Journal of Artificial Intelligence Research 1 (2019) 1-45 Submitted 7/2019; published 7/2019

Pattern-Based Approach to the Workflow Satisfiability Problem with User-Independent Constraints[*]

Daniel Karapetyan DANIEL.KARAPETYAN@GMAIL.COM
Institute for Analytics and Data Science, University of Essex, Colchester CO4 3SQ, UK

Andrew J. Parkes ANDREW.PARKES@NOTTINGHAM.AC.UK
School of Computer Science, University of Nottingham, Nottingham NG8 1BB, UK

Gregory Gutin G.GUTIN@RHUL.AC.UK
Department of Computer Science, Royal Holloway, University of London, Egham TW20 0EX, UK

Andrei Gagarin GAGARINA@CARDIFF.AC.UK
School of Mathematics, Cardiff University, Cardiff CF10 3AT, UK

Abstract

The fixed parameter tractable (FPT) approach is a powerful tool in tackling computationally hard problems. In this paper, we link FPT results to classic artificial intelligence (AI) search techniques to show how they complement each other. Specifically, we consider the workflow satisfiability problem (WSP) which asks whether there exists an assignment of authorised users to the steps in a workflow specification, subject to certain constraints on the assignment. It was shown by Cohen et al. (JAIR 2014) that WSP restricted to the class of user-independent (UI) constraints, covering many practical cases, admits FPT algorithms, i.e. can be solved in time exponential only in the number of steps k and polynomial in the number of users n. Since usually $k \ll n$ in WSP, such FPT algorithms are of great practical interest.

We present a new interpretation of the FPT nature of the WSP with UI constraints giving a decomposition of the problem into two levels. Exploiting this two-level split, we develop a new FPT algorithm that is by many orders of magnitude faster than the previous state-of-the-art WSP algorithm and also has only polynomial-space complexity. We also introduce new pseudo-Boolean (PB) and Constraint Satisfaction (CSP) formulations of the WSP with UI constraints which efficiently exploit this new decomposition of the problem and raise the novel issue of how to use general-purpose solvers to tackle FPT problems in a fashion that meets FPT efficiency expectations. In our computational study, the phase transition (PT) properties of the WSP are investigated for the first time, under a model for generation of random instances. We show how PT studies can be extended, in a novel fashion, to support empirical evaluation of scaling of FPT algorithms.

Keywords: fixed parameter tractability; workflow satisfiability problem; phase transition; pseudo-boolean formulation; hypergraph list colouring.

$*$. A preliminary version of some portions of this paper was published in the proceedings of FAW 2015 (Karapetyan, Gagarin, & Gutin, 2015).

1. Introduction

The combinatorial explosion, in the running times of algorithms for many important problems, has been an ongoing computational challenge within Artificial Intelligence (AI). AI has developed, and continues to develop, many powerful techniques to address this challenge. One of them, originating from theoretical computer science but recently also used in AI (see, e.g., (De Haan et al., 2015; Ordyniak & Szeider, 2013)), is the theory of fixed parameter tractable (FPT) algorithms, which is concerned with the parametrisation of hard problems that reveals that these problems are tractable under certain conditions. In this paper, we link together AI and FPT results and apply them to an important access control problem arising in many organisations.

Specifically, many organisations often have to solve 'workflow' problems in which multiple sets of tasks, or *steps*, need to be assigned to workers, or *users*, subject to constraints that are designed to ensure effective, safe and secure processing of the tasks. For example, security might require that some sets of tasks are performed by a small group of workers or maybe just one worker. Alternatively, some sets might need to be performed by at least two users, for example, so as to ensure independent processing or cross-checking of work, etc. (Basin et al., 2014; Crampton et al., 2013; Roy et al., 2015; Wang & Li, 2010). Furthermore, different users have different capabilities and security permissions, and will generally not be authorised to process all of the steps. In the *Workflow Satisfiability Problem* (WSP), the aim is to assign authorised users to the steps in a workflow specification, subject to constraints arising from business rules and practices. (Note the term "workflow" originally arose from the flow of the steps between users, however, in this context, the time ordering is not relevant – the challenge is to make a feasible assignment for all the steps.) The WSP has important applications and has been extensively studied in the security research community (Basin et al., 2014; Bertino et al., 1999; Bertolissi et al., 2018; Crampton, 2005; Crampton et al., 2015, 2017a, 2017b; dos Santos et al., 2017; Wang & Li, 2010).

