from vertex $\text{curr}(v)$
$\Pi = \langle A^1, \dots, A^m \rangle$ <i>//initially all <math>A^\tau = \phi</math></i>	or from the <i>don't care</i> vertex $\times$ (if any),
<b>foreach</b> time step $\tau = 1$ to $m$ <b>step</b> 1	terminate in the same vertex, and satisfy
$A^\tau = \text{selectParallelActions}(\mathcal{V})$	conditions $\rho$ on the edge labels $\langle \alpha, \rho, \epsilon \rangle$
<b>if</b> $A^\tau = \phi$ <b>then return failure</b>	<b>if</b> no two edges in $E$ have actions $\alpha, \alpha'$ in
<b>foreach</b> action $\alpha \in A^\tau$ <i>//update vars</i>	their labels such that $\alpha, \alpha'$ have any
<b>if</b> $v \Leftarrow e_\alpha$ <b>then</b> $\text{curr}(v) = e_\alpha(v)$	<b>parallel conflicts</b> in their preconditions
<b>if</b> $\exists v \Leftarrow \mathcal{G} \text{curr}(v) \neq \mathcal{G}(v)$ , <b>return failure</b>	or in effects <i>//E</i> be constructed carefully
<b>else return success and parallel plan</b> $\Pi$	<b>return</b> the set of actions in the edges in $E$
	<b>else return</b> an empty set of actions

---

In Algorithm 2, we show a non-deterministic search algorithm that finds a parallel plan of a given makespan  $m$ . This algorithm just conceptually shows how a search for planning can be performed on the DTG-based planning formulation. In Function `performSearch`, each state variable is first assigned a value specified in the initial state of the planning problem. This actually represents the initial state in the DTG of the variable. Initially, the parallel plan achieved so far has no actions in any time step. Next, in a loop for each time step, we call Function `selectParallelActions` to non-deterministically select a set of actions that will run in parallel. In Function `selectParallelActions`, at least one edge is selected from  $\text{DTG}(v)$  for each  $v \in \mathcal{V}$ . Edges chosen from the same  $\text{DTG}(v)$  must originate from vertex  $\text{curr}(v)$  or



from the *don't care* vertex  $\times$  (if any), terminate in the same vertex to produce a valid state within the DTG, and satisfy conditions  $\rho$  on the edge label  $\langle \alpha, \rho, \epsilon \rangle$ . Each such selected edge gives an action but these actions must not have any *parallel conflicts* between each other in terms of the semantic of the parallel plan defined in Definition 7. Exactly when two actions can be in parallel conflict is discussed below in details. After a set of parallel actions is selected, based on the destination vertex of the related edges, the current state of the DTGs are updated in Function `performSearch`. At the end of time step  $\tau$ , if the goal conditions are met, we have a parallel plan; otherwise the search is said to have failed. Since selected actions could be *no-ops*, we might have to omit them from the final plan returned (if any). Notice that Function `selectParallelActions` at any time step can select a set of parallel actions in many different ways, a deterministic implementation of Algorithm 2 would need to explore all these possible branches before deciding search failure.

Since in a parallel plan, several actions can take place at the same time step, for plan validity, we need to ensure that no two actions take place in parallel where the resultant state is invalid. In a valid state, each state variable can have only one value. Two actions have *parallel conflicts* if there is a shared variable in their preconditions and effects, and the values are conflicting. The formal definition of parallel conflicts of two actions is given below. The definition uses the three disjoint partitions  $\text{PnotE}(\alpha)$ ,  $\text{EnotP}(\alpha)$ , and  $\text{PandE}(\alpha)$  of the variables based on whether they appear only in the preconditions, or only in the effects, or in both of the preconditions and effects of an action  $\alpha$ . Considering these partitions, we can have nine different combinations for two actions that share a variable. Because of possible symmetries, these nine combinations could however be covered by only six cases. Note that actions not sharing a variable have no parallel conflicts between them.

**Definition 13** (Parallel Conflict). *Two actions  $\alpha \neq \alpha'$  have a parallel conflict, denoted by  $\alpha \otimes \alpha'$  if and only if any of the following six conditions hold.*

1.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PnotE}(\alpha')] \text{ such that } p_\alpha(v) \neq p_{\alpha'}(v).$
2.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{EnotP}(\alpha')] \text{ such that } p_\alpha(v) \neq e_{\alpha'}(v) \text{ or } \exists v \in [\text{EnotP}(\alpha) \cap \text{PnotE}(\alpha')] \text{ such that } e_\alpha(v) \neq p_{\alpha'}(v).$
3.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PandE}(\alpha')] \text{ or } \exists v \in [\text{PandE}(\alpha) \cap \text{PnotE}(\alpha')].$
4.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{EnotP}(\alpha')] \text{ such that } e_\alpha(v) \neq e_{\alpha'}(v).$
5.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{PandE}(\alpha')] \text{ or } \exists v \in [\text{PandE}(\alpha) \cap \text{EnotP}(\alpha')].$
6.  $\exists v \in [\text{PandE}(\alpha) \cap \text{PandE}(\alpha')].$

The following lemma establishes the connection between serialisability of a set of actions and the existence of a parallel conflict between any two actions in the set.

**Lemma 2** (Serialisability Conditions). *A set of actions  $A$  is serialisable meaning each permutation  $\vec{A} = \langle \alpha_1, \dots, \alpha_n \rangle$  of the actions in  $A$  produces the same state  $s'$  from a given state  $s$  if and only if the following two conditions hold:*

1. *Each action  $\alpha \in A$  is applicable on state  $s$ .*

2. No two actions  $\alpha, \alpha'$  exist in  $A$  where  $\alpha \otimes \alpha'$

*Proof.* For the only if part, assume  $A$  is serialisable. As per Definition 6,  $s' = \text{app}(s, A) = \text{app}(s, \vec{A})$  for any arbitrary permutation  $\vec{A}$ . This means any action  $\alpha \in A$  is applicable on  $s$ , since any action can be the first action of a permutation. This proves the first assertion. To prove the second assertion by contradiction, assume  $\alpha, \alpha' \in A$  are two actions such that  $\alpha \otimes \alpha'$ . For this, we show  $A$  is not serialisable in any of the six cases in Definition 13.

1.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PnotE}(\alpha')]$  such that  $p_\alpha(v) \neq p_{\alpha'}(v)$ . Since a single state  $s$  cannot satisfy both  $p_\alpha(v)$  and  $p_{\alpha'}(v)$ , only one of  $\alpha$  or  $\alpha'$  is applicable on  $s$  but not both. This clearly contradicts the only if part of the first assertion of this lemma.
2.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{EnotP}(\alpha')]$  such that  $p_\alpha(v) \neq e_{\alpha'}(v)$ . This means  $\alpha$  cannot be applied after  $\alpha'$ , contradicting our assumption that  $A$  is serialisable.
3.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PandE}(\alpha')]$ . If  $p_\alpha(v) \neq p_{\alpha'}(v)$ , using the same argument as in Case 1, we show contradiction here. On the contrary, if  $p_\alpha(v) = p_{\alpha'}(v)$  then, because of  $p_{\alpha'}(v) \neq e_{\alpha'}(v)$  as is specified in Definition 2,  $p_\alpha(v) \neq e_{\alpha'}(v)$ . Now using the same argument as in Case 2, we show contradiction in this situation.
4.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{EnotP}(\alpha')]$  such that  $e_\alpha(v) \neq e_{\alpha'}(v)$ . In this case  $s'(v)$  will be  $e_\alpha(v)$  if  $\alpha$  is after  $\alpha'$  and  $e_{\alpha'}(v)$  if  $\alpha'$  is after  $\alpha$ . This means  $s'(v)$  depends on the ordering of the actions, contradicting our assumption that  $A$  is serialisable.
5.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{PandE}(\alpha')]$ . If  $e_\alpha(v) \neq e_{\alpha'}(v)$ , using the same argument as in Case 4, we show contradiction here. On the contrary, if  $e_\alpha(v) = e_{\alpha'}(v)$  then, because of  $p_{\alpha'}(v) \neq e_{\alpha'}(v)$  as is specified in Definition 2,  $e_\alpha(v) \neq p_{\alpha'}(v)$ . Now using the same argument as in Case 2, we show contradiction in this situation.
6.  $\exists v \in [\text{PandE}(\alpha) \cap \text{PandE}(\alpha')]$ . If  $e_\alpha(v) \neq e_{\alpha'}(v)$ , using the same argument as in Case 4, we show contradiction here. On the contrary, if  $e_\alpha(v) = e_{\alpha'}(v)$  then, because of  $p_\alpha(v) \neq e_\alpha(v)$  as is specified in Definition 2,  $p_\alpha(v) \neq e_{\alpha'}(v)$ . Now using the same argument as in Case 2, we show contradiction in this situation.

For the if part, assume the two conditions hold. So no two actions  $\alpha, \alpha'$  are in parallel conflict i.e.  $\alpha \otimes \alpha'$ . This means no action destroys applicability preconditions of another action. This also means every ordering of any two actions produces consistent effects. The resulting state will therefore be the same. Therefore  $A$  is serialisable.  $\square$

The following lemma shows when a sequence of sets of actions is a parallel plan.

**Lemma 3** (Parallel Planning). *Let  $\Pi = \langle A^1, \dots, A^m \rangle$  be a sequence of sets of actions from a multi-valued planning task  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ .  $\Pi$  is parallel plan for  $\mathcal{P}$  as per Definition 7 if and only if there exists a sequence of states  $\langle s_0, s_1, \dots, s_m \rangle$  such that (i)  $s_0 = \mathcal{I}$ , (ii) for all  $\tau \in [1, m]$ ,  $\text{app}(s_{\tau-1}, A^\tau) = s_\tau$ , (iii) for all  $\tau \in [1, m]$ , there exist no two actions  $\alpha, \alpha' \in A^\tau$  such that  $\alpha \otimes \alpha'$ , and (iv)  $\mathcal{G}(v) = s_m(v)$  for each variable  $v \in \mathcal{G}$ .*

*Proof.* The proof is very straightforward. Conditions (i), (ii), and (iv) are obvious from conditions (i), (ii), and (iii) in the definition of a parallel plan in Definition 7. Condition (iii) comes from the property (ii) of Lemma 2, which is proved to satisfy the necessary and sufficient conditions of serialisability.  $\square$

The lemma below addresses the soundness and completeness of running Algorithm 2 for a given makespan level. It also addresses the optimality of running Algorithm 2 for a series of makespan levels starting from 1 up to  $m$  when a plan is found.

**Lemma 4** (Search Properties). *Running Algorithm 2 for makespan  $0, 1, \dots, m$  returns a correct and makespan-optimal parallel plan for a given multi-valued planning task  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ , if and only if a parallel plan exists within the given makespan limit  $m$ .*

*Proof.* Algorithm 2 is correct for a given  $m$  because it starts by initialising the current state of the DTG( $v$ ) for each state variable  $v$ . In each iteration, it changes the state of each DTG( $v$ ) or remains at the same state, but as per the definition of parallel conflicts in Definition 13 avoids conflicting actions at each iteration. Lemma 2 proves that actions obtained from the selected transitions are serialisable and so the sequence of sets of actions returned at the end when the goal conditions are satisfied is indeed a valid parallel plan. Putting altogether, Lemma 3 proves the soundness of the approach for a given  $m$ . Algorithm 2 is complete for a given  $m$  because Function `selectParallelActions` non-deterministically select a set of non-conflicting actions at each time step. In terms of determinism, this means all possible search branches are explored. One could argue that at one time step, actions selected could all be `no-ops` or loops and this could happen at each time step. This is true in some search branches, but in other branches—remember all search branches are explored—this is not the case. So if there exists a parallel plan within the given makespan, it will definitely be found in some branch, although the returned plan might not be makespan optimal. Nevertheless, this proves the completeness of Algorithm 2 for a given makespan  $m$ . Now if we start from makespan value 0 and every time increase it by 1 until a given limit, a plan will be found if exists within the limit. This proves the overall completeness mentioned in the lemma. The soundness of running Algorithm 2 a number of times also comes from soundness of the individual runs. For optimality of the series of runs, we argue that a higher makespan level is tried only if a plan is not found by running Algorithm 2 with a lower makespan level. Because of the completeness of the algorithm at each run, if there is a plan at a lower makespan level, it must have been found earlier. This proves the optimality of the series run of Algorithm 2 with progressively larger makespan.  $\square$

## 5. Our TCPP Planner

Given Lemma 4, we obtain a deterministic implementation of the Algorithm 2 via constraint satisfaction techniques. For this, we develop a constraint model to capture a DTG-based parallel planning problem. We use a CSP solver to perform complete search within a given makespan. Lastly, if a CSP solution is found, we decode it to get a parallel plan.

### 5.1 Encoding DTGs into CSP

The encoder described in Algorithm 3 takes as input the DTGs extracted from the multi-valued representation of the planning problem and also the makespan  $m$  and outputs a CSP, modelling the problem of finding the plan with this makespan. In the CSP model, there is a CSP variable  $v^\tau$  for each state variable  $v$  and timestep  $\tau \in [0, m]$ . Each of these  $(m+1)|\mathcal{V}|$  CSP variables will have the same domain as the state variable  $v$  has. Transitions in the DTGs are encoded as transition table constraints. To ensure action serialisability at each time step, we also need parallelism constraints, which are implemented through parallelism variables. For each variable  $v$  we may have a parallelism variable  $p_v$  for timestep  $\tau \in [0, m]$ . So there could be at most  $m|\mathcal{V}|$  parallelism variables and therefore at most  $(2m+1)|\mathcal{V}|$  CSP variables in our model. Moreover, for better efficiency of our planner, optionally we also use the mutex-groups that are produced by the PDDL to SAS<sup>+</sup> translator.

---

**Algorithm 3** DTGs To CSP Encoding

---

```

1 Planing Problem  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ , makespan  $m$ 
2 //variables:  $v, v'$ , values:  $k, k'$ , actions:  $\alpha, \alpha'$ 
3 Foreach  $(v, k) \in \mathcal{I}$ , Add constraint  $\text{eq}(v^0, k)$ 
4 Foreach  $(v, k) \in \mathcal{G}$ , Add constraint  $\text{eq}(v^m, k)$ 
5 For timestep  $\tau = 0$  to  $m - 1$  do
6   Foreach  $v \in \mathcal{V}$ , Add transition constraint  $\text{tc}(v^\tau)$ 
7   Foreach pair  $\alpha, \alpha' \in \mathcal{A}$  such that  $\alpha \otimes \alpha'$ 
8     Add parallelism variables and their values to the
9     transition constraints of the shared variables
10  // the mutex constraints below are optional
11  For timestep  $\tau = 0$  to  $m$  do
12    Foreach pair  $\{v = k, v' = k'\}$  in mutex groups
13      Add a mutex constraint  $\text{mc}(v, k, v', k', \tau)$ 
```

---

### 5.2 Initial State and Goal Constraints

To encode the initial state and the goals of the planning problem, we need to add constraints specifying the values of state variables at time 0 and  $m$ . Since the initial state is fully specified, we need  $|\mathcal{V}|$  equality constraints that specify the values of each CSP variable  $v^0$  for a state variable  $v \in \mathcal{V}$ . For the goals, we only have the values for some of the state variables  $v^m$ s and we need constraints to specify their values at time  $m$ . These constraints are added to the constraint model by Lines 3 and 4 of Algorithm 3.

### 5.3 Our State Transition Constraints

To encode the state transitions in the DTGs of the state variables extracted from the multi-valued representation of the planning problem, we have transition constraints in our model (Line 6 in Algorithm 3). These constraints specify on what conditions the values of the state variables can change between time steps  $\tau$  and  $\tau+1$ . The edges in the DTG of each variable are responsible for the changes of values between consecutive time steps. This change is

caused by actions. Since actions have some conditions that appear on the edges, the change can only occur if these conditions are met. Therefore, we need a constraint that specifies the values that other variables should take to let this change happen. The variables on the corresponding edge are the only variables whose values should be specified. There may be other variables on the other edges of the graph that are considered *don't care* for this transition. Inspired by the use of table constraints in (Barták, 2011a), we have used this kind of table constraints to represent transitions.

### 5.3.1 DRIVERLOG EXAMPLE

We explain how we encode the state transitions in the redefined DTG's in Figures 7 and 8 to transition constraints shown in Figure 9. Suppose we want to encode the DTG of variable **d-loc** to a transition constraint. First of all, we need to specify the variables involved in this constraint; these are the columns of the table. Variables of the table are those variables occurring on all edges of the graph. Therefore, we need to look at each edge and the corresponding action. All the state variables in the preconditions and effects of that action are included as CSP variables in the constraint table both for time  $\tau$  and  $\tau + 1$ . For the DTG of **d-loc**, the state variables are **t-loc** and **t-occ**. We also need the state variable of the DTG itself that is **d-loc**. These variable should be considered at time  $\tau$  and  $\tau + 1$ . So we have 6 columns in the table: **d-loc** $^\tau$ , **d-loc** $^{\tau+1}$ , **t-loc** $^\tau$ , **t-loc** $^{\tau+1}$ , **t-occ** $^\tau$ , and **t-occ** $^{\tau+1}$ .

The next step in encoding the DTG as a constraint table is defining the rows of the table. Each edge of the graph is responsible for the change of the value of **d-loc** and would become a row in the table specifying a valid assignment of values to the variables of the table. We have 10 non-loop edges in the DTG of **d-loc** so we will have 10 rows in the table for the state transitions. The edge from **A** to **t** labelled with  $\langle 1, \rho_1, \epsilon_1 \rangle$  denotes the action **embark-truck**. Due to this edge, the value of **d-loc** can change from **A** to **t** at time  $\tau$  if truck **t** is at location **A** and is not occupied. After the action is executed the value of **d-loc** changes to truck **t** and the truck is now occupied. The location of truck must not change. We, therefore, have the first row in the table with values  $\langle \mathbf{A}, \mathbf{t}, \mathbf{A}, \mathbf{A}, \perp, \top \rangle$  for columns  $\langle \mathbf{d-loc}^\tau, \mathbf{d-loc}^{\tau+1}, \mathbf{t-loc}^\tau, \mathbf{t-loc}^{\tau+1}, \mathbf{t-occ}^\tau, \mathbf{t-occ}^{\tau+1} \rangle$  respectively. The second row corresponds to the edge from **t** to **A** for action **debark-truck**. Since **t-occ** is not a precondition of the corresponding action, notice that the value of **t-occ** $^\tau$  is  $\times$ . Rows 3 to 6 are for the other edges of the graph from **B** to **t**, **t** to **B**, **C** to **t**, and **t** to **C** respectively. Now consider the edge from **C** to **D**. The corresponding action is **driver-walk** and driver can change his location by walking from **C** to **D**. There is no condition on this edge and this means that variables **t-loc** and **t-occ** can take any values at time  $\tau$  and  $\tau + 1$ ; so their values are considered *don't cares* in Row 7. Rows 8 to 10 are for the other **driver-walk** actions in the remaining edges of the graph. Lastly, the driver can stay at the same location between successive time steps. For all the loops in the DTG, we therefore have rows 11-15 with **d-loc** having the same values at time  $\tau$  and  $\tau + 1$  and the other variables having *don't cares* as their values.

### 5.3.2 TRANSITION ENCODING PROCEDURE

The detailed procedure that encodes a DTG to a table constraint is represented in Algorithm 4. For a state variable  $v$  with  $\text{DTG}(v)$ , the table  $T^\tau$  at time  $\tau$  has columns  $v^\tau, v^{\tau+1}, \bar{v}^\tau, \bar{v}^{\tau+1}$ s, where  $\bar{v}$ s are variables appearing on all edges of  $\text{DTG}(v)$  (Line 3). Each transi-

TC for d-loc					
d-loc		t-loc		t-occ	
$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$
A	t	A	A	$\perp$	$\top$
t	A	A	A	$\times$	$\perp$
B	t	B	B	$\perp$	$\top$
t	B	B	B	$\times$	$\perp$
C	t	C	C	$\perp$	$\top$
t	C	C	C	$\times$	$\perp$
C	D	$\times$	$\times$	$\times$	$\times$
D	C	$\times$	$\times$	$\times$	$\times$
A	D	$\times$	$\times$	$\times$	$\times$
D	A	$\times$	$\times$	$\times$	$\times$
A	A	$\times$	$\times$	$\times$	$\times$
B	B	$\times$	$\times$	$\times$	$\times$
C	C	$\times$	$\times$	$\times$	$\times$
D	D	$\times$	$\times$	$\times$	$\times$
t	t	$\times$	$\times$	$\times$	$\times$

TC for t-occ					
t-occ		d-loc		t-loc	
$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$
$\perp$	$\top$	A	t	A	A
$\perp$	$\top$	B	t	B	B
$\perp$	$\top$	C	t	C	C
$\times$	$\perp$	t	A	A	A
$\times$	$\perp$	t	B	B	B
$\times$	$\perp$	t	C	C	C
$\perp$	$\perp$	$\times$	$\times$	$\times$	$\times$
$\top$	$\top$	$\times$	$\times$	$\times$	$\times$

TC for t-loc			
t-loc		d-loc	
$\tau$	$\tau + 1$	$\tau$	$\tau + 1$
A	B	t	t
B	A	t	t
B	C	t	t
C	B	t	t
A	A	$\times$	$\times$
B	B	$\times$	$\times$
C	C	$\times$	$\times$

Figure 9: Encoding transitions in DTGs in Figures 7 and 8 using table constraints. In the figure, Driver: d; Truck: t; Locations: A, B, C, D; Boolean: true  $\top$ , false  $\perp$ ; loc: location; occ: occupied; TC: transition constraint;  $\times$ : *don't care*;  $\tau$ : time step.

tion constraint is therefore an  $l$ -ary constraint where  $l = 2(l' + 1)$  and  $l'$  is the number of variables appearing on the edges of  $\text{DTG}(v)$ . Next, we extract the rows of the table for each edge in  $\text{DTG}(v)$  (Line 4). Suppose  $\langle k, k' \rangle$  is an edge in  $\text{DTG}(v)$  and  $\text{tr}$  is the corresponding row to be added to the table. With respect to the edge, row  $\text{tr}$  will have appropriate values in the relevant columns and *don't cares*  $\times$  in the irrelevant columns. Because of edge  $\langle k, k' \rangle$ , clearly,  $v^\tau$  is  $k$  and  $v^{\tau+1}$  is  $k'$ ; note  $k$  could be  $\times$  in the  $\text{DTG}(v)$  (Line 5-7). We then consider every variable  $\bar{v}$  that appears on the  $\rho$  and  $\epsilon$  components of the edge label  $\langle \alpha, \rho, \epsilon \rangle$ . There could be three possible cases depending on whether a variable appears either in  $\rho$  or in  $\epsilon$  or in both. In an edge label,  $\alpha$  is the action responsible for the transition. Lines 9-11 in Algorithm 4 cover the following three cases:

1.  **$\bar{v}$  only in  $\rho$ :** In this case,  $\bar{v}$  remains the same at times  $\tau$  and  $\tau + 1$  ensuring that during the execution of  $\alpha$ ,  $\bar{v}$ 's value does not change by any other action.
2.  **$\bar{v}$  in  $\rho$  and  $\epsilon$  both:** In this case  $\bar{v}$  changes the value between times  $\tau$  and  $\tau + 1$ . So the value at time  $\tau$  is  $\rho(\bar{v})$  and at time  $\tau + 1$  is  $\epsilon(\bar{v})$ . Note  $\rho(\bar{v})$  and  $\epsilon(\bar{v})$  are values assigned to  $\bar{v}$  by the assignments  $\rho$  and  $\epsilon$  respectively.
3.  **$\bar{v}$  only in  $\epsilon$ :** In this case, we need to specify the value of  $\bar{v}$  only at time  $\tau + 1$ , which is  $\epsilon(\bar{v})$ . At time  $\tau$ , the value of  $\bar{v}$  is *don't care*  $\times$ , since  $\bar{v} \notin \rho$ .

To summarise, we have  $m|\mathcal{V}|$  transition constraints for a planning problem where  $\mathcal{V}$  is the set of multi-valued state variables and  $m$  is the current makespan. The number of rows

**Algorithm 4** Transition constraint  $\text{tc}(v^\tau)$  using a table

---

```

1 DTG( $v$ ): DTG for the state variable  $v$ ,  $T^\tau$ : table for transitions of CSP variable  $v^\tau$ 
2  $V = \{v\} \cup \{\bar{v} : \bar{v} \Leftarrow \rho \vee \bar{v} \Leftarrow \epsilon \text{ where } \langle \alpha, \rho, \epsilon \rangle \text{ is the label of an edge in DTG}(v)\}$ 
3 Foreach  $\bar{v} \in V$ , table  $T^\tau$  has two columns  $\bar{v}^\tau$  and  $\bar{v}^{\tau+1}$ , for two time steps
4 Foreach edge  $\langle k, k' \rangle$  in DTG( $v$ ) //  $k, k'$  are values of  $v$ ,  $k$  may equal to  $k'$ 
5   tr: a new row in  $T^\tau$  where each column has initially don't care  $\times$ 
6   // Modify tr in the following ways to capture the transition
7    $\text{tr}[v^\tau] = k$ ,  $\text{tr}[v^{\tau+1}] = k'$  // note  $k$  could be a don't care  $\times$ 
8   Assume  $\langle \alpha, \rho, \epsilon \rangle$  is the label of the edge  $\langle k, k' \rangle$  in the DTG( $v$ )
9   Foreach variable  $\bar{v} \in V$  with  $(\bar{v} \Leftarrow \rho \wedge \bar{v} \not\Leftarrow \epsilon)$ ,  $\text{tr}[\bar{v}^\tau] = \text{tr}[\bar{v}^{\tau+1}] = \rho(\bar{v})$ 
10  Foreach variable  $\bar{v} \in V$  with  $(\bar{v} \Leftarrow \rho \wedge \bar{v} \Leftarrow \epsilon)$ ,  $\text{tr}[\bar{v}^\tau] = \rho(\bar{v})$ ,  $\text{tr}[\bar{v}^{\tau+1}] = \epsilon(\bar{v})$ 
11  Foreach variable  $\bar{v} \in V$  with  $(\bar{v} \not\Leftarrow \rho \wedge \bar{v} \Leftarrow \epsilon)$ ,  $\text{tr}[\bar{v}^{\tau+1}] = \epsilon(\bar{v})$ 
12 Add columns for parallelism variables and set their values in rows (see Section 5.4)
```

---

in the table constraint for a DTG is the number of edges (both loops and non-loops) in the DTG. However, the number of columns of the table totally depends on the planning problem. When the number of variables on the edges of the DTG is huge, we have a table with a large number of columns. This is the case when the operators responsible for changing the values of a state variable have many preconditions on the values of other variables. Note that depending on the parallel conflicts between actions, in some of the tables, we may also have columns for parallelism variables. The values of the parallelism variables capture the parallelism constraints needed to ensure the parallel conflicts. This is discussed in details later. Nevertheless, the number of *don't care* values in the rows depends on the number of shared variables on the edges of the graph. If the edges share the same set of variables, we will have actual values for these variables in the rows of the table corresponding to the edges. However, if each edge has a separate set of variables on it, the values of these variables will be *don't care* for the rows of the table corresponding to the other edges. To illustrate this, we use a simple example from the blocks world domain.

### 5.3.3 BLOCKS WORLD EXAMPLE

In the blocks world example in Figure 10, we have two blocks A and B. The blocks are on the table in the initial state and we want to put the block A on the block B in the final state. The PDDL representation shows the five predicates *clear*, *ontable*, *empty*, *holding*, and *on*. The four actions in this domain are *pick-up*, *put-down*, *stack*, and *unstack*. The SAS<sup>+</sup> representation of this problem has 5 state variables: *A-top*, *B-top*, *hand*, *A-loc*, and *B-loc*. Variables *A-top* and *B-top* specify if the respective blocks have their top clear or not. Variable *hand* specifies if hand is empty or not. Variables *A-loc* and *B-loc* denote the locations of blocks A and B. A block could be at location *onT* denoting the block is on table or at location *onx* meaning on another block  $x$  (A or B in this case), or at hand denoted by *H*. Thus, the domains of these five variables *A-top*, *B-top*, *hand*, *A-loc*, and *B-loc* are  $\{C, NC\}$ ,  $\{C, NC\}$ ,  $\{E, NE\}$ ,  $\{H, onB, onT\}$ , and  $\{H, onA, onT\}$  respectively.

PDDL representation	SAS <sup>+</sup> representation
Predicates: clear( $x$ ), ontable( $x$ ), empty, holding( $x$ ), on( $x$ , $y$ ) Actions: pick-up( $x$ ) preconds: empty, clear( $x$ ), ontable( $x$ ) effects: $\neg$ clear( $x$ ), $\neg$ ontable( $x$ ), $\neg$ empty, holding( $x$ ) put-down( $x$ ) preconds: holding( $x$ ) effects: $\neg$ holding( $x$ ), clear( $x$ ), empty, ontable( $x$ ) stack( $x$ , $y$ ) preconds: holding( $x$ ), clear( $y$ ) effects: $\neg$ holding( $x$ ), $\neg$ clear( $y$ ), clear( $x$ ), on( $x$ , $y$ ), empty unstack( $x$ , $y$ ) preconds: empty, on( $x$ , $y$ ), clear( $x$ ) effects: $\neg$ clear( $x$ ), $\neg$ on( $x$ , $y$ ), $\neg$ empty, holding( $x$ ), clear( $y$ )	Variables: A-top: {C, NC}, B-top: {C, NC}, //clear? hand: {E, NE} //empty? A-loc {H, onB, onT}, B-loc {H, onA, onT} Operators: pick-up( $x$ ) preconds: $x$ -top = C, hand = E, $x$ -loc = onT Effects: $x$ -top = NC, hand = NE, $x$ -loc = H put-down( $x$ ) preconds: $x$ -loc = H effects: $x$ -top = C, hand = E, $x$ -loc = onT stack( $x$ , $y$ ) preconds: $y$ -top = C, $x$ -loc = H effects: $x$ -top = C, $y$ -top = NC, hand = E, $x$ -loc = ony unstack( $x$ , $y$ ) preconds: $x$ -top = C, hand = E, $x$ -loc = ony effects: $x$ -top = NC, $y$ -top = C, hand = NE, $x$ -loc = H

Figure 10: The blocks world domain in PDDL and SAS<sup>+</sup>

We only depict the DTGs of variables *hand* and *A-loc* in Figure 11. In the DTG for *hand*, the edge labelled with  $\langle 2, \rho_2, \epsilon_2 \rangle$  is for action *put-down(A)*. We have the variable *A-loc* in preconditions  $\rho_2$  and it needs to take the value H. After *put-down(A)* is executed, the effects will be *A-top* = C, *A-loc* = onT, which are in  $\epsilon_2$ . Similarly, other edges in DTG(*hand*) and all edges in DTG(*A-loc*) could be constructed by using Algorithm 1.

The transition constraints for the two DTGs are in Figure 12. To encode the DTG of *hand* as a table constraint, we have included all the state variables on the edges of the graph at time  $\tau$  and  $\tau + 1$ . The first row of the table is for edge from  $\times$  to E labelled with  $\langle 2, \rho_2, \epsilon_2 \rangle$  which is for action *put-down(A)*. As you can see, the variable *A-top* is not among the preconditions but it is in the effects with value C. So we set values  $\times$  and C for variables *A-top* at time  $\tau$  and  $\tau + 1$ . The value of *A-loc* changes from H to onT by action *put-down(A)*. The value for other variables are considered *don't cares*. The other rows are extracted in a similar manner. We can see from the tables, there are comparatively a large number of *don't care* values in these tables. Considering the conditions on the edges of the DTGs, it is obvious that they share a small number of variables. This is in contrast with our previous example from driverlog domain where the variables of the conditions on edges were almost the same. That is why we had a small number of *don't care* values in those tables.



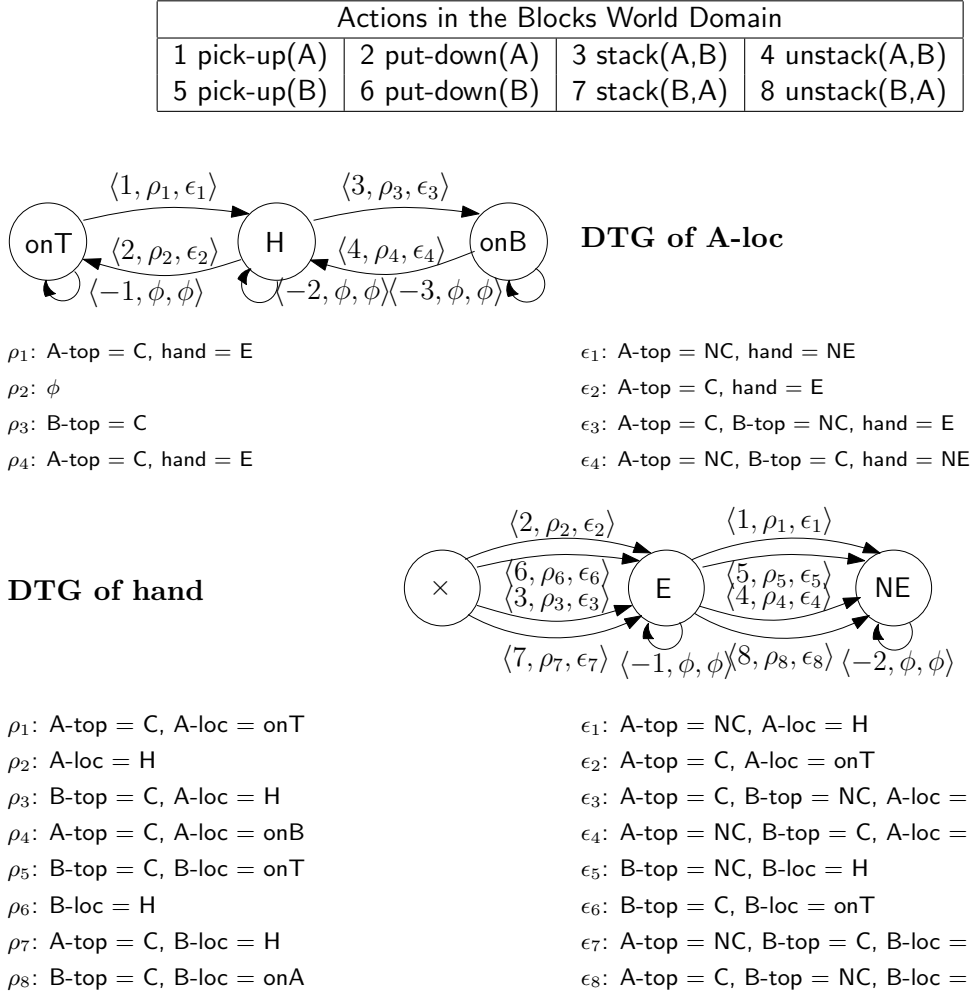


Figure 11: Top: actions in the example blocks world problem. Bottom: DTGs for two sample variables for an example from the blocks world domain. The positive numbers on the edges are the action identifiers from the table at the top and the negative numbers are distinct identifiers for the *no-op* actions.

#### 5.4 Handling Parallelism Conflicts

As described already, the transition constraints encode the possible changes of values of the state variables. Since many variables can change their values at the same time step, we can have several actions taking place at the same time. To prevent actions that have parallel conflicts among them from happening at the same time, in our model (Lines 7–9 of Algorithm 3), we add parallelism variables to the corresponding transition table constraints. These parallelism variables and their values are to capture the conditions in Definition 13 and thus to implement the parallelism constraints required to prevent parallel conflicts. If we have conflicting actions in the transition table constraint of a state variable, we will

TC for hand									
hand		A-top		B-top		A-loc		B-loc	
$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$
×	E	×	C	×	×	H	onT	×	×
×	E	×	×	×	C	×	×	H	onT
×	E	×	C	C	NC	H	onB	×	×
×	E	C	NC	×	C	×	×	H	onA
E	NE	C	NC	×	×	onT	H	×	×
E	NE	×	×	C	NC	×	×	onT	H
E	NE	C	NC	×	C	onB	H	×	×
E	NE	×	C	C	NC	×	×	onA	H
E	E	×	×	×	×	×	×	×	×
NE	NE	×	×	×	×	×	×	×	×

TC for A-loc									
A-loc		A-top		B-top		hand			
$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$	$\tau$	$\tau + 1$		
onT	H	C	NC	×	×	E	NE		
H	onT	×	C	×	×	×	E		
H	onB	×	C	C	NC	×	E		
onB	H	C	NC	×	C	E	NE		
H	H	×	×	×	×	×	×		
onB	onB	×	×	×	×	×	×		
onT	onT	×	×	×	×	×	×		

Figure 12: Encoding state transitions in DTGs in Figure 11 using table constraints.

consider a parallelism variable for that table constraint. This variable will take different values in the corresponding rows of conflicting actions. These parallelism variables having different values in the rows for conflicting actions, prevent them from happening at the same time.

**Example.** Figure 13 shows two actions  $\alpha$  and  $\alpha'$  such that  $\alpha \otimes \alpha'$  as per Condition 6 in Definition 13. To make it clearer,  $\alpha$  and  $\alpha'$  conflict because both actions have  $v_3$  in their effects as well as in their preconditions. These actions correspond to different rows in the transition constraint table of  $v_3$ . To prevent these actions from happening at the same time we add an extra parallelism variable  $p_{v_3}^\tau$  to the table. This variable will have different values  $k$  and  $k'$  in the rows for actions  $\alpha$  and  $\alpha'$ . Note that actions  $\alpha$  and  $\alpha'$  will also appear in the transition constraint tables of  $v_4$  and  $v_5$  respectively. To synchronise the transitions across different tables, we must add this parallelism variable  $p_{v_3}^\tau$  to the transition constraint tables of  $v_4$  and  $v_5$ . This variable will have value  $k$  in the corresponding row for action  $\alpha$  in transition constraint table of  $v_4$  and will have value  $k'$  in the corresponding row for action  $\alpha'$  in transition constraint table of  $v_5$ . Note that because of conflicting actions in other transition constraint tables, further parallelism variables may also be added to this table.

Now suppose we have another action  $\alpha''$  (shown in Figure 13), whose preconditions are the union of preconditions of  $\alpha$  and  $\alpha'$ , and the effects are the union of effects of  $\alpha$  and  $\alpha'$ . For this action we will have another row in the table and since this action is conflicting with  $\alpha$  and  $\alpha'$  based on the same condition 6 in Definition 13, we will have a separate value  $k''$  for the parallelism variable in this row. Now it is obvious that any of these actions can be selected to be included in the plan and preventing  $\alpha$  and  $\alpha'$  from happening at the same time does not prevent  $\alpha''$ .

We now formally describe how our model prevents actions having parallel conflicts from happening at the same time. From the six types of condition in Definition 13, some conditions are prevented automatically in our model. For the others, we add parallelism variables to transitions tables and assign appropriate row values to them to prevent the corresponding actions from happening at the same time.

<b>action <math>\alpha</math></b>	<b>action <math>\alpha'</math></b>	<b>action <math>\alpha''</math></b>
<b>preconds:</b> $v_1 = 1, v_3 = 2$	<b>preconds:</b> $v_2 = 5, v_3 = 2, v_5 = 6$	<b>preconds:</b> $v_1 = 1, v_2 = 5, v_3 = 2, v_5 = 6$
<b>effects:</b> $v_3 = 3, v_4 = 4$	<b>effects:</b> $v_3 = 3, v_5 = 7$	<b>effects:</b> $v_3 = 3, v_4 = 4, v_5 = 7$

Transition Table $tc(v_3)$											
$v_1^\tau$	$v_1^{\tau+1}$	$v_3^\tau$	$v_3^{\tau+1}$	$v_4^\tau$	$v_4^{\tau+1}$	$v_2^\tau$	$v_2^{\tau+1}$	$v_5^\tau$	$v_5^{\tau+1}$	$\dots$	$p_{v_3}^\tau$
1	1	2	3	$\times$	4	$\times$	$\times$	$\times$	$\times$	$\times$	$k$
$\times$	$\times$	2	3	$\times$	$\times$	5	5	6	7	$\times$	$k'$
1	1	2	3	$\times$	4	5	5	6	7	$\times$	$k''$

Figure 13: Adding parallelism variables for two conflicting actions

**Lemma 5** (Parallelism Constraints). *Given the transition constraints constructed by Algorithm 4, we need constraints only for the Conditions 5 and 6 of Definition 13 to avoid selection of actions that have parallel conflicts. Other conditions are implicitly enforced.*

*Proof.* We analyze one after another each condition of Definition 13 below.

1.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PnotE}(\alpha')]$  such that  $p_\alpha(v) \neq p_{\alpha'}(v)$ . For  $\alpha$ ,  $v^\tau = p_\alpha(v)$  but for  $\alpha'$ ,  $v^\tau = p_{\alpha'}(v)$ . However,  $v^\tau$  can take only one value at one time. So this condition can never happen and we do not need any constraint for this case
2.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{EnotP}(\alpha')]$  such that  $p_\alpha(v) \neq e_{\alpha'}(v)$ . For  $\alpha$ ,  $v^{\tau+1} = v^\tau = p_\alpha(v)$  but  $v^{\tau+1} = e_{\alpha'}(v)$  for  $\alpha'$ . Clearly, this is not possible as  $v^{\tau+1}$  can take only one value at one time. So we do not need any constraint for this case.
3.  $\exists v \in [\text{PnotE}(\alpha) \cap \text{PandE}(\alpha')]$ . For  $\alpha$ ,  $v^\tau = v^{\tau+1} = p_\alpha(v)$  and for  $\alpha'$ ,  $v^\tau = p_{\alpha'}(v)$ ,  $v^{\tau+1} = e_{\alpha'}(v)$ . Since  $p_{\alpha'}(v) \neq e_{\alpha'}(v)$ ,  $p_\alpha(v)$  cannot be equal to both of them at the same time. So we do not need any constraint for this case.
4.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{EnotP}(\alpha')]$  such that  $e_\alpha(v) \neq e_{\alpha'}(v)$ . For  $\alpha$ ,  $v^{\tau+1} = e_\alpha(v)$  but for  $\alpha'$ ,  $v^{\tau+1} = e_{\alpha'}(v)$ . However,  $v^{\tau+1}$  can take only one value at one time. So this condition can never happen and we do not need any constraint for this case.
5.  $\exists v \in [\text{EnotP}(\alpha) \cap \text{PandE}(\alpha')]$ . For  $\alpha$ ,  $v^{\tau+1} = e_\alpha(v)$  and for  $\alpha'$ ,  $v^\tau = p_{\alpha'}(v)$  and  $v^{\tau+1} = e_{\alpha'}(v)$  where  $p_{\alpha'}(v) \neq e_{\alpha'}(v)$ . If  $e_\alpha(v) \neq e_{\alpha'}(v)$ , we do not need any constraint because  $v^{\tau+1}$  cannot take two values at one time. However, if  $e_\alpha(v) = e_{\alpha'}(v)$ , we have to implement a parallelism constraint by adding a parallelism variable and along with its appropriate values in the corresponding rows of the transition table constraint.
6.  $\exists v \in [\text{PandE}(\alpha) \cap \text{PandE}(\alpha')]$ . For  $\alpha$ ,  $v^\tau = p_\alpha(v)$  and  $v^{\tau+1} = e_\alpha(v)$  where  $p_\alpha(v) \neq e_\alpha(v)$ . For  $\alpha'$ ,  $v^\tau = p_{\alpha'}(v)$  and  $v^{\tau+1} = e_{\alpha'}(v)$  where  $p_{\alpha'}(v) \neq e_{\alpha'}(v)$ . If  $p_\alpha(v) \neq$

$p_{\alpha'}(v)$ , we do not need any constraint because  $v^\tau$  cannot take two values at one time. Similarly, if  $e_\alpha(v) \neq e_{\alpha'}(v)$ , we do not need any constraint because  $v^{\tau+1}$  cannot take two values at one time. However, if  $p_\alpha(v) = p_{\alpha'}(v)$  and  $e_\alpha(v) = e_{\alpha'}(v)$  then we have to implement a parallelism constraint by adding a parallelism variable and along with its appropriate values in the corresponding rows of the transition table constraint.

We therefore need parallelism constraints only for Conditions 5 and 6, and still partially.  $\square$

In order to identify the actions that could have parallel conflicts among them due to the parts of conditions 5 and 6 of Lemma 5, we define two sets of actions below.

**Definition 14.** *Given a state variable  $v$  and two values  $k, k'$  from its domain  $D(v)$ , we define the following two sets of actions:*

$\overline{\text{PandE}}(v, k, k')$ : *a set of actions  $\{\alpha | v \in \text{PandE}(\alpha) \wedge p_\alpha(v) = k \wedge e_\alpha(v) = k'\}$  where each action has  $(v = k)$  in its precondition and  $(v = k')$  in its effects. These actions are on the edges between any two vertexes  $k \neq k'$  in  $\text{DTG}(v)$  of the state variable  $v$ , where  $k, k'$  are not the don't care. As per Condition 6 of Lemma 5, any two actions in this set have parallel conflict with each other.*

$\overline{\text{EnotP}}(v, k')$ : *a set of actions  $\{\alpha | v \in \text{EnotP}(\alpha) \wedge e_\alpha(v) = k'\}$  where each action has  $(v = k')$  in its effects. Clearly, these actions are on the edges from the don't care vertex  $\times$  (if any) to the vertex  $k'$  in  $\text{DTG}(v)$  of the state variable  $v$ . No two actions in this set have parallel conflict with each other.*

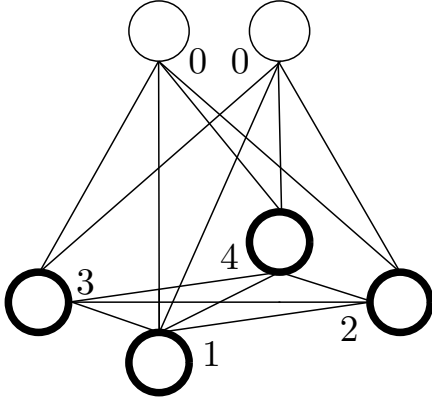
As per Condition 5 of Lemma 5, any  $\alpha \in \overline{\text{EnotP}}(v, k')$  and any  $\alpha' \in \overline{\text{PandE}}(v, k, k')$  have parallel conflict between each other.

Given the two sets of actions as defined above, below we define a parallel conflict graph that represents the parallel conflicts between these actions.

**Definition 15** (Parallel Conflict Graph). *Given a variable  $v$  and two values  $k, k' \in D(v)$ , the parallel conflict graph  $\text{PCG}(v, k, k')$  has the set of vertexes  $\overline{\text{PandE}}(v, k, k') \cup \overline{\text{EnotP}}(v, k')$  and a set of edges  $\{(\alpha, \alpha') | \alpha \in \overline{\text{PandE}}(v, k, k') \wedge \alpha' \in (\overline{\text{PandE}}(v, k, k') \cup \overline{\text{EnotP}}(v, k'))\}$ .*

In order to avoid actions having parallel conflict from happening at the same time, we add to the transition constraint table  $\text{tc}(v)$  a parallelism variable  $p_v$  that will take different values for each two actions in parallel conflict. Assigning these values is essentially equivalent to solving the *graph colouring problem* on the parallel conflict graph. We, therefore, need to assign different labels to each two neighbouring vertexes in the parallel conflict graph. From the characteristics of this type of graph, as are shown in Figure 14, it is obvious that we will have  $|\overline{\text{PandE}}(v, k, k')| + 1$  labels. All the vertexes in  $\overline{\text{EnotP}}(v, k')$  could get (and does get in our implementation) the same label 0 and the vertexes in  $\overline{\text{PandE}}(v, k, k')$  could be labelled by sequential numbers from 1 to  $|\overline{\text{PandE}}(v, k, k')|$  in any permutation order.

We now need to find a way to assign a unique label for each action that appears in the edges of a given  $\text{DTG}(v)$ . Note that an action  $\alpha \in \overline{\text{PandE}}(v, k, k')$  can appear only in one parallel conflict graph  $\text{PCG}(v, k, k')$  over all pairs of values  $k \neq k'$ . However, for a given value  $k'$ , an action  $\alpha' \in \overline{\text{EnotP}}(v, k')$  could appear in all parallel conflict graphs  $\text{PCG}(v, k, k')$



- Thin circles: actions  $\overline{\text{EnotP}}(v, k')$
- Thick circles: actions  $\overline{\text{PandE}}(v, k, k')$
- Vertex Labels: Colours assigned by a graph colouring algorithm
- Thick circles all are in a clique.
- Each thin circle is the centre of a star.

Figure 14: A typical parallel conflict graph and a possible colouring of its vertexes.

for  $k \neq k'$ . In order to get a unique label for such action  $\alpha$  across all parallel conflict graphs  $\text{PCG}(v, k, k')$  for  $k \neq k'$ , we assign a predefined label 0 (without loss of generality) to any action  $\alpha' \in \overline{\text{EnotP}}(v, k')$ . Only the actions in  $\overline{\text{PandE}}(v, k, k')$  for any  $k \neq k'$  could actually be labelled by a graph colouring algorithm. A *no-op* action on an edge from  $k'$  to  $k'$  can happen in parallel with any actions in  $\overline{\text{EnotP}}(v, k')$ , but not with any actions in  $\overline{\text{PandE}}(v, k, k')$ . No two *no-op* actions can happen at the same time. So all *no-op* actions in the  $\text{DTG}(v)$  could get a predefined label 0. We thus have a unique label  $L_v(\alpha)$  for each  $\alpha$  in  $\text{DTG}(v)$ . The unique label  $L_v(\alpha)$  will be used as the value of column  $p_v$  and row  $\alpha$  in the table constraint  $\text{tc}(v)$  of the state variable  $v$ .

In Algorithm 5, we implement the addition of the parallelism variables to the transition table constraints and assigning the row values to the parallelism variables as well. In Procedure **ParallelismVariables**, for each state variable  $v$ , and for each pair of values  $k \neq k'$  of  $v$ , we create a parallel conflict graph by calling Procedure **GenerateGraph**. The *no-ops* appearing in the loops of the  $\text{DTG}(v)$  are assigned label 0. Also, the actions in  $o = \overline{\text{EnotP}}(v, k')$  gets label 0. Procedure **LabelGraph** assigns labels to the actions in  $o' = \overline{\text{PandE}}(v, k, k')$ . Having the labels, we just add a parallelism variable  $p_v$  to the transition table initialising values at all rows with 0. Then, for each action  $\alpha \in o'$ , we consider  $L_v(\alpha)$  as the value of the parallelism variable in the corresponding row. Besides this, to synchronise the transitions in all transition tables, we also add this parallelism variable and its value to the transitions tables of all other variables appearing in the effects of this action. These transition tables are the only tables that this action will appear in them as a row.

Although the above addresses the parallelism constraint implementation issue, we however, do some optimisation during generation of the parallel conflict graphs by Procedure **GenerateGraph**. When we make an attempt to add an edge between two conflicting actions  $\alpha$  and  $\alpha'$ , we check whether these actions have an automatically prevented conflict on some other variable. If so, we do not need any edge for this case. This checking is performed in Function **ElsePrevented**. Doing this we can reduce the number of labels needed. Since these labels will be the values of the parallelism variable that will be added to the transition table, the domain size of the parallelism variable will be smaller by this optimisation. Given this optimisation in the parallel conflict graph, a straightforward graph colouring as described before might not be very good. However, solving a graph colouring

**Algorithm 5** Adding Parallelism Variables to Transition Constraints

---

```

proc ParallelismVariables()
  foreach  $v \in \mathcal{V}$ 
    foreach  $k \in D(v)$ ,  $L_v(\alpha) = 0$ , where  $\alpha$  is on the loop  $\langle k, k \rangle$  in  $\text{DTG}(v)$ 
    foreach  $k, k' \in D(v) : k \neq k'$ 
       $o = \{\alpha | v \in \text{EnotP}(\alpha) \wedge e_\alpha(v) = k'\} // \overline{\text{EnotP}}(v, k')$ 
       $o' = \{\alpha | v \in \text{PandE}(\alpha) \wedge p_\alpha(v) = k, e_\alpha(v) = k'\} // \overline{\text{PandE}}(v, k, k')$ 
       $G = \text{GenerateGraph}(o, o')$ 
      Assign  $L_v(\alpha) = 0$  for all  $\alpha \in o$ 
      NumLabels = LabelGraph( $G$ )
      if (NumLabels == 0) continue;
      add column  $p_v$  to  $\text{tc}(v)$  with value 0 in all rows (if not added before)
      foreach  $\alpha$  in  $o \cup o'$ 
         $\text{tr} = \text{corresponding row of } \alpha \text{ in } \text{tc}(v)$ 
        if ( $\alpha \in o$ )  $\text{tr}[p_v] = L_v(\alpha)$ 
        foreach  $v' \in \text{PandE}(\alpha) \cup \text{EnotP}(\alpha) // \text{means in effects, no matter in preconds}$ 
          add column  $p_v$  to  $\text{tc}(v')$  with value 0 in all rows (if not added before)
           $\text{tr}' = \text{corresponding row of } \alpha \text{ in } \text{tc}(v')$ 
           $\text{tr}'[p_v] = L_v(\alpha)$ 

func GenerateGraph( $o, o'$ )
   $G = \text{empty graph}$ 
  foreach  $\alpha$  in  $o$ 
    Add  $v_\alpha$  to  $G$ 
  foreach  $\alpha$  in  $o'$ 
    Add  $v_\alpha$  to  $G$ 
  foreach  $\alpha \in o'$ 
    foreach  $\alpha' \in o$ 
      if  $\neg \text{ElsePrevented}(\alpha, \alpha')$ 
        add edge  $(v_\alpha, v_{\alpha'})$  to  $G$ 
    foreach  $\alpha' \in o'$ 
      if  $\neg \text{ElsePrevented}(\alpha, \alpha')$ 
        add edge  $(v_\alpha, v_{\alpha'})$  to  $G$ 
  return  $G$ 

func ElsePrevented( $\alpha, \alpha'$ ) returns bool
  foreach  $v \leftarrow p_\alpha$  foreach  $v' \leftarrow p_{\alpha'}$ 
    if  $v = v' \wedge p_\alpha(v) \neq p_{\alpha'}(v')$ 
      then return true
    //uncomment below if mutexes used
    //if  $v \neq v' \wedge \text{mutex}\{v = p_\alpha(v), v' = p_{\alpha'}(v')\}$ 
    // then return true
  foreach  $v \leftarrow e_\alpha$  and foreach  $v' \leftarrow e_{\alpha'}$ 
    if  $v = v' \wedge (e_\alpha(v) \neq e_{\alpha'}(v') \vee L_v(\alpha) \neq L_{v'}(\alpha'))$ 
      then return true
    //uncomment below if mutexes used
    //if  $v \neq v' \wedge \text{mutex}\{v = e_\alpha(v), v' = e_{\alpha'}(v')\}$ 
    // then return true
  return false

```

---

problem optimally is in general very hard. So we use the well-known greedy algorithm Welsh-Powell (Welsh & Powell, 1967) for colouring the graph.

To summarise, after all the transition constraints are constructed, to prevent conflicting actions from happening at the same time, parallelism variables will be added to the transition tables. Therefore, in addition to the  $(m+1)|\mathcal{V}|$  CSP variables we mentioned before we may have  $m|\mathcal{V}_p|$  CSP variables for parallelism variables where  $\mathcal{V}_p$  is the set of parallelism variables.  $|\mathcal{V}_p|$  will be at most equal to  $|\mathcal{V}|$  but it is practically much smaller than that.

### 5.5 Mutex Constraints

During translation from PDDL to SAS<sup>+</sup>, the translator (Helmert, 2006), along with the SAS<sup>+</sup> representation, also produces the mutex-groups. The mutex-groups are sets of variable-

value pairs that cannot occur at the same time. Mutexs are produced as part of the identification of monotonicity invariant candidates (Helmert, 2009). In our driverlog example, we have a **mutex-group**  $\{(d\text{-loc} = t), (t\text{-occ} = \perp)\}$ , which denotes that when the variable  $d\text{-loc}$  takes the value  $t$ , the variable  $t\text{-occ}$  cannot take the value false and vice versa. In blocks world example, we have a **mutex-group**  $\{(A\text{-top} = C), (A\text{-loc} = H), (B\text{-loc} = \text{onA})\}$ . These three variable-value pairs are mutually exclusive. To ensure that no pair from these **mutex-group** occurs at the same time, we use **mutex** constraints, which are 2-ary negative table constraints. A **mutex-group** that involves only one variable could actually be ignored because such a mutual exclusion condition is already captured by the characteristics of a multi-valued variable. By **mutex-groups**, we therefore mainly mean **mutex-groups** that involve more than one variable. To summarise, for each **mutex** pair  $\{(v = k), (\bar{v} = \bar{k})\}$  in **mutex-groups**, we add a negative **mutex** table  $\text{mc}(v, k, \bar{v}, \bar{k}, \tau)$  with columns  $v^\tau, \bar{v}^\tau$  and a row with column values  $k, \bar{k}$ . Figure 15 represents the negative **mutex** tables for the above **mutex-group** from the blocks world domain. Note that these tables do not necessarily have just one row. We may have other **mutex** groups that insert other rows for these tables. For instance, we have two other **mutex-groups** in our example from the blocks world  $\{(B\text{-top} = C), (A\text{-loc} = \text{onB}), (B\text{-loc} = H)\}$  and  $\{(\text{hand} = E), (A\text{-loc} = H), (B\text{-loc} = H)\}$  that insert two other rows  $\langle \text{onB}, H \rangle$  and  $\langle H, H \rangle$  to the **mutex** table for  $A\text{-loc}^\tau$  and  $B\text{-loc}^\tau$ .

$A\text{-top}^\tau$	$A\text{-loc}^\tau$	$A\text{-top}^\tau$	$B\text{-loc}^\tau$	$A\text{-loc}^\tau$	$B\text{-loc}^\tau$
C	H	C	onA	H	onA

Figure 15: Mutex constraints for a sample mutex group from blocks world domain.

Note that **mutex-groups** are used optionally in our model. When used they could lead to more efficient search performance. Also note that when **mutex-groups** are used, we can reduce the domain size of parallelism variables. A **mutex** group could subsume a parallel conflict and thus the respective edge could be removed from a parallel conflict graph. This might result in a better graph colouring of the parallel conflict graph in terms of the number of colours used. Refer to the commented lines in Function `ElsePrevented` in Algorithm 5; the commented pseudocode lines are to be uncommented to obtain this effect.

## 5.6 Decoding CSP Solutions to Plans

After encoding a planning problem of a given makespan as a CSP, we solve the CSP using a CSP solver. If the problem is satisfiable, the CSP solver will output the solution as a complete assignment to the variable  $v^\tau$ s and  $p_v^\tau$ s. We then need to extract the final plan from the assignment. Algorithm 6 describes the procedure. At each time step, if the value of  $v$  is changed from  $k$  at time step  $\tau$  to  $k'$  at time step  $\tau + 1$ , we need to check the actions on the labels of the edges from  $k$  to  $k'$  along with edges from *don't-care* to  $k'$  in the DTG of  $v$ . If the value of  $v$  is the same value  $k'$  at time steps  $\tau$  and  $\tau + 1$ , we only need to check the actions on the labels of the edges from *don't-care* (note that every value can be considered as *don't-care*) to  $k'$  in the DTG of  $v$ . From these sets of actions, those whose preconditions are satisfied at  $\tau$  and whose effects are satisfied at  $\tau + 1$  are the actions that are responsible for the change. Note that we can not have two actions changing the value of  $v$  from  $k$  to  $k'$ , because we are preventing them from happening at the same time by using parallelism

variables  $p_v$ s. So if we have more than one candidate action that changes the value of  $v$  from  $k$  to  $k'$ , and the preconditions and effects are satisfied, we can easily select one of them based on the value of parallelism variable  $p_v$  in the transition table of  $v$ . However, we may have more than one action changing the value of  $v$  from *don't-care* to  $k'$ . Since these actions are not conflicting with each other, if their preconditions and effects are satisfied and the values of relevant parallelism variables match with the corresponding transition table cell values, we select all of them.

---

**Algorithm 6** Decoding CSP Solution to Plan

---

```

input  $m$ : makespan value when the CSP solution is found
input  $\{(v^\tau = k)\}$  for each  $\tau \in [0, m]$  and for each state variable  $v$ : CSP solution
input  $\{(p^\tau = l)\}$  for each  $\tau \in [0, m)$  and for each parallelism variable  $p_v$ : CSP solution
output: parallel plan  $\Pi = \langle A^1, \dots, A^m \rangle // A^\tau$  actions at  $\tau$ 
foreach time step  $\tau = 1$  to  $m$  // number of action layers
  foreach  $v$  at time steps  $v^{\tau-1}$  and  $v^\tau$ 
    If  $v^{\tau-1}$  and  $v^\tau$  have different values
      foreach edge  $\langle v^{\tau-1}, v^\tau \rangle$  with label  $\langle \alpha, \rho, \epsilon \rangle$  in  $\text{DTG}(v)$ 
        if  $\forall(\bar{v} \Leftarrow \rho)[\rho(\bar{v}) = \bar{v}^{\tau-1}]$  and  $\forall(\bar{v} \Leftarrow \epsilon)[\epsilon(\bar{v}) = \bar{v}^\tau]$  and  $p_v^\tau = L_v(\alpha)$ 
          add  $\alpha$  to  $A^\tau$ ; break // only one such action exists
        // in the following, don't care  $\times$  could be equal or not equal to  $v^\tau$ 
      foreach edge  $\langle \times, v^\tau \rangle$  with label  $\langle \alpha, \rho, \epsilon \rangle$  in  $\text{DTG}(v)$ 
        if  $\forall(\bar{v} \Leftarrow \rho)[\rho(\bar{v}) = \bar{v}^{\tau-1}]$  and  $\forall(\bar{v} \Leftarrow \epsilon)[\epsilon(\bar{v}) = \bar{v}^\tau]$  and  $\forall(\bar{v} \Leftarrow \rho \cup \epsilon)p_v^\tau = L_{\bar{v}}(\alpha)$ 
          add  $\alpha$  to  $A^\tau$  // we allow all such actions, some may be redundant

```

---

**Lemma 6** (CSP to Plan). *There could be more than one parallel plans for each CSP solution returned. However, we generate just one plan, which may include redundant actions.*

*Proof.* The proof is straightforward from Algorithm 6. For a given  $v$ , when we extract actions that are on the edges from the *don't care* vertex to a value  $k'$ , we include all actions that satisfy the preconditions and effects. By selecting all of those actions, we might include some redundant actions in our plan. For example, the redundancy may arise when one such action's preconditions and effects are respectively subsets of those of another such action. Redundant actions may also come when an action changes the values of variables, and those variables and the values are neither required as preconditions of any latter actions in the plan nor are specified in the goal condition. However, we do not have any concern about this, since our objective is to produce plans of optimal makespan and not of optimal plan length. Of course a post-processing step could eliminate the redundant actions if a shorter plan is desired. Thus, multiple plans are possible from the same CSP solution.  $\square$

## 5.7 Correctness and Complexities

We prove the correctness of our encoding and decoding algorithms showing how they correspond to the DTG-based planning formulation. In Lemma 4, we already have shown the soundness, completeness and optimality of the DTG-based planning formulation.



**Lemma 7** (TCPP Model). *There exists a parallel plan of a given makespan if and only if the CSP model constructed by Algorithm 3 along with Algorithms 4 and 5 is satisfiable. A parallel plan can be correctly decoded from the CSP solution by using Algorithm 6.*

*Proof.* For the if part, assume a CSP solution is found and we will show that a parallel plan exists. To show the existence, we rather generate such a parallel plan; which also proves the last part of the lemma. Given a CSP solution to our model, we have values of all  $v^\tau$ s and all  $p_v^\tau$ s. From the changes of a variable  $v$  from  $v^\tau$  to  $v^{\tau+1}$ , we can easily identify corresponding actions from the corresponding DTG of  $v$  and using  $p_v^\tau$ s we can also identify which of the conflicting actions is actually to be extracted. Algorithm 6 exactly finds these actions and generates a sequence of sets of actions  $\Pi = \langle A^1, \dots, \dots, A^m \rangle$ . Although Algorithm 6 does not extract the *no-ops*, we can optionally include them in the sets of actions as well. Now, if the CSP solver does return a solution, the solution certainly satisfies the initial state and the goal constraints. From the values of  $v^\tau$ s and  $v^{\tau+1}$ s, we can easily get a sequence of states  $\langle s_0, s_1, \dots, s_m \rangle$ . Each transition table implements application of each action appearing on the edges of a DTG. So all transition tables at all timestep together implement  $\text{app}(s_{\tau-1}, A^\tau) = s_\tau$ . Lastly, no two actions  $\alpha \otimes \alpha'$  are in a set of actions  $A^\tau$  since values assigned to the parallelism variables inherently prevented this. As per Lemma 3, the generated sequence of sets of action  $\Pi$  is therefore a parallel plan.

For the only if part, assume a parallel plan exists and for this we will show that we have a satisfiable CSP model. Let  $\Pi = \langle A^1, \dots, \dots, A^m \rangle$  be such a parallel plan for the multi-valued planning task  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ . As per Lemma 3, we therefore have a sequence of states  $\langle s_0, s_1, \dots, s_m \rangle$  such that (i)  $s_0 = \mathcal{I}$ , (ii) for all  $\tau \in [1, m]$ ,  $\text{app}(s_{\tau-1}, A^\tau) = s_\tau$ , (iii) for all  $\tau \in [1, m]$ , there exist no two actions  $\alpha, \alpha' \in A^\tau$  such that  $\alpha \otimes \alpha'$ , and (iv)  $\mathcal{G}(v) = s_m(v)$  for each variable  $v \in \mathcal{G}$ . In our CSP model, for each state variable  $v \in \mathcal{V}$  and for each time step  $0 \leq \tau \leq m$ , we have a CSP variable  $v^\tau$ , which has the same domain as the state variable  $v$  has. Thus, a set of variables  $\{v^\tau : v \in \mathcal{V}\}$  represents a state  $s_\tau$  for any given  $\tau \in [0, m]$ . Now considering the assignment of values to  $v^\tau$ s and  $p_v^\tau$ s where  $v^\tau = s_\tau(v)$  and  $p_v^\tau = L_\alpha(v)$ , our initial state and goal constraints are satisfied trivially. Our transition constraint table for each  $v$  and so  $v^\tau$  contains rows for all edges in  $\text{DTG}(v)$  (including loops) and so for the actions (including *no-ops*) appearing on the edges. The rows in the table using other  $\bar{v}^\tau$ s and  $\bar{v}^{\tau+1}$ s completely describe all the actions in the DTGs. Thus, for a given set of actions  $A^\tau$  at time step  $\tau$ , since  $A^\tau$  contains at least one action from each *DTG* (may include *no-ops*), we must be able to find some rows in all the table constraints and thus all the table constraints will be satisfiable. Note that when an action is selected from a transition table constraint as a row, it will be selected from all other tables in which it appears. In all rows in different tables corresponding to this action, the value of common parallelism variables will be the same so all these table constraints are satisfied. All these prove that our CSP model is satisfiable. A complete CSP search is therefore able to find a CSP solution.  $\square$

We also analyse the theoretical time complexity of our encoding and decoding procedures although these are practically negligible compared to the search times.

**Lemma 8** (Encoding Complexity). *Transition and mutex constraints are all constructed once and used for each time step. In Algorithm 3, construction of*

1. all initial state and goal constraints need  $O(|\mathcal{V}|)$  time and space.
2. all transition constraints by using Algorithm 4 require  $O(\widehat{E}^2|\mathcal{V}|\widehat{V}_a)$  time and space;
3. adding parallelism variables to transitions constraints by using Algorithm 5 requires  $O(|\mathcal{V}|\widehat{E}^2 + \widehat{M}\widehat{V}_m)$  time and  $O(|\mathcal{V}|\widehat{V}_p\widehat{E})$  space;
4. all mutex constraints require  $O(\widehat{M}\widehat{V}_m^2)$  time and space;

where

$\widehat{E}$  is the maximum number of edges in a DTG;

$\widehat{V}_a$  is the maximum number of variables appearing in the preconditions and the effects of an action (i.e. the maximum parameter size of an operator);

$\widehat{V}_p$  is the maximum number of parallelism variables;

$\widehat{M}$  is the number of mutex-groups; and

$\widehat{V}_m$  is the maximum number of variable-value pairs in a mutex-group.

*Proof.* We analyse the algorithms to find the time and space complexities.

1. Initial states are fully specified while the goals could be partially specified.
2. In Algorithm 4, the maximum number of columns possible in a table is  $2\widehat{E}\widehat{V}_a$ . This is because there are maximum  $\widehat{E}$  edges in a DTG and each edge on its label has an action that has at most  $V_a$  variable in its preconditions and effects. The maximum number of rows possible in a table is  $\widehat{E}$ . Lines 2-3 in Algorithm 4 take  $O(\widehat{E}\widehat{V}_a)$  times (i.e. the number of columns). Moreover, the loop in Line 4 runs  $O(|\widehat{E}|)$  times, each iteration taking  $O(\widehat{E}\widehat{V}_a)$  time to fill in the table row. There are  $|\mathcal{V}|$  state variables and we need a table for each variable's DTG.
3. In Algorithm 5, the first loop in Procedure `ParallelismConstraints` needs  $|\mathcal{V}|$  times. Function `GenerateGraph` considers each pair of edges from  $k$  to  $k'$  and from *don't-care* to  $k'$  so its complexity is quadratic in number of edges. Therefore, generating conflicting graphs for all values in the DTG in the worst case is  $O(\widehat{E}^2)$ . Also the complexity of Function `LabelGraph` which uses Welsh-Powel colouring algorithm is linear in the number of vertexes. Since the vertexes of the conflicting graphs are actions or edges of DTG, the worst case complexity of labelling all conflicting graphs in a DTG is  $O(\widehat{E})$ . Function `ElsePrevented` needs  $\widehat{V}_a^2$  time for the two-level nested loops and  $O(\widehat{M}\widehat{V}_m)$  to construct a look-up table and  $O(\widehat{M})$  time to determine whether two given variable-value pairs are mutex in the lookup table. The look-up table comprises for each variable-value pair a list of *mutex-groups* where the pair appears. Lastly, assigning labels to the parallelism variable in rows needs  $\widehat{E}$  time. There are  $\widehat{V}_p$  parallelism variables which may appear in the worst case in all transition tables and each table has at most  $\widehat{E}$  rows. So we need  $O(|\mathcal{V}|\widehat{V}_p\widehat{E})$  space.

4. We need  $O(\widehat{M}\widehat{V}_m^2)$  time and space to construct the mutex constraint tables, each of them has 2 columns and 1 row.

Given the same number ( $|\mathcal{V}|$ ) of CSP variables at each time step, we need to refer to the three types of table in each of the  $m$  time steps when  $m$  is the given makespan.  $\square$

**Lemma 9** (Decoding Complexity). *The time taken to decode CSP solution to plan is  $O(m|\mathcal{V}|\widehat{E}_m\widehat{V}_a)$  where  $m$  is the given makespan,  $\mathcal{V}$  is the set of state variables,  $\widehat{E}_m$  is the maximum number of edges (multi-edge) between two vertexes in a DTG, and  $\widehat{V}_a$  is the maximum number of variables appearing in the preconditions and effects of an action.*

*Proof.* The proof is straightforward from the loops in Algorithm 6. For each makespan and for each state variable, we select an edge among  $\widehat{E}_m$  edges between corresponding vertexes and selecting them needs checking  $\widehat{V}_a$  state variables and at most  $\widehat{V}_a$  parallelism variables.  $\square$

## 6. PaP2 Reconstruction

To compare our constraint model with PaP2’s model, we reconstruct PaP2 and name it PaPR. To encode the DTG-based planning formulation as CSP, we need to encode the relationships between the state variables and the actions at consecutive time steps. In our TCPP model, we ignore the actions and our *transition constraints* directly encode the relationships between the state variables in two consecutive time steps by using the state transitions in the DTGs. Moreover, we use parallelism variables to specify which action pairs cannot be in parallel with each other in the same time step. In the PaP2 model (Barták, 2011b), the main constraints, we refer to them by *action succession constraints*, encode the relationships between actions at two consecutive time steps ignoring the state variables connected by them. The relationships between actions and their preconditions and effects are encoded by another type of constraints named *synchronisation constraints*. Nevertheless, we also significantly enhance PaPR by using *don’t cares* instead of *sets* and also using mutex-groups.

### 6.1 Constructing FSA

In PaP2 (Barták, 2011b), the action succession constraints are actually encoded by using FSA generated from a multi-valued representation of the problem. These FSA are similar to the DTGs and they represent how the state variables change their values. The difference is that unlike DTG’s in which there are conditions on the edges, in these automata only actions are represented on the edges. When an action changes the value of a variable from  $k$  to  $k'$ , it occurs on the edge from vertex  $k$  to vertex  $k'$ . Also, if a state variable is only on the effects of an action, that action occurs on edges from all values to the value in the effect. The other difference is that in the FSA, there are loops for every vertex in the graph denoting the *no-ops* for each value of the state variable. We provide the formal definition of FSA in Definition 16 and also show FSA for the driverlog example in Figure 16.

**Definition 16** (Finite State Automaton). *Assume  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$  be a multi-valued planning task. The finite state automaton  $\text{FSA}(v)$  of  $v \in \mathcal{V}$  is an edge-labelled directed graph with the vertexes  $\mathcal{D}(v)$  and the following edges:*

1. For each action  $\alpha \in \mathcal{A}$ , there will be an edge  $\langle k, k' \rangle$  from  $k$  to  $k'$  with label  $\alpha$ , if  $v \in \text{PandE}(\alpha)$ , and  $p_\alpha(v) = k$  and  $e_\alpha(v) = k'$ .
2. For each action  $\alpha \in \mathcal{A}$ , there will be an edge  $\langle k, k' \rangle$  from each  $k \in (\mathcal{D}(v) \setminus \{k'\})$  to  $k'$  with label  $\alpha$ , if  $v \in \text{EnotP}(\alpha)$  and  $e_\alpha(v) = k'$ .
3. For each value  $k \in \mathcal{D}(v)$ , there will be a loop  $\langle k, k \rangle$  from vertex  $k$  to vertex  $k$  to represent a no-op with label  $\alpha$  where for each loop,  $\alpha$  is a distinct no-op action with preconditions  $\{(v = k)\}$  and no effects.

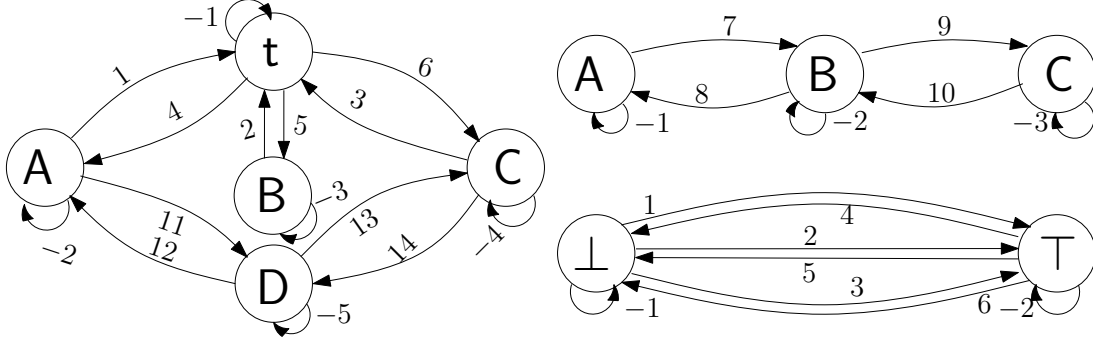


Figure 16: FSA for the driverlog problem in Figure 1. Left: FSA for state variable **d-loc**, Right-Top: FSA for state variable **t-loc**, and Right-Bottom: FSA for state variable **t-occ**. In the labels of the edges the positive numbers are the action identification numbers from Table 1 and the negative numbers are distinct identifiers for no-ops or loops. Boolean values are true  $\top$  and false  $\perp$ .

## 6.2 Encoding FSA into CSP

In PaP2, for each state variable  $v$  at time  $\tau$ , a CSP variable  $v^\tau$  is considered. The domain of  $v^\tau$  comprises all the actions occurring on the edges of  $\text{FSA}(v)$ . These actions include the *no-ops* as well. The FSA are encoded with four kinds of constraint: *initial constraints*, *goal constraints*, *action succession constraints*, and *synchronisation constraints*. In the original PaP2, auxiliary constraints are used to specify that at each time step, at least one non-loop action should be selected; we do the same in PaPR, our reconstructed version of PaP2. Although not used in the original PaP2, in our enhanced version of PaPR, wherever possible we use *don't cares* instead of sets and also optionally use constraints encoding mutex-groups produced by the PDDL-to-SAS<sup>+</sup> translator. Algorithm 7 describes PaPR's encoding procedure.

## 6.3 Initial State and Goal Constraints

To encode the initial state and the goals of the planning problem, PaP2 needs to add constraints restricting the actions to be selected at time 1 and  $m$ . Since the initial state is fully specified, we need  $|\mathcal{V}|$  table constraints listing the actions that are allowed in each variable's FSA at time 1. These actions are on the outgoing edges of the vertex representing

**Algorithm 7** FSA To CSP Encoding

---

```

1 Planing Problem  $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ , makespan  $m$ 
2 //variables:  $v, v'$ , values:  $k, k'$ 
3 Foreach  $(v, k) \in \mathcal{I}$ , Add constraint  $\text{ic}(v^1, k)$ 
4 Foreach  $(v, k) \in \mathcal{G}$ , Add constraint  $\text{gc}(v^m, k)$ 
5 For timestep  $\tau = 1$  to  $m$  do
6   Foreach state variable  $v \in \mathcal{V}$ 
7     // asc below is added up to  $\tau = m - 1$ 
8     Add action succession constraint  $\text{asc}(v^\tau, v^{\tau+1})$ 
9     Add synchronisation constraint  $\text{sc}(v^\tau)$ 
10    Add auxiliary constraint  $\bigvee_{v \in \mathcal{V}} (v^\tau \geq 0)$  //non-loop
11    //mutex constraints below are optional for enhancement
12    Foreach pair  $\{v = k, v' = k'\}$  in mutex groups
13      Add a mutex constraint  $\text{mc}(v, k, v', k', \tau)$ 

```

---

the initial value of the variable. For the goals, we only have the values for some of the state variables. So we need at most  $|\mathcal{V}|$  table constraints listing actions that are allowed at time  $m$ . These actions are on the incoming edges of the vertex representing the value of the variable in a goal.

#### 6.4 Action Succession Constraints

The action succession constraints are represented by binary constraints between CSP variables for a state variable in two consecutive times and list all the action pairs that can occur consecutively in the automaton. These constraints are easily constructed from the automata of the state variables by examining all pairs of consecutive edges possible in the automata. Note that actions are in the labels of the edges in a FSA. Nevertheless, action  $\alpha$  can be *followed by* another action  $\alpha'$  if for all  $v \in \mathcal{V}$ ,  $(v \Leftarrow e_\alpha \wedge v \Leftarrow p_{\alpha'})$  logically implies  $e_\alpha(v) = p_{\alpha'}(v)$  i.e. preconditions of  $\alpha'$  are compatible with the effects of  $\alpha$ . Algorithm 8 constructs the table for the action succession constraint between variables  $v^\tau$  and  $v^{\tau+1}$ . For a given FSA, all actions  $\alpha'$  that can follow a given action  $\alpha$  are however stored as a set in the same cell of the table to obtain a compact representation.

**Algorithm 8** Action Succession Constraint  $\text{asc}(v^\tau, v^{\tau+1})$  using a table

---

```

1  $T^\tau$ : action succession constraint table for  $v^\tau, v^{\tau+1}$ 
2 Columns of  $T^\tau$  are  $v^\tau$  and  $v^{\tau+1}$  to hold actions
3 foreach edge  $\langle k, k' \rangle$  in  $\text{FSA}(v)$  with label  $\alpha$ 
4   tr: a new row in table  $T^\tau$ 
5    $E$ : all edges outgoing from  $k'$ 
6   foreach edge  $\langle k', k'' \rangle \in E$  with label  $\alpha'$ 
7     //in the line below if  $\alpha$  cannot be followed by  $\alpha'$ 
8     if  $\exists (v \in \mathcal{V}) [v \Leftarrow e_\alpha \wedge v \Leftarrow p_{\alpha'} \wedge e_\alpha(v) \neq p_{\alpha'}(v)]$ 
9        $E = E \setminus \{ \langle k', k'' \rangle \}$ 
10   $\text{tr}[v^\tau] = \alpha, \text{tr}[v^{\tau+1}] = \{ \alpha' : \alpha' \text{ is on an edge in } E \}$ 

```

---

Figure 17 shows the action succession constraints obtained for the FSA in Figure 16 for the driverlog example. In the first row of the table for d-loc in Figure 17, we see action 1 can be followed by actions -1, 4, 5, 6. Verify these from the FSA of d-loc in Figure 16 and the action names in Table 1. Also, verify other rows and tables in a similar way.

ASC for d-loc	
d-loc	d-loc
$\tau$	$\tau + 1$
1	$\{-1,4,5,6\}$
2	$\{-1,4,5,6\}$
3	$\{-1,4,5,6\}$
4	$\{-2,1,11\}$
5	$\{-3,2\}$
6	$\{-4,3,14\}$
11	$\{-5,12,13\}$
12	$\{-2,1,11\}$
13	$\{-4,3,14\}$
14	$\{-5,12,13\}$
-1	$\{-1,4,5,6\}$
-2	$\{-2,1,11\}$
-3	$\{-3,2\}$
-4	$\{-4,3,14\}$
-5	$\{-5,12,13\}$

ASC for t-occ	
t-occ	t-occ
$\tau$	$\tau + 1$
1	$\{-2,4,5,6\}$
2	$\{-2,4,5,6\}$
3	$\{-2,4,5,6\}$
4	$\{-1,1\}$
5	$\{-1,2\}$
6	$\{-1,3\}$
-1	$\{-1,1,2,3\}$
-2	$\{-2,4,5,6\}$

ASC for t-loc	
t-loc	t-loc
$\tau$	$\tau + 1$
7	$\{-2,8,9\}$
8	$\{-1,7\}$
9	$\{-3,10\}$
10	$\{-2,8,9\}$
-1	$\{-1,7\}$
-2	$\{-2,8,9\}$
-3	$\{-3,10\}$

Figure 17: Action succession constraints from the FSA in Figure 16 for driverlog example.

## 6.5 Synchronisation Constraints

The synchronisation constraints ensure that if an action changes several state variables then the same action must be selected in each of the respective FSA. Also, if a variable appears only in the preconditions of an action, we need to be sure that a *no-op* action is selected in the corresponding FSA. A synchronisation constraint is considered for each state variable  $v$  and for each time  $\tau$  and is encoded as a table. For all variables  $v$ , this constraint will have a column  $v^\tau$ . Each edge in the FSA corresponds to one row in the table.

Algorithm 9 describes the procedure used to construct the synchronisation constraint tables. Let  $\alpha$  be one of the edges of the FSA of variable  $v$ . Then, in the corresponding row in the synchronisation table for  $v$ , the value of  $v$  and all variables occurring in effects of this action will have value  $\alpha$ . Variables just in the preconditions of  $\alpha$  will have the corresponding *no-op* as their values. All the other columns can have any values from their domains. At the end, some columns from the synchronisation table could be removed, particularly those whose corresponding variable can take any value from its domain.

Figure 18 shows the tables for the synchronisation constraints obtained from the FSA in Figure 16 for the driverlog example. Look at the first row of the table for d-loc. When action 1 *embark-truck* (see Table 1) takes place at location A, both d-loc and t-occ change their values but t-loc does not. The negative number in t-loc indicates the corresponding

**Algorithm 9** Synchronisation Constraint  $\text{sc}(v^\tau)$  using a table

---

```

1   $T^\tau$ : synchronisation table for variable  $v^\tau$ 
2   $\mathcal{D}(v)$ : the domain of the state variable  $v$ 
3  foreach  $\bar{v} \in \mathcal{V}$ , table  $T^\tau$  has a column  $\bar{v}^\tau$ 
4  foreach edge in  $\text{FSA}(v)$  with label  $\alpha$ 
5      tr: a new row in  $T^\tau$ , each column  $\bar{v}^\tau$ 
6          has initially value  $\mathcal{D}(\bar{v})$ , but  $\text{tr}[v^\tau] = \alpha$ 
7      foreach  $\bar{v} \in \mathcal{V}$  with  $\bar{v} \Leftarrow p_\alpha \wedge \bar{v} \not\Leftarrow e_\alpha$ 
8           $\text{tr}[\bar{v}] =$  the no-op at vertex  $p_\alpha(\bar{v})$  in  $\text{FSA}(\bar{v})$ 
9      foreach  $\bar{v} \in \mathcal{V}$  with  $\bar{v} \Leftarrow e_\alpha$ ,  $\text{tr}[\bar{v}] = \alpha$ 
10 delete all columns  $v^\tau$  with all cells having values  $\mathcal{D}(v)$ 
11 merge rows into one if they differ only in one column

```

---

SC for d-loc		
d-loc	t-loc	t-occ
$\tau$	$\tau$	$\tau$
1	-1	1
2	-2	2
3	-3	3
4	-1	4
5	-2	5
6	-3	6
$S_4$	$S_2$	$S_3$

SC for t-occ		
t-occ	d-loc	t-loc
$\tau$	$\tau$	$\tau$
1	1	-1
2	2	-2
3	3	-3
4	4	-1
5	5	-2
6	6	-3
$S_5$	$S_1$	$S_2$

SC for t-loc	
t-loc	d-loc
$\tau$	$\tau$
7	-1
8	-1
9	-1
10	-1
$S_6$	$S_1$

Figure 18: Synchronisation constraints from the FSA in Figure 16 for driverlog example. In the tables,  $S_1 = \{-5, \dots, -1, 1, \dots, 6, 11, \dots, 14\}$ ,  $S_2 = \{-3, -2, -1, 7, \dots, 10\}$ ,  $S_3 = \{-1, -2, 1, \dots, 6\}$ ,  $S_4 = \{-5, \dots, -1, 11, \dots, 14\}$ ,  $S_5 = \{-1, -2\}$ , and  $S_6 = \{-1, -2, -3\}$  are sets used as cell values.

*no-op* action in the FSA of t-loc, while positive numbers in d-loc and t-occ denote the action 1 in the both variables' FSA. Other rows in the table and other tables could be explained in the same way. Notice that certain cells have a set of values such as  $S_1$ ,  $S_2$  and  $S_3$ . PaP2 uses sets to encode them, but these sets are actually the entire domains of the corresponding variables. So one can replace them with *don't cares*  $\times$ . We do that in PaPR to get a variant and to see the effect on the search performance.

## 6.6 Mutex Constraints

For each mutex pair  $\{(v = k), (\bar{v} = \bar{k})\}$  in *mutex-groups*, a negative mutex table  $\text{mc}(v, k, \bar{v}, \bar{k}, \tau)$  with columns  $v^\tau, \bar{v}^\tau$  is added. The rows in this table will be the pairs of actions  $\alpha, \bar{\alpha}$  where  $\alpha$  is on the label of an outgoing edge of vertex  $k$  in  $\text{FSA}(v)$  and  $\bar{\alpha}$  is on the label of an outgoing edge of vertex  $\bar{k}$  in  $\text{FSA}(\bar{v})$ . The table also has the rows for actions pairs  $\alpha, \bar{\alpha}$

where  $\alpha$  is on the label of an incoming edge of vertex  $k$  in  $\text{FSA}(v)$  and  $\bar{\alpha}$  is on the label of an incoming edge of vertex  $\bar{k}$  in the  $\text{FSA}(\bar{v})$ . Algorithm 10 describes the procedure to construct such a negative table. Note that to reduce the numbers of these negative tables, mutex tables having the same columns are merged into one table.

---

**Algorithm 10** Mutex Constraint  $\text{mc}(v, k, \bar{v}, \bar{k}, \tau)$  using a negative table

---

```

1   $T^\tau$ : mutex constraint table with columns  $v^\tau$  and  $\bar{v}^\tau$ 
2  for all edges  $\langle k, k' \rangle$  in  $\text{FSA}(v)$  with label  $\alpha$ 
3    for all edges  $\langle \bar{k}, \bar{k}' \rangle$  in  $\text{FSA}(\bar{v})$  with label  $\bar{\alpha}$ 
4      tr: a new row in  $T^\tau$  with  $\text{tr}[v^\tau] = \alpha$ ,  $\text{tr}[\bar{v}^\tau] = \bar{\alpha}$ 
5  for all edges  $\langle k', k \rangle$  in  $\text{FSA}(v)$  with label  $\alpha$ 
6    for all edges  $\langle \bar{k}', \bar{k} \rangle$  in  $\text{FSA}(\bar{v})$  with label  $\bar{\alpha}$ 
7      tr: a new row in  $T^\tau$  with  $\text{tr}[v^\tau] = \alpha$ ,  $\text{tr}[\bar{v}^\tau] = \bar{\alpha}$ 

```

---

## 6.7 Decoding CSP Solutions to Plans

In PaP2 model, the CSP solution returned is itself the parallel plan. This is because in the CSP solution, the value of each variable is an action, although we have to ignore the *no-ops*, if any. We have a CSP variable for each state variable and for each time step  $\tau \in [1, m]$ . So the decoding step has only the cost of removing the *no-ops*.

## 6.8 Reconstructed PaP2 on Minion

In PaP2 implementation, the table constraints allow sets of values in their cells and the SICStus solver that has been used supports this kind of table constraints. However, as we have seen in some of the synchronisation constraint tables, the sets could be replaced by *don't cares*. In PaPR, we use only sets and then sets plus *don't cares* to compare the performances with the difference in the encodings. The other difference between our PaPR and PaP2 is in the auxiliary constraints. In PaP2, these constraints are modelled by “greater than zero” constraints. Since the *no-ops* have values less than zero, the constraints are that maximum of the values for CSP variables at each time step should be greater than or equal to zero; which enforces that at least one positive value (not *no-op* i.e. a regular action) should be selected for each time step. While using Minion, we could not model this constraint this way and we used or-constraint to model it. For each time step, we use an or-constraint that enforces at least one of the CSP variables for actions should take positive value i.e. a regular action is selected. Other than the difference in the auxiliary constraints, in terms of the constraints used and their table constraint forms, PaPR constraint model very closely matches with the PaP2 constraint model. While running PaPR, we use different variable selection heuristics and constraint propagation techniques to compare performances of the resulting variants.

## 6.9 Correctness and Complexities

Our modification to PaP2 model is to use *don't cares* instead of sets whenever possible, and optionally use mutex constraints to improve the performance. We refer (Barták, 2011b) for



the correctness of the basic PaP2 model. Below we provide the theoretical time and memory complexity of our PaPR encoding method, although the times are practically negligible compared to the search times.

**Lemma 10** (Encoding Complexity). *Action succession, synchronisation, auxiliary, and mutex constraints are all constructed once and used for each time step. In Algorithm 7, construction of*

1. *all initial state and goals require  $O(|\mathcal{V}|\widehat{E}_d)$  time and space.*
2. *all action succession constraints by using Algorithm 8 require  $O(\widehat{E}\widehat{E}_d\widehat{V}_a^2|\mathcal{V}|)$  time and  $O(\widehat{E}\widehat{E}_d|\mathcal{V}|)$  space;*
3. *all synchronisation constraints by using Algorithm 9 require  $O(\widehat{E}|\mathcal{V}|^2)$  time and space;*
4. *all auxiliary constraints require  $O(|\mathcal{V}|)$  time, no space as given in analytical form.*
5. *all mutex constraints require  $O(\widehat{M}\widehat{V}_m^2\widehat{E}_d^2)$  time and space;*

where

$\widehat{E}$  is the maximum number of edges in an FSA;

$\widehat{E}_d$  is the maximum degree of a vertex in an FSA.

$\widehat{V}_a$  is the maximum number of variables appearing in the preconditions and the effects of an action (i.e. the maximum parameter size of an operator);

$\widehat{M}$  is the number of mutex-groups; and

$\widehat{V}_m$  is the maximum number of variable-value pair in a mutex-group.

*Proof.* We analyse the algorithms to find the time and space complexities.

1. For each  $(v = k)$  in  $\mathcal{I}$  (or  $\mathcal{G}$ ), we list all the actions in outgoing (or incoming) edges from vertex  $k$  of  $\text{FSA}(v)$ .
2. Line 3 in Algorithm 8 runs  $O(\widehat{E})$  time, Line 6 runs  $O(\widehat{E}_d)$  time, and Line 8 needs  $O(\widehat{V}_a^2)$  time. The number of columns in an action succession constraint table is 2 while the number of rows is  $\widehat{E}$ . However, each table cell can have at most  $\widehat{E}_d$  numbers as a set. We need an action succession constraint for each state variable.
3. In Algorithm 9, a synchronisation table has  $|\mathcal{V}|$  columns and  $\widehat{E}$  rows. Computing each row needs  $O(|\mathcal{V}|)$  time.
4. For auxiliary constraints, we need to write an analytical formula involving each variable  $v^\tau$  in the same time step.
5. For each pair  $\{(v = k), (v' = k')\}$ , we have a table with 2 columns and  $\widehat{E}_d^2$  rows.

Given the same number  $(|\mathcal{V}|)$  of CSP variables at each time step, we need to refer the four types of constraint in each of the  $m$  time step when  $m$  is the given makespan.  $\square$

## 7. Experimental Results

We ran all experiments reported in this paper on the same high performance computing cluster *Gowonda* at *Griffith University*. Each node of the cluster is equipped with Intel Xeon CPU E5-2670 processors @2.60 GHz, FDR 4x InfiniBand Interconnect, having system peak performance 18949.2 Gflops. We ran experiments with 4GB memory limit and 60 minute timeout. Our time measurement starts from the input PDDL to the output plans.

As our benchmark planning domains, we use 21 classical planning domains from international planning competitions. These domains are airport, blocks, driverlog, freecell, grid, gripper, logistics00, logistics98, miconic, mprime, mystery, pathway, psr-small, pipes-no-tankage, pipes-tankage, rovers, satellite, storage, tpp, and zenotravel. Among these, only in 9 domains namely airport, blocks, depot, driverlog, freecell, grid, gripper, pipes-tankage, and storage, the PDDL-to-SAS<sup>+</sup> translator produces **mutex-groups**. There are in total 786 problem instances in all 21 domains and we use them to evaluate the planners. However, in the charts, we will mainly use the problem instances that are solved by at least one planner in the respective charts. Problem instances that are not solved by any planner in the respective charts are thus ignored. Moreover, problem instances are often sorted in order of the best solver’s time; the monotonously increasing line in a chart will indicate this. Lastly, the missing points in the charts mean those problems are not solved by the respective planners within the timeout. For more meaningful comparison, we emphasise on the number of problem instances solved, although we will also look at the time performances.

As mentioned in Section 3, we use Minion (Gent et al., 2006) as our constraint solver. Minion supports several options for variable ordering, but in our experiments we use the most well-known two: *smallest domain first (sdf)* and *most conflict variable first (conflict)*. The sdf heuristic is also known as *min-dom* heuristic and the conflict heuristic is based on *dom/wdeg* (i.e. the ratio of the current domain size to the weighted degree of a variable) (Boussemart, Hemery, Lecoutre, & Sais, 2004). Minion uses the lexicographic ordering to break ties. The other variable ordering heuristics that we do not use for various typical reasons are the largest domain first, static ordering, random ordering, smallest ratio of current domain size to initial domain size ordering. For propagation, Minion supports *generalised arc consistency (gac)*, *singleton arc consistency (sac)*, and a few variants of sac. In our experiments, we use gac and the basic sac as our constraint propagation strategies. Unfortunately Minion does not provide any significant value ordering heuristic other than ascending or descending ordering. In our experiments, we only use the ascending value ordering. Minion supports *don’t cares* and *sets* as cell values in a table constraint. While only some sets could be replaced by *don’t cares*, any *don’t care* could be replaced by a set. Lastly, given the **mutex-groups** are produced during the PDDL-to-SAS<sup>+</sup> translation, we either use them or ignore them. Combining all these options, we have experimented with the 12 versions of TCPP and 12 versions of PaPR along with the original PaP2. In order to avoid cluttering in the charts and the tables, we will often show results of the best performing versions only.

- TCPP encoding variants running on Minion with different configurations
  1. TCPPx: uses *don’t cares* (x) only, considered as the base version of TCPP
  2. TCPPsm: uses sets (s) and **mutex-groups** (m), to study the effect of using sets

- 3. TCPPxm: uses *don't cares* (x) and *mutex-groups* (m), final version of TCPP
- PaPR encoding variants running on Minion with different configurations
  1. PaPRs: uses sets (s) only, the reconstructed version which is equivalent to PaP2
  2. PaPRsx: uses sets (s), *don't cares* (x) where possible, enhancing using *don't cares*
  3. PaPRsxm: uses sets (s), *don't cares* (x) and *mutex-groups* (m), final PaPR
- Each of TCPP and PaPR variants running on Minion with the following configurations
  1. sdf-gac: smallest domain first variable ordering and generalised arc consistency
  2. sdf-sac: smallest domain first variable ordering and singleton arc consistency
  3. conf-gac: conflict variable ordering and generalised arc consistency
  4. conf-sac: conflict variable ordering and singleton arc consistency
- PaP2 on SICStus Prolog with specially developed variable and value ordering

### 7.1 TCPP Encoding Characteristics

Figure 19 shows the performances of TCPPxm and TCPPsm with all four Minion configurations on all 21 domains. TCPPxm appears to be consistently performing better than TCPPsm. Since TCPPxm uses *don't cares* while TCPPsm uses sets, from this, we conclude that using *don't cares* is significantly better than using sets in our constraint model regardless of the Minion configuration used. This is because the table constraints with *don't cares* are much more compact than that with the sets. Moreover, checking a cell value to match with a given value is very efficient with *don't cares* than with sets.

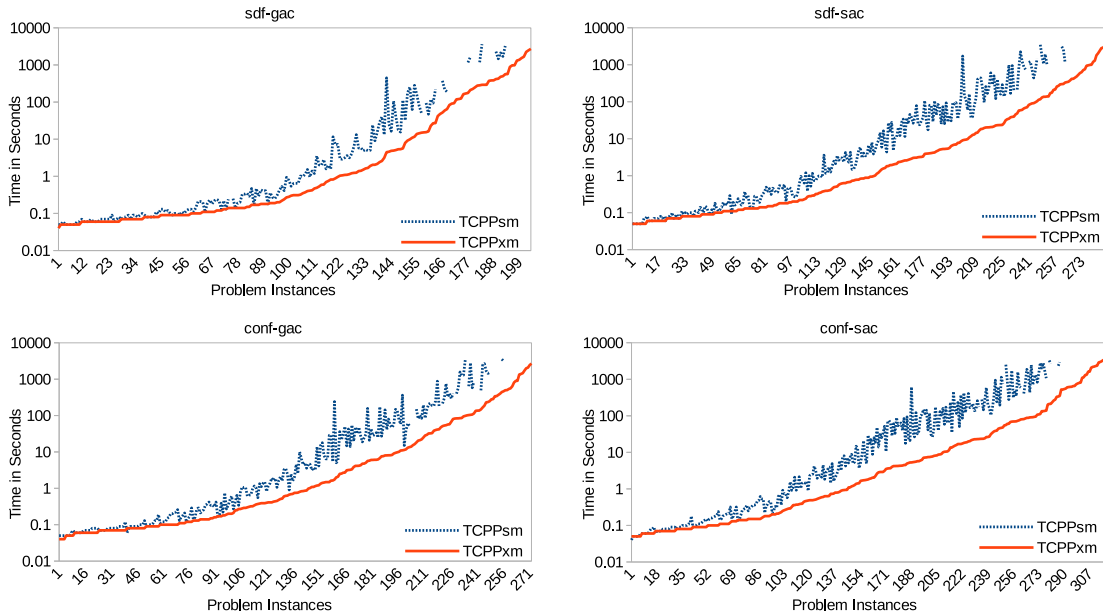


Figure 19: Enhancing TCPP using *don't cares* rather than sets

Figure 20 shows the performances of TCPPxm and TCPPx on the problem instances from the 9 domains where **mutex-groups** are produced. In the other 12 domains where no **mutex-groups** are produced, TCPPxm and TCPPx are actually equivalent and hence not compared. Nevertheless, for all four Minion configurations, TCPPxm consistently performs better than TCPPx. Since TCPPxm uses **mutex-groups** while TCPPx does not, we conclude that using **mutex-groups** (along with *don't cares*) is significantly better in our constraint model than not using them. This is because **mutex-groups** are additional knowledge in the form of *nogoods* that help the CSP solver prune the search space. We represent **mutex-groups** using very small binary tables. We also reduce the domain size of parallelism variables using mutex groups (see Algorithm 5 Function ElsePrevented). The optional use of **mutex-groups** in our encoding thus brings about more efficiency in TCPPxm than in TCPPx.

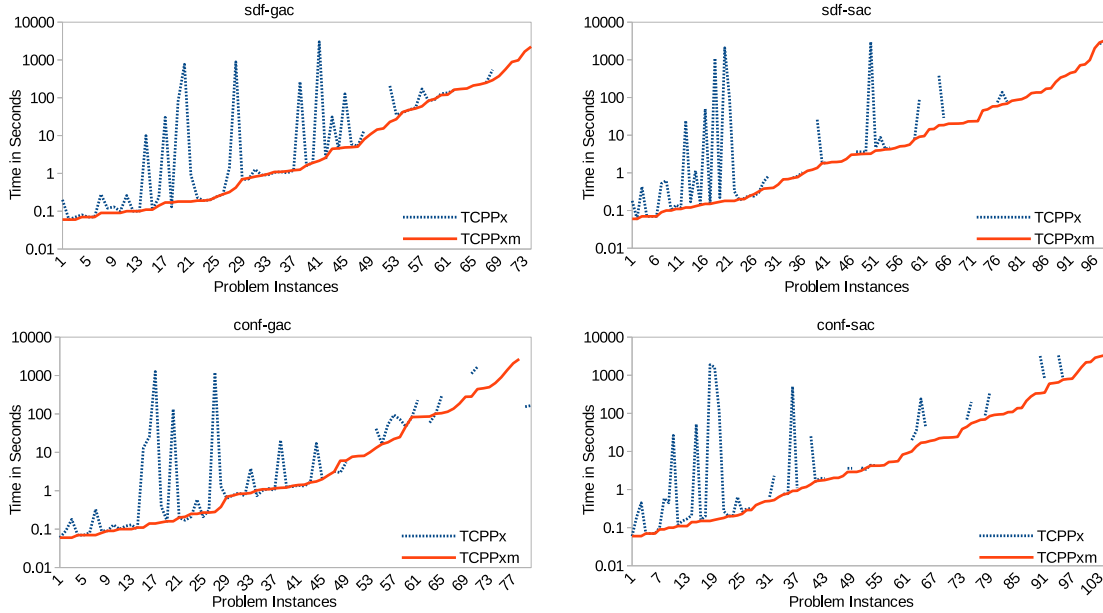


Figure 20: Enhancing TCPP using **mutex-groups**

Table 2 shows the numbers of problem instances solved by different TCPP versions. For each given Minion configuration, TCPPxm, our final TCPP version, is the best performing solver compared to TCPPsm and TCPPx. Among all 12 versions, TCPPxm-conf-sac appears to be the best, solving 317 instances in total, while the second best TCPPxm-sdf-sac solves 287 instances and the third best TCPPsm-conf-sac solves 281 instances. Overall, conf-sac appears to be the most suitable Minion configuration for TCPP encodings. On the other hand, the sdf-gac configuration appears to be the least suitable. Given a variable ordering (sdf or conf), the sac constraint propagation appears to be consistently better than the gac with all three versions TCPPxm, TCPPsm, and TCPPx. On the other hand, given a propagation strategy (gac or sac), the conf variable ordering appears to be consistently better than the sdf with all three versions TCPPxm, TCPPsm, and TCPPx.

Table 2: Numbers of problem instances solved by TCPP versions

Encoding Variants →		TCPPsm				TCPPxm				TCPPx				Mutex
Solver Config →		sdf	sdf	conf	conf	sdf	sdf	conf	conf	sdf	sdf	conf	conf	Groups
Domains	#Ins	gac	sac	gac	sac	gac	sac	gac	sac	gac	sac	gac	sac	Exist
airport	50	14	15	15	15	21	20	21	21	19	18	21	19	yes
blocks	35	13	29	13	29	14	32	14	32	8	8	8	8	yes
depot	22	2	6	5	9	5	7	6	11	2	2	1	2	yes
driverlog	20	11	14	11	13	12	14	12	13	11	14	11	13	yes
freecell	20	3	3	3	3	3	4	4	4	3	3	4	3	yes
grid	5	1	2	1	2	1	2	2	2	1	1	1	1	yes
gripper	20	1	2	1	2	1	2	2	2	1	1	1	1	yes
logistics00	28	11	19	19	22	12	19	19	24	12	19	19	24	no
logistics98	35	1	6	6	7	1	7	7	9	1	7	7	9	no
miconic	150	31	30	27	27	33	33	29	29	33	33	29	29	no
mprime	35	12	18	20	23	13	20	22	26	13	20	22	26	no
mystery	30	8	13	13	14	10	17	14	17	10	17	14	17	no
pathways	30	5	5	5	5	5	5	5	5	5	5	5	5	no
pipes-no-tankage	50	7	7	7	6	11	8	10	8	11	8	10	8	no
pipes-tankage	50	6	7	7	7	8	8	8	9	8	8	8	8	yes
psr-small	50	21	38	43	42	21	42	43	44	21	42	43	44	no
rovers	40	8	15	17	21	8	16	18	22	8	16	18	22	no
satellite	36	2	3	5	5	2	5	6	7	2	5	6	7	no
storage	30	9	9	9	9	9	9	9	10	9	9	9	9	yes
tpp	30	5	8	8	8	5	8	9	10	5	8	9	10	no
zenotravel	20	8	9	12	12	9	9	12	12	9	9	12	12	no
Total	786	179	258	247	281	204	287	272	317	192	253	258	277	

## 7.2 Reconstructing PaPR from PaP2

Figure 21 shows the performances of the original PaP2 and our reconstructed version PaPR. PaP2 runs on SICStus Prolog with customised variable and value ordering heuristics. We run PaPR on Minion with the four configurations mentioned before. Note that the PaPR version considered here is PaPRs, which uses sets in the table constraints and does not use mutex-groups. PaPRs is thus the closest structurally matching reconstructed version.

Nevertheless, compared to PaP2’s, we see that PaPRs’ performances vary much from one problem instance to another although a statistically fitted line, if drawn, perhaps would be similar. To look at this further, in Figure 22, we compare the numbers of problem instances solved by PaP2 and PaPR when different timeout limits are given. It appears that PaPRs when run on Minion with conf-gac configuration performs closest to PaP2 (actually slightly better). Moreover, PaPRs with sdf-gac performs the much worse compared to PaP2.

Table 3 shows the numbers of problem instances solved by PaP2 and PaPRs. While in most domains the numbers of instances solved by PaP2 and PaPRs are very close, the significant differences are in logistics00 and psr-small. Given the close matching of PaP2’s constraint model and that of PaPRs, the difference mainly comes from the solvers SICStus Prolog and Minion. The exact reason behind this depends on the specific details of the respective solvers.

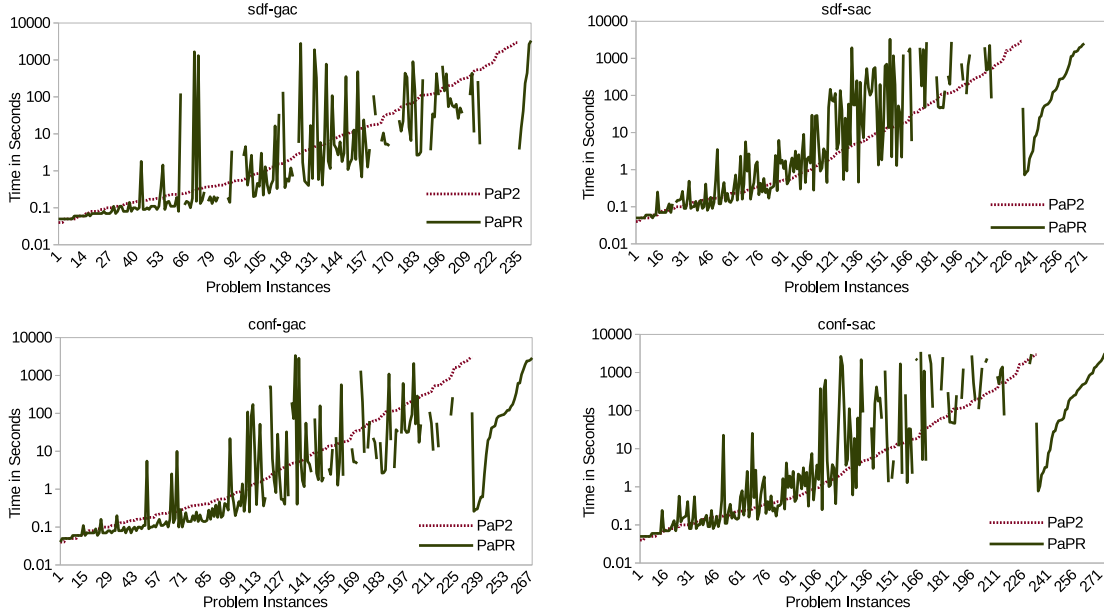


Figure 21: Reconstructing PaP2 to obtain PaPR that runs on Minion.

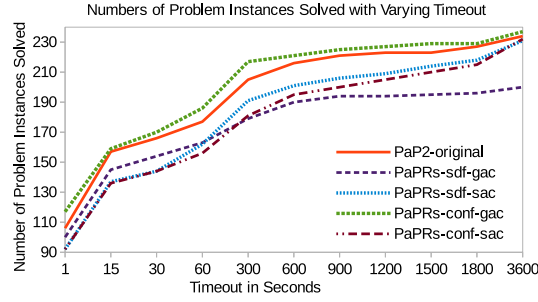


Figure 22: PaP2 and PaPR with different timeout limits.

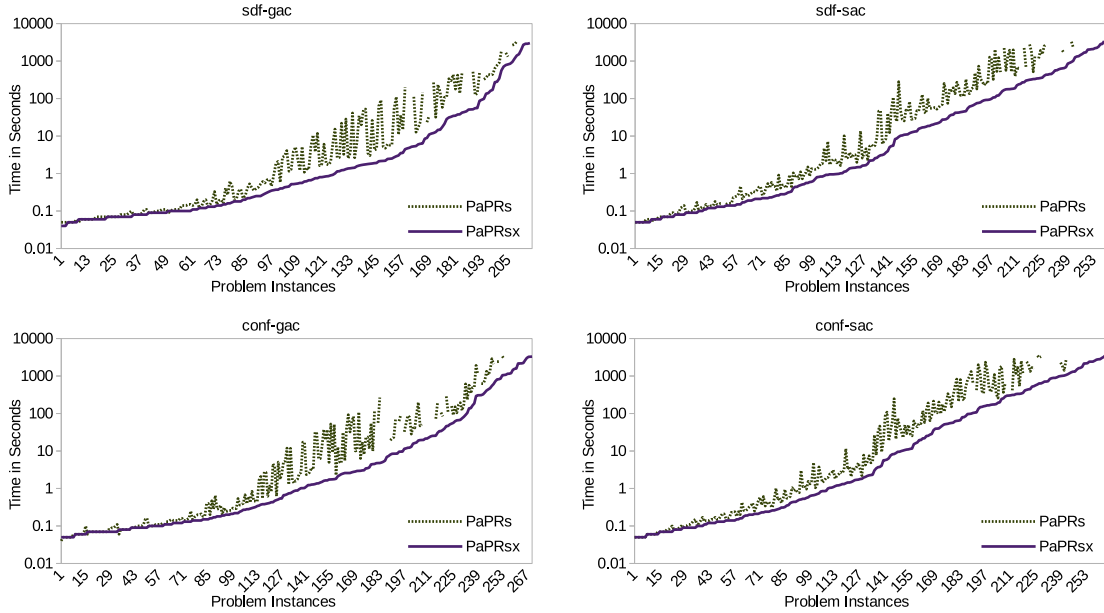
### 7.3 Enhancing PaPR with *Don't Cares* and Mutexes

Figure 23 shows performances of PaPR when encoded only with sets (PaPRs) and when certain sets are replaced by *don't cares* (PaPRsx). For all four Minion configurations, PaPRsx appears to be better than PaPRs. This is because the table constraints with *don't cares* are much more compact than that with the sets. Moreover, checking a cell value to match with a given value is very efficient with *don't cares* than with sets.

Figure 24 shows the performances of PaPRsxm and PaPRsx on the problem instances from the 9 domains where *mutex-groups* are produced. In the other 12 domains where no *mutex-groups* are produced, PaPRsxm and PaPRsx are actually equivalent and hence not compared. Nevertheless, for all four Minion configurations, PaPRsxm consistently performs better than PaPRsx. Since PaPRsxm uses *mutex-groups* while PaPRsx does not, we conclude that using *mutex-groups* along with *don't cares* is significantly better in PaP2 constraint model than not using them. This is because *mutex-groups* are additional knowledge

Table 3: Numbers of problem instances solved by PaP2 and PaPRs

Encoding Variants →		PaP2	PaPRs on Minion			
Solver Config →		SICStus	sdf	sdf	conf	conf
Domains	#Ins	Prolog	gac	sac	gac	sac
airport	50	14	14	12	16	12
blocks	35	13	8	12	10	11
depot	22	3	3	5	4	6
driverlog	20	13	14	13	12	12
freecell	20	3	2	2	3	2
grid	5	1	1	1	1	1
gripper	20	2	2	1	2	2
logistics00	28	15	12	19	22	24
logistics98	35	18	12	9	12	9
miconic	150	34	32	29	25	23
mprime	35	20	18	10	18	14
mystery	30	11	10	11	11	11
pathways	30	2	1	5	5	5
pipes-no-tankage	50	6	3	4	7	5
pipes-tankage	50	6	5	5	5	5
psr-small	50	28	19	42	37	43
rovers	40	11	10	18	16	17
satellite	36	4	3	2	3	5
storage	30	10	9	9	9	8
tpp	30	8	8	10	8	8
zenotravel	20	12	14	12	11	9
Total	786	234	200	231	237	232

Figure 23: Enhancing PaPR using *don't cares*

in the form of *nogoods* that help the CSP solver prune the search space. We represent **mutex-groups** using very small binary tables that have pairs of actions having parallel conflicts. Although synchronisation constraints in a different way enforces selection of non-conflicting actions, explicit use of a negative table to list the conflicting actions due to mutexes eliminated combinations of actions during search. The optional use of **mutex-groups** in PaP2's encoding thus brings about more efficiency in PaPRsxm than in PaPRsx.

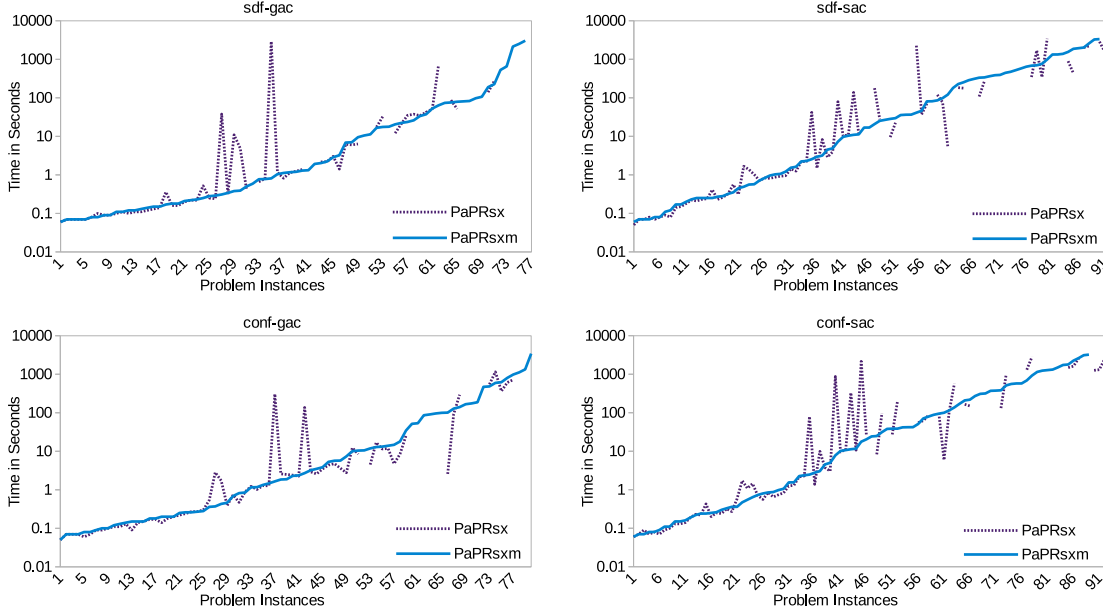


Figure 24: Enhancing PaPR using mutex-groups.

Table 4 shows the numbers of problem instances solved by different PaPR versions. For each given Minion configuration, PaPRsxm, the final PaPR version, is the best performing solver compared to PaPRsx and PaPRs. Among all 12 versions, PaPRsxm-conf-sac and PaPRsxm-conf-gac appear to be the two best, each solving 281 instances in total, while the second best PaPRsxm-sdf-sac solves 278 instances and the third best PaPRsx-conf-gac solves 270 instances. Overall, conf-gac/sac appears to be the most suitable Minion configuration for PaPR encodings. On the other hand, the sdf-gac configuration appears to be the least suitable. Given a variable ordering sdf, the sac constraint propagation appears to be consistently better than the gac with all three versions PaPRsxm, PaPRsx, and PaPRs. In contrast, given a conf variable ordering, the gac constraint propagation appears to be consistently better than sac. Nevertheless, given a propagation strategy (gac or sac), the conf variable ordering appears to be consistently better than the sdf with all three versions PaPRsxm, PaPRsx, and PaPRs.

#### 7.4 Time Performance Comparison

Besides PaP2, we have selected planners from IPC-8 deterministic track to compare our planners TCPP and PaPR with them. We have selected SymBA\*-2, which is the winner of the sequential optimal track. SymBA\*-2 is a cost-optimal planner and performs several



Table 4: Numbers of problem instances solved by PaPR versions

Encoding Variants →		PaPRs				PaPRsx				PaPRsxm				Mutex
Solver Config →		sdf	sdf	conf	conf	sdf	sdf	conf	conf	sdf	sdf	conf	conf	Groups
Domains	#Ins	gac	sac	gac	sac	gac	sac	gac	sac	gac	sac	gac	sac	Exists
airport	50	14	12	16	12	17	16	17	16	17	17	22	18	yes
blocks	35	8	12	10	11	9	13	10	13	18	29	16	30	yes
depot	22	3	5	4	6	4	8	6	8	6	10	6	8	yes
driverlog	20	14	13	12	12	14	13	12	13	14	13	12	12	yes
freecell	20	2	2	3	2	2	3	3	3	3	3	3	3	yes
grid	5	1	1	1	1	1	1	1	1	1	1	1	1	yes
gripper	20	2	1	2	2	2	2	2	2	2	2	2	2	yes
logistics00	28	12	19	22	24	12	19	23	25	12	19	23	25	no
logistics98	35	12	9	12	9	12	11	16	13	12	11	16	13	no
miconic	150	32	29	25	23	34	29	27	24	34	29	27	24	no
mprime	35	18	10	18	14	21	16	24	18	21	16	24	18	no
mystery	30	10	11	11	11	12	14	15	13	12	14	15	13	no
pathways	30	1	5	5	5	1	5	5	5	1	5	5	5	no
pipes-no-tankage	50	3	4	7	5	4	7	11	8	4	7	11	8	no
pipes-tankage	50	5	5	5	5	5	6	9	7	6	6	9	6	yes
psr-small	50	19	42	37	43	19	43	38	44	19	43	38	44	no
rovers	40	10	18	16	17	10	19	16	18	10	19	16	18	no
satellite	36	3	2	3	5	4	3	5	5	4	3	5	5	no
storage	30	9	9	9	8	9	9	9	9	9	9	9	9	yes
tpp	30	8	10	8	8	8	10	9	9	8	10	9	9	no
zenotravel	20	14	12	11	9	15	12	12	10	15	12	12	10	no
Total	786	200	231	237	232	215	259	270	264	228	278	281	281	

symbolic bidirectional A\* searches on different state spaces. SymBA\*-2 also uses abstraction heuristics. We have also selected YAHSP3 and Madagascar, which are the first and second place holder in the agile track of the competition. YAHSP3 is a forward state-space heuristic search planner, but the version submitted to agile track is not a cost-optimal planner. The quality of plans generated by this planner are not good when compared to that of Madagascar or other optimal planners. Madagascar planner is a SAT-based planner. Since it encodes the planning problem as a SAT problem, we have decided to compare our approach with it. SAT is often viewed as a Boolean CSP. It is therefore worth observing the performance difference between SAT-based and CSP-based planners.

Figure 25 shows the total numbers of problem instances solved by different planners in 21 selected benchmark domains. Moreover, Table 5 shows the total number of instances solved in each domain by different planners over a 60-minute timeout. From the charts and the table, we observe that YAHSP3 outperforms all other planners with a great margin, although a large part of the margin comes from its performance in the miconic domain. Nevertheless, one can easily see the huge differences in the performance of the constraint-based planners from that of the competition winning planners. Constraint-based planners need to make huge improvements in order for them to be competitive enough. In this analysis, we only consider the best performing versions of TCPP and PaPR.

TCPPxm-conf-sac performs significantly better than PaPRsxm-conf-gac/sac which are much more significantly better than original PaP2 and our reconstructed PaPRs. In 21

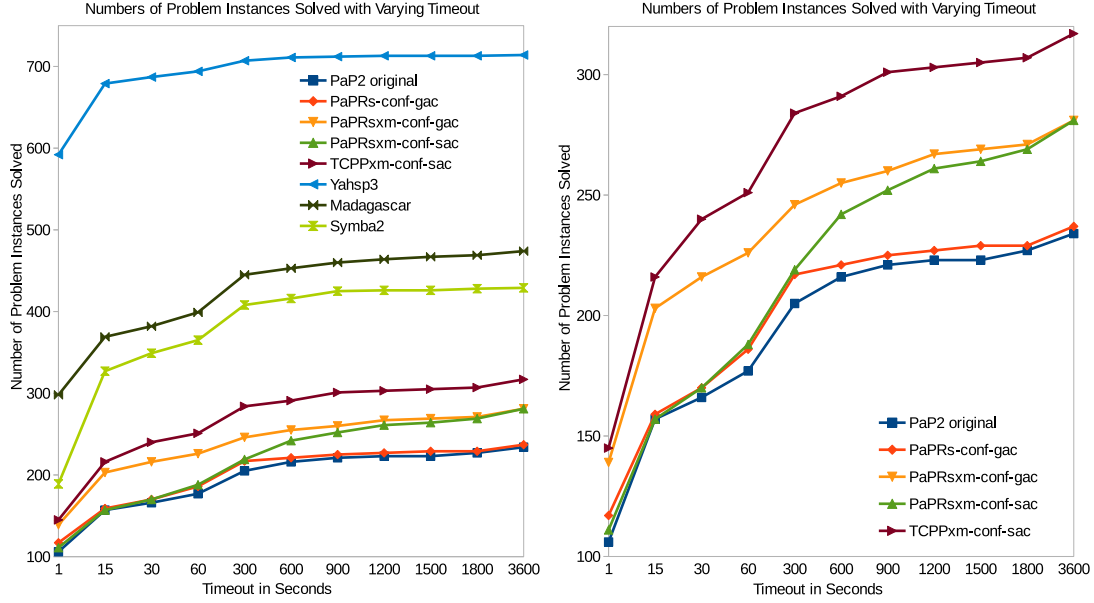


Figure 25: The total numbers of problem instances solved by different planners over different timeouts. Left: comparison with international planning competition winners. Right: comparison of only TCPP, PaP2, and PaPR.

domains that we have experimented with, TCPPxm-conf-sac solves the most numbers of problems in 15 domains while PaPRsxm-conf-gac solves in 5 domains and PaPRsxm-conf-sac solves 2 domains. Between PaPRsxm-conf-gac and PaPRsxm-conf-sac, the former performs much better than the latter with shorter timeouts, but with long timeouts both perform similar. Between TCPPxm-conf-sac and PaPRsxm-conf-gac, the former appears to be consistently performing better than the latter. It is interesting to observe these behaviours since TCPP and PaPR view DTGs from two orthogonally different perspectives: TCPP views state transitions and PaPR views action successions in the DTGs.

## 7.5 Plan Length Comparison

In Figure 26, we compare lengths of the plans produced by each planner. We are not optimising the plan length, but optimising the makespan, which has implications in terms of the plan length. While we do not claim any contribution in terms of the plan length comparison, this part of our analysis is mainly to observe how good are the plan lengths compared to that from the other planners. Since YASHP in the agile track of the planning competition only emphasises on speed without looking at the plan length, we exclude YASHP from our comparison. SymBA\*-2 optimises plans over a cost function, but in the absence of any function, the plan length is optimised. Madagascar is a SAT planner and is optimal in makespan. Note that TCPPxm-conf-sac, PaP2, and PaPRsxm-conf-gac all produces plans having the optimal makespan, but the sequential plan lengths might not be optimal. Nevertheless, to obtain an overall idea about the plan quality, in Figure 26, we

Table 5: Numbers of problem instances solved by state-of-the-art planners. In the table, Mada is Madagascar, PaPR<sup>0</sup> is PaPRs-conf-gac, PaPR<sup>1</sup> is PaPRsxm-conf-gac, PaPR<sup>2</sup> is PaPRsxm-conf-sac, TCPP\* is TCPPxm-conf-sac.

Domains	#Ins	Mada	Yahsp	SymBA	PaP2	PaPR <sup>0</sup>	PaPR <sup>1</sup>	PaPR <sup>2</sup>	TCPP*
airport	50	45	45	27	14	16	<b>22</b>	18	21
blocks	35	35	35	33	13	10	16	30	<b>32</b>
depot	22	15	22	7	3	4	6	8	<b>11</b>
driverlog	20	15	20	14	13	12	12	12	<b>13</b>
freecell	20	4	20	5	3	3	3	3	<b>4</b>
grid	5	2	5	2	1	1	1	1	<b>2</b>
gripper	20	3	20	20	2	2	2	2	2
logistics00	28	28	28	20	15	22	23	<b>25</b>	24
logistics98	35	28	34	5	18	12	<b>16</b>	13	9
miconic	150	49	150	111	34	25	27	24	<b>29</b>
mprime	35	31	35	24	20	18	24	18	<b>26</b>
mystery	30	18	17	15	11	11	15	13	<b>17</b>
pathways	30	9	0	5	2	5	5	5	5
pipes-notank	50	26	44	15	6	7	<b>11</b>	8	8
pipes-tank	50	10	40	16	6	5	<b>9</b>	6	<b>9</b>
psr-small	50	50	50	50	28	37	38	<b>44</b>	<b>44</b>
rovers	40	33	40	14	11	16	16	18	<b>22</b>
satellite	36	16	36	11	4	3	5	5	<b>7</b>
storage	30	14	23	15	10	9	9	9	<b>10</b>
tpp	30	28	30	8	8	8	9	9	<b>10</b>
zenotravel	20	15	20	12	12	11	<b>12</b>	10	<b>12</b>
Total	786	474	714	429	234	237	281	281	<b>317</b>

show the total plan length of the plans produced by the planners over each domain; only the problem instances that are solved by all three planners are taken into account.

We observe that SymBA\*-2 produces the shortest plan lengths in all domains. Moreover, PaPRsxm-conf-gac produces plans that are slightly longer than SymBA\*-2's plans. Furthermore, PaP2's plans are longer than PaPRsxm-conf-gac's; TCPPxm-conf-sac's plans are longer than PaP2s; and Madagascar's plans are the longest. Since we use the ascending value ordering and in PaP2 and PaPRsxm-conf-gac, the values less than zero are set for *no-ops*, these planners first try *no-ops* in the search and that is why the number of real actions in the final plan in PaP2 and PaPRsxm-conf-gac are less than that in TCPPxm-conf-sac. The value ordering in this case affects the quality of the plans produced.

## 7.6 Encoding Statistics

As we have discussed before, in TCPPxm, we have transition constraints while in PaPRsxm, we have action succession constraints along with synchronisation constraints. Both planners use table constraints with *don't cares* to encode their constraints. In Table 6, we report the average numbers of DTGs, tables, rows, and columns in the encodings over the problem instances that are solved by at least one of the two planners TCPPxm-conf-sac and PaPRsxm-conf-gac.

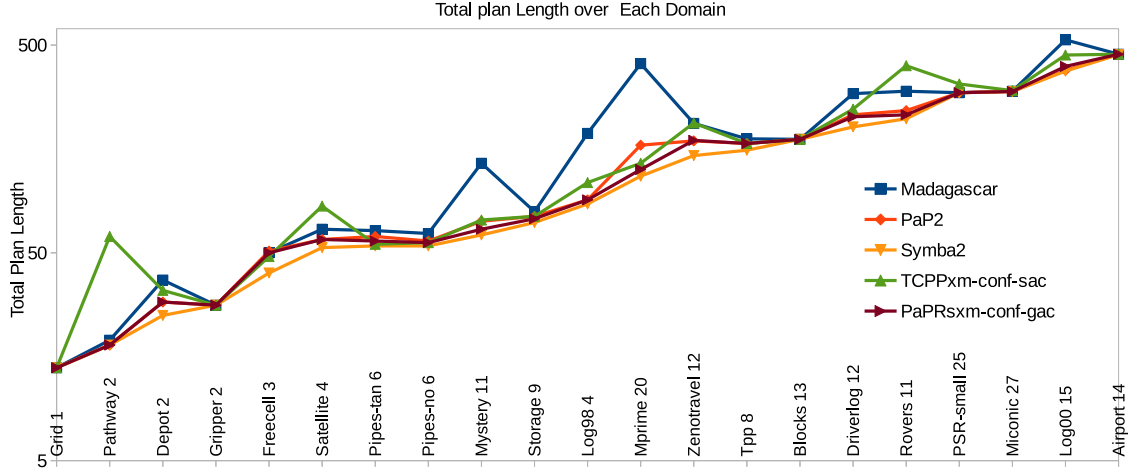


Figure 26: Total plan length for the plans produced by different planners over each domain

To summarise Table 6, the numbers of CSP variables in TCPX are one time step more than that in PaPR (one can say in PaP2) because TCPX’s variables are for states, while, PaPR’s variables are for transitions. However, this is not an important factor because TCPX performs good even in the airport domain where the number of DTGs is huge. The domain sizes of CSP variables in PaPR are much larger than that in TCPX. This significantly affects the efficiency of the CSP search. The branching factor in the search depends on the domain size. Also, the cost of the SAC algorithm depends on this factor. The numbers of positive tables in PaPR are roughly twice of that of TCPX and are proportional to the numbers of variables in the SAS<sup>+</sup> representations. The numbers of rows in the positive tables in both TCPX’s and PaPR’s models are about the same. The numbers of columns in TCPX’s transition constraints are about twice the numbers of columns in PaPR’s synchronisation constraints except in domains pipes-notankage, and pipes-tankage. The number of columns in PaPR’s action succession constraints is always 2, but each column could have a set of values. In terms of inspection of certain values in the table, we actually have to examine each value in the set. This means rather than considering the number of cell in a table, we need to consider the number of values that needs to be checked. Using don’t cares (which are used in high percentages), the checking is minimised but it does not happen with sets. Nevertheless, the numbers of tables for mutex-groups are the same for both TCPX and PaPR models. These tables have 2 columns but the numbers of rows in PaPR are much more than that in TCPX. Learning of which planner and which configuration works better in a domain or an instance could be performed using the encoding statistics, However, that is clearly out of scope of this paper.

Table 6: Different statistics about the TCPP and PaPR encodings. In the table, pv: parallel variables, tc: transitions constraints, mc: mutex constraints, sc: synchronisation constraints, asc: action succession constraints, neg: negative tables, Ts: #tables, Rs: #rows, Cs: #columns, pCs: #columns for parallel variables. Tables for mc and asc have 2 columns. The numbers of DTGs for TCPP is the same as the numbers of FSA for PaPR.

			TCPPxm									PaPRsxm						
Planning Domains	# Ins	# of DTG	Dom Size	pv Size	pv Dom	tc			neg mc		don't care%	Dom Size	sc		asc	neg mc		don't care%
airport	26	841	2	2	2	9	54	0.4	304	2	86	9	8	35	9	304	19	81
blocks	32	19	6	0	0	39	30	0	143	4	80	39	34	15	39	143	125	75
depot	12	36	7	0	0	142	54	0	180	7	85	144	139	27	144	180	1356	81
driverlog	14	13	7	0	0	44	9	0	8	1	58	44	38	4	44	8	62	52
freecell	4	34	4	2	15	138	24	2	72	2	80	138	135	12	138	72	405	81
grid	2	20	18	0	0	342	17	0	50	1	75	342	325	8	342	50	146025	74
gripper	2	8	4	0	0	14	9	0	11	2	57	14	11	4	14	11	13	48
logistics00	25	13	8	0	0	21	8	0	0	0	67	22	16	4	22	0	0	61
logistics98	19	40	23	0	0	112	16	0	0	0	91	150	128	8	150	0	0	90
miconic	34	9	3	0	0	4	5	0	0	0	45	10	8	3	10	0	0	33
mprime	27	32	10	15	13	479	24	6.9	0	0	80	459	451	12	459	0	0	80
mystery	17	34	10	15	8	284	19	3.6	0	0	70	284	275	9	284	0	0	70
pathways	5	78	2	15	4	5	7	1.3	0	0	71	5	4	4	5	0	0	72
pipe-notankage	11	140	2	140	10	31	67	35	0	0	88	31	30	35	31	0	0	85
pipe-tankage	9	62	3	39	5	163	60	22	43	4	82	163	161	35	163	43	14320	81
psr-small	45	24	2	1	1	16	13	0.1	0	0	56	16	15	7	16	0	0	48
rover	21	86	3	6	6	16	9	0.7	0	0	54	16	14	5	16	0	0	55
satellite	7	43	2	1	2	5	14	0.2	0	0	67	10	8	7	10	0	0	58
storage	10	17	3	4	2	30	19	2.4	6	1	68	30	28	10	30	6	278	60
tpp	10	32	3	16	1	8	8	1.1	0	0	49	8	7	4	8	0	0	44
zenotravel	15	12	7	0	0	186	6	0	0	0	12	186	179	3	186	0	0	9

## 7.7 Performance on Domains

Figures 27 and 28 shows the performance of TCPPxm and PaPRsxm on 14 domains. These domains are chosen to include examples where TCPP or PaPR or SAC or GAC is better, and performance differences are clearly visible.

In the charts, we mainly show the best TCPP and PaPR versions namely TCPPxm and PaPRsxm respectively. We already have shown these versions significantly outperform PaP2 and its reconstructed version PaPRs. Moreover, we only show the configurations where conf variable ordering is used. As we see from our results, and also from the literature on variable selection heuristics (Balafoutis & Stergiou, 2008), conf appears to be better than sdf. Nevertheless, it is obvious from the charts that the performance of TCPP is better than that of PaPR when both use SAC. The reason is that the complexity of SAC is proportional or quadratic to the domain size of CSP variables based on the algorithm selected for enforcing SAC (Bessiere, Cardon, Debruyne, & Lecoutre, 2011). As shown in Table 6, PaPR's encoding has larger domains than TCPP's encoding. Overall, we see TCPP outperforms PaPR in most domains.

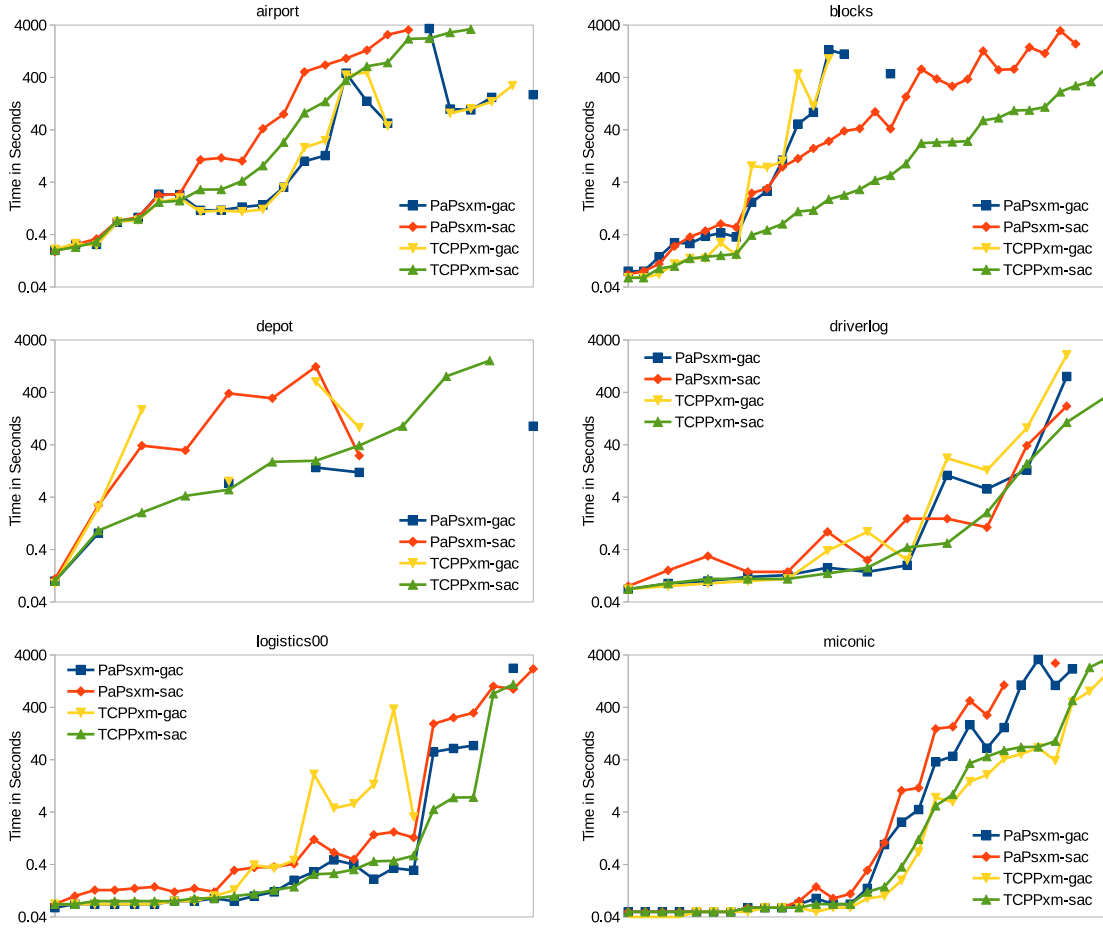


Figure 27: Time performance of TCPPxm, PaPRsxm on 6 Domains

## 7.8 Effect of Minion Configurations

Table 7 and Figure 29 compare performances of PaPRsxm and TCPPxm versions across different Minion configurations. Given a propagation strategy (gac or sac), the conf variable ordering appears to be better than the sdf for both TCPP and PaPR although for PaPR with sac appears very close; and also in miconic, sdf is rather better. The finding that the conf variable ordering is in most cases better than the sdf is consistent with the findings in the CSP research (Balafoutis & Stergiou, 2008).

For TCPP, given a variable ordering (sdf or conf), the sac constraint propagation appears to be consistently better than the gac. For PaPR, given a variable ordering sdf, the sac constraint propagation appears to be consistently better than the gac, but with conf, the gac is better in the short timeouts and almost the same in long timeouts.

Figure 30 shows the effect of gac and sac propagation strategies on CSP search for planning. We show two instances from the airport domain such that gac strategy is much faster than the sac strategy. We also show two instances from the blocks domain such that the sac strategy is much faster than the gac strategy. As we can see, the search nodes

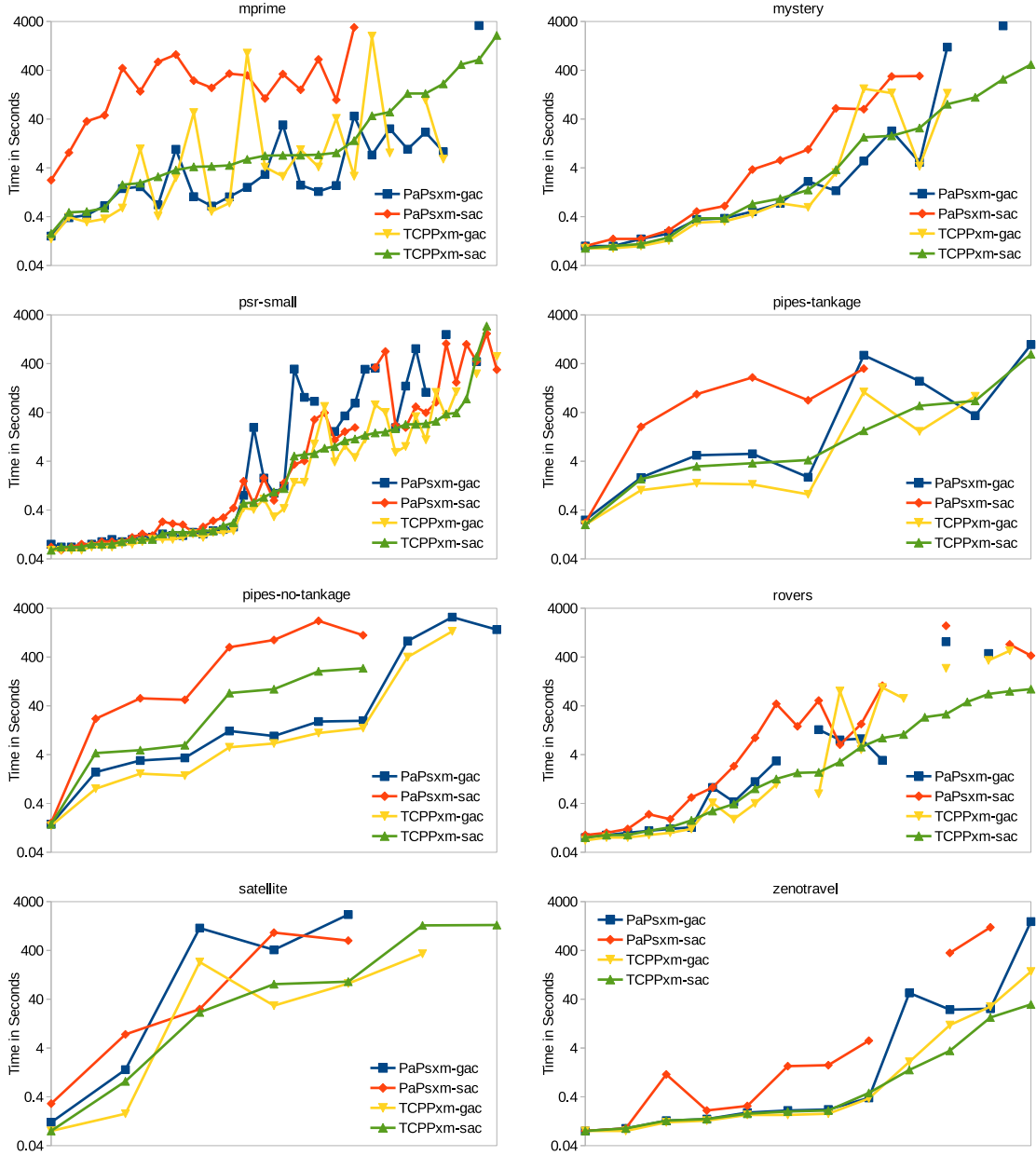


Figure 28: Time performance of TCPPxm, PaPRsxm on 8 Domains

visited by sac is often 0 at different makespan level. This indicates sac is mostly used in the preprocessing step and is able to determine unsolvability without performing search. When we checked the total running time to solve an instance, we found that most part of the running time is spent at the last makespan level when the constraint model is satisfiable.

Using sac during search is known to be a very costly operation. However, so far in our experiments, when we used sac or gac, we used the same in both the preprocessing and the search phases. Based on the above observation that using sac, we often do not need search

Table 7: Numbers of problem instances solved by PaPRsxm and TCPPxm versions

Encoding Variants →		PaPRsxm				TCPPxm			
Solver Config →		sdf	sdf	conf	conf	sdf	sdf	conf	conf
Domains	#Ins	gac	sac	gac	sac	gac	sac	gac	sac
airport	50	17	17	22	18	21	20	21	21
blocks	35	18	29	16	30	14	32	14	32
depot	22	6	10	6	8	5	7	6	11
driverlog	20	14	13	12	12	12	14	12	13
freecell	20	3	3	3	3	3	4	4	4
grid	5	1	1	1	1	1	2	2	2
gripper	20	2	2	2	2	1	2	2	2
logistics00	28	12	19	23	25	12	19	19	24
logistics98	35	12	11	16	13	1	7	7	9
miconic	150	34	29	27	24	33	33	29	29
mprime	35	21	16	24	18	13	20	22	26
mystery	30	12	14	15	13	10	17	14	17
pathways	30	1	5	5	5	5	5	5	5
pipes-no-tankage	50	4	7	11	8	11	8	10	8
pipes-tankage	50	6	6	9	6	8	8	8	9
psr-small	50	19	43	38	44	21	42	43	44
rovers	40	10	19	16	18	8	16	18	22
satellite	36	4	3	5	5	2	5	6	7
storage	30	9	9	9	9	9	9	9	10
tpp	30	8	10	9	9	5	8	9	10
zenotravel	20	15	12	12	10	9	9	12	12
Total	786	228	278	281	281	204	287	272	317

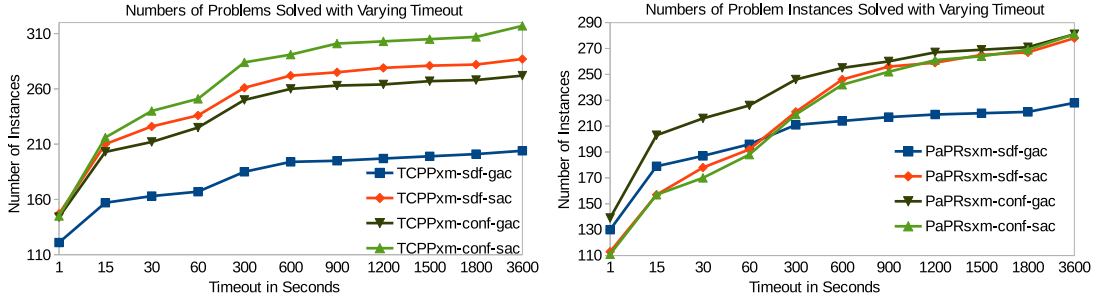


Figure 29: TCPPxm and PaPRsxm with different timeout limits

particularly in the unsatisfiable constraint models, we further ran both TCPPxm-conf and PaPRsxm-conf with sac in the preprocessing step and gac in the search step.

Table 8 shows the numbers of problem instances solved when different combinations of sac and gac are used in the preprocessing and search of TCPPxm-conf and PaPRsxm-conf. TCPPxm-conf with sac-gac combination appears to be solving fewer problem instances than with the sac-sac combination. On the other hand, PaPRsxm-conf with sac-gac appears to be solving few more problems than with the sac-sac or the gac-gac combination.



Makespan	Steps →		20	21	22	23	24	25	26	time
TCPPxm-conf	airport8	gac	0	9	21	40	299	1377	1457	18.23
		sac	0	0	0	0	0	0	11	84.48
PaPR-sxm-conf	airport8	gac	0	0	0	9	94	173	30	10.04
		sac	0	0	0	0	0	0	10	510.52
TCPPxm-conf	airport14	gac	0	9	21	36	291	1551	1634	24.93
		sac	0	0	0	0	0	0	14	138.44
PaPR-sxm-conf	airport14	gac	0	0	0	9	94	173	32	12.86
		sac	0	0	0	0	0	0	12	689.71
Makespan	Steps →		14	15	16	17	18	19	20	time
TCPPxm-conf	blocks12	gac	959	4363	19250	85344	373311	1685275	2083670	464.49
		sac	0	0	0	0	0	0	10	1.1
PaPRsxm-conf	blocks12	gac	43	112	916	1921	11773	36064	946	51.147
		sac	0	0	0	0	0	0	4	11.27
TCPPxm-conf	blocks13	gac	2773	15340	83950	438665	462718			113.92
		sac	0	0	0	0	5			1.17
PaPRsxm-conf	blocks13	gac	569	2053	10639	37995	13795			86.18
		sac	0	0	0	0	3			17.66

Figure 30: The numbers of search nodes visited by the CSP search for planning

Table 8: Total numbers of problem instances solved when different combinations of sac and gac are used in preprocessing and search of TCPPxm-conf and PaPRsxm-conf

TCPPxm-conf configs			PaPRsxm-conf configs		
gac-gac	sac-gac	sac-sac	gac-gac	sac-gac	sac-sac
272	303	317	281	288	281

configuration: preprocess-search e.g. sac-gac

Figure 31 shows the time performance of TCPPxm-conf and PaPR-sxm-conf when different combinations of sac and gac are used in the preprocessing and search. The problem instances are from all domains and include instances that are solved by at least one planner in the respective chart. TCPPxm-conf is slightly better in speed with sac-gac configuration than with sac-sac and clearly better than with gac-gac but solves fewer problem instances than with sac-sac. On the other hand PaPRsxm-conf is slightly better with sac-gac than with sac-sac but is worse with sac-gac than with gac-gac.

Overall, we observe that when the sac can prune many nodes during the preprocessing or search, enforcing sac pays off and we can see the improvement in the performance in domains such as blocks, depot, logistics00 and rovers. In the domains such as airport where pruning is not considerable compared to the time needed to prune, sac does not pay off and so in these domains, gac is much more efficient. Moreover, because of the chains of variables connected through the constraint paths in planning problems, sac can prune many values from the domains of CSP variables. Since the cost of propagation depends on the domain size of the variables, sac works better in TCPP than in PaPR. TCPP is better than PaPR in most domains because of the smaller domains of CSP variables and so reduced branching factors during search and reduced cost of sac.

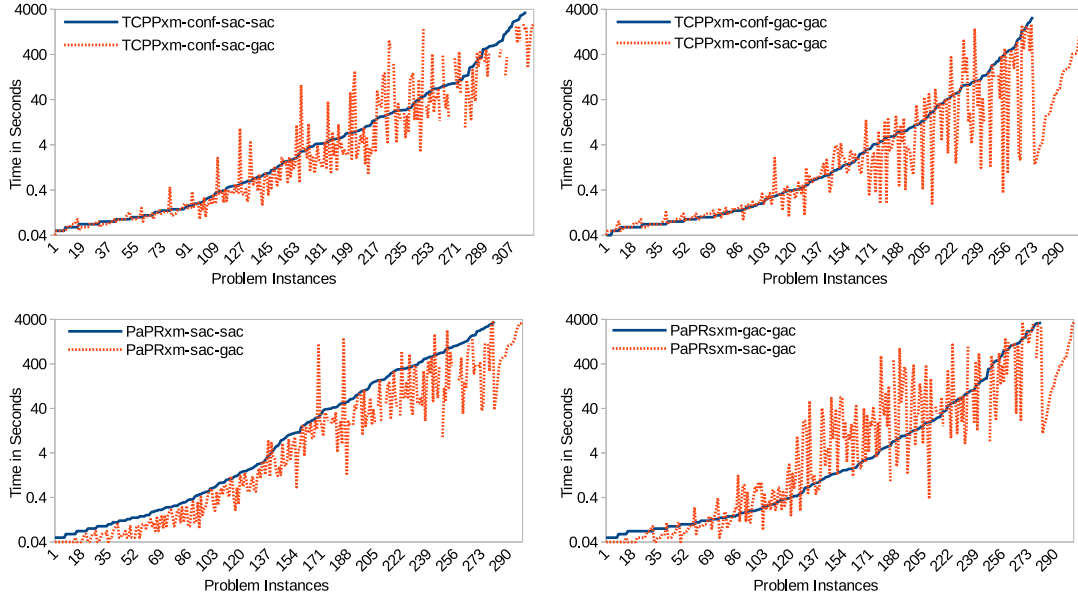


Figure 31: TCPPxm and PaPRxm with sac and gac in preprocessing and search

## 7.9 Effect of Graph Colouring

In Algorithm 5, we used a greedy graph colouring algorithm to assign labels to the vertexes in the conflicting graph. By using this algorithm we indeed reduce the domain size of parallelism variables. To see the effect of using this algorithm, we conducted an experiment in which we did not use any colouring algorithm and just labelled the vertexes with sequential numbers. Figure 32 shows the effect of the greedy graph colouring algorithm on different Minion configurations. As we observe, using fewer labels produced by the greedy graph colouring algorithm than using just sequential numbers do improve the performance over all four Minion configurations.

## 8. Conclusions

In this paper, we have described a constraint-based automated planner named Transition Constraints for Parallel Planning (TCPP). TCPP constructs its constraint model from domain transition graphs (DTGs) and encodes state transitions in the DTGs by table constraints allowing *don't cares*. We also have reconstructed the existing state-of-the-art planner PaP2 and have significantly enhanced it to obtain a new planner named PaPR by using *don't cares* and mutex constraints. Both TCPP and PaPR use Minion as their constraint solver. Our experiments on a number of standard planning benchmark domains demonstrate TCPP's efficiency over PaPR and PaP2, and also PaPR's efficiency over PaP2. In future, we will explore the effect of path heuristics in variable and value selection for constraint satisfaction models representing planning problems.

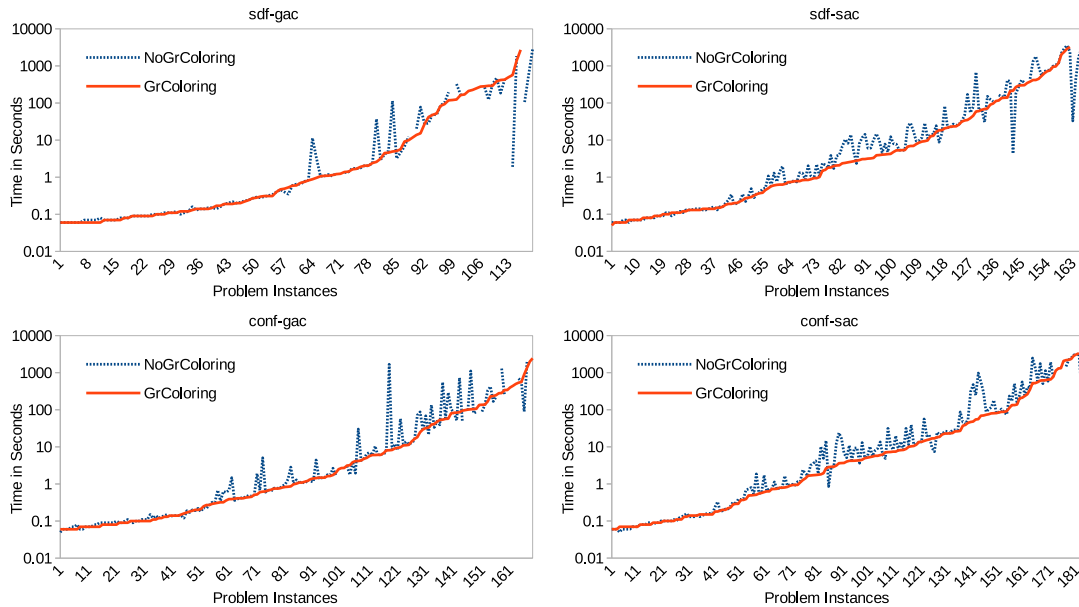


Figure 32: TCPPxm with graph colouring and without graph colouring

## Acknowledgments

A preliminary report of this work (a brief description of TCPP and its comparison with PaP2 and FastDownward+LMCut) has been published in the Proceedings of the Twenty-Ninth National Conference on Artificial intelligence (AAAI) (Ghooshchi et al., 2015); We thank the anonymous reviewers for their useful constructive feedbacks.

## References

- Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, 11, 625–656.
- Balafoutis, T., & Stergiou, K. (2008). Experimental evaluation of modern variable selection strategies in constraint satisfaction problems. In *Proceedings of the 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, online at CEUR Workshop Proceedings (ISSN 1613-0073)*, Vol. 451.
- Barták, R. (2011a). A novel constraint model for parallel planning. In *Proceedings of the International FLAIRS Conference*.
- Barták, R. (2011b). On constraint models for parallel planning: The novel transition scheme. In *Proceedings of the Eleventh Scandinavian Conference on Artificial Intelligence*, pp. 50–59.
- Barták, R., & Toropila, D. (2008). Reformulating constraint models for classical planning. In *Proceedings of the International FLAIRS Conference*, pp. 525–530.

- Barták, R., & Toropila, D. (2009a). Enhancing constraint models for planning problems. In *Proceedings of the International FLAIRS Conference*.
- Barták, R., & Toropila, D. (2009b). Revisiting constraint models for planning problems. In *International Symposium on Methodologies for Intelligent Systems*, pp. 582–591.
- Beacham, A., Chen, X., Sillito, J., & van Beek, P. (2001). Constraint programming lessons learned from crossword puzzles. In *Advances in Artificial Intelligence*, pp. 78–87. Springer.
- Bessiere, C. (2006). *Handbook of constraint programming (chapter3)*.
- Bessiere, C., Cardon, S., Debruyne, R., & Lecoutre, C. (2011). Efficient algorithms for singleton arc consistency. *Constraints*, 16(1), 25–53.
- Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1), 1636–1642.
- Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *ECAI*, Vol. 16, p. 146.
- Cesta, A., & Fratini, S. (2008). The timeline representation framework as a planning and scheduling software development environment. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group*.
- Do, M. B., & Kambhampati, S. (2001). Planning as constraint satisfaction: Solving the planning-graph by compiling it into CSP. *Artificial Intelligence*, 132, 151–182.
- Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4), 189–208.
- Fox, M., & Long, D. (2003). Pddl2. 1: An extension to pddl for expressing temporal planning domains.. *J. Artif. Intell. Res.(JAIR)*, 20, 61–124.
- Gent, I. P., Jefferson, C., & Miguel, I. (2006). MINION: a fast, scalable, constraint solver. In *Proceedings of the European Conference on Artificial Intelligence*.
- Ghallab, M., Knoblock, C., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith, D. E., Sun, Y., & Weld, D. (1998). PDDL: the planning domain definition language. In *Technical Report, Yale University*.
- Ghallab, M., & Laruelle, H. (1994). Representation and control in ixtet, a temporal planner.. In *AIPS*, Vol. 1994, pp. 61–67.
- Ghallab, M., Nau, D. S., & Traverso, P. (2004). *Automated Planning - Theory and Practice*. Elsevier.
- Ghooshchi, N. G., Namazi, M., Newton, M. H., & Sattar, A. (2015). Transition constraints for parallel planning. In *Proceedings of the Twenty-Ninth National Conference on Artificial intelligence (AAAI)*.
- Gregory, P., Long, D., & Fox, M. (2010). Constraint based planning with composable substate graphs. In *Proceedings of the European conference on Artificial Intelligence*, pp. 453–458.
- Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26, 191–246.

- Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5), 503–535.
- Hooker, J. N. (2005). A search-infer-and-relax framework for integrating solution methods. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 243–257. Springer.
- Huang, R., Chen, Y., & Zhang, W. (2010). A novel transition based encoding scheme for planning as satisfiability. In *Proceedings of the Twenty-Fourth National Conference on Artificial intelligence (AAAI)*.
- Jefferson, C., & Nightingale, P. (2013). Extending simple tabular reduction with short supports. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*.
- Judge, M., & Long, D. (2011). Heuristically guided constraint satisfaction for planning. In *Proceedings of the 29th Workshop of the UK Planning and Scheduling Special Interest Group*.
- Lecoutre, C. (2011). Str2: Optimized simple table reduction for table constraints. *Constraints*, 16(4), 341371.
- Lopez, A., & Bacchus, F. (2003). Generalizing graphplan by formulating planning as a CSP. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 954–960. Morgan Kaufmann Publishers.
- Rintanen, J., Heljanko, K., & Niemelä, I. (2006). Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12), 1031–1080.
- van Beek, P., & Chen, X. (1999). CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial intelligence, AAAI '99/IAAI '99*, pp. 585–590.
- Verfaillie, G., Pralet, C., & Lemaître, M. (2010). How to model planning and scheduling problems using constraint networks on timelines. *Knowledge Engineering Review*, 25(3), 319–336.
- Vidal, V. (2004). Branching and pruning: An optimal temporal POCL planner baed on constraint programming. In *Artificial Intelligence*, pp. 570–577.
- Vidal, V., & Geffner, H. (2006). Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3), 298335.
- Welsh, D. J., & Powell, M. B. (1967). An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1), 85–86.