Encoding Domain Transitions for Constraint-Based Planning

Nina Ghanbari Ghooshchi[a,b] nina.ghanabri@griffithuni.edu.au
Majid Namazi[a] m.namazi@griffith.edu.au
M.A.Hakim Newton[a] mahakim.newton@griffith.edu.au
Abdul Sattar [a,b] a.sattar@griffith.edu.au
[a] *Institute for Integrated and Intelligent Systems (IIIS), Griffith University, Brisbane, Australia*
[b] *Data61, CSIRO, NICTA, 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 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$^+$ 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 DTGs (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)[1]. 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$^+$ 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. TCPP encodes parallelism constraints by using negative table constraints to ensure parallel actions in a plan are non-conflicting. 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 model and returns the parallel plan. Minion can efficiently handle table constraints with

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)

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 (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 debark 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.

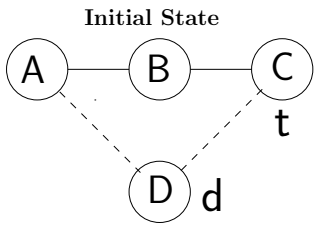
Table 1: Actions i.e. the instantiated or grounded operators in the driverlog domain. The action identification numbers are used henceforth to refer to these actions.

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.

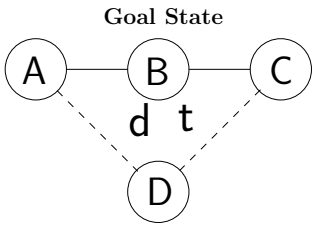
Definition 1 (Classical Planning Task)**.** *A* classical planning task $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ *comprises a set of* logical atoms $\mathcal{V}$*, an* initial state $\mathcal{I} \subseteq \mathcal{V}$*, a* goal state $\mathcal{G} \subseteq \mathcal{V}$*,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

An alternative to PDDL is to use the SAS$^+$ formalism (Bäckström & Nebel, 1995). The SAS$^+$ formalism has become popular after being used in Fast-Downward planner (Helmert, 2006). In the SAS$^+$ 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$^+$ 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$^+$ (i.e. they are actions) although in the figure, they are shown as parameterised. Since action prototypes in SAS$^+$ are the same as in PDDL, Table 1 lists the SAS$^+$ 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$^+$ planning task. Below we provide a formal definition of a multi-valued planning task (Helmert, 2006).



Plan

driver-walk(d,D,C)
embark-truck(d,t,C)
drive-truck(d,t,C,B)
debark-truck(d,t,B)



PDDL Description

Predicates:

at(o, l), empty(t), road(x, y),
path(x, y), driving(d, t)

Operators:

embark-truck(d, t, l)
  preconds:
    at(t, l), at(d, l), empty(t)
  effects:
    ¬at(d, l), ¬empty(t),
    driving(d, t)

debark-truck(d, t, l)
  preconds:
    at(t, l), driving(d, t)
  effects:
    at(d, l), empty(t),
    ¬driving(d, t)

drive-truck(d, t, l, l')
  preconds:
    at(t, l), driving(d, t),
    road(l, l')
  effects:
    ¬ at(t, l), at(t, l')

driver-walk(d, l, l')
  preconds:
    at(d, l), path(l, l')
  effects:
    ¬ at(d, l), at(d, l')

SAS$^+$ Description

Variables:

d-loc {A, B, C, D, t} // driver location
t-loc {A, B, C} // truck location
t-occ {false, true} // truck occupied

Operators:

embark-truck(d, t, l)
  preconds:
    t-loc $= l$, d-loc $= l$,
    t-occ $=$ false
  effects:
    d-loc $=$ t, t-occ $=$ true

debark-truck(d, t, l)
  preconds:
    t-loc $= l$, d-loc $=$ t
  effects:
    d-loc $= l$, t-occ $=$ false

drive-truck(d, t, l, l')
  preconds:
    t-loc $= l$, d-loc $=$ t
  effects:
    t-loc $= l'$

driver-walk(d, l, l')
  preconds:
    d-loc $= l$
  effects:
    d-loc $= l'$

SAS$^+$ actions are grounded,
although shown parameterised

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$^+$ (bottom-right).

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* goal state $\mathcal{G}$*, 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 goal state* $\mathcal{G}$ *is a partial assignment. Each action* $\alpha \in \mathcal{A}$ *has two partial assignments* p_α *and* e_α *that respectively denote the* preconditions *and the* effects *of* α. *Given the current state* s*, which is a complete assignment, an action* α *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* α *on the state* s *is denoted by* $s' = \mathsf{app}(s, \alpha)$ *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* α*, 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 α being applicable on a state s. For both types of planning, we also have provided the definitions of the effects of the action α in terms of the computation of $\mathsf{app}(s, \alpha)$ to produce the resultant state s' from a given state s where the preconditions of α 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* α*, we define the following mutually disjoint sets of variables that appear in the preconditions and/or effects of* α*:*

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

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

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

Definition 5 (Sequential Plan)**.** *A* sequential plan $\Sigma = \langle \alpha_1, \ldots, \alpha_n \rangle$ *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* $\langle s_0, s_1, \ldots, s_n \rangle$ *for which* (i) $s_0 = \mathcal{I}$, (ii) $\forall_{\tau \in [1,n]} \mathsf{app}(s_{\tau-1}, \alpha_\tau) = s_\tau$, *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* $\mathcal{P}$.

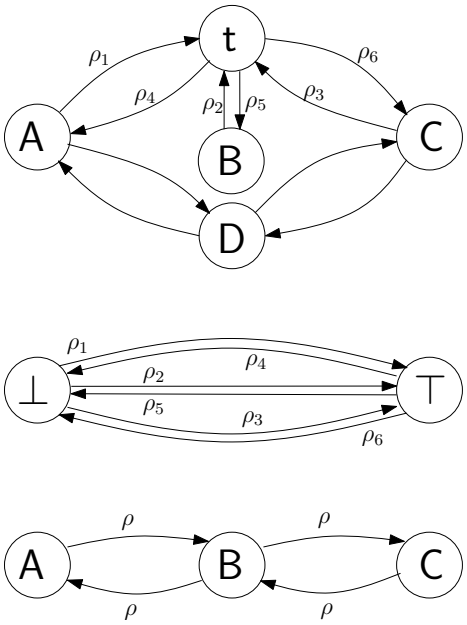
Definition 6 (Action Serialisability)**.** *A set of actions* A *is* serialisable *if for each permutation* $\vec{A} = \langle \alpha_1, \ldots, \alpha_{|A|} \rangle$ *of the actions in* A, $\mathsf{app}(s, \vec{A})$ *produces the same resulting state* s' *from a given state* s. *For a serialisable set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

Definition 7 (Parallel Plan)**.** *A* parallel plan $\Pi = \langle A^1, \ldots, A^m \rangle$ *of* makespan m *for a (classical or multi-valued) planning problem $\mathcal{P}$ is a sequence of sets of serialisable actions A^τ such that there exists a sequence of states $\langle s_0, s_1, \ldots, s_m \rangle$ for which* (i) $s_0 = \mathcal{I}$*,* (ii) *for all $\tau \in [1, m]$ $\mathsf{app}(s_{\tau-1}, A^\tau) = s_\tau$, 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 $\langle \alpha^\tau_1, \ldots \alpha^\tau_{|A^\tau|} \rangle$ of the actions in A^τ, for a parallel plan Π, we have a serialised plan $\vec{\Pi} = \langle \alpha^1_1, \ldots, \alpha^1_{|A^1|}, \ldots, \alpha^\tau_1, \ldots, \alpha^\tau_{|A^\tau|}, \alpha^{\tau+1}_1, \ldots, \alpha^{\tau+1}_{|A^{\tau+1}|}, \ldots, \alpha^m_1, \ldots, \alpha^m_{|A^m|} \rangle$. A serialised plan is thus a sequential plan of length $n = |A^1| + \ldots + |A^m|$.*

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

2.2 Domain Transition Graphs

From the SAS$^+$ 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.



DTG of d-loc
ρ_1: t-loc = A, t-occ = $\bot$
ρ_2: t-loc = B, t-occ = $\bot$
ρ_3: t-loc = C, t-occ = $\bot$
ρ_4: t-loc = A
ρ_5: t-loc = B
ρ_6: t-loc = C

DTG of t-occ
ρ_1: d-loc = A, t-loc = A
ρ_2: d-loc = B, t-loc = B
ρ_3: d-loc = C, t-loc = C
ρ_4: d-loc = t, t-loc = A
ρ_5: d-loc = t, t-loc = B
ρ_6: d-loc = t, t-loc = C

DTG of t-loc
ρ: d-loc = t

Figure 2: Domain transition graphs for the driverlog problem in Figure 1. The labels ρ_js 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 ρ_j written on it to hold before the transition. Boolean values are true $\top$ and false $\bot$.

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 $\mathsf{DTG}(v)$ *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 $\langle k, k' \rangle$ from k to k' with label $\rho = p_\alpha \setminus \{(v = k)\}$, if $v \in \mathsf{PandE}(\alpha)$, 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 $\rho = p_\alpha$, if $v \in \mathsf{EnotP}(\alpha)$ 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 ρ_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 ρ_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 ν (denoted by $x = \nu$) from its finite domain $D(x)$. An assignment σ 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 selects one row 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:

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

$x_1 + x_2 + x_3 < 3$

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

$x_1 \leq x_2$

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

$x_2 + 2x_3 \geq 3$

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 *$\widetilde{T}$ 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.

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

$x_1 + x_2 + x_3 < 3$

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

$x_1 \leq x_2$

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

$x_2 + 2x_3 \geq 3$

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

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

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.

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

$x_1 + x_2 + x_3 < 3$

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

$x_1 \leq x_2$

x_1	x_2
0	0
0	1
1	1

$x_2 + 2x_3 \geq 3$

x_2	x_3
0	2
1	1
1	2

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

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

Definition 11 (Arc-Consistency)**.** *Value ν 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 ν 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 ν 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 ν for variable x, denoted by $x = \nu$, is* singleton arc-consistent *if after assigning the value ν 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 ν 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 ν, $x = \nu$ 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, goal state, 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$^+$ 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.

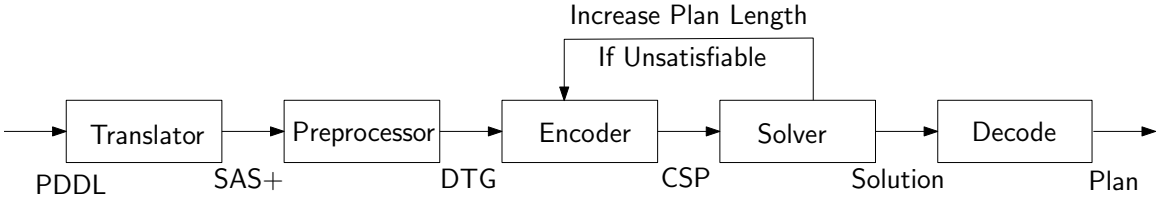


Figure 6: Architecture of our constraint-based planner

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$^+$ 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$^+$ translator, along with the SAS$^+$ 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-loc = t, t-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 goal state, we look in their DTGs for the shortest path from their values in the initial state to their values in the goal state. 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 τ and $\tau + 1$ to see which ones are changed. If a 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 variable changes from v_1 to v_2, we need to look at the edges from vertex v_1 to vertex v_2 in the DTG of this state variable. There may be more than one edge with different actions on them, but we need the action that its preconditions are satisfied at time τ and its effects are matching at time $\tau + 1$. Since there may be more than one action for each time, 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 α, if $v \in \mathsf{EnotP}(\alpha)$ 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 $\mathsf{DTG}(v)$ *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 $\mathsf{DTG}(v)$ 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.*

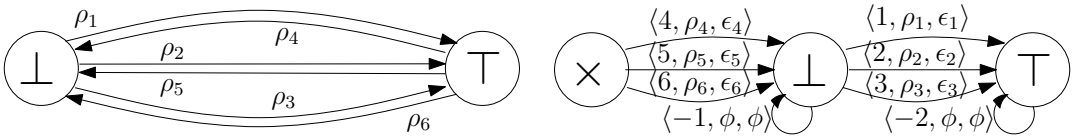
1. *For each action $\alpha \in \mathcal{A}$, there is an edge $\langle p_\alpha(v), e_\alpha(v) \rangle$ from $p_\alpha(v)$ to $e_\alpha(v)$ with label $\langle \alpha, \rho, \epsilon \rangle$, if $v \in \mathsf{PandE}(\alpha)$. Here, $\rho = (p_\alpha \setminus \{(v = k)\})$ and $\epsilon = (e_\alpha \setminus \{(v = k')\})$.*

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 $v \in \mathsf{EnotP}(\alpha)$. Here, $\rho = p_\alpha$ and $\epsilon = (e_\alpha \setminus \{(v = k')\})$.*

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 (α, ϕ, ϕ) where for each loop, α is a distinct* no-op *action with preconditions $\{(v = k)\}$ and no effects, and ϕ is an empty assignment.*

The original and the redefined domain transition graphs for variable t-occ in the driverlog problem in Figure 1 is 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.

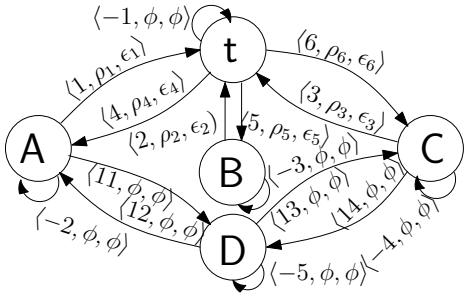
Algorithm 1 describes the procedure to construct the redefined DTGs in Definition 12. For each variable $v \in \mathcal{V}$, we have a DTG with vertexes for each $k \in \mathcal{D}(v) \cup \{\times\}$. Then, for each action $\alpha \in \mathcal{A}$, we look at the preconditions and effects of α. If a variable v appears in both preconditions and effects of α, then we add an edge $\langle p_\alpha(v), e_\alpha(v) \rangle$ to $\mathsf{DTG}(v)$. 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 $\mathsf{DTG}(v)$. There is no edge for the variables that appear only in the



ρ_1: d-loc = A, t-loc = A
ρ_2: d-loc = B, t-loc = B
ρ_3: d-loc = C, t-loc = C
ρ_4: d-loc = t, t-loc = A
ρ_5: d-loc = t, t-loc = B
ρ_6: d-loc = t, t-loc = C

ϵ_1: d-loc = t
ϵ_2: d-loc = t
ϵ_3: d-loc = t
ϵ_4: d-loc = A
ϵ_5: d-loc = B
ϵ_6: d-loc = C

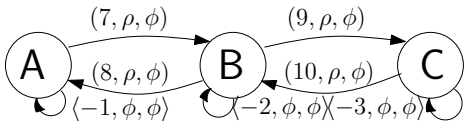
Figure 7: Original and redefined domain transition graphs for variable t-occ in the driverlog problem in Figure 1. Top-Left: original DTG as per Definition 8 and (Helmert, 2006); Top-Right: our redefined DTG. Positive numbers on the edges in our DTG denote the action identification numbers in Table 1.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



DTG of d-loc

ρ_1: t-loc = A, t-occ = $\bot$
ρ_2: t-loc = B, t-occ = $\bot$
ρ_3: t-loc = C, t-occ = $\bot$
ρ_4: t-loc = A
ρ_5: t-loc = B
ρ_6: t-loc = C

ϵ_1: t-occ = $\top$
ϵ_2: t-occ = $\top$
ϵ_3: t-occ = $\top$
ϵ_4: t-occ = $\bot$
ϵ_5: t-occ = $\bot$
ϵ_6: t-occ = $\bot$



DTG of t-loc

ρ: d-loc = t

Figure 8: Redefined domain transition graphs for variables d-loc and t-loc of the driverlog problem in Figure 1. The ρ_js 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 ϵ_js in the labels are the effects of the transitions on other variables. Boolean values are true $\top$ and false $\bot$; and *don't care* is $\times$.

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.

Algorithm 1 Construction of Redefined DTGs

```
//Add vertexes to DTGs
foreach variable v ∈ V
  foreach value k ∈ D(v)
    add vertex k to DTG(v)
  add vertex × to DTG(v)
//Add edges to DTGs
foreach action α ∈ A
  foreach variable v ∈ PandE(α)
    add edge ⟨p_α(v), e_α(v)⟩ to DTG(v)
  foreach variable v ∈ EnotP(α)
    add edge ⟨×, e_α(v)⟩ to DTG(v)
//Delete useless don't cares
foreach variable v ∈ V
  if no edge from vertex ×
    remove × from DTG(v)
```

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. □

Algorithm 2 A Non-Deterministic Search for Parallel Planning

```
func performSearch(P, m) returns Π
  P = ⟨V, I, G, A⟩: multi-valued plan prob
  m: makespan level to search the plan for
  foreach v ∈ V, curr(v) = I(v)//curr val
  Π = ⟨A^1, ..., A^m⟩ //initially all A^τ = φ
  foreach time step τ = 1 to m step 1
    A^τ = selectParallelActions(V)
    if A^τ = φ then return failure
    foreach action α ∈ A^τ//update vars
      if v ⇐ e_α then curr(v) = e_α(v)
  if ∃_{v⇐G} curr(v) ≠ G(v), return failure
  else return success and parallel plan Π

func selectParallelActions(V)
  E: set of |V| edges one for each v ∈ V from
    its DTG(v) such that the edge originates
    from vertex curr(v) or from the don't care
    vertex × (if any), and satisfies conditions
    ρ when the edge label is ⟨α, ρ, ϵ⟩
  if no two edges in E have actions α, α' in
      their labels such that α, α' have any
      parallel conflicts in their preconditions
      or in effects//E be constructed carefully
    return the set of actions in the edges in E
  else return φ
```

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, exactly one edge is selected from each DTG(v) for $v \in \mathcal{V}$ where the edge originates from the current vertex curr(v) in DTG(v) or from the *don't care* vertex × (if any) in the same DTG(v). The edge also needs to satisfy the preconditions in ρ related to other state variables where the edge label is $\langle \alpha, \rho, \epsilon \rangle$. 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.

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 τ, 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. Also, only one transition can happen at a time from the current state of a variable's DTG to another state in it. These are typical properties of a domain transition graph. 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 $\mathsf{PnotE}(\alpha)$, $\mathsf{EnotP}(\alpha)$, and $\mathsf{PandE}(\alpha)$ 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 α. 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. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PnotE}(\alpha')]$ *such that* $p_\alpha(v) \neq p_{\alpha'}(v)$.
2. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ *such that* $p_\alpha(v) \neq e_{\alpha'}(v)$ *or*
   $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{PnotE}(\alpha')]$ *such that* $e_\alpha(v) \neq p_{\alpha'}(v)$.
3. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PandE}(\alpha')]$ *or* $\exists v \in [\mathsf{PandE}(\alpha) \cap \mathsf{PnotE}(\alpha')]$.
4. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ *such that* $e_\alpha(v) \neq e_{\alpha'}(v)$.
5. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{PandE}(\alpha')]$ *or* $\exists v \in [\mathsf{PandE}(\alpha) \cap \mathsf{EnotP}(\alpha')]$.
6. $\exists v \in [\mathsf{PandE}(\alpha) \cap \mathsf{PandE}(\alpha')]$.

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 $\vec{A} = \langle \alpha_1, \ldots, \alpha_n \rangle$ 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 α, α' exists in A where $\alpha \otimes \alpha'$*

Proof. For the only if part, assume A is serialisable. As per Definition 6, $s' = \mathsf{app}(s, A) = \mathsf{app}(s, \vec{A})$ 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. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PnotE}(\alpha')]$ 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 α or α' is applicable on s but not both. This clearly contradicts the only if part of the first assertion of this lemma.

2. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ such that $p_\alpha(v) \neq e_{\alpha'}(v)$. This means α cannot be applied after α', contradicting our assumption that A is serialisable.

3. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PandE}(\alpha')]$. 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. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ such that $e_\alpha(v) \neq e_{\alpha'}(v)$. In this case $s'(v)$ will be $e_\alpha(v)$ if α is after α' and $e_{\alpha'}(v)$ if α' is after α. This means $s'(v)$ depends on the ordering of the actions, contradicting our assumption that A is serialisable.

5. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{PandE}(\alpha')]$. 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. $\exists v \in [\mathsf{PandE}(\alpha) \cap \mathsf{PandE}(\alpha')]$. 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 α, α' 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. □

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

Lemma 3 (Parallel Planning)**.** *Let* $\Pi = \langle A^1, \ldots, \ldots, A^m \rangle$ *be a sequence of sets of actions from a multi-valued planning task* $\mathcal{P} = \langle \mathcal{V}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$*.* Π *is parallel plan for* $\mathcal{P}$ *as per Definition 7 if and only if there exists a sequence of states* $\langle s_0, s_1, \ldots, s_m \rangle$ *such that* (i) $s_0 = \mathcal{I}$*,* (ii) *for all* $\tau \in [1, m]$*,* $\mathrm{app}(s_{\tau-1}, A^\tau) = s_\tau$*,* (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* $v \Leftarrow \mathcal{G}$*.*

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. □

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* $0, 1, \ldots, m$ *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 $\mathsf{DTG}(v)$ for each state variable v. In each iteration, it makes a transition at each $\mathsf{DTG}(v)$, 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. □

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^τ for each state variable v and timestep $\tau \in [0, m]$. Each CSP variable v^τ will have the same domain as the state variable v has. Thus, we have $(m+1)|\mathcal{V}|$ CSP variables in our model. To ensure action serialisability at each time step, we use parallelism constraints.

Moreover, for better efficiency of our planner, optionally we also use the mutex-groups that are produced by the PDDL to SAS$^+$ translator.

Algorithm 3 DTGs To CSP Encoding

```
1  Planing Problem P = ⟨V, I, G, A⟩, makespan m
2  //variables: v, v', values: k, k', actions: α, α'
3  Foreach (v, k) ∈ I, Add constraint eq(v^0, k)
4  Foreach (v, k) ∈ G, Add constraint eq(v^m, k)
5  For timestep τ = 0 to m − 1 do
6      Foreach v ∈ V, Add transition constraint tc(v^τ)
7      Foreach pair α, α' ∈ A such that α ⊗ α'
8          Add a parallelism constraint pc(α, α', τ)
9      // the mutex constraints below are optional
10 For timestep τ = 0 to m do
11     Foreach pair {v = k, v' = k'} in mutex groups
12         Add a mutex constraint mc(v, k, v', k', τ)
```

5.2 Initial and Goal State Constraints

To encode the initial state and the goal state 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 $v \in \mathcal{V}$. In the goal state, we only have the values for some of the state variables v^ms 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 τ 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 one action. Since this action has some preconditions that appear on the edges, the change can only occur if these preconditions 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 τ 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 τ 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 τ 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 $\langle$A, t, A, A, $\bot$, $\top\rangle$ for columns $\langle$d-loc$^{\tau}$, d-loc$^{\tau+1}$, t-loc$^{\tau}$, t-loc$^{\tau+1}$, t-occ$^{\tau}$, 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 τ 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 τ 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 DTG(v), the table T^{τ} at time τ has columns v^{τ}, $v^{\tau+1}$, $\bar{v}^{\tau}$s, $\bar{v}^{\tau+1}$s, where $\bar{v}$s are variables appearing on all edges of DTG(v) (Line 3). Each transition constraint is therefore an l-ary constraint where $l = 2(l' + 1)$ and l' is the number of variables appearing on the edges of DTG(v). Next, we extract the rows of the table for each edge in DTG(v) (Line 4). Suppose $\langle k, k' \rangle$ is an edge in DTG(v) and tr is the corresponding row to be added to the table. With respect to the edge, row 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^{τ} is k and $v^{\tau+1}$ is k'; note k could be $\times$ in the DTG(v) (Line 5-7). We then consider every variable $\bar{v}$ that appears on the ρ and ϵ components of the edge label $\langle \alpha, \rho, \epsilon \rangle$. There could be three possible cases depending on whether a variable appears either in ρ or in ϵ or in both. In an edge label, α is the action responsible for the transition. Lines 9-11 in Algorithm 4 cover the following three cases:

1. $\bar{v}$ **only in** ρ**:** In this case, $\bar{v}$ remains the same at times τ and $\tau + 1$ ensuring that during the execution of α, $\bar{v}$'s value does not change by any other action.

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

TC for t-loc			
t-loc		d-loc	
τ	$\tau+1$	τ	$\tau+1$
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 $\bot$; loc: location; occ: occupied; TC: transition constraint; $\times$: *don't care*; τ: time step.

2. $\bar{v}$ **in** ρ **and** ϵ **both:** In this case $\bar{v}$ changes the value between times τ and $\tau+1$. So the value at time τ 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 ρ and ϵ respectively.

3. $\bar{v}$ **only in** ϵ**:** In this case, we need to specify the value of $\bar{v}$ only at time $\tau+1$, which is $\epsilon(\bar{v})$. At time τ, the value of $\bar{v}$ is *don't care* $\times$, since $\bar{v} \not\Leftarrow \rho$.

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

DTG(v): DTG for the state variable v, T^τ: table for transitions of CSP variable v^τ
$V = \{v\} \cup \{\bar{v} : \bar{v} \Leftarrow \rho \vee \bar{v} \Leftarrow \epsilon$ where $\langle \alpha, \rho, \epsilon \rangle$ is the label of an edge in DTG$(v)\}$
Foreach $\bar{v} \in V$, table T^τ has two columns $\bar{v}^\tau$ and $\bar{v}^{\tau+1}$, for two time steps
Foreach edge $\langle k, k' \rangle$ in DTG(v) // *k, k' are values of v, k may equal to k'*
    tr: a new row in T^τ where each column has initially *don't care* $\times$
    // *Modify tr in the following ways to capture the transition*
    tr$[v^\tau] = k$, tr$[v^{\tau+1}] = k'$ // *note k could be a* don't care $\times$
    Assume $\langle \alpha, \rho, \epsilon \rangle$ is the label of the edge $\langle k, k' \rangle$ in the DTG(v)
    ForEach variable $\bar{v} \in V$ with $(\bar{v} \Leftarrow \rho \wedge \bar{v} \not\Leftarrow \epsilon)$, tr$[\bar{v}^\tau]$ = tr$[\bar{v}^{\tau+1}] = \rho(\bar{v})$
    ForEach variable $\bar{v} \in V$ with $(\bar{v} \Leftarrow \rho \wedge \bar{v} \Leftarrow \epsilon)$, tr$[\bar{v}^\tau] = \rho(\bar{v})$, tr$[\bar{v}^{\tau+1}] = \epsilon(\bar{v})$
    ForEach variable $\bar{v} \in V$ with $(\bar{v} \not\Leftarrow \rho \wedge \bar{v} \Leftarrow \epsilon)$, tr$[\bar{v}^{\tau+1}] = \epsilon(\bar{v})$

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 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. On the other hand, 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.

PDDL representation

Predicates:
clear(x), ontable(x),
empty, holding(x),
on(x, y)
Actions:
pick-up(x)
preconds: empty, clear(x),
ontable(x)
effects: ¬clear(x), ¬ontable(x),
¬empty, holding(x)
put-down(x)
preconds: holding(x)
effects: ¬holding(x), clear(x),
empty, ontable(x)
stack(x, y)
preconds: holding(x), clear(y)
effects: ¬holding(x), ¬clear(y),
clear(x), on(x, y), empty
unstack(x, y)
preconds: empty, on(x, y),
clear(x)
effects: ¬clear(x), ¬on(x, y),
¬empty, holding(x), clear(y)

SAS$^+$ representation

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 = on$y$
unstack(x, y)
preconds: x-top = C, hand = E,
x-loc = on$y$
effects: x-top = NC, y-top = C,
hand = NE, x-loc = H

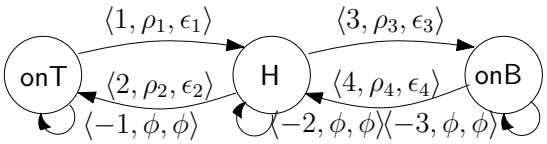
Figure 10: The blocks world domain in PDDL and SAS$^+$

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$^+$ 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 on$x$ 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.

Actions in the Blocks World Domain			
1 pick-up(A)	2 put-down(A)	3 stack(A,B)	4 unstack(A,B)
5 pick-up(B)	6 put-down(B)	7 stack(B,A)	8 unstack(B,A)

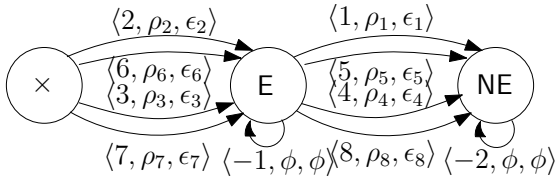


DTG of A-loc

ρ_1: A-top = C, hand = E
ρ_2: ϕ
ρ_3: B-top = C
ρ_4: A-top = C, hand = E

ϵ_1: A-top = NC, hand = NE
ϵ_2: A-top = C, hand = E
ϵ_3: A-top = C, B-top = NC, hand = E
ϵ_4: A-top = NC, B-top = C, hand = NE

DTG of hand



ρ_1: A-top = C, A-loc = onT
ρ_2: A-loc = H
ρ_3: B-top = C, A-loc = H
ρ_4: A-top = C, A-loc = onB
ρ_5: B-top = C, B-loc = onT
ρ_6: B-loc = H
ρ_7: A-top = C, B-loc = H
ρ_8: B-top = C, B-loc = onA

ϵ_1: A-top = NC, A-loc = H
ϵ_2: A-top = C, A-loc = onT
ϵ_3: A-top = C, B-top = NC, A-loc = onB
ϵ_4: A-top = NC, B-top = C, A-loc = H
ϵ_5: B-top = NC, B-loc = H
ϵ_6: B-top = C, B-loc = onT
ϵ_7: A-top = NC, B-top = C, B-loc = onA
ϵ_8: A-top = C, B-top = NC, B-loc = H

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.

We only depict the DTGs of variables hand and A-loc in Fiqure 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 ρ_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 ϵ_2. Similarly, other edges in DTG(hand) and all edges in DTG(A-loc) could be constructed by using Algorithm 1.

TC for hand									
hand		A-top		B-top		A-loc		B-loc	
τ	$\tau+1$	τ	$\tau+1$	τ	$\tau+1$	τ	$\tau+1$	τ	$\tau+1$
×	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	
τ	$\tau+1$	τ	$\tau+1$	τ	$\tau+1$	τ	$\tau+1$
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.

The transition constraints for the two DTGs are in Fiqure 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 τ and $\tau+1$. The first row of the table is for edge from × 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 × and C for variables A-top at time τ 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.

5.4 Parallelism Constraints

As described already, the transitions 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 and 8 of Algorithm 3), we have parallelism constraints that capture the conditions in Definition 13. These parallelism constraints are represented by negative table constraints.

Example. Let us see an example of parallelism constraints. Figure 13 shows the constraint for two actions α and α' such that $\alpha \otimes \alpha'$ as per Condition 6 in Definition 13. To make it clearer, α and α' conflicts because both actions have v_3 in their effects as well as in their preconditions. We use a negative table constraint to represent this. The table has the columns v_1^τ, v_3^τ, $v_3^{\tau+1}$, $v_4^{\tau+1}$, v_2^τ, v_5^τ, $v_5^{\tau+1}$ one at time τ for each variable that appears in the preconditions of either action, and another at time $\tau+1$ for each variable that appears

in the effects of either action. There is only one row in this table listing the values for the variables. During search this combination of variables and values will be avoided.

action α
preconds:
$v_1 = 1, v_3 = 2$
effects:
$v_3 = 3, v_4 = 4$

action α'
preconds:
$v_2 = 5, v_3 = 2, v_5 = 6$
effects:
$v_3 = 3, v_5 = 7$

Negative Table PC for Action Pair α, α'						
v_1^τ	v_3^τ	$v_3^{\tau+1}$	$v_4^{\tau+1}$	v_2^τ	v_5^τ	$v_5^{\tau+1}$
1	2	3	4	5	6	7

Figure 13: Negative table for a parallelism constraint between two actions

We then describe how our model prevents conflicting actions from happening at the same time. From the six types of condition in Definition 13, some are prevented automatically in our model. For the others, we add negative tables to prevent them.

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 analyse one after another each condition of Definition 13 below.

1. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PnotE}(\alpha')]$ such that $p_\alpha(v) \neq p_{\alpha'}(v)$. For α, $v^\tau = p_\alpha(v)$ but for α', $v^\tau = p_{\alpha'}(v)$. However, v^τ 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. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ such that $p_\alpha(v) \neq e_{\alpha'}(v)$. For α, $v^{\tau+1} = v^\tau = p_\alpha(v)$ but $v^{\tau+1} = e_{\alpha'}(v)$ for α'. 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. $\exists v \in [\mathsf{PnotE}(\alpha) \cap \mathsf{PandE}(\alpha')]$. For α, $v^\tau = v^{\tau+1} = p_\alpha(v)$ and for α', $v^\tau = p_{\alpha'}(v)$, $v^{\tau+1} = e_{\alpha'}(v)$. 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. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{EnotP}(\alpha')]$ such that $e_\alpha(v) \neq e_{\alpha'}(v)$. For α, $v^{\tau+1} = e_\alpha(v)$ but for α', $v^{\tau+1} = e_{\alpha'}(v)$. 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. $\exists v \in [\mathsf{EnotP}(\alpha) \cap \mathsf{PandE}(\alpha')]$. For α, $v^{\tau+1} = e_\alpha(v)$ and for α', $v^\tau = p_{\alpha'}(v)$ 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 add a parallelism constraint, since at one time two different transitions cannot take place within the same $\mathsf{DTG}(v)$.

6. $\exists v \in [\mathsf{PandE}(\alpha) \cap \mathsf{PandE}(\alpha')]$. For α, $v^\tau = p_\alpha(v)$ and $v^{\tau+1} = e_\alpha(v)$ where $p_\alpha(v) \neq e_\alpha(v)$. For α', $v^\tau = p_{\alpha'}(v)$ 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^τ 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 add a parallelism constraint, since at one time two different transitions cannot take place within the same $\mathsf{DTG}(v)$.

We therefore need constraints only for Conditions 5 and 6, and still partially. □

Algorithm 5 Constructing Negative Tables for Parallelism Constraints

proc PrallelismConstraints(τ)
  foreach $\bar{v} \in \mathcal{V}$ **foreach** $\alpha : \bar{v} \in \mathsf{PandE}(\alpha)$
    foreach $\alpha' : \bar{v} \in \mathsf{EnotP}(\alpha')$
      if $e_\alpha(\bar{v}) = e_{\alpha'}(\bar{v})$
        if $\neg$**ElsePrevented**(α, α')
          add neg table pc(α, α', τ)
    //Condition 5 above, Condition 6 below
    foreach $\alpha' : \bar{v} \in \mathsf{PandE}(\alpha')$
      if $p_\alpha(\bar{v}) = p_{\alpha'}(\bar{v}) \wedge e_\alpha(\bar{v}) = e_{\alpha'}(\bar{v})$
        if $\neg$**ElsePrevented**(α, α')
          add neg table pc(α, α', τ)
  //Optionally merge tables to reduce the number
  foreach two tables with the same columns
    gather rows from both into one table

func ElsePrevented(α, α') 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** $v \neq v' \wedge$ mutex$\{v = p_\alpha(v), v' = p_{\alpha'}(v')\}$
      // **then return true**
  foreach $v \Leftarrow e_\alpha$ and **foreach** $v' \Leftarrow e_{\alpha'}$
    if $v = v' \wedge e_\alpha(v) \neq e_{\alpha'}(v')$
      then return true
    //*uncomment below if mutexes used*
    //**if** $v \neq v' \wedge$ mutex$\{v = e_\alpha(v), v' = e_{\alpha'}(v')\}$
      // **then return true**
  return false

proc pc(α, α', τ) constructs a negative table
  $\widetilde{T}^\tau$: a neg table for parallelism constraint
  foreach $v \Leftarrow (\mathsf{PnotE}(\alpha) \cup \mathsf{PnotE}(\alpha'))$ then $\widetilde{T}^\tau$ has a column v^τ
  foreach $v \Leftarrow (\mathsf{EnotP}(\alpha) \cup \mathsf{EnotP}(\alpha'))$ then $\widetilde{T}^\tau$ has a column $v^{\tau+1}$
  foreach $v \Leftarrow (\mathsf{PandE}(\alpha) \cup \mathsf{PandE}(\alpha'))$ then $\widetilde{T}_\tau$ has two columns v^τ and $v^{\tau+1}$
  tr: a new row in $\widetilde{T}_\tau$ // all cells will be assigned below
  if $v \in \mathsf{PnotE}(\alpha)$ **then** tr$[v^\tau] = p_\alpha(v)$
  if $v \in \mathsf{PnotE}(\alpha')$ **then** tr$[v^\tau] = p_{\alpha'}(v)$
  if $v \in \mathsf{EnotP}(\alpha)$ **then** tr$[v^{\tau+1}] = e_\alpha(v)$
  if $v \in \mathsf{EnotP}(\alpha')$ **then** tr$[v^{\tau+1}] = e_{\alpha'}(v)$
  if $v \in \mathsf{PandE}(\alpha)$ **then** tr$[v^\tau] = p_\alpha(v)$, tr$[v^{\tau+1}] = e_\alpha(v)$
  if $v \in \mathsf{PandE}(\alpha')$ **then** tr$[v^\tau] = p_{\alpha'}(v)$, tr$[v^{\tau+1}] = e_{\alpha'}(v)$
  return $\widetilde{T}^\tau$

In Algorithm 5, we describe the algorithm to compute the negative tables for parallelism constraints. Instead of Lines 7-8 in Algorithm 3, we actually call Procedure PrallelismConstraints in Algorithm 5. Since actions having parallel conflicts must have a shared variable, we compute the parallelism constraints among the actions in the edges of the DTGs. Pairwise all actions appearing in the edges of a DTG are to be considered for parallel conflict, but only for Conditions 5 and 6 of Definition 13. However, we do some optimisation during construction of the constraints. If a conflict is detected for the given variable v based on Condition 5 or 6, we do not directly encode it as a negative table. Rather we check whether the same action pair also has an automatically prevented conflict on some other variable. If so, we do not need any constraint for this case. This is done in Function ElsePrevented. Nevertheless, if a negative table is needed, it is constructed in Procedure pc(α, α', τ) where the columns are identified from the variables and then a row entry is made using the values of the variables. To increase efficiency, lastly in Procedure PrallelismConstraints, any two tables having the same columns are merged into one table by putting the rows together. This is possible because in a negative table all rows are avoided all the time.

5.5 Mutex Constraints

During translation from PDDL to SAS$^+$, the translator (Helmert, 2006), along with the SAS$^+$ 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 $\{(\text{d-loc} = \text{t}), (\text{t-occ} = \bot)\}$, which denotes that when the variable d-loc takes the value t, the variable t-occ cannot take the value false and vice versa. In blocks world example, we have a mutex-group $\{(\text{A-top} = \text{C}), (\text{A-loc} = \text{H}), (\text{B-loc} = \text{onA})\}$. 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 14 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 $\{(\text{B-top} = \text{C}), (\text{A-loc} = \text{onB}), (\text{B-loc} = \text{H})\}$ and $\{(\text{hand} = \text{E}), (\text{A-loc} = \text{H}), (\text{B-loc} = \text{H})\}$ that insert two other rows $\langle \text{onB}, \text{H} \rangle$ and $\langle \text{H}, \text{H} \rangle$ to the mutex table for A-loc$^\tau$ and B-loc$^\tau$.

A-top$^\tau$	A-loc$^\tau$
C	H

A-top$^\tau$	B-loc$^\tau$
C	onA

A-loc$^\tau$	B-loc$^\tau$
H	onA

Figure 14: 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 might be able to throw away some of the parallelism constraints. If a parallelism constraint is subsumed by a mutex-group, we do not create a negative table for the parallelism constraint. 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^τs. We then need to extract the final plan from the assignment. To do so, at each time step, we extract each state variable v whose value is changed in that time step. If the value of v is changed from k at time step τ to k' at time step $\tau + 1$, we need to check the actions on the labels of the edges from k to k' in the DTG of v. The action whose preconditions are satisfied at τ and whose effects are satisfied at $\tau + 1$ is the action that is responsible for the change. Algorithm 6 describes the procedure.

Lemma 6 (CSP to Plan)**.** *There is one parallel plan for each CSP solution returned.*

Algorithm 6 Decoding CSP Soluton to Plan

input m: makespan value when the CSP solution is found
input $\{(v^\tau = k) : \tau \in [0, m] \wedge v \in \mathcal{V}\}$: the CSP solution
output: parallel plan $\Pi = \langle A^1, \ldots, A^m \rangle // A^\tau$ actions at τ
foreach time step $\tau = 1$ to m // number of action layers
  foreach v such that $v^{\tau-1}, v^\tau$ have different values
    foreach edge $\langle v^{\tau-1}, v^\tau \rangle$ with label $\langle \alpha, \rho, \epsilon \rangle$ in DTG(v)
      if $\forall(\bar{v} \Leftarrow \rho)[\rho(\bar{v}) = \bar{v}^{\tau-1}]$ and $\forall(\bar{v} \Leftarrow \epsilon)[\epsilon(\bar{v}) = \bar{v}^\tau]$
        add α to A^τ, break // only one for each v

Proof. The proof is straightforward from Algorithm 6. In the algorithm, for each state variable, we select one action that changes its value between two time steps. □

5.7 Correctness and Complexities

We prove the correctness of our encoding and decoding algorithms showing how they corresponds 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. The parallel plan can be correctly decoded from the CSP solution by using Algorithm 6.*

Proof. Given the set of multi-valued variables $\mathcal{V}$, the number of DTGs is $|\mathcal{V}|$. We consider a CSP variable v^τ for each state variable $v \in \mathcal{V}$ and for each time step $0 \leq \tau \leq m$. The domains of these CSP variables are the same as v's domain. These CSP variables capture the values of the state variables at each time step. We use constraints to specify the values of the CSP variables at time step 0 and m, which exactly models the initial and goal state constraints mentioned in (i) and (iv) of Lemma 3. Our transition constraints constructed by Algorithm 4 clearly specify the values of the CSP variables at time step $\tau + 1$ given their values at τ. Only one row from each respective table is selected at a time ensuring exactly one action to be selected from each DTG. A transition constraint table contains each transition in the respective DTG. All these ensures (ii) from Lemma 3. Lastly, conflicting action pairs (as per Definition 13) are directly encoded in the negative tables for the parallelism constraints. This corresponds to (iii) of Lemma 3. The constraint model is thus correct. If a complete CSP solver finds a solution, we will get the values of the CSP variables. If v^τ and $v^{\tau+1}$ are the same for a state variable v, then a *no-op* has been selected; otherwise, using other preconditions and effects, we identify the action involved in the change. Between each τ and $\tau+1$, we can have one action per state variable. Algorithm 6 exactly finds these actions and thus correctly returns a parallel plan. Now, if the CSP solver does return a solution, the solution certainly satisfies the initial and the goal constraints, the values of v^τs and $v^{\tau+1}$s also satisfy the transition constraints and the parallelism constraints. So the actions responsible for the changes of values from v^τs to $v^{\tau+1}$s represent the transitions on the respective DTGs. Moreover, values of all v^τs constitute the state at time step τ. So as per Lemma 3, the plan decoded is indeed a parallel plan and the process is parallel planning. □

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, parallelism and* mutex *constraints are all constructed once and used for each time step. In Algorithm 3, construction of*

1. *all initialisation and goal state 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. *all parallelism constraints by using Algorithm 5 require $O(|\mathcal{V}|\widehat{E}^2\widehat{V_a}^2\widehat{M} + \widehat{M}\widehat{V_m})$ time and $O(\widehat{E}^2\widehat{V_a})$ 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);*

$\widehat{M}$ *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 goal states 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, Procedure PrallelismConstraints in its three-level nested loops needs $|\mathcal{V}|\widehat{E}^2$ times where $\widehat{E}$ is the maximum number of edges in a DTG. 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, Procedure pc requires $O(\widehat{V_a})$ time to construct a table. There are $O(\widehat{E}^2)$ tables each with $O(\widehat{V_a})$ columns and just one row. Cost of table merging will be similar to that of table creation.
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. □

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 $\widehat{V_a}$ variables. □

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 constraints 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*. While synchronisation constraints are implicit in our transition constraints, parallelism constraints are implicit in action succession constraints in PaP2. Nevertehless, 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 14 and also show FSA for the driverlog example in Figure 15.

Definition 14 (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 $\mathsf{FSA}(v)$ *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* α*, if* $v \in \mathsf{PandE}(\alpha)$*, 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 α, if $v \in \mathsf{EnotP}(\alpha)$ 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 α where for each loop, α is a distinct* no-op *action with preconditions $\{(v = k)\}$ and no effects.*

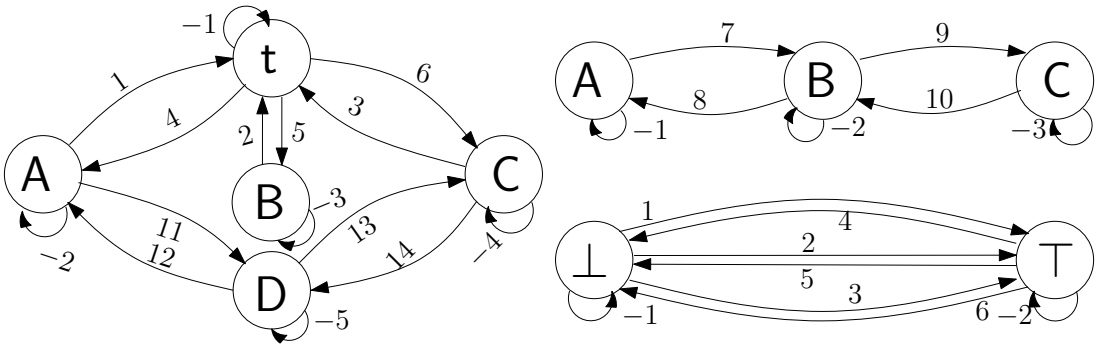


Figure 15: FSA for the driverlog problem in Figure 1. Left: FSA for state variable d-loc, Right-Top: FSA for state variable t-loc, and Right-Bottom: FSA for state variable t-occ. In the labels of the edges the positive numbers are the action identification numbers from Table 1 and the negative numbers are distinct identifiers for no-ops or loops. Boolean values are true $\top$ and false $\bot$.

6.2 Encoding FSA into CSP

In PaP2, for each state variable v at time τ, a CSP variable v^τ is considered. The domain of v^τ comprises all the actions occurring on the edges of $\mathsf{FSA}(v)$. 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$^+$ translator. Algorithm 7 describes PaPR's encoding procedure.

6.3 Initial and Goal State Constraints

To encode the initial state and the goal state 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 the initial value of the variable. In the goal state, 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

Algorithm 7 FSA To CSP Encoding

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

time m. These actions are on the incoming edges of the vertex representing the value of the variable in a goal state.

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 α can be *followed by* another action α' 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 α' are compatible with the effects of α. Algorithm 8 constructs the table for the action succession constraint between variables v^τ and $v^{\tau+1}$. For a given FSA, all actions α' that can follow a given action α are however stored as a set in the same cell of the table to obtain a compact representation.

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

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

Figure 16 shows the action succession constraints obtained for the FSA in Figure 15 for the driverlog example. In the first row of the table for d-loc in Figure 16, we see action 1 can be followed by actions -1, 4, 5, 6. Verify these from the FSA of d-loc in Figure 15 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
τ	$\tau+1$
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
τ	$\tau+1$
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
τ	$\tau+1$
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 16: Action succession constraints from the FSA in Figure 15 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 τ and is encoded as a table. For all variables v, this constraint will have a column v^τ. 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 α 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 α. Variables just in the preconditions of α 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 17 shows the tables for the synchronisation constraints obtained from the FSA in Figure 15 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 $\mathsf{sc}(v^\tau)$ using a table

```
1  T^τ: synchronisation table for variable v^τ
2  D(v): the domain of the state variable v
3  foreach v̄ ∈ V, table T^τ has a column v̄^τ
4  foreach edge in FSA(v) with label α
5     tr: a new row in T^τ, each column v̄^τ
6        has initially value D(v̄), but tr[v^τ] = α
7     foreach v̄ ∈ V with v̄ ⇐ p_α ∧ v̄ ⇍ e_α
8        tr[v̄]= the no-op at vertex p_α(v̄) in FSA(v̄)
9     foreach v̄ ∈ V with v̄ ⇐ e_α, tr[v̄] = α
10 delete all columns v^τ with all cells having values D(v)
11 merge rows into one if they differ only in one column
```

SC for d-loc		
d-loc	t-loc	t-occ
τ	τ	τ
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
τ	τ	τ
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
τ	τ
7	-1
8	-1
9	-1
10	-1
S_6	S_1

Figure 17: Synchronisation constraints from the FSA in Figure 15 for driverlog example. In the tables, $S_1 = \{-5, \ldots, -1, 1, \ldots, 6, 11, \ldots, 14\}$, $S_2 = \{-3, -2, -1, 7, \ldots, 10\}$, $S_3 = \{-1, -2, 1, \ldots, 6\}$, $S_4 = \{-5, \ldots, -1, 11, \ldots, 14\}$, $S_5 = \{-1, -2\}$, 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 $\mathsf{mc}(v, k, \bar{v}, \bar{k}, \tau)$ with columns $v^\tau, \bar{v}^\tau$ is added. The rows in this table will be the pairs of actions $\alpha, \bar{\alpha}$ where α is on the label of an outgoing edge of vertex k in $\mathsf{FSA}(v)$ and $\bar{\alpha}$ is on the label of an outgoing edge of vertex $\bar{k}$ in $\mathsf{FSA}(\bar{v})$. The table also has the rows for actions pairs $\alpha, \bar{\alpha}$

where α is on the label of an incoming edge of vertex k in $\mathsf{FSA}(v)$ and $\bar{\alpha}$ is on the label of an incoming edge of vertex $\bar{k}$ in the $\mathsf{FSA}(\bar{v})$. 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 $\mathsf{mc}(v, k, \bar{v}, \bar{k}, \tau)$ using a negative table

```
1  T^τ: mutex constraint table with columns v^τ and v̄^τ
2  for all edges ⟨k, k'⟩ in FSA(v) with label α
3      for all edges ⟨k̄, k̄'⟩ in FSA(v̄) with label ᾱ
4          tr: a new row in T^τ with tr[v^τ] = α, tr[v̄^τ] = ᾱ
5  for all edges ⟨k', k⟩ in FSA(v) with label α
6      for all edges ⟨k̄', k̄⟩ in FSA(v̄) with label ᾱ
7          tr: a new row in T^τ with tr[v^τ] = α, tr[v̄^τ] = ᾱ
```

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 and goal states 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 $\mathsf{FSA}(v)$.
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^τ 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. □

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-notankage, 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$^+$ 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$^+$ 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 18 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.

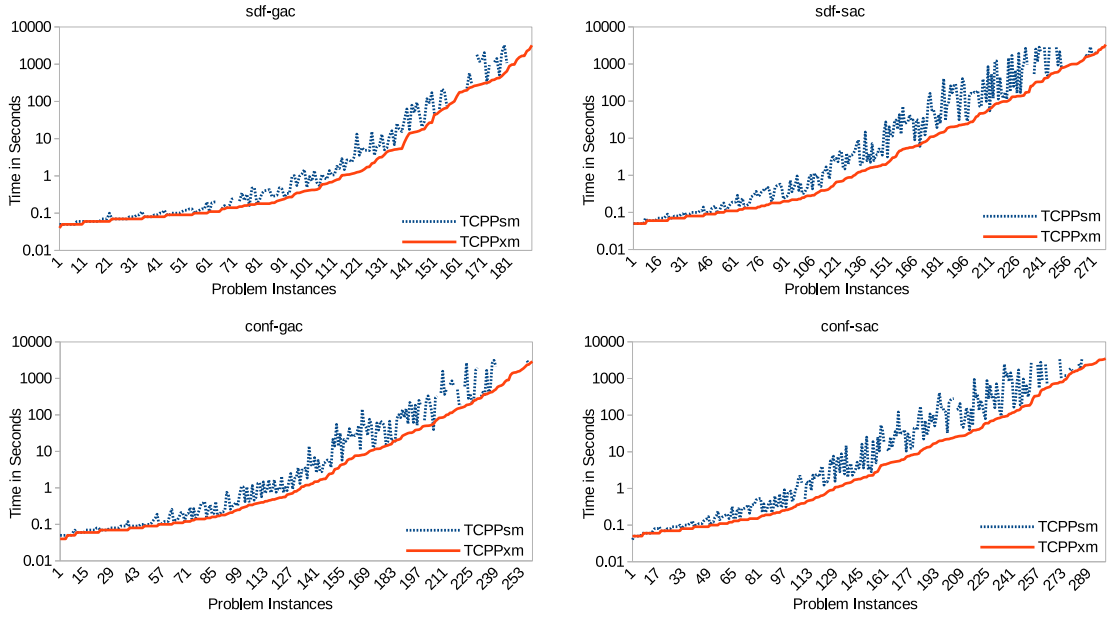


Figure 18: Enhancing TCPP using *don't cares* rather than sets.

Figure 19 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. Moreover, we also eliminate certain rows from the negative tables for the parallelism constraints when those rows are subsumed by two variable-value pairs in a mutex-group (see Algorithm 5 Function ElsePrevented). This is because smaller *nogoods* are more effective than subsumed larger *nogoods* in terms of the area of the search space that could be pruned out. The optional use of mutex-groups in our encoding thus brings about more efficiency in TCPPxm than in TCPPx.

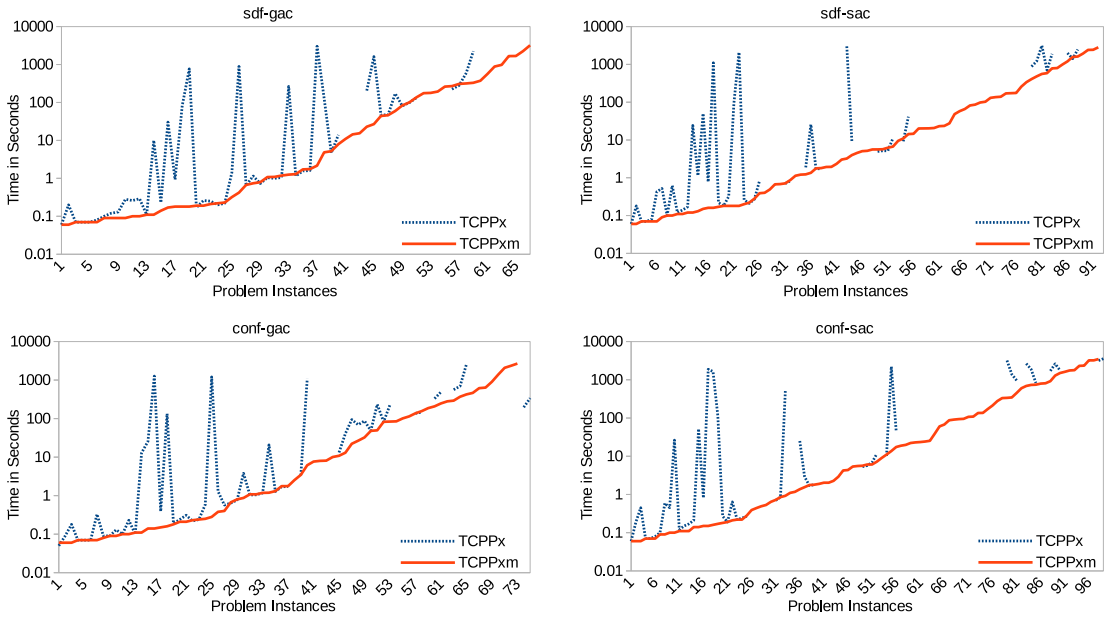


Figure 19: Enhancing TCPP using mutex-groups.

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 300 instances in total, while the second best TCPPxm-sdf-sac solves 278 instances and the third best TCPPsm-conf-sac solves 269 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

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	14	21	19	21	20	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	2	2	2	2	2	2	2	2	1	2	2	2	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	14	22	22	21	14	24	22	25	14	24	22	25	no
mystery	30	9	14	12	11	9	15	12	14	9	15	12	14	no
pathways	30	4	5	4	5	4	5	4	5	4	5	4	5	no
pipes-no-tankage	50	4	4	5	4	4	4	5	4	4	4	5	4	no
pipes-tankage	50	3	5	5	6	3	6	5	6	3	4	5	5	yes
psr-small	50	20	38	42	42	21	41	43	44	21	41	43	44	no
rovers	40	8	16	17	19	8	16	18	20	8	16	18	20	no
satellite	36	2	3	5	5	2	5	6	6	2	5	6	6	no
storage	30	8	8	9	8	8	8	9	10	7	7	9	8	yes
tpp	30	6	8	8	9	6	8	9	10	6	8	9	10	no
zenotravel	20	8	9	12	12	9	9	12	12	9	9	12	12	no
Total	786	173	257	241	269	190	278	259	300	176	243	245	261	

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.

7.2 Reconstructing PaPR from PaP2

Figure 20 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 21, 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

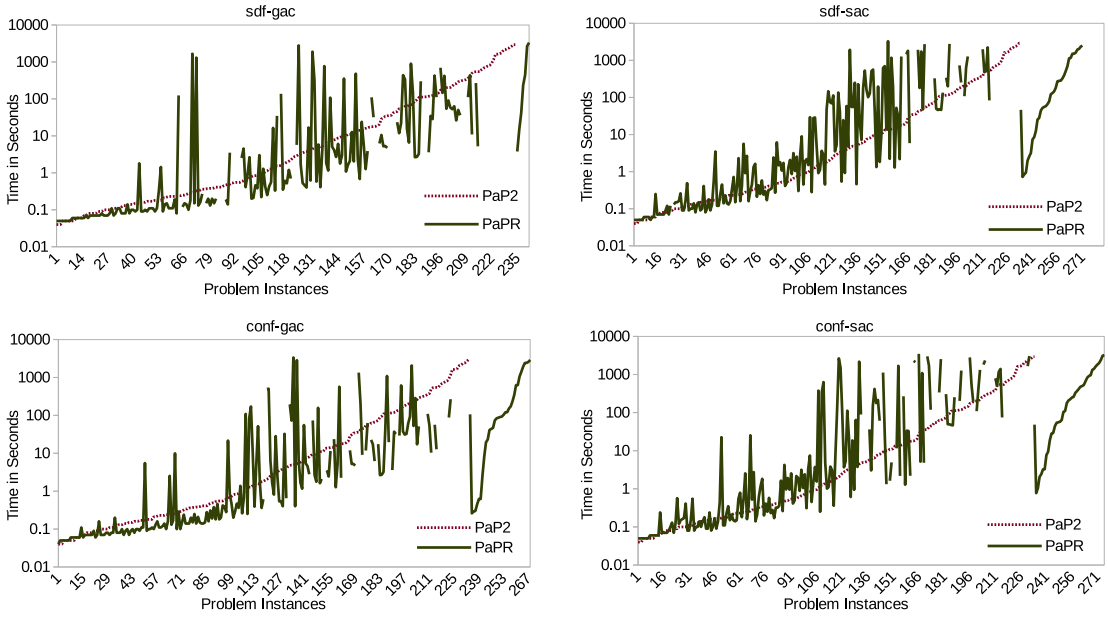


Figure 20: Reconstructing PaP2 to obtain PaPR that runs on Minion.

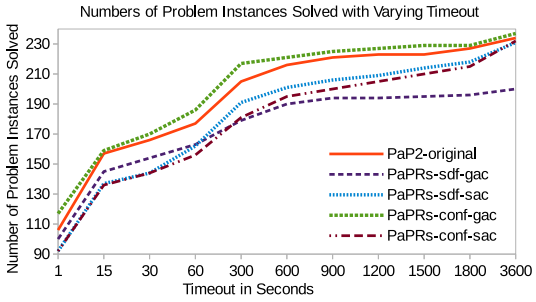


Figure 21: PaP2 and PaPR with different timeout limits.

Prolog and Minion. The exact reason behind this depends on the specific details of the respective solvers.

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

Figure 22 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 23 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 per-

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

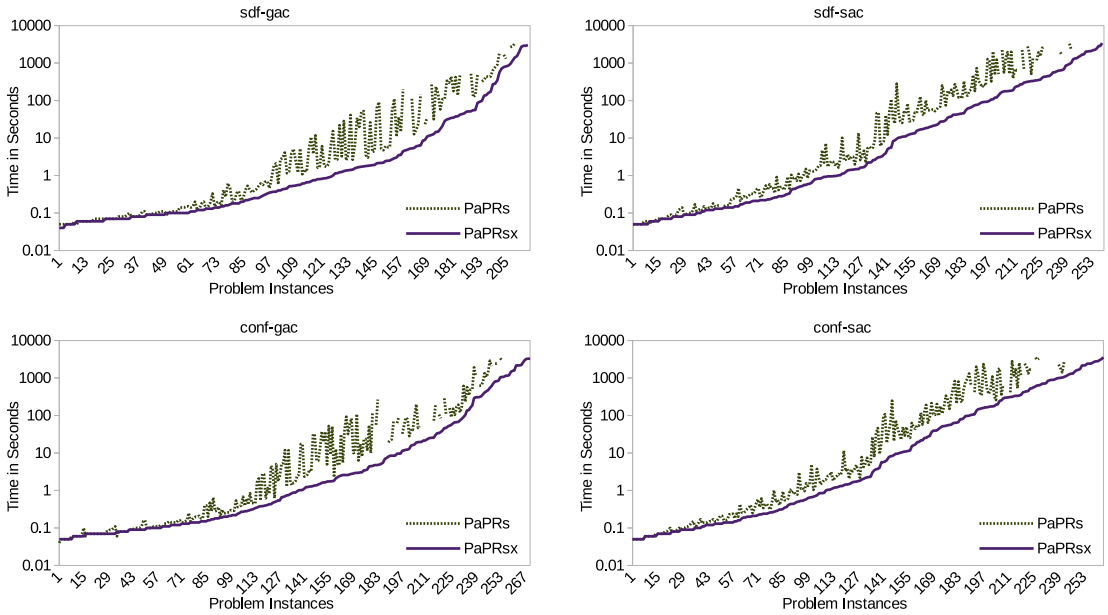


Figure 22: Enhancing PaPR using *don't cares*

forms 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 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.

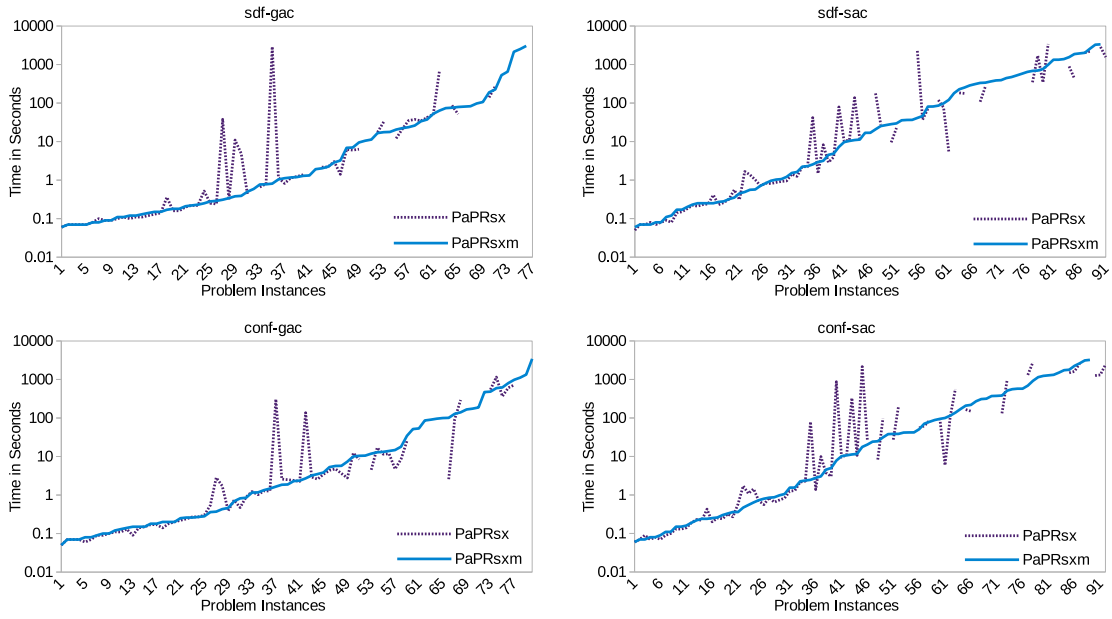


Figure 23: Enhancing PaPR using mutex-groups.

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.

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	

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 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 24 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-

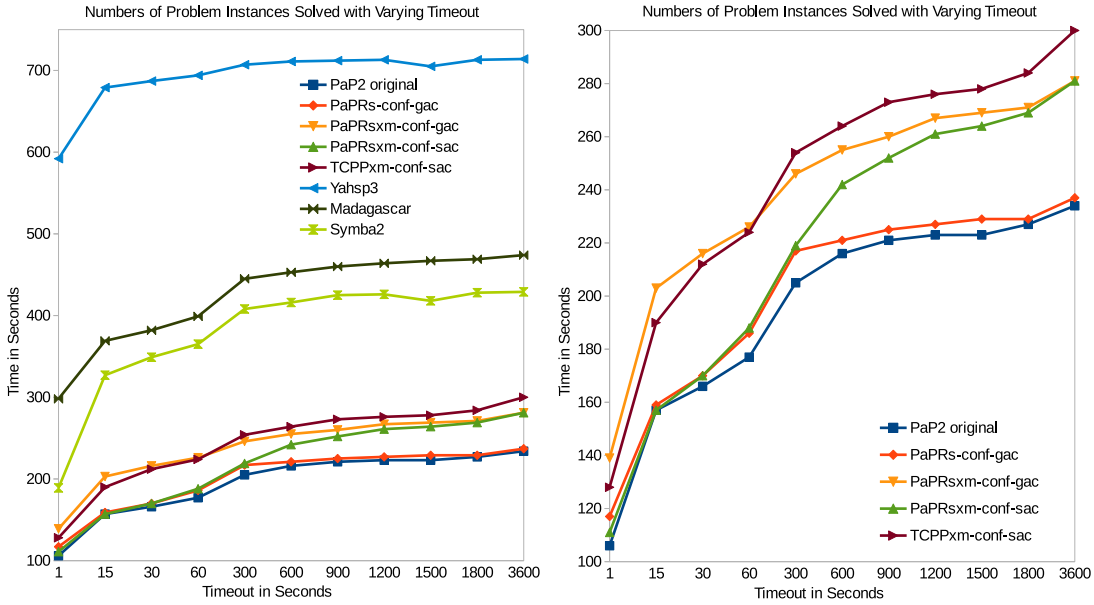


Figure 24: The total numbers of problem instances solved by different planners over different timeouts. Left: comparison with international planning competition winners. Right: comparison of only TCPP, PaP2, and PaPR.

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 domains that we have experimented with, TCPPxm-conf-sac solves the most numbers of problems in 12 domains while PaPRsxm-conf-gac solves in 6 domains and PaPRsxm-conf-sac solves 3 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 performs slightly worse than the latter with shorter timeouts. However, with long timeouts, 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 25, 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 plan-

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

Domains	#Ins	Mada	Yahsp	SymBA	PaP2	$PaPR^0$	$PaPR^1$	$PaPR^2$	TCPP*
airport	50	45	45	27	14	16	**22**	18	20
blocks	35	35	35	33	13	10	16	30	**32**
depot	22	15	22	7	3	4	6	8	**11**
driverlog	20	15	20	14	13	12	12	12	**13**
freecell	20	4	20	5	3	**3**	**3**	**3**	2
grid	5	2	5	2	1	1	1	1	**2**
gripper	20	3	20	20	2	2	2	2	2
logistics00	28	28	28	20	15	22	23	**25**	24
logistics98	35	28	34	5	18	12	**16**	13	9
miconic	150	49	150	111	34	25	27	24	**29**
mprime	35	31	35	24	20	18	24	18	**25**
mystery	30	18	17	15	11	11	**15**	13	14
pathways	30	9	0	5	2	5	5	5	5
pipes-notank	50	26	44	15	6	7	**11**	8	4
pipes-tank	50	10	40	16	6	5	9	6	6
psr-small	50	50	50	50	28	37	38	**44**	**44**
rovers	40	33	40	14	11	16	16	18	**20**
satellite	36	16	36	11	4	3	5	5	**6**
storage	30	14	23	15	10	9	9	9	**10**
tpp	30	28	0	8	8	8	9	9	**10**
zenotravel	20	15	20	12	12	11	**12**	10	**12**
Total	786	474	714	429	234	237	281	281	**300**

ning 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 25, we 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.

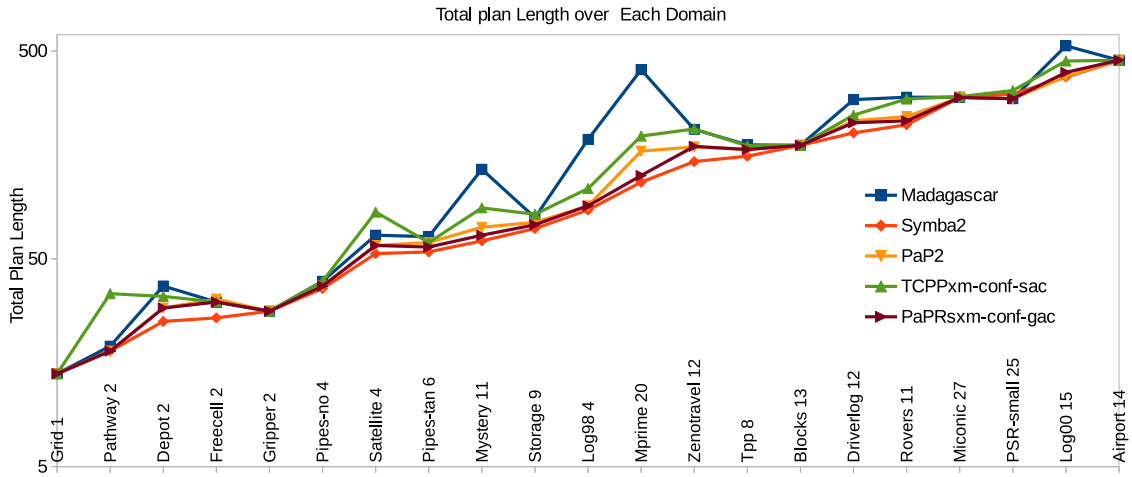


Figure 25: Total plan length for the plans produced by different planners over each domain

7.6 Encoding Statistics

As we have discussed before, in TCPPxm, we have transition constraints along with the parallelism 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.

To summarise Table 6, the numbers of CSP variables in TCPP are one time step more than that in PaPR (one can say in PaP2) because TCPP's variables are for states, while, PaPR's variables are for transitions. However, this is not an important factor because TCPP 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 TCPP. 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 TCPP and are proportional to the numbers of variables in the SAS$^+$ representations. The numbers of rows in the positive tables in both TCPP's and PaPR's models are about the same. The numbers of columns in TCPP's transition constraints are about twice the numbers of columns in PaPR's synchronisation constraints. 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 TCPP and PaPR models. These tables have 2 columns but the numbers of rows in PaPR are much more than that in TCPP. In some domains e.g. freecell,

Table 6: Different statistics about the TCPP and PaPR encodings. In the table, tc: transitions constraints, pc: parallelism constraints, mc: mutex constraints, sc: synchronisation constraints, asc: action succession constraints, neg: negative tables, Ts: #tables, Rs: #rows, Cs: #columns. 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	#	# of	Dom	tc		neg pc			neg mc		don't	Dom	sc		asc	neg mc		don't
Domains	Ins	DTG	Size	Rs	Cs	Ts	Rs	Cs	Ts	Rs	care%	Size	Rs	Cs	Rs	Ts	Rs	care%
airport	26	841	2	9	54	165	1	10	304	2	86	9	7	35	9	304	19	81
blocks	32	19	6	39	30	0	0	0	143	4	80	39	34	15	39	143	125	75
depot	12	36	6	142	54	0	0	0	180	7	85	144	139	27	144	180	1356	81
driverlog	14	13	7	44	8	0	0	0	8	1	58	44	38	4	44	8	62	52
freecell	3	30	4	129	23	2270	15	12	56	3	75	129	127	12	129	56	372	74
grid	2	20	18	342	17	0	0	0	50	1	57	342	325	8	342	50	146025	48
gripper	2	8	4	14	9	0	0	0	11	2	67	14	11	4	14	11	13	61
logistics00	25	13	8	21	8	0	0	0	0	0	91	22	16	4	22	0	0	90
logistics98	19	39	23	112	16	0	0	0	0	0	45	150	127	8	150	0	0	33
miconic	34	9	3	4	5	0	0	0	0	0	56	10	8	3	10	0	0	55
mprime	26	31	9	459	23	1172	433	6	0	0	50	459	451	12	459	0	0	44
mystery	16	31	9	239	17	922	45	7	0	0	12	239	230	9	239	0	0	9
pathways	5	78	2	5	7	217	3	6	0	0	80	5	4	4	5	0	0	80
pipe-notankage	10	77	2	29	63	8082	1	13	0	0	69	29	28	34	29	0	0	68
pipe-tankage	8	50	3	131	54	16849	1	19	19	3	56	131	129	32	131	19	3963	48
psr-small	45	24	2	16	13	1	1	10	0	0	89	16	15	7	16	0	0	84
rover	21	86	3	16	9	226	5	5	0	0	81	16	14	5	16	0	0	80
satellite	6	38	2	5	13	3	1	6	0	0	67	9	7	6	9	0	0	60
storage	10	17	3	30	19	98	3	9	6	1	74	30	28	10	30	6	278	72
tpp	10	32	2	8	8	30	5	7	0	0	81	8	7	4	8	0	0	79
zenotravel	15	12	7	186	6	0	0	0	0	0	81	186	179	3	186	0	0	55

pipes-notankage, and pipes-tankage, TCPP has very large numbers of negative tables for parallelism constraints. These really reduces TCPP's efficiency in these domains. 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.

7.7 Performance on Domains

Figures 26 and 27 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, or negative tables are an issue, 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

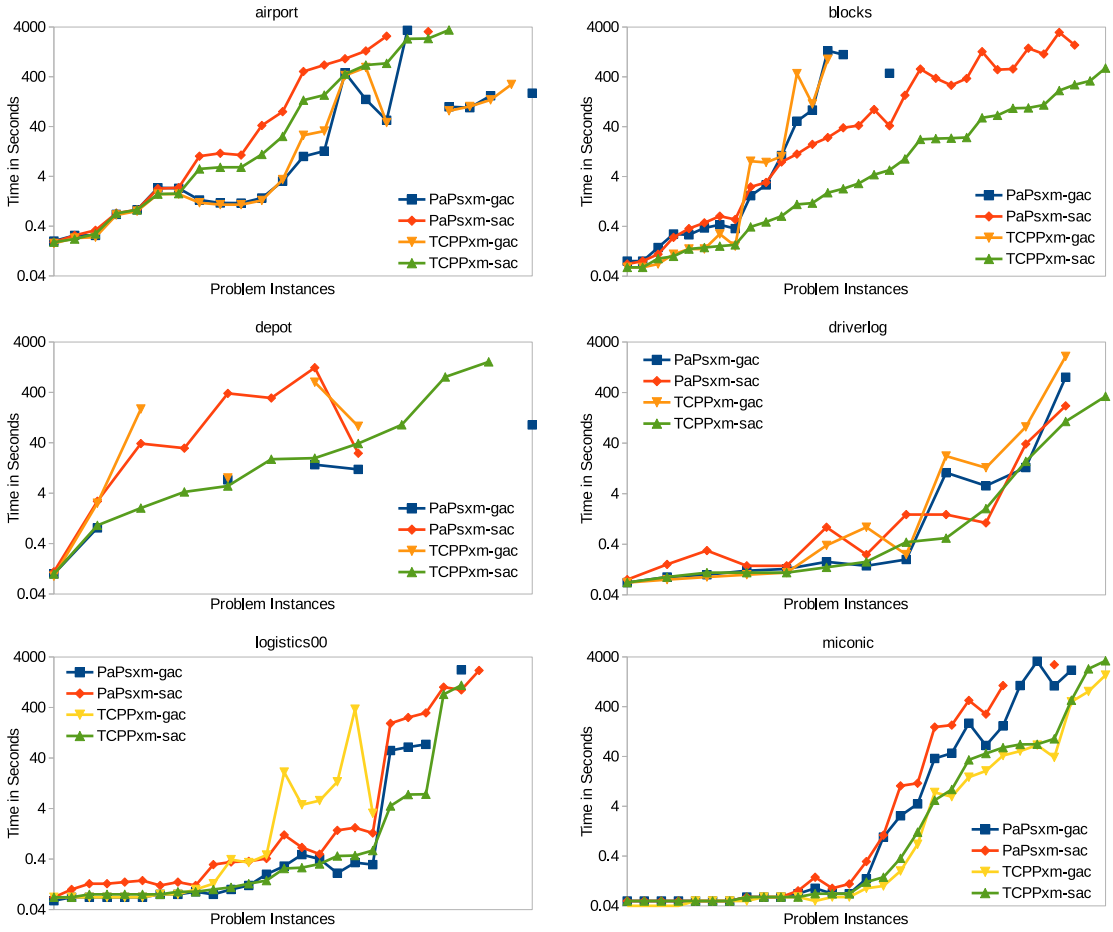


Figure 26: Time performance of TCPPxm, PaPRsxm on 6 Domains

selected for enforcing SAC (Bessiere, Cardon, Debruyne, & Lecoutre, 2011). As shown in Table 6, PaPR's encoding has larger domains than TCPP's encoding. We also see TCPP outperforms PaPR in most domains except in those that have large numbers of negative parallelism constraints, e.g. pipes-tankage, pipes-notankage, mprim and mystery.

7.8 Effect of Minion Configurations

Table 7 and Figure 28 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

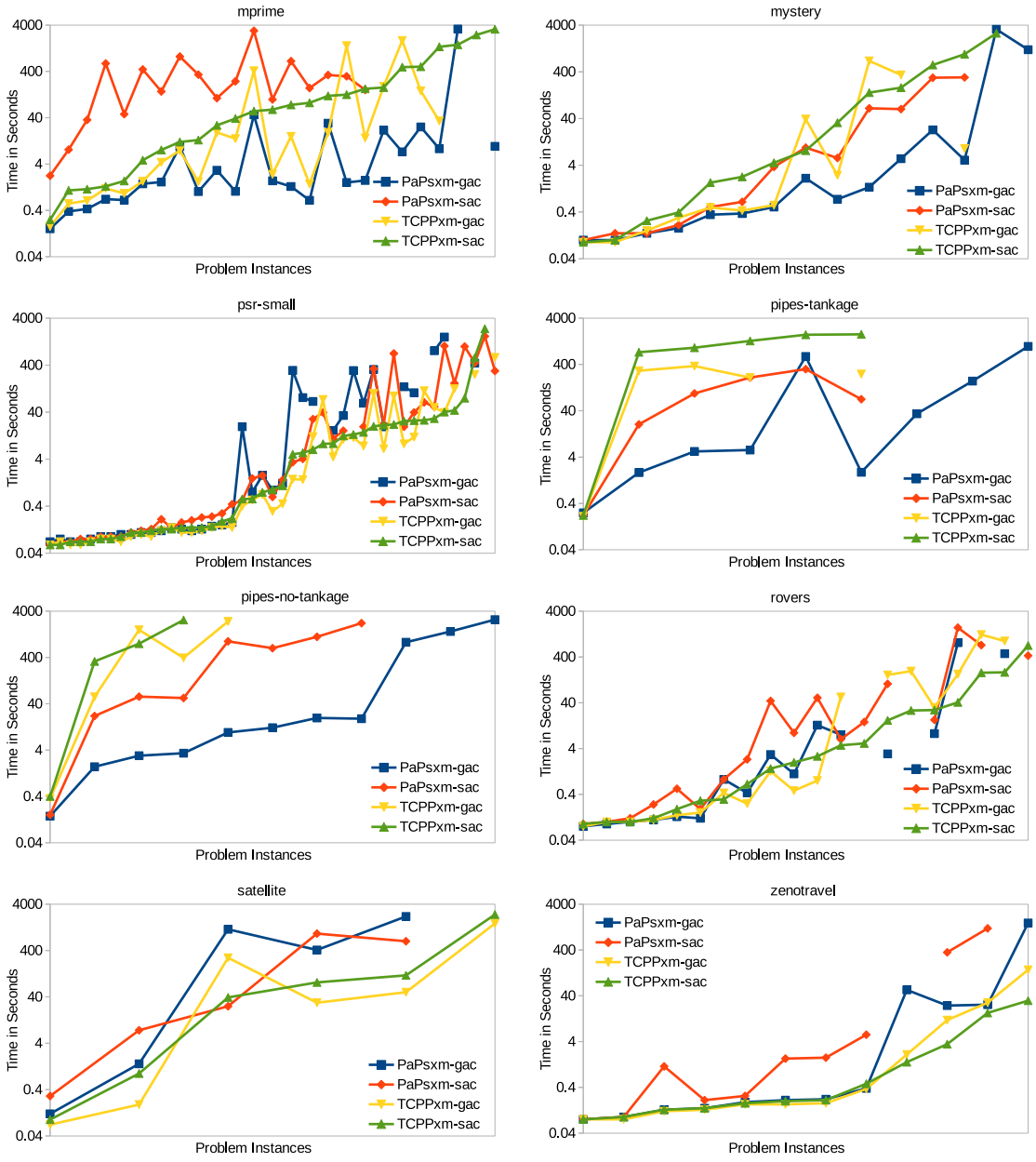


Figure 27: Time performance of TCPPxm, PaPRsxm on 8 Domains

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 29 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 visited by sac is often 0 at different makespan level. This indicates sac is mostly used in the

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	19	21	20
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	2	2	2	2
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	14	24	22	25
mystery	30	12	14	15	13	9	15	12	14
pathways	30	1	5	5	5	4	5	4	5
pipes-no-tankage	50	4	7	11	8	4	4	5	4
pipes-tankage	50	6	6	9	6	3	6	5	6
psr-small	50	19	43	38	44	21	41	43	44
rovers	40	10	19	16	18	8	16	18	20
satellite	36	4	3	5	5	2	5	6	6
storage	30	9	9	9	9	8	8	9	10
tpp	30	8	10	9	9	6	8	9	10
zenotravel	20	15	12	12	10	9	9	12	12
Total	786	228	278	281	281	190	278	259	300

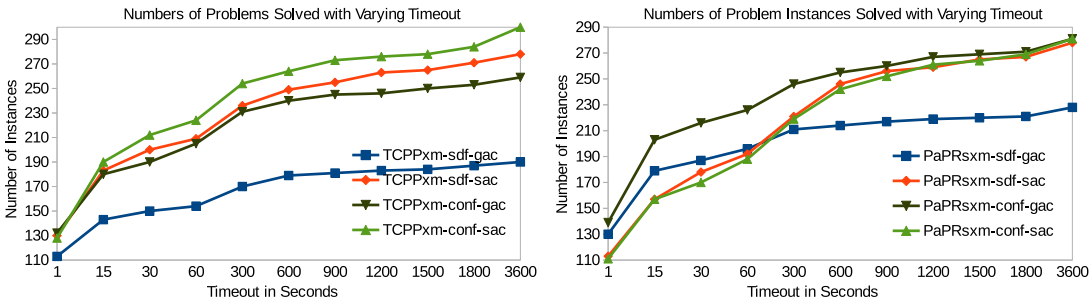


Figure 28: TCPPxm and PaPRsxm with different timeout limits.

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 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.

Makespan Steps →			20	21	22	23	24	25	26	time
TCPPxm-conf	airport8	gac	0	7	19	38	284	1347	1432	26.69
		sac	0	0	0	0	0	0	11	135.32
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	7	19	34	276	1476	1564	32.43
		sac	0	0	0	0	0	0	14	171.35
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 29: 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
259	293	300	281	288	281

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

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 PaPR-sxm-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.

Figure 30 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 with both sac-gac configuration than with sac-sac or gac-gac. 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, rovers, zenotravel, psr-small, and satellite. In the domains 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. This is the case in domains airport, mprime, and mystery. Moreover, because of the chains of variables

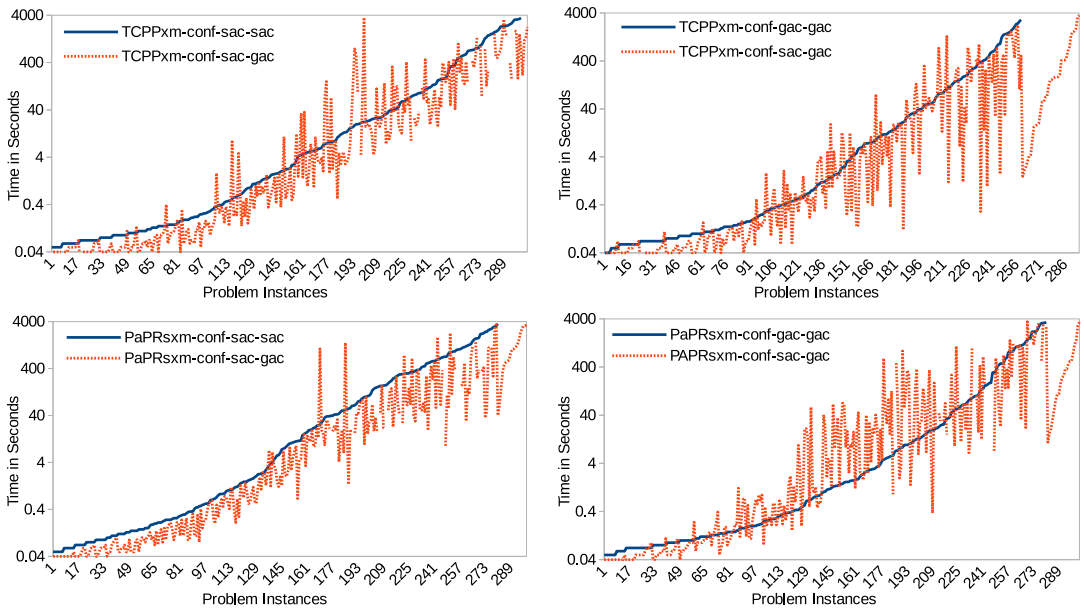


Figure 30: TCPPxm and PaPRsxm with sac and gac in preprocessing and search

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. However, in domains with large numbers of negative parallelism constraints, TCPP is worse than PaPR e.g. in domains such as pipes-no-tankage, pipes-tankage, mprime, mystery.

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 contraints. 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.

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. We are grateful for the support from NICTA. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

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.

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 programmming. *Artificial Intelligence*, *170*(3), 298335.