The WSP is NP-complete, and it has been difficult to solve, even for some moderately-sized instances (Cohen et al., 2016; Roy et al., 2015). Work in WSP has attempted to render solving of the WSP practical by finding a subclass of problems that admit *fixed parameter tractable* (FPT) algorithms; informally speaking, this means that there is a small parameter k such that the problem is exponential in k but polynomial in the size of the problem. In the case of the WSP, the parameter k is naturally the number of steps – in real-life instances this number is usually much smaller than the number n of users (Wang & Li, 2010).

It has been shown (Cohen et al., 2014) that the WSP is FPT if it includes only authorisations and *user-independent* (UI) constraints, i.e. constraints whose satisfaction does not depend on specific user identities. Then existing methods (Cohen et al., 2016) achieve a runtime that is polynomial in n, and exponential only in k. (Constraints described in the WSP literature are relatively simple, so it was assumed by Cohen et al. (2014, 2016) that it is possible to test whether a solution satisfies any single constraint in polynomial time. We will use the same assumption in this paper. We will also assume that an authorisation test – whether a user is authorised to a step – takes constant time, as we store authorisation lists in the form of bitmaps.)

The major contributions of this paper are:

1. A new understanding of the FPT nature of the WSP with UI constraints that decomposes the problem into two levels: 'upper' level corresponding to UI constraints, and 'lower' level corresponding to user assignment and authorisations.

2. An effective backtracking search method, 'Pattern Backtracking' (PBT), exploiting the two level decomposition and supported by heuristics and pruning. (The PBT implementation presented in this paper is a significant improvement over the algorithm introduced in the preliminary report (Karapetyan et al., 2015) and even more so over the FPT algorithm of (Cohen et al., 2016) which unlike PBT has an exponential space complexity.)
3. A declarative method that we use explicitly in a pseudo-Boolean (Boros & Hammer, 2002; Le Berre & Parrain, 2010) and implicitly in a CSP formulations, which also exploits the two level decomposition of the problem.
4. Experimental studies of the algorithm performances, focussing on average case complexity, and supported by a new methodology that carefully exploits work in AI on phase transition (PT) phenomena (Bollobas, 1985; Huberman & Hogg, 1987; Cheeseman et al., 1991; Mitchell et al., 1992) but in a fashion extended and adapted for the needs of an FPT study.

To fully exploit the two level decomposition of the problem, we use special structures called *patterns* as introduced by Cohen et al. (2014). Patterns capture the decisions concerned with UI constraints but generally do not fix user assignments. In particular, they specify which steps are to be performed by the same user and which steps are to be performed by different users.

The notion of patterns is a convenient tool for handling the decomposition of WSP into two levels: an upper level corresponding to UI constraints, where the decisions can be encoded with a pattern, and a lower level corresponding to user assignments (note that authorisations are intrinsically user dependent and hence cannot be handled with patterns). This is used in our two-level algorithm which we call Pattern Backtracking (PBT). Its upper level implements a tree search in the space of patterns (thus not fixing user assignments), and the lower level searches for a user assignment restricted by a pattern. The space of upper level solutions has size exponential in k (and not depending on n), and the lower level can be reduced to a bipartite matching problem, i.e. admits a polynomial-time algorithm. Thus, PBT has running time exponential in k only.

We will show that PBT is not only FPT but also has polynomial space usage. Note that, the complexity class FPT does not in itself directly restrict the space usage; the previous algorithms had a space usage that was exponential in k, and this restricted their application to (roughly) $k \leq 20$. Moreover, due to the two-level structure of PBT, together with careful design of pruning methods and branching heuristics, the resulting implementation is many orders of magnitude faster than previous methods, with much improved scaling behaviour. The reachable values of k, i.e. the number of steps in the workflow, jump from about $k \leq 20$ to about $k \leq 50$; the reachable number of users is also now extended to being in millions. This is a significant improvement for an NP-complete problem, and also can be expected to be sufficient for practical-sized WSP instances.

We also provide a new pseudo-Boolean (PB) formulation of the problem: 'Pattern Based PB' (PBPB). Existing PB or integer programming encodings of the WSP with UI constraints (Cohen et al., 2016; Wang & Li, 2010) are based on the binary decision variables x_{su}, indicating whether step s is assigned to user u (we refer to these encodings as 'UDPB' for "user-dependent PB" encodings). To exploit the user-independent nature of our problem, we also use a set of M-variables which are not directly linked to specific users:

$$M_{ij} = 1 \text{ iff steps } i \text{ and } j \text{ are assigned to the same user, 0 otherwise.} \tag{1}$$

The M-variables in the PBPB formulation are used to allow an 'off-the-shelf' PB solver to better exploit the two-level decomposition of the problem as branching on M-variables captures the UI property of the constraints and corresponds to the upper level of the search (in the space of patterns). Notice that such variables have also been used extensively in powerful semi-definite programming approaches to graph colouring (Lovász, 1979), and have been referred to as the 'colouring matrix', e.g. see (Dukanovic & Rendl, 2008). By using a generic PB solver, SAT4J (Le Berre & Parrain, 2010), we show that the performance of the new PBPB formulation is several orders of magnitude better than the previous 'UDPB' formulation. Moreover, PBPB can even compete with PBT in terms of scaling behaviour, in some circumstances, though not in terms of constant factors.

We also give a Constraint Satisfaction Problem (CSP) formulation and solve it with the state-of-the-art OR-Tools[1] (in the latest MiniZinc constraint solvers competition, OR-Tools was the clear winner (Stuckey, 2018)). Even though the formulation and solver do not explicitly use our knowledge of the FPT nature of the problem, the performance of the CSP-based solver is similar to that of PBPB. We discuss a possible reason for this similarity.

Regarding the computational complexity, FPT problems and algorithms have been mainly studied from the perspective of worst case analysis, well-known to be over-pessimistic in many cases. In terms of the study of the potential practical usages of proposed algorithms, it is vital to study the performance averaged over specific instances. Ideally, it would be useful to have a benchmark suite of real-world instances. However, in the case of WSP, there is not yet any such suite publicly available, and even if it were, then it would probably not be suitable for scaling studies due to the diverse nature of instances in such suites. Hence, as commonly accepted in computational studies of WSP (Bertolissi et al., 2018; Cohen et al., 2016; Wang & Li, 2010), we use a generator of artificial instances.

When studying average case performance, it is vital to have a systematic way to decide on the parameters used in the generation of test instances, and in a fashion that gives the best chance of obtaining a meaningful and reliable insight into the behaviour of algorithms. With this motivation, in the latter parts of the paper we study the WSP from the perspective of phase transition (PT) or threshold phenomena. It has long been known that complex systems can exhibit threshold phenomena, see e.g. (Bollobas, 1985; Huberman & Hogg, 1987). They are characterised by a sharp change, or PT, in the properties of problem instances when a parameter in the instance generation is changed. An important discovery, e.g. (Cheeseman et al., 1991; Mitchell et al., 1992; Selman & Kirkpatrick, 1996; Mézard et al., 2005), was that such thresholds are also associated with decision problems that are the most challenging for search algorithms: the PT region is the source of the hardest instances of the associated decision problem. In the context of NP-complete problems, this generally means the empirical average time complexity is exponential in n in the PT region, i.e. it has a form that matches the worst case expectations, though usually with a reduced coefficient in the exponent. Outside of the PT region, the average complexity can drop significantly, resulting in what is usually informally called an "easy-hard-easy" transition.

Testing instances without a study of the associated PT properties has the danger of accidentally picking an easy region and, as a result, obtaining overly optimistic results. Since the WSP is a decision problem, a fair and effective testing of the scaling of the algorithms is best done by focussing on the PT region. (We also argue later that real-world instances are likely to be close to the PT region.)

The empirical average case analysis of FPT algorithms is more challenging than a typical PT study, since FPT problems have not just one but several size parameters. We give a novel empirical-

1. `https://developers.google.com/optimization/` (accessed Jan. 2019)

average-case study that has been systematically organised for the case of FPT studies. We will show how the instance generator we use leads to hard problems (in the FPT sense), and also exhibits other behaviours expected from a PT.

1.1 Structure of the paper

Section 2 gives the needed background on the theory of FPT, and the WSP. Section 3 introduces the notion of a pattern, central for the algorithms we discuss. Section 4 describes the new PBT algorithm in detail and demonstrates that it is FPT. Section 5 provides the new PB formulation PBPB of the WSP, (associated with this, Appendix A, see (Karapetyan, Parkes, Gutin, & Gagarin, 2019) gives details of encoding of various UI constraints.), along with the CSP encoding. Section 6 provides a descriptive comparison of the workings of the PBT algorithm, the existing algorithm (Cohen et al., 2016) which we call here Pattern User-Iterative (PUI), and the PBPB encoding.

Section 7 introduces the instance generator that we use for computational experiments. Some more technical aspects related to the PT are given in Appendices B and C, see (Karapetyan et al., 2019). Section 8 provides the results of empirical comparisons of the algorithms and their scaling behaviours with reference to the PT, with some more results reported in Appendix D, see (Karapetyan et al., 2019). Finally, in Section 9 we discuss overall conclusions and potential for future work.

2. Background

To make the paper reasonably self-contained, this section provides background and discusses related work on parameterized algorithmics and the WSP.

2.1 Parameterized Algorithms and Complexity

A parameterized problem Π can be considered as a set of pairs (I, k) where I is the *problem instance* and k (usually a nonnegative integer) is the *parameter*. Π is called *fixed-parameter tractable (FPT)* if membership of (I, k) in Π can be decided by an algorithm of runtime $O(f(k)|I|^c)$, where $|I|$ is the size of I, $f(k)$ is a computable function of the parameter k only, and c is a constant independent from k and I. Such an algorithm is called an *FPT* algorithm. The parameterised algorithmics literature also uses the notation "O*" to denote a version of big-Oh that suppresses polynomial factors in both k and $|I|$ (in the same fashion that the $\tilde{O}$ notation suppresses logarithmic factors), and so we can write $f(k)|I|^{O(1)}$ as $O^*(f(k))$. There is a general expectation that a problem admitting an FPT algorithm is "easy" as high-order polynomials are not so likely to occur.

When the decision time is replaced by the much more powerful $|I|^{O(f(k))}$, we obtain the class XP, where each problem is polynomial-time solvable for any fixed value of k. There is a hierarchy of parameterized complexity classes between FPT and XP (for each integer $t \geq 1$, there is a class W[t]):

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq XP.$$

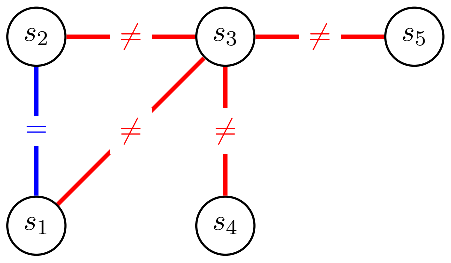
For the definition of classes W[t], see, e.g., (Downey & Fellows, 2013). Due to a number of results obtained on the topic, it is widely believed that FPT$\neq$W[1], i.e. no W[1]-hard problem admits an FPT algorithm (Downey & Fellows, 2013).

Note that, in general, the WSP parameterized by the number of steps k is W[1]-hard (Wang & Li, 2010), but the WSP with only user-independent constraints is FPT (Cohen et al., 2014). For more information on parameterized algorithms and complexity, see, e.g., (Downey & Fellows, 2013).

2.2 The WSP

In the WSP, we are given a set U of n *users*, a set S of k *steps*, a set $\mathcal{A} = \{A(u) \subseteq S : u \in U\}$ of *authorisation lists*, and a set C of *(workflow) constraints*. In general, a *constraint* $c \in C$ can be described as a pair $c = (T_c, \Theta_c)$, where $T_c \subseteq S$ is the *scope* of the constraint and Θ_c is a set of functions from T_c to U which specifies those assignments of steps in T_c to users in U that satisfy the constraint (authorisations are disregarded).

If $W = (S, U, \mathcal{A}, C)$ is the *workflow* and $T \subseteq S$ is a set of steps, then we say that a function $\pi : T \to U$ is a *plan*. A plan is called *authorised* if $\pi^{-1}(u) \subseteq A(u)$ for all $u \in U$ (each user is authorised to the steps they are assigned), and *eligible* if for all $c \in C$ such that $T_c \subseteq T$, $\pi|_{T_c} \in \Theta_c$ (every constraint with scope contained in T is satisfied). A plan that is both authorised and eligible is called a *valid plan*. If $T = S$, then the plan is *complete*. A workflow W is *satisfiable* if and only if there exists a complete valid plan.



user	s_1	s_2	s_3	s_4	s_5
u_1	−	−	+	−	−
u_2	−	+	−	−	−
u_3	+	+	−	+	+
u_4	+	−	−	−	−
u_5	−	−	+	+	+
u_6	+	−	+	−	−

Figure 1: An example of a WSP instance with $k = 5$ and $n = 6$. On the left, each step is represented with a node, and each constraint with an edge (in general, a constraint is represented with a hyper-edge but in this example all the constraints have scopes of size two). Red edges (marked with "$\neq$") represent not-equals constraints, and blue edges (marked with "$=$") represent equals constraints. The table on the right gives authorisations $A(u)$, where '+' means authorised and '−' means unauthorised.

Consider an example of a WSP instance with UI constraints in Figure 1. This particular instance includes just two types of constraints: "equals", defined for a scope of two steps, that requires those two steps to be assigned the same user; and "not-equals", also defined for a scope of two steps, that requires those two steps to be assigned different users. Then the following are three examples of complete plans for this instance:

$$\pi(s_1) = u_1,\ \pi(s_2) = u_1,\ \pi(s_3) = u_2,\ \pi(s_4) = u_1,\ \pi(s_5) = u_1 : \text{ eligible, non-authorised;} \tag{2}$$

$$\pi(s_1) = u_3,\ \pi(s_2) = u_2,\ \pi(s_3) = u_1,\ \pi(s_4) = u_3,\ \pi(s_5) = u_5 : \text{ ineligible, authorised;} \tag{3}$$

$$\pi(s_1) = u_3,\ \pi(s_2) = u_3,\ \pi(s_3) = u_6,\ \pi(s_4) = u_3,\ \pi(s_5) = u_5 : \text{ authorised, eligible, i.e. valid.} \tag{4}$$

The existence of a valid complete plan (4) implies that this instance is satisfiable.

Clearly, not every workflow is satisfiable, and hence it is important to be able to determine whether a workflow is satisfiable or not and, if it is satisfiable, to find a valid complete plan. Unfortunately, the WSP is NP-hard (Wang & Li, 2010). However, since the number k of steps is usually relatively small in practice (usually $k \ll n = |U|$ and we assume, in what follows, that $k < n$), Wang and Li (2010) introduced its parameterisation by k (we use terminology of the recent monograph (Downey & Fellows, 2013) on parameterised algorithms and complexity). Algorithms for this parameterised problem were also studied in (Cohen et al., 2014, 2016; Crampton et al., 2013). While in general the WSP is W[1]-hard (Wang & Li, 2010), the WSP restricted to some practically important families of constraints, but still allowing arbitrary authorisations, is FPT (Cohen et al., 2014; Crampton et al., 2013; Roy et al., 2015; Wang & Li, 2010).

Many business rules are not concerned with the identities of the users that perform a set of steps. Accordingly, we say a constraint $c = (T, \Theta)$ is user-independent (UI) if, whenever $\theta \in \Theta$ and $\phi : U \to U$ is a permutation, then $\phi \circ \theta \in \Theta$. In other words, given a complete plan π that satisfies c and any permutation $\phi : U \to U$, the plan $\pi' : S \to U$, where $\pi'(s) = \phi(\pi(s))$, also satisfies c. The class of UI constraints is general enough in many practical cases; for example, all the constraints defined in the ANSI RBAC standard (American National Standards Institute, