

Encoding Domain Transitions for Constraint-Based Planning

Nina Ghanbari Ghooshchi^{a,b}

Majid Namazi^a

M.A.Hakim Newton^a

Abdul Sattar^{a,b}

NINA.GHANABRI@GRIFFITHUNI.EDU.AU

M.NAMAZI@GRIFFITH.EDU.AU

MAHAKIM.NEWTON@GRIFFITH.EDU.AU

A.SATTAR@GRIFFITH.EDU.AU

^a*Institute for Integrated and Intelligent Systems (IIIS),
Griffith University, Brisbane, Australia*

^b*National ICT Australia (NICTA), Brisbane, Australia*

Abstract

We describe a constraint-based automated planner named Transition Constraints for Parallel Planning (TCPP). TCPP constructs its constraint model from the domain transition graphs (DTG) of a given planning problem. TCPP encodes *state transitions* in the 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 the parallel plan. We empirically compare TCPP with the current state-of-the-art constraint-based parallel planner PaP2. PaP2 encodes *action successions* in state transition automata as table constraints with cells containing sets of values and uses SICStus Prolog as its constraint solver. Our experiments on a number of standard planning benchmark domains demonstrate TCPP's efficiency over PaP2. We also model PaP2-style action succession constraints from DTGs and encode them using table constraints with don't cares. This new planning system named PaPr running on Minion significantly improves over PaP2.

1. Introduction

The use of inference-based pruning techniques and path heuristic-based choice guidance 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 algorithms. 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 to the solution being not of concern, path heuristics are in general not developed. In this paper, we take the approach of using CSP techniques to solve planning problems and describe our recently developed constraint-based planners.

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. While many key characteristics of a planning problem are overlooked in this way, these planners rather 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.

Two possible research directions discussed below deserve investigation in this respect. First, the cost of constraint propagation such as local consistency checking greatly increases with the increase of solution depth, but in planning, causal dependencies often create long and entangled chains. A reachability analysis in the form of path heuristics might be able to capture characteristics of a long causal chain and thus might improve variable and value choices during search. Second, 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. In this paper, we investigate the second 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 manually in (van Beek & Chen, 1999) and also in automated fashion in (Do & Kambhampati, 2001) and (Lopez & Bacchus, 2003). These models translate the so called planning graph structures (Blum & Furst, 1995) into CSPs with a view to generating parallel plans. These planners mostly use Boolean variables and constraints with logical formulas; which follows the propositional nature of the PDDL language (Ghallab et al., 1998). Constraint models for planning problems are later represented extensionally by table constraints and are developed by using 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 consistency, and **no-good** learning (Barták & Toropila, 2009a, 2009b). Few recent 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 one in (Judge & Long, 2011) applies goal and variable/value heuristics and uses meta-CSP variables. In (Barták, 2011a), the state transition automata that are similar to DTGs (Helmert, 2006) are used to build the constraint models.

In this paper, we describe a constraint-based automated planner named Transition Constraints for Parallel Planning (TCPP)¹. TCPP constructs its constraint model from the DTGs of a given planning problem. 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 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 general arc consistency propagation. 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 benchmark domains.

Constraint-based planner PaP2 encodes the *action successions* in automata as table constraints with cells containing *sets of values* and uses SICStus Prolog as its constraint solver. These automata 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. Also, there are loops in the automata for no-ops and edges from each value to a given value if the given value is in the effect but there is no precondition for it. Action successions in automata are pairs of actions such that in the automata, the latter action follows the former action satisfying their causal dependency. While in TCPP, we use table constraints with *don't cares* to encode state transitions in DTGs, in this paper, we further attempt to do the same to encode PaP2-style action successions but using the same DTGs. We then use Minion to solve this new constraint model. This is done to observe the interaction between the CSP model and the CSP solver. On the same benchmark domains as mentioned before, this new planning system named 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 describes the architecture of our constraint-based planner TCPP; Section 4 describes our encoding of planning problems into CSPs, particularly the encoding of transition constraints from DTGs using table constraints with don't cares; Section 5 presents our reconstructed planner PaPr that models PaP2-style action successions from DTGs; Section 6 presents our experimental results and analyses; and Finally, Section 7 presents the conclusion of this paper.

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.

2.1 Classical Planning

Given an initial state and a desired goal state, the *planning problem* is to find a sequence of actions 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. In classical planning, the world is considered finite, fully observable and deterministic. We, therefore, have a finite number of states in the world, the state of the world is fully known before and after performing the actions, and the actions are the only matters that change the world.

To illustrate a planning problem, we use a simplified driverlog domain (see Figure 1). 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

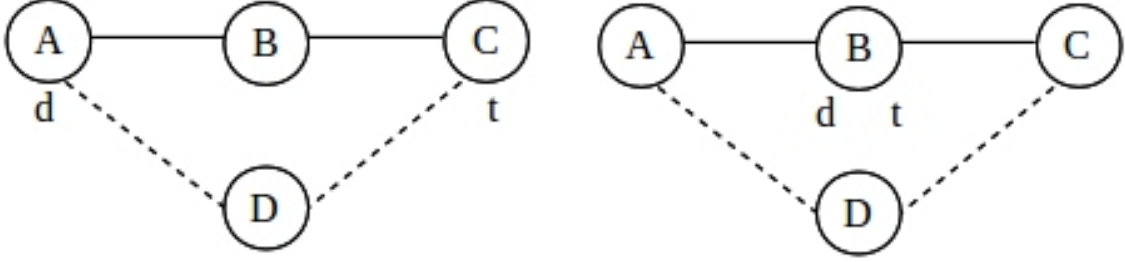


Figure 1: A driverlog problem instance: initial state (left) and goal state (right)

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 A and the truck t is at location C. To get to the goal state, the driver needs to catch the truck and drive it to the location B.

The most common language used in describing planning problems is PDDL (Ghallab et al., 1998). The PDDL representation of the problem in Figure 1 is shown in Figure 2 (left). An alternative approach which has become popular after being used in Fast-Downward planner (Helmert, 2006) is the SAS+ formalism (Bäckström & Nebel, 1995). In this formalism, planning problems are represented by multi-valued state variables; moreover, mutually exclusive predicates do not appear in the state description. Also, extracting structural information such as DTGs is also straightforward in this representation. The SAS+ representation of our example is shown in Figure 2 (right). As one can see, in this representation of the problem, we have two state variables $d\text{-loc}$ and $t\text{-loc}$ for locations of the driver and the truck respectively, and a state variable $t\text{-occ}$ to denote whether the truck is occupied by a driver. The domain for each of the variables is shown in the figure. Note that in SAS+ representation of a planning problem, preconditions and effects of actions are represented as *prevail* and *pre-post* conditions. Prevails are variable-value pairs that specify the values of variables that occur only in the preconditions of actions and not in the effects. Pre-post conditions are triples variable-value1-value2 that specify the values of variables before and after the execution of actions. The action can occur at time τ if variable has the value1 at time τ and after execution of action the value of variable changes to value2.

2.2 Domain Transition Graphs

From the SAS+ representation of a planning problem, we extract a domain transition graph (Helmert, 2006) for every state variable to show how these variables can change their values.

For every value that a state variable can take, there is one vertex in its DTG. The edges describe how the values of the state variables change. If there is an action having a precondition $v = d$ and an effect $v = d'$, we have an arc from the vertex with value d to the vertex with value d' . If an action has $v = d$ as its effect but no precondition relating to variable v , then in the original DTG, there is an edge from every vertex to the vertex representing the value d . In our version of DTGs, we simplify this by using a vertex with value '-' to denote a don't care value and draw an arc from this vertex to the vertex with value d . If any of the transitions is based on another variable's value, that is if a condition

PDDL Description

Predicates:

$\text{at}(o, l)$, $\text{empty}(t)$, $\text{road}(x, y)$,
 $\text{path}(x, y)$, $\text{driving}(d, t)$

Actions:

1. $\text{embark_truck}(d, t, l)$
preconds:
 $\text{at}(t, l)$, $\text{at}(d, l)$, $\text{empty}(t)$
effects:
 $\neg \text{at}(d, l)$, $\neg \text{empty}(t)$,
 $\text{driving}(d, t)$
2. $\text{debark_truck}(d, t, l)$
preconds:
 $\text{at}(t, l)$, $\text{driving}(d, t)$
effects:
 $\text{at}(d, l)$, $\text{empty}(t)$,
 $\neg \text{driving}(d, t)$
3. $\text{drive_truck}(d, t, x, y)$
preconds:
 $\text{at}(t, x)$, $\text{driving}(d, t)$,
 $\text{road}(x, y)$
effects:
 $\neg \text{at}(t, x)$, $\text{at}(t, y)$
4. $\text{driver_walk}(d, x, y)$
preconds:
 $\text{at}(d, x)$, $\text{path}(x, y)$
effects:
 $\neg \text{at}(d, x)$, $\text{at}(d, y)$

SAS+ Description

Variables:

$d\text{-loc}$ {A, B, C, D, t }
 $t\text{-loc}$ {A, B, C}
 $t\text{-occ}$ {false, true}

Operators:

1. $\text{embark_truck}(d, t, l)$
preconds:
 $t\text{-loc} = l$, $d\text{-loc} = l$, $t\text{-occ} = \text{false}$
effects:
 $d\text{-loc} = t$, $t\text{-occ} = \text{true}$
2. $\text{debark_truck}(d, t, l)$
preconds:
 $t\text{-loc} = l$, $d\text{-loc} = t$
effects:
 $d\text{-loc} = l$, $t\text{-occ} = \text{false}$
3. $\text{drive_truck}(d, t, x, y)$
preconds:
 $t\text{-loc} = x$, $d\text{-loc} = t$
effects:
 $t\text{-loc} = y$
4. $\text{driver_walk}(d, x, y)$
preconds:
 $d\text{-loc} = x$
effects:
 $d\text{-loc} = y$

l, x, y : locations, d : driver, t : truck
occ: occupied, loc: location

Figure 2: The driverlog problem instance in Figure 1 represented in PDDL (left) and SAS+ (right). PDDL actions are parameterised. SAS+ operators are actually grounded although in the figure, for convenience, they are shown using variables.

such as $v' = d''$ is also included in the precondition of the action, this condition is mentioned on the transition edge. Actions responsible for changes are also included on the edges.

DTGs for the example in Figure 1 are shown in Figure 3. In the DTG of variable $d\text{-loc}$, we have an edge from A to t labelled with c_2 which is for action $\text{embark_truck}(d, t, A)$. This action changes the value of $d\text{-loc}$ from A to t . Since variables $t\text{-loc}$ and $t\text{-occ}$ appear in the preconditions of this action, they all appear on the condition c_2 of the corresponding edge

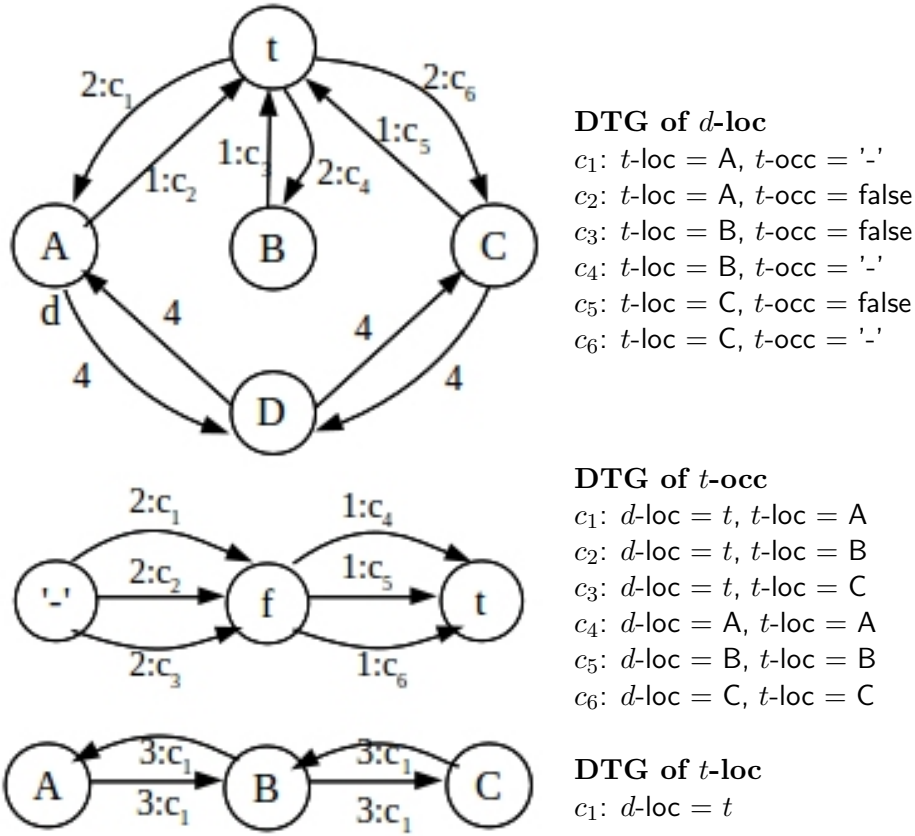


Figure 3: Domain transition graphs for the driverlog problem in Figure 1

meaning that to change the value of $d\text{-loc}$, these variables should have values A and false respectively. You can see an edge labelled c_1 from '-' to f in the DTG of variable $t\text{-occ}$. This change is caused by action `debark_truck(d , t , A)`. This action has variable $t\text{-occ}$ with value false in its effects but not in its preconditions. That is why we have an edge from '-' to A. The condition c_1 on the edge includes variable $d\text{-loc}$ and $t\text{-loc}$ that occur in the preconditions of action `debark_truck`. As you can see from the graph, we have multiple edges from vertex '-' to vertex f . This is not the case in the original DTG.

2.3 Constraint Satisfaction Problem

A *constraint satisfaction problem* is a tuple (X, D, C) where X is a set of n variables $\{X_1, X_2, \dots, X_n\}$, D is a set of finite domains $\{D_1, D_2, \dots, D_n\}$ for each of the variables respectively, and C is a set of m constraints $\{C_1, C_2, \dots, C_m\}$. Every constraint C_i is defined on a set of variables named $\text{Scope}(C_i)$ and specifies the combinations of allowed assignments for the variables in $\text{Scope}(C_i)$. A solution to a CSP is an assignment of a value to each variable in X that satisfies all the constraints. If there exists at least one solution for a CSP, we say the given CSP is satisfiable, otherwise it is unsatisfiable.

In our constraint satisfaction model for planning, we will use a special kind of constraints named *table constraints*. The scope of such a constraint is the set of variables occurring as the table columns and the rows list the set of values for these variables that are allowed by the constraint. Moreover, each row in the table is called a *support* for the constraint. As an example, consider a CSP with 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 $X_2 + 2X_3 \geq 3$. The table constraints for this CSP are shown in Figure 4. Look at the table constraint of $X_1 + X_2 + X_3 < 3$. Each row of the constraint $X_1 + X_2 + X_3 < 3$ is a set of assignments to variables X_1 , X_2 , and X_3 that satisfy the inequality. Similar statements hold for other constraints and their corresponding table constraints.

An example CSP:

$X_1 + X_2 + X_3 < 3$

$X_1 \leq X_2$

$X_2 + 2X_3 \geq 3$

X_1	X_2	X_3
0	0	0
0	0	1
0	0	2
0	1	0
0	1	1
0	2	0
1	0	0
1	0	1
1	1	0
2	0	0

X_1	X_2
0	0
0	1
0	2
1	1
1	2
2	2

X_2	X_3
0	2
1	1
1	2
2	1
2	2

Figure 4: Table constraints for a CSP example

We briefly describe generalised arc consistency for table constraints. We say a variable-value pair (X_i, a) is *generalised arc-consistent* if for every constraint having X_i in its columns, there exists a row that has value a for variable X_i . A table constraint is *generalised arc consistent* if for every variable X_i in its scope and for every value a in the corresponding domain, there exists a support for (X_i, a) , that is there is a row with value a for variable X_i . A CSP is *generalised arc-consistent* if all of its constraints are generalised arc-consistent (Bessiere, 2006). In the example in Figure 4, the first two constraints are generalised arc-consistent but the third constraint is not. Consider the table constraint for $X_1 + X_2 + X_3 < 3$. For every variable in this constraint and for every value in its domain, we have a row in the table. For example, rows 1–6 are the supports for pair $(X_1, 0)$, rows 7–9 are the supports for pair $(X_1, 1)$, and row 10 is the support for $(X_1, 2)$. Also, rows 1–3 and rows 7, 8, 10 are the supports for the pair $(X_2, 0)$, rows 4, 5, 9 are the supports for the pair $(X_2, 1)$ and row 6 is the support for the pair $(X_2, 2)$. Similarly, we have supports for all values of variable X_3 . Therefore, $X_1 + X_2 + X_3 < 3$ is generalised arc consistent. We can 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 supports for the pair $(X_3, 0)$ because we do not have any row having value 0 for variable X_3 . If we maintain arc-consistency, we should omit this value from the domain of variable X_3 . The

new 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 5.

	$X_1 + X_2 + X_3 < 3$			$X_1 \leq X_2$		$X_2 + 2X_3 \geq 3$	
	X_1	X_2	X_3	X_1	X_2	X_2	X_3
$X_1 + X_2 + X_3 < 3$	0	0	1	0	0	0	2
$X_1 \leq X_2$	0	0	2	0	1	1	1
$X_2 + 2X_3 \geq 3$	0	1	1	0	2	1	2
	1	0	1	1	1	2	1
				1	2	2	2
				2	2		

Figure 5: Table constraints after first stage of maintaining generalised arc-consistency

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 support for pairs $(X_1, 2)$ 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 6. Note that the domains of the variables are now $X_1 : \{0, 1\}$, $X_2 : \{0, 1\}$ and $X_3 : \{1, 2\}$ 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	X_1	X_2	X_2	X_3
$X_1 + X_2 + X_3 < 3$	0	0	1	0	0	0	2
$X_1 \leq X_2$	0	0	2	0	1	1	1
$X_2 + 2X_3 \geq 3$	0	1	1	1	1	1	2
	1	0	1				

Figure 6: Final table constraints after maintaining generalised arc-consistency

We describe singleton arc-consistency. A variable-value pair (X_i, a) is called *singleton arc-consistent* if after assigning the value a to variable X_i , the CSP can be made generalised arc-consistent. A CSP is *singleton arc-consistent* if for each variable X_i in its scope and for each value a in its domain, (X_i, a) is singleton arc-consistent (Bessiere, 2006). In our example in Figure 6, the variable-value pair $(X_1, 1)$ 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, as you can see with this assignment, we will not have any supports in the middle table in Figure 6 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 all the variable-value pairs are singleton arc-consistent.

Many algorithms have been developed to maintain generalised arc-consistency during search. These algorithms prune the domains of the variables and reduce the branching factor of the search. Since maintaining generalised arc-consistency and singleton-arc-consistency are time-consuming, we need to consider the trade off with their advantages. 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 plan length in advance. Since in planning the plan length 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; 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 time-consuming and equivalency checking is also waste of time. Recent CSP-based planners compute parallel plans.

Constraint satisfaction techniques are first applied to AI planning in (van Beek & Chen, 1999), but by designing the constraint models manually. 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. All the above mentioned planners use Boolean variables to encode the planning problem as CSP. Another constraint model that uses Boolean variables 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 these kinds of constraint, more inconsistencies are filtered out than could be done by using the original model 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 is 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, CSP variables are considered for state variables and actions at each time step and constraints are used for encoding the arcs 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.

3. Our Planner Architecture

Based on the DTGs extracted from the SAS+ representation of the planning problem, we 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 transition on the DTGs of state variables. The overall approach of our planner is shown in Figure 7.

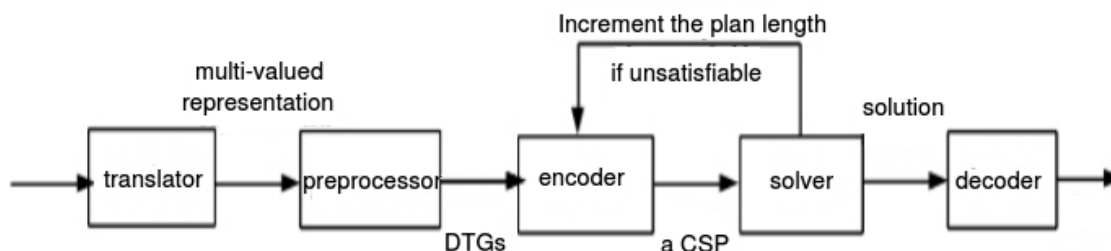


Figure 7: Architecture of our constraint-based planner

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

is used as a common language in the international planning competitions. We use the translator proposed by (Helmert, 2006) to translate the problem into SAS+ formalism with multi-valued variables and corresponding instantiated actions.

Preprocessing. After translation, the multi-valued representation of the planning problem is pre-processed to extract the DTGs for multi-valued state variables. This is done by using our own extractor program which is similar to the one in Fast-Downward planner (Helmert, 2006). The difference is that in the DTGs in Fast-Downward planner there exists only one edge between different values of a state variable if there is at least one action responsible for the change and if there are multiple actions with this condition, no multiple edge is considered for the transitions. However, in our DTGs, we have an edge for every action changing the value of the variable, that is we may have multiple edges between two vertices in the graph. Also, for each DTG, we have an additional vertex with value '-'.

Encoding. In this step, imposing a fixed bound on the makespan, 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 initial state to their values at the goal state. We select the maximum path length as the first makespan to start. Further details on our encoding are explained later in the paper.

Solving. As is 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. This solver also have 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 time 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 operators on them, but we need the operator 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.

4. Our Planning To CSP Encoding

We explain the encoding step of our planner with further details.

Procedure 1 PlanningToCSP(n) // makespan n

//variables: v^i, v^j , values: p, q , actions: A, B

```

Foreach  $(v^i, p)$  in initial state, Add constraint  $\text{eq}(v_0^i, p)$ 
Foreach  $(v^i, p)$  in goal state, Add constraint  $\text{eq}(v_n^i, p)$ 
For timestep  $\tau = 0$  to  $n - 1$  do
  Foreach  $v^i$ , Add transition constraint  $\text{tc}(v_\tau^i)$ 
  Foreach pair  $(A, B) : A$  is inconsistent with  $B$ 
    Add a negative parallelism constraint
  Foreach pair  $(v^i = p, v^j = q)$  in mutex groups
    Add a negative mutex table  $\text{mt}(v^i, p, v^j, q)$ 

```

The encoder takes as input the DTGs extracted from the multi-valued representation of the planning problem and also the makespan n and outputs a CSP, modelling the problem of finding the plan with this makespan. Procedure 1 presents the overall process. We assume that the planning problem has m state variables and so we have m DTGs. We consider a CSP variable v_τ^i for each state variable v^i and for any time $0 \leq \tau \leq n$. The domains of these CSP variables are the same as v^i 's domains. Therefore, we have $m(n + 1)$ CSP variables. We use four kinds of constraint in our model: initial state constraints, goal state constraints, transition constraints, and negative constraints. Our contribution is in the transition constraints that represent a new encoding of the state transitions in the DTGs of a planning problem. Below we describe each type of constraints.

4.1 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 n . Since the initial state is fully specified, we need m equality constraints that specify the values of variables $v_0^0, v_0^1, \dots, v_0^{m-1}$. In the goal state, we only have the values for some of the state variables and we need constraints for specifying their values at time n .

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

4.2.1 DRIVERLOG EXAMPLE

We now explain in detail how we encode the state transitions in the DTG's in Figure 3 to transition constraints shown in Figure 8. Suppose we want to encode the DTG of variable

$d\text{-loc}$ 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 operator. All the state variables in the preconditions and effects of that operator are included as CSP variables in the constraint table both for time τ and $\tau + 1$. For the DTG of $d\text{-loc}$, the state variables are $t\text{-loc}$ and $t\text{-occ}$. We also need the state variable of the DTG itself that is $d\text{-loc}$. These variable should be considered at time τ and $\tau + 1$. So we have 6 columns in the table: $d\text{-loc}_\tau$, $d\text{-loc}_{\tau+1}$, $t\text{-loc}_\tau$, $t\text{-loc}_{\tau+1}$, $t\text{-occ}_\tau$, and $t\text{-occ}_{\tau+1}$.

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 $d\text{-loc}$ and would become a row in the table specifying a valid assignment of values to the variables of the table. We have 10 edges in the DTG of $d\text{-loc}$ so we will have 10 rows in the table for the state transitions. The edge from A to t labelled with c_2 denotes the operator `embark_truck`. Due to this edge, the value of $d\text{-loc}$ can change from A to t at time τ if truck t is at location A and is not occupied. After the operator is executed the value of $d\text{-loc}$ changes to truck t and the truck is now occupied. The location of truck is not changed. We, therefore, have the first row in the table with values (A, t , A, A, false, true) for columns ($d\text{-loc}_\tau$, $d\text{-loc}_{\tau+1}$, $t\text{-loc}_\tau$, $t\text{-loc}_{\tau+1}$, $t\text{-occ}_\tau$, $t\text{-occ}_{\tau+1}$) respectively. The second row corresponds to the edge from t to A for operator `debark_truck`. Since $t\text{-occ}$ is not a precondition of the corresponding operator, notice that the value of $t\text{-occ}_\tau$ is '-'. 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 operator 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 $t\text{-loc}$ and $t\text{-occ}$ can take any values at time τ and $\tau + 1$; so their values are considered don't care in Row 7. Rows 8 to 10 are for the other `driver_walk` operators in the remaining edges of the graph. Lastly, the driver can stay at the same location between successive time steps. We therefore have rows 11-15 with $d\text{-loc}$ having the same values at time τ and $\tau + 1$ and the other variables having don't cares as their values.

4.2.2 TRANSITION ENCODING PROCEDURE

The detailed procedure that encodes a DTG to a table constraint is represented as Procedure 2. For a state variable v^i with DTG G^i , the table T_τ^i at time τ has columns v_τ^i , $v_{\tau+1}^i$, v_τ^j s, $v_{\tau+1}^j$ s, where v^j s are variables appearing on all edges of G^i . Each transition constraint is therefore a k -ary constraint where $k = 2(l + 1)$ and l is the number of variables appearing on the edges of G^i . Next, we extract the rows of the table for each edge in the G^i (see Step 2). Suppose (p, q) is an edge in G^i 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 '-' in the irrelevant columns. Because of edge (p, q) , clearly, v_τ^i is p and $v_{\tau+1}^i$ is q ; note p could be '-' in the G^i .

Procedure 2 $\text{tc}(v_\tau^i)$ // *transition-constraint*

G^i : DTG for v^i , T_τ^i : table for transitions of v_τ^i
 $V = \{v^i\} \cup \{v^x : v^x \text{ appears on an edge in } G^i\}$

1. Foreach $v^k \in V$, T_τ^i has two columns v_τ^k and $v_{\tau+1}^k$
2. Foreach edge (p, q) in G^i // p, q are values of v^i

TC for d-loc					
d-loc		t-loc		t-occ	
τ	$\tau + 1$	τ	$\tau + 1$	τ	$\tau + 1$
A	t	A	A	F	T
t	A	A	A	-	F
B	t	B	B	F	T
t	B	B	B	-	F
C	t	C	C	F	T
t	C	C	C	-	F
C	D	-	-	-	-
D	C	-	-	-	-
A	D	-	-	-	-
D	A	-	-	-	-
A	A	-	-	-	-
B	B	-	-	-	-
C	C	-	-	-	-
D	D	-	-	-	-
t	t	-	-	-	-

TC for t-occ					
t-occ		d-loc		t-loc	
τ	$\tau + 1$	τ	$\tau + 1$	τ	$\tau + 1$
F	T	A	t	A	A
F	T	B	t	B	B
F	T	C	t	C	C
-	F	t	A	A	A
-	F	t	B	B	B
-	F	t	C	C	C
F	F	-	-	-	-
T	T	-	-	-	-

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	-	-
B	B	-	-
C	C	-	-

Figure 8: Encoding domain transitions in DTGs in Figure 3 using table constraints. In the figure, Driver: d ; Truck: t ; Locations: A, B, C, D; Boolean: T, F; loc: location; occ: occupied; TC: transition constraint; '-': don't care; τ : time step.

tr: a new row in T_τ^i , each col has don't care '-'
 $\text{tr}[v_\tau^i] = p$, $\text{tr}[v_{\tau+1}^i] = q$ // note p could be '-'
 // Modify tr in the following ways
 Foreach condition $v^x = r$ on the edge (p, q)
 a_{pq} : the action associated with edge (p, q)
 If v^x only in precondition of a_{pq}
 $\text{tr}[v_\tau^i] = \text{tr}[v_{\tau+1}^i] = r$
 Elseif v^x both in precond and effect of a_{pq}
 Assume $v^x \leftarrow s$ in effect of a_{pq}
 $\text{tr}[v_\tau^i] = r$ and $\text{tr}[v_{\tau+1}^i] = s$
 Elseif v^x only in effect of a_{pq} i.e. $r = '-'$
 $\text{tr}[v_{\tau+1}^i] = s$ where $v^x \leftarrow s$ in effect of a_{pq}
 3. Foreach value p in domain of v^i
 tr: a new row in T_τ^i , each col has don't care '-'
 // Modify tr in the following ways
 $\text{tr}[v_\tau^i] = \text{tr}[v_{\tau+1}^i] = p$

We now consider every condition $v^x = r$ on the edge and three cases to determine other relevant column values. Assume a_{pq} is the action corresponding to the edge (p, q) .

1. **v^x only in a_{pq} 's precondition:** In this case, v^x is a prevailing condition for a_{pq} and remains the same at time τ and $\tau + 1$ ensuring that during the execution of a_{pq} , v^x 's value does not change by any other action.
2. **v^x in a_{pq} 's precondition and effect both:** Assuming $v^x \leftarrow s$ is in the effect, the values of v^x at time τ and $\tau + 1$ in this case are trivially r and s respectively.
3. **v^x only in a_{pq} 's effect i.e. r is '-':** In this case, we need to specify the value of v^x only at time $\tau + 1$, which we assume is s . At time τ , the value of v^x is don't care '-'

Lastly, to allow v^i 's value to be the same between successive time steps τ and $\tau + 1$, we need one row for each value in the v^i 's domain assuming don't care values for other variables in the row (see Step 3 in Procedure 2).

To summarise, we have mn transition constraints for a planning problem where m is the number of DTGs and n is the current makespan. The number of rows in the table constraint for a DTG is $e + d$ where e is the number of edges in the DTG and d is the domain size of the variable. 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 tables 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 real values for these variables in the rows of tables 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 tables corresponding to the other edges. To illustrate this, we use a simple example from the blocks world domain.

4.2.3 BLOCKS WORLD EXAMPLE

In the blocks world example in Figure 9, 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 `onx` meaning on another block x (A or B in this case), or at hand denoted by H. Thus, the domains of these five variables `A-top`, `B-top`, `hand`, `A-loc`, and `B-loc` are $\{C, NC\}$, $\{C, NC\}$, $\{E, NE\}$, $\{H, onB, onT\}$, and $\{H, onA, onT\}$ respectively.

We only depict the domain transition graphs for SAS+ variables `hand` and `A-loc` in Figure 10. In the DTG for `hand`, the edge labelled with condition c_1 is for action `put-down(A)`. We have the variable `A-loc` in the precondition and it needs to take the value H. We also have `A-top` in the condition on the edge. This state variable is not in the preconditions of the action but it occurs only in the effects; that is why we specified its value as don't care. The edge labelled with condition c_2 is for action `put-down(B)` and the conditions are similar. The edges labelled by c_3 and c_4 are for actions `stack(A,B)` and

PDDL representation

Predicates:

clear(x), ontable(x),
empty, holding(x),
on(x , y)

Actions:

- 1: pick-up(x)
preconds: empty, clear(x),
ontable(x), empty
effects: \neg clear(x), \neg ontable(x),
 \neg empty, holding(x)
- 2: put-down(x)
preconds: holding(x)
effects: \neg holding(x), clear(x),
empty, ontable(x)
- 3: stack(x , y)
preconds: holding(x), clear(y)
effects: \neg holding(x), \neg clear(y),
clear(x), on(x , y), empty
- 4: unstack(x , y)
preconds: empty, on(x , y),
clear(x)
effects: \neg clear(x), \neg on(x , y),
 \neg 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:

- 1: pick-up(x)
preconds: x -top = C, hand = E,
 x -loc = onT
Effects: x -top = NC, hand = NE,
 x -loc = H
- 2: put-down(x)
preconds: x -loc = H
effects: x -top = C,
hand = E, x -loc = onT
- 3: stack(x , y)
preconds: y -top = C, x -loc = H
effects: x -top = C, y -top = NC,
hand = E, x -loc = ony
- 4: unstack(x , y)
preconds: x -top = C, hand = E,
 x -loc = ony
effects: x -top = NC, y -top = C,
hand = NE, x -loc = H

Figure 9: The blocks world domain in PDDL and SAS+

stack(B,A). In order for action stack(A,B) to occur, B-top should be clear and block A should be in hand. These are the conditions appearing in the preconditions of the actions and so in the conditions on the edges. Variable A-top is appearing only in the effects of the action and so is considered don't care on the edge. The edges labelled by conditions c_5 and c_6 are for actions put-down(A) and put-down(B) and those labelled by c_7 and c_8 are for actions unstack(A,B) and unstack(B,A). In the DTG of the state variable A-loc, conditions c_1 , c_2 , c_3 , and c_4 are for actions pick-up(A), put-down(A), stack(A,B) and stack(B,A) respectively. The conditions are extracted from the preconditions and effects of the actions.

The transition constraints for these DTGs are in Figure 11. 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 c_1 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 putting down the block. The value for other variables are considered don't care. 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

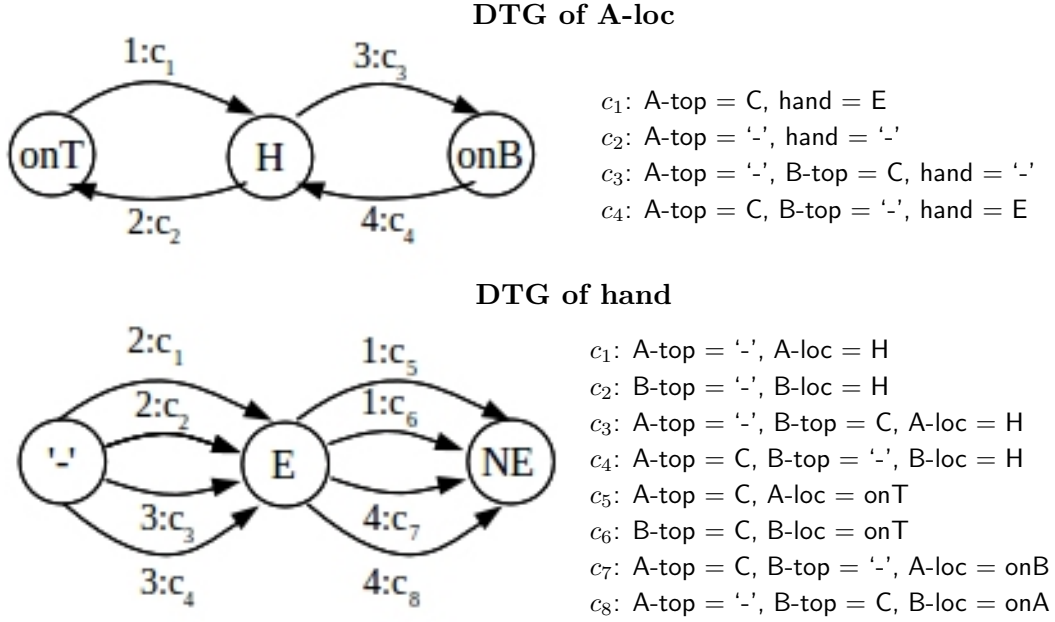


Figure 10: DTGs for two sample variables for an example from the blocks world domain

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	$\tau + 1$	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 11: Encoding state transitions in DTGs in Figure 10 using table constraints.

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.

4.3 Negative Constraints

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 operators taking place at the same time. To prevent the inconsistent operators from happening in parallel, we have negative constraints in our model. Also, we use negative constraints to

encode the mutex groups in the SAS+ representation. In our model, we have two kinds of negative constraints: mutex constraints and parallelism constraints, both of which are represented by negative table constraints. However, after extracting all the mutex and parallelism negative constraints, we also integrate the tables with the same set of variables into a compact negative table. This reduces the number of such negative tables.

4.3.1 MUTEX CONSTRAINTS

In SAS+ representation of some planning problems, there are mutex groups. These are groups of variable-value pairs that are mutually exclusive to each other. In our first example from the driverlog domain, we have a mutex group ($d\text{-loc} = t$, $t\text{-occ} = \text{false}$) which denotes that when the variable $d\text{-loc}$ takes the value t , the variable $t\text{-occ}$ can not take the value false and vice versa. In our second example from the blocks world domain, we have a mutex group ($A\text{-top} = C$, $A\text{-loc} = H$, $B\text{-loc} = \text{onA}$). These three variable-values pairs are mutually exclusive. To ensure that no pair from these group occurs at the same time, we use mutex constraints which are 2-ary negative table constraints. To summarise, for each mutex pair $(v^i = p, v^j = q)$ in mutex groups, we add a negative mutex table $\text{mt}(v^i, p, v^j, q)$ with columns v_τ^i, v_τ^j and a row with column values p, q . Figure 12 represents the negative mutex tables for the above mutex group from blocks world domain.

A-top _τ	A-loc _τ
C	H

A-top _τ	B-loc _τ
C	onA

A-loc _τ	B-loc _τ
H	onA

Figure 12: Negative mutex constraints for a sample mutex group from blocks world domain.

4.3.2 PARALLELISM CONSTRAINTS

As mentioned before, in our encoding, several operators can take place at the same time step and thus our planner produces parallel plans. However, for plan validity, we need to ensure that no mutually inconsistent operators occur at the same time. Two instantiated actions are inconsistent if there is a common state variable in their preconditions and effects, and the values are conflicting. More precisely instantiated actions X and Y are inconsistent if any of the following four conditions hold (Barták, 2011b).

1. Preconditions of X, Y share a variable with conflicting values.

$$\exists_{(v^i=p) \in \text{preconds}(X)} \exists_{(v^j=q) \in \text{preconds}(Y)} [v^i = v^j \wedge p \neq q]$$

2. A variable in X 's effect also appears in Y 's effect.

$$\exists_{(v^i \leftarrow p) \in \text{effects}(X)} \exists_{(v^j \leftarrow q) \in \text{effects}(Y)} [v^i = v^j]$$

3. A variable in X 's effect appears in Y 's precondition.

$$\exists_{(v^i \leftarrow p) \in \text{effects}(X)} \exists_{(v^j=q) \in \text{preconds}(Y)} [v^i = v^j]$$

4. A variable in Y's effect appears in X's precondition.

$$\exists_{(v^i \leftarrow p) \in \text{effects}(Y)} \exists_{(v^j = q) \in \text{preconds}(X)} [v^i = v^j]$$

Here, condition 1 means if there is a common variable with different values in the preconditions of two actions, these actions are inconsistent. Condition 2 means if the effects of actions X and Y share a variable no matter with the same value or not, these actions are inconsistent. Conditions 3 and 4 say that if there is a common variable in the preconditions of one action and in the effects of the other action, these actions are inconsistent. As discussed before, in SAS+ representation, preconditions are actually categorised into prevail and pre-posts. Based on those categorisation, we categorise state variables as follows:

1. in prevail (appearing only in preconditions and not in effects)
2. in pre-post with pre value '-' (appearing only in effects and not in preconditions)
3. in pre-post with pre value not '-' (appearing both in preconditions and effects)

Using the above categorisation, we give a new definition for action inconsistency. Actions X and Y are *inconsistent* if any of the six conditions below hold (for simplicity, we have omitted the converse cases). We then describe how our model handles these cases.

1. Prevails of A, B share a variable but the values are conflicting.
2. A variable appearing in A's prevails also appears only in B's effects.
3. A variable in A's prevails also appears in both B's preconditions and effects.
4. A variable appearing in only A's effects also appears in only B's effects.
5. A variable in only A's effects also appears in both B's preconditions and effects.
6. A variable appearing in both A's preconditions and effects also appears in both B's preconditions and effects.

Condition 1 is automatically prevented in our encoding because a variable can take one value at one time. Suppose we have a variable v^y at the prevail of action X with $v^y = p$ and at the prevail of action B with $v^y = q$. These actions are on different edges in the DTGs. These edges can not be traversed at the same time. The reason is that in the corresponding row for the edge for action A, variables v_τ^y and $v_{\tau+1}^y$ both should take value p and in the corresponding row for action B, variables v_τ^y and $v_{\tau+1}^y$ both should take value q . These variables can not take different values at the same time and therefore we do not need any constraint for this condition.

For condition 2, suppose we have a variable v^y in the prevails of action A with $v^y = p$ and only in the effects of action B with $v^y = q$. In the corresponding row for action A, v_τ^y and $v_{\tau+1}^y$ should take the same value p and in the corresponding row for action B variable v_τ^y and $v_{\tau+1}^y$ should take values '-' and q respectively. So if $p \neq q$, these actions are automatically prevented from happening at the same time. The problem is when $p = q$. In this case, in

our encoding, these actions can take place at the same time and this is not correct, so we should add a negative constraint.

For condition 3, suppose we have a variable v^y at the prevail of action A with $v^y = p$ and both at preconditions (with $v^y = q$) and effects (with $v^y = r$) of action B. In the corresponding row for action A, variables v_τ^y and $v_{\tau+1}^y$ should take the same value p and in the corresponding row for action B variables v_τ^y and $v_{\tau+1}^y$ should take values q and r respectively. So if $p \neq q$ or $p \neq r$ these actions are automatically prevented from happening at the same time. The problematic situation is when $p = q$ and $p = r$. In this case, we need to add a negative constraint to refrain it from happening.

For condition 4, suppose we have a variable v^y at only the effects of action A with $v^y = p$ and at only the effects of action B with $v^y = q$. In the corresponding row for action A, variables v_τ^y and $v_{\tau+1}^y$ should take values '-' and p respectively and in the corresponding row for action B variables v_τ^y and $v_{\tau+1}^y$ should take values '-' and q respectively. So if $p \neq q$ these actions are automatically prevented from happening at the same time. The problematic situation is when $p = q$ where we add a negative constraint.

For condition 5, suppose we have a variable v^y at only the effects of action A with $v^y = p$ and both at preconditions (with $v^y = q$) and effects with ($v^y = r$) of action B. In the corresponding row for action A variables v_τ^y and $v_{\tau+1}^y$ should take values '-' and p and in the corresponding row for action B, variables v_τ^y and $v_{\tau+1}^y$ should take values q and r . So if $p \neq r$, these actions are automatically prevented from happening at the same time. The problematic situation is when $p = r$ where we need a negative constraint.

Finally, for condition 6, suppose we have a variable v^y both at preconditions (with $v^y = q$) and effects with ($v^y = r$) of action A and both at preconditions (with $v^y = q'$) and effects (with $v^y = r'$) of action B. In the corresponding row for action A, variables v_τ^y and $v_{\tau+1}^y$ should take values q and r and in the corresponding row for action B variables v_τ^y and $v_{\tau+1}^y$ should take values q' and r' . So if $q \neq q'$ or $r \neq r'$ these actions are automatically prevented from happening at the same time. The problem is when $q = q'$ and $r = r'$. In this case, in our encoding, these actions can take place at the same time and we should add a negative constraint to stop it from happening.

Example. Let us see an example of parallelism negative constraints. The negative constraints for inconsistent actions A and B involve all the state variables in the prevail and pre-post conditions of the actions if the values are not '-'. The row of this negative table constrains the values of these variables in the corresponding actions. Considering actions with prevail condition $v^1 = r$ and pre-post conditions $v^3 : p \rightarrow q$ and $v^4 : '-' \rightarrow t$ and action B with prevail condition $v^2 = s$ and pre-post conditions $v^3 : p \rightarrow q$ and $v^5 : u \rightarrow v$. These actions are inconsistent and because of condition 6, we need a negative constraint. Variables of this condition are $v_\tau^1, v_\tau^3, v_{\tau+1}^3, v_{\tau+1}^4, v_\tau^2, v_\tau^5, v_{\tau+1}^5$. The row will be (r, p, q, t, s, u, v) .

5. PaP2 Reconstruction Using DTGs

A parallel planning problem could be represented by a planning graph (Blum & Furst, 1995) shown in Figure 13. The planning graph sketches the relationships between actions taking place at different time steps and their preconditions and effects. If we assume the makespan of a parallel plan is n , we will have $n + 1$ time steps and each time step consists of two layers: one for state variables and another for actions. In the figure, circles represent the

state variables and squares the actions. Actions are connected to their preconditions at the same time step and to their effects at the following time step. The last time step contains only the state variables. Planning graphs provide enormous useful information.

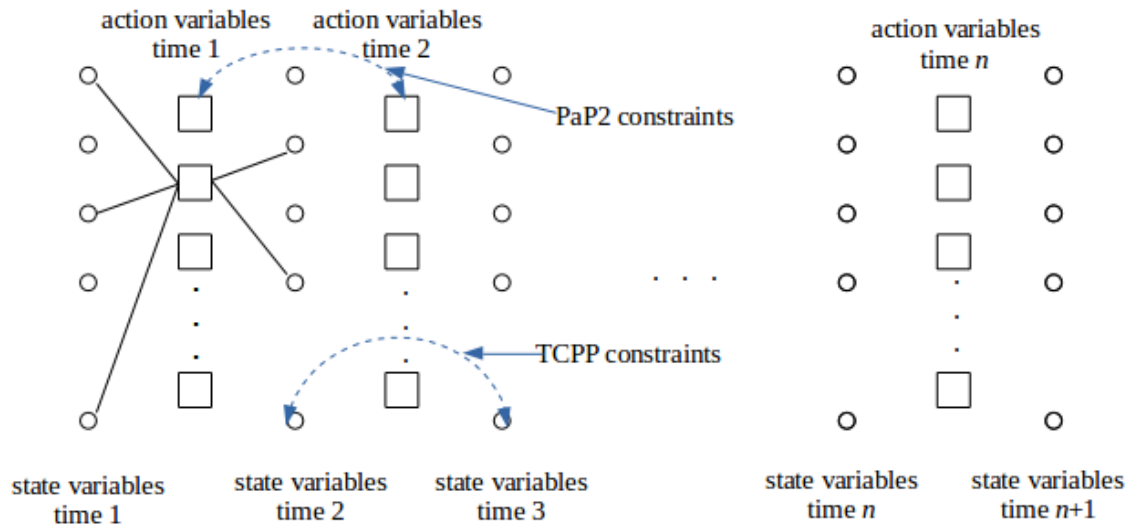


Figure 13: A typical planning graph, PaP2, and TCPP

To encode this parallel planning problem as a CSP, we need to encode the relationships between state variables and actions at consecutive time steps. 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 each action and its preconditions and effects are encoded by another type of constraint named *synchronisation constraints*. 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. We however do that by using the state transitions in the DTGs. Synchronisation constraints are implicit in our transition constraints. These are the major differences between PaP2's and TCPP's encoding.

In PaP2 (Barták, 2011b), the action succession constraints are actually encoded by using automata generated from a multi-valued representation of the problem. These automata 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 a to b it occurs on the edge from vertex a to vertex b . 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 automata, there are loops for every vertex in the graph denoting the no-ops for each value of the state variable.

In PaP2, the automata are encoded as a CSP using three kinds of constraints: action succession, synchronisation, and auxiliary constraints. For each state variable at time τ , a CSP variable is considered. The domain of this variable comprises all the actions occurring in the edges of automata for the state variable. These actions occur on the edges and also

includes the no-ops. The state variable describes the actions responsible for the changes in the state variable at time τ . 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.

In PaP2, for each state variable having its own automaton, there is a k -ary synchronisation constraint where k is the number of state variables in the planning problem. For each action from the automaton of this state variable, we need to specify the actions of other automata compatible with this action. For example, if this action changes the value of a state variable, the action responsible for this change should be considered for that state variable and if this action have a state variable only in its preconditions, the corresponding no-op should be considered for that state variable. Lastly, The auxiliary constraints specify that at each time step, at least one real action should take place.

Besides TCPP, in this paper, we also present a reconstructed PaP2 planner, which we named PaPr. This new planner PaPr is implemented to investigate the interaction between the constraint model and the constraint solver. In a constraint satisfaction problem, the constraint model, the search algorithm and the selection heuristics interact with each other (Beacham et al., 2001) and their choices should not be made independently. Although the planning model is always given as a PDDL description, there are many ways to convert a planning problem into constraint models and not all CSP solver are equally efficient in solving a given constraint model. Therefore, we extract PaP2-style action succession constraints from DTGs rather than from automata. The extracted constraints are encoded by using table constraints with don’t cares. This resulting new system PaPr runs on Minion solver. We cannot use SICStus Prolog since it does not support don’t cares.

In PaP2, the table constraints allow sets of values in their cells and the SICStus solver that has been used supports this kind of table constraints. Our CSP solver Minion does not support such table constraints, rather it supports don’t cares as cell values. We therefore have modelled the table constraints using don’t cares. This is a key difference between PaP2 and PaPr. The other difference between 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 sum 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. In 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.

6. 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 E5-2650 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.

6.1 Configuration Selection

Given the constraint models and the solver chosen to solve the constraint models, we next have to choose the selection heuristics available for the solver. For CSP solvers, variable and value selection heuristics are the key factors. The constraint propagation algorithm is also very important. CSP solver Minion supports two options for variable ordering: shortest domain first (sdf) and most conflict variable first (conflict). For propagation, Minion supports two options: generalised arc consistency (GAC) and singleton arc consistency (SAC). Unfortunately Minion does not provide any significant value ordering heuristic other than ascending or descending order. Mixing propagation method and variable selection heuristics, we obtain four different configuration for both of our planners TCPP and PaPr; For value selection, we just choose ascending order always.

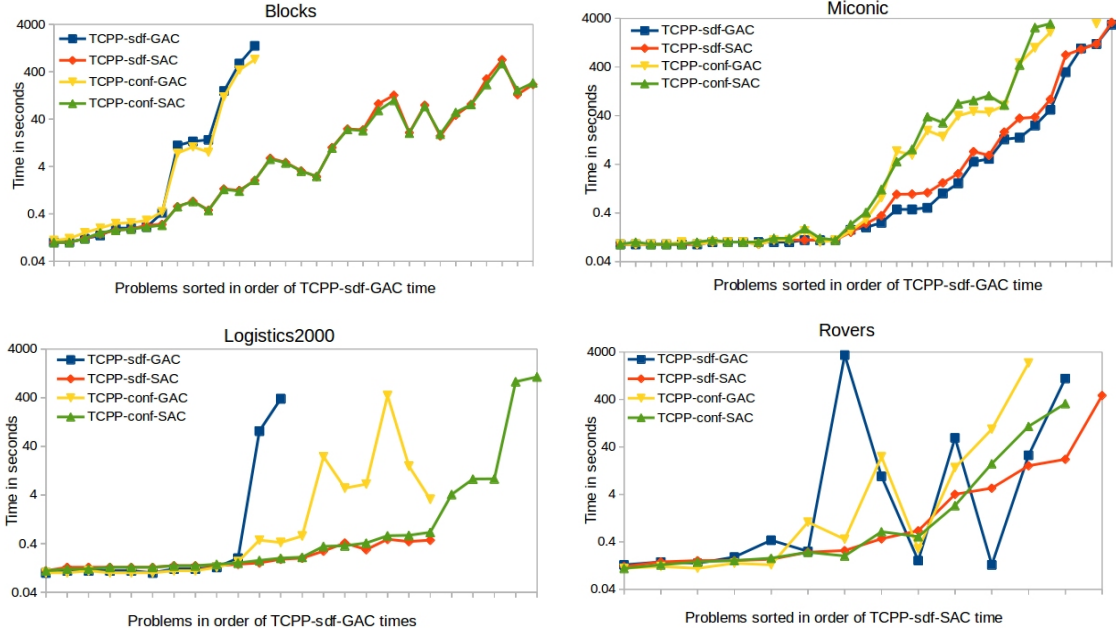


Figure 14: Effect of different configurations on TCPP’s performance on different domains

Figure 14 shows TCPP’s performance in blocks, miconic, rover, and logistics00 domains with the four configurations. In the blocks domain, propagation strategies appeared to have made significant differences in the performance. The SAC propagation outperforms the GAC propagation with great margin. The variable selection heuristics, sdf vs conflict, appeared to have no significant effect. In miconic domain, the opposite situation happens: propagation strategies have no effect while the variable selection heuristic makes significant performance difference with sdf strategy performing better than conflict strategy. In logistics00, all configurations have significantly different performance from each other with conflict-SAC outperforming other configurations. In rovers domain, no configuration clearly wins over another configuration. Overall, no clear selection of one of these TCPP configurations could be made from these domain-wise analysis. We have considered 21 domains collected from the international planning competition benchmarks.

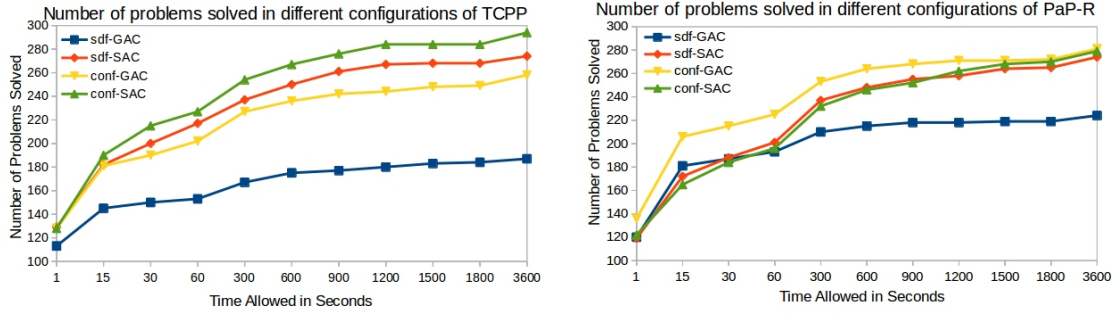


Figure 15: Performance of TCPP and PaPr configurations with different time cutoffs.

To select one configuration from the four mentioned above, we then use the total number of problem instances solved over all the domains by each of the configuration given different time cutoffs. Using this criterion, from Figure 15 left, we see the conflict-SAC configuration outperforms the other configurations. Henceforth, we select this conflict-SAC configuration to be our final TCPP planner and denote this with just the name TCPP. In a similar way, for PaPr, we observe from Figure 15 right that the conflict-GAC configuration outperforms the other configurations over the span of the time cutoffs. In contrast to TCPP configurations, PaPr configurations in a domain-wise analysis show similar performance characteristics. Henceforth, we just use the name PaPr to denote its conflict-GAC configuration.

6.2 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 problem. It is therefore worth observing the performance difference between SAT-based and CSP-based planners.

Figure 16 shows the total numbers of problem instances solved by different planners in 21 selected benchmark domains. Moreover, Table 1 shows the total number of instances solved in each domain by different planners over a 60-minute time-cutoff. From the charts and the table, we observe that YAHSP3 outperforms all other planners with a great margin, although a large part of the margin comes from its performance in the miconic domain. Nevertheless, one can easily see the huge differences in the performance of the constraint-based planners from that of the competition winner planners. Constraint-based planners need to make huge improvement in order for them to be competitive enough. Among the constraint based planners, TCPP performs slightly better than PaPr while both of them

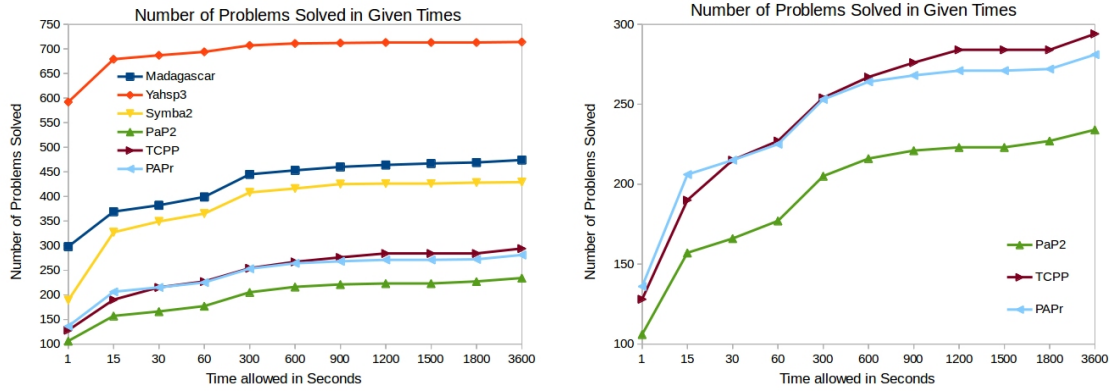


Figure 16: The total numbers of problem instances solved by different planners over different time cutoffs. Left: comparison with international planning competition winners. Right: comparison of only TCPP, PaP2, and PaPr.

are significantly better than PaP2. In 21 domains that we have experimented with, TCPP solves the most numbers of problems in 15 domains while PaPr and PaP2 solve in 10 and 7 domains respectively. Between TCPP and PaPr, the former performs slightly worse than the latter with shorter time cutoffs. However, with long cutoffs, TCPP appears to be consistently performing better than PaPr. It is interesting to observe these behaviours since the two planners view DTGs from two orthogonally different perspectives: TCPP views state transitions vs PaPr views action successions in the DTGs.

We show time performance of constraint-based planners TCPP, PaPr, and PaP2 on 16 of the 21 benchmarks domains where more (than five) instances are solved. Figure 17 shows for 8 such domains while Figure 18 shows for the other 8 domains. Splitting the domains in this way is just for convenience of fitting in them in the page’s space constraints.

Airport. PaPr performs the best and TCPP the second outperforming PaP2 in speed. Moreover, PaPr can solve many instances that are not solved by TCPP and PaP2.

Blocks. TCPP shows outstanding performance by solving problem instances with greater speed and by solving many more problem instances than PaP2 and PaPr.

Depots. TCPP solves problem instances faster and solves more problem instances than PaPr and PaP2. Moreover, PaPr solves more problem instances than PaP2.

Driverlog. All three planners perform very similarly. PaPr, however, could solve one problem instance less than TCPP and PaP2.

Logistics00. All three planners show similar speed in the problem instances that PaP2 could solve. However, TCPP and PaPr both could solve many more instances than PaP2.

Logistics98. PaP2 solves the most number of problem instances, but in most instances that PaPr could solve, it solves them with much greater speed. TCPP performs much worse.

Miconic. This is the best domain for PaP2, which solves more problem instances and with greater speed than TCPP and PaPr. TCPP is slightly better than PaPr.

Table 1: The total number of problem instances solved by different planners over 60-minute cutoff. The emboldened numbers are the winners of PaP2, TCPP, and PaPr only.

Problem	Count	Madagascar	Yahsp	SymBA	PaP2	TCPP	PaPr
Airport	50	45	45	27	14	12	28
Blocks	35	35	35	33	13	32	11
Depot	22	15	22	7	3	12	6
Driverlog	20	15	20	14	13	13	12
Freecell	20	4	20	5	3	2	4
Grid	5	2	5	2	1	2	1
Gripper	20	3	20	20	2	2	2
Logistics00	28	28	28	20	15	24	23
Logistics98	35	28	34	5	18	9	15
Miconic	150	49	150	111	34	29	27
Mprime	35	31	35	24	20	27	23
Mystery	30	18	17	15	11	15	15
Pathways	30	9	0	5	2	5	5
Pipes-notank	50	26	44	15	6	4	11
Pipes-tank	50	10	40	16	6	6	9
PSR-small	50	50	50	50	28	44	37
Rovers	40	33	40	14	11	20	17
Satellite	36	16	36	11	4	4	4
Storage	30	14	23	15	10	10	9
Tpp	30	28	0	8	8	10	10
Zenotravel	20	15	20	12	12	12	12
Total	786	474	714	429	234	294	281

Mprime. PaPr solves problem instances faster than TCPP, which is faster than PaP2, but overall TCPP solves more problem instances than PaP2 and PaPr.

Mystery. PaPr is faster than TCPP in most problem instances. Both PaPr and TCPP are faster than PaP2 and also solve more problem instances.

PSR small. TCPP is faster than PaPr and solves more problem instances. PaP2 is worst both in solution speed and in the number of problem instances solved.

Pipes Tankage. PaPr significantly improves over PaP2 both in solution speed and in the number of problem instances solved. TCPP is the worst in this domain.

Pipes No-Tankage. PaPr significantly improves over PaP2 both in solution speed and in the number of problem instances solved. TCPP is the worst in this domain.

Rovers. PaP2 performs the worst both in solution speed and in the number of problem instances solved. TCPP is slightly slower in problem instances taking less time, but in problem instances taking large solution times, TCPP is fast and solves more instances.

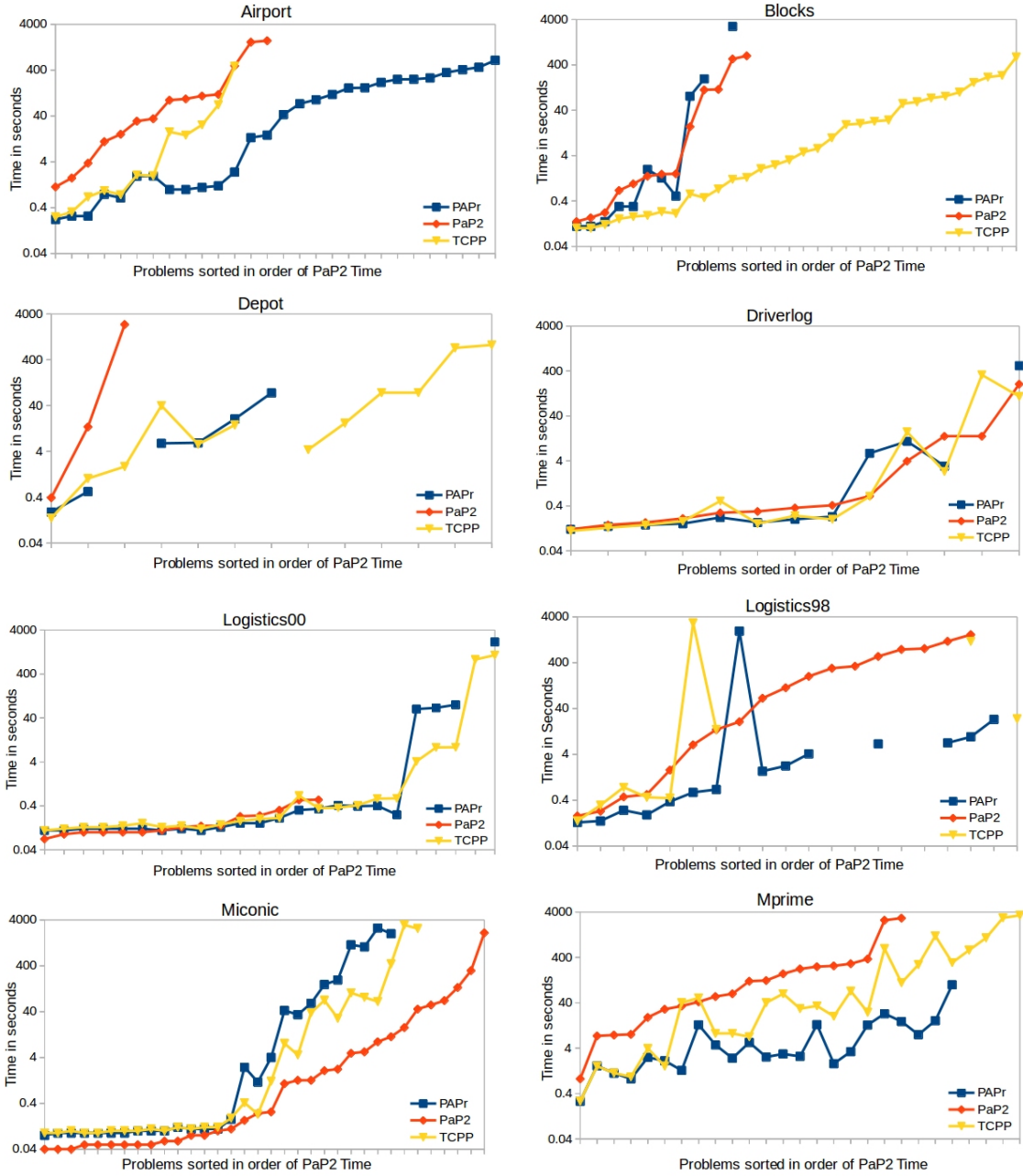


Figure 17: Time performance of TCPP, PaPr, and PaP2 on 8 different domains.

Storage. TCPP is worse than PaP2 in solution speed. PaPr is slower than TCPP and PaP2, and also solves comparatively fewer problem instances.

TPP. TCPP is slightly slower and PaPr is slightly faster than PaP2. PaP2 however solves fewer problem instances than both TCPP and PaPr.

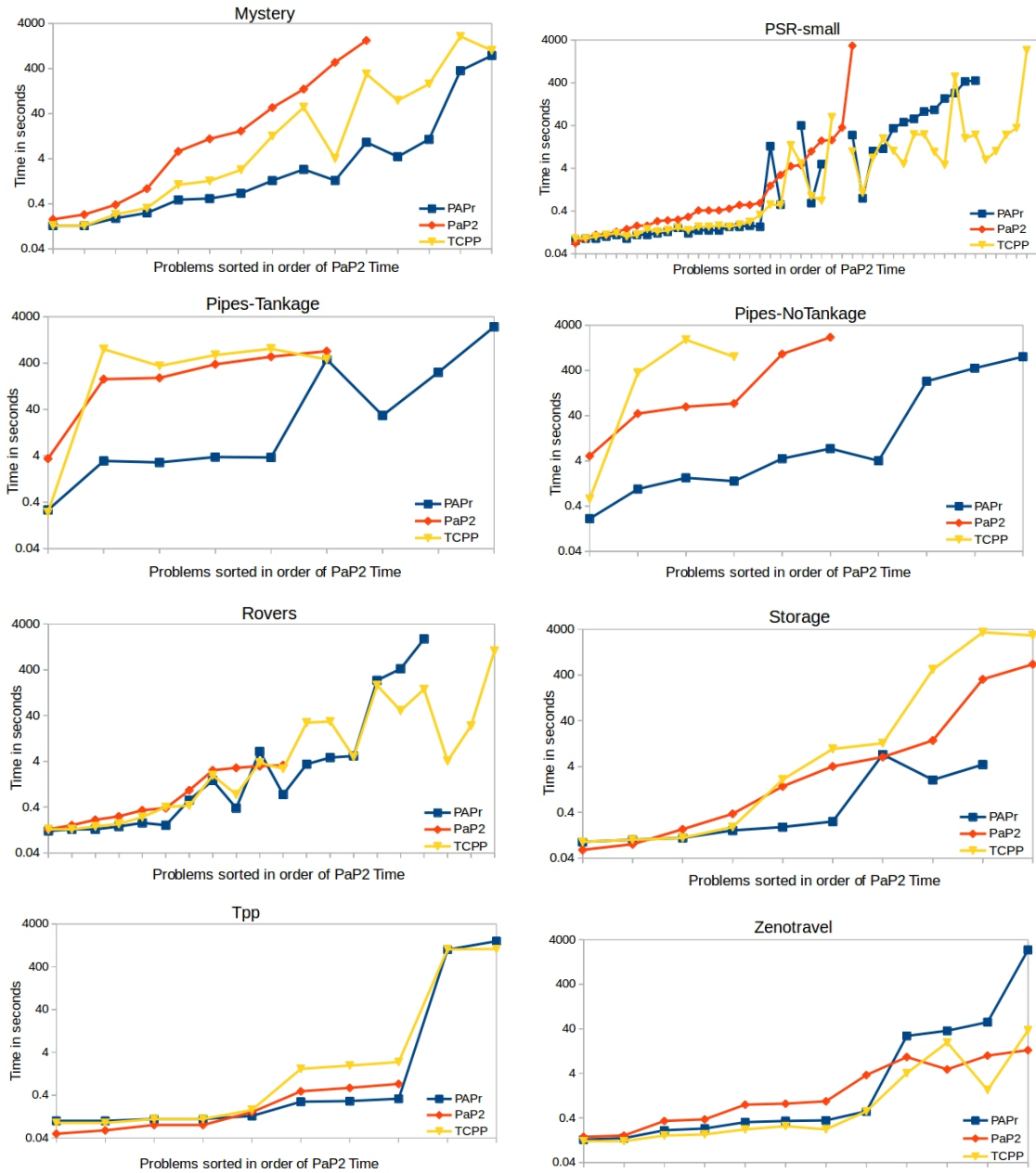


Figure 18: Time performance of TCPP, PaPr, and PaP2 on 8 different domains.

Zenotravel. TCPP is faster in most problems than PaP2. PaPr is faster than PaP2 in problems taking shorter times, but in those taking larger times, PaPr is significantly slower.

Overall Conclusion. Given 60 mins time cutoff in the 16 benchmark domains where time performance is considered, TCPP (279) solves more problem instances than PaPr (265) does; PaP2 (222) solves much fewer problem instances. In terms of solution speed in problem instances that planners compared both could solve, TCPP is clearly better

than PaPr in 4 domains such as Blocks, Miconic, PSR-Small, and Zenotravel; in contrast, PaPr is better than TCPP in 8 domains such as Airport, Logistics98, Mprime, Mystery, Pipes-Tankage, Pipes-NoTankage, Storage, and Tpp. TCPP and PaPr both of them are faster than PaP2 in six domains such as Airport, Depot, Mprime, Mystery, PSR-Small, and Rovers. PaP2 is clearly better than PaPr and TCPP only in Miconic.

6.3 Plan Comparison

In Figure 19, we compare lengths of the plans produced by each planner. Since YASHP in the agile track of the planning competition only emphasises on speed without looking at the plan length, we exclude YASHP from our comparison. SymBA*-2 optimises plans over a cost function, but in the absence of any function, the plan length is optimised. Madagascar is a SAT planner and is optimal in makespan. Note that TCPP, PaP2, and PaPr 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 19, 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.

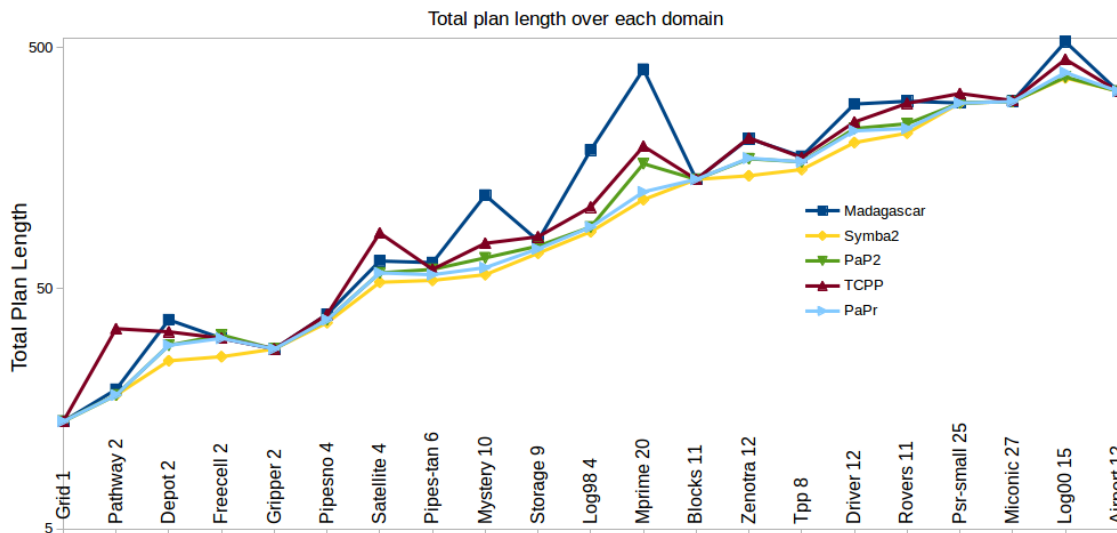


Figure 19: Total plan length for the plans produced by different planners over each domain

We observe that SymBA*-2 produces the shortest plan lengths in most domains. PaPr produces plans that are slightly longer than SymBA*-2's plans. Moreover, PaP2's plans are longer than PaPr's; TCPP's plans are longer than PaP2's; and Madagascar's plans are the longest. Since we use the ascending value ordering and in PaP2 and PaPr, 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 PaPr are less than that in TCPP. The value ordering in this case affects the quality of the plans produced.

6.4 Encoding Statistics

As we have discussed before, in TCPP, we have transition constraints along with the negative constraints while in PaPr (actually inherited from PaP2), we have action succession constraints along with synchronisation constraints. Both planners use table constraints with don't cares to encode their constraints. In Table 2, we report the maximum number DTGs and also the maximum numbers of rows and columns in the tables in each domain.

In our TCPP model, we have table constraints with large numbers of columns and also large numbers of don't cares. In most of the domains, the percentage of don't care values in tables is greater than 75% except in domains Zenotravel and Miconic in which the percentage is less than 50%. Although at the first glance, one might think that large numbers of columns could decrease the solution speed, but experiments do not show this. Because of the don't cares in the tables, actually checking is not needed for each cell in the row. Moreover, we merge negative constraints with the same variables into one table; which also improves the efficiency. In PaPr (and so PaP2) model, action succession constraints have only two columns but the numbers of rows are enormous; synchronisation constraints however have varying numbers of rows and columns in the tables.

Table 2: Different statistics about the TCPP and PaPr encodings

Benchmark Domains	PaPr				Max Number of DTGs Also, see footnote	TCPP					
	Synchronise Constraints		Action Succ Constraints			Transition Constraints			Negative Constraints		
	Max Row	Max Col	Max Row	Max Col		Max Col	Max Row	Max '·' %	Max Row	Max Col	Total Count
Airport	751	4625	11082	2	9145	731	6146	97	1	84	66568
Blocks	580	35	229145	2	35	580	70	90	0	0	0
Depot	1465	95	1247142	2	101	1465	190	95	0	0	0
Driverlog	1108	13	65044	2	45	1108	26	78	0	0	0
Freecell	21334	110	12188842	2	110	13289	188	94	242	14	44675
Grid	2188	21	45281	2	35	2188	42	81	0	0	0
Gripper	211	44	3991	2	45	211	88	90	0	0	0
Logistics00	57	8	167	2	22	57	16	79	0	0	0
Logistics98	3450	109	103823	2	141	3450	218	98	0	0	0
Miconic	3600	3	216000	2	61	60	6	45	0	0	0
Mprime	6076	48	1520020	2	83	6076	96	92	9108	7	12490
Mystery	5548	48	413956	2	81	5548	96	91	300	7	9074
Pathways	2449	100	168873	2	553	2449	151	93	35	9	5811
Pipes-notank	3047	883	3442665	2	1216	1139	714	98	1	15	239156
Pipes-tank	16308	236	73216399	2	264	1883	180	97	48	20	176187
PSR-small	9218	30	84962306	2	58	137	60	80	0	0	0
Rovers	1750	113	7652	2	613	2002	226	80	196	7	7056
Satellite	65025	54	16581375	2	1375	205	96	92	1	7	74818
Storage	5002	282	994102	2	290	2672	244	95	32	13	90012
Tpp	871	18	93055	2	332	871	36	76	1944	8	4592
Zenotravel	5463	6	192045	2	35	5463	12	20	0	0	0

Number of DTGs = Number of state variables in SAS+ = Number of transitions constraints of TCPP
= Number of synchronisation constraints of PaPr = Number of sequencing constraints of PaPr.

7. 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 the new system, being significantly different, is named as PaPr. PaPr also constructs its constraint model from DTGs but encoding action successions using table constraints allowing don't cares. 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.
- 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)*.

- Blum, A. L., & Furst, M. L. (1995). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1), 1636–1642.
- 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.
- 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., 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.

- 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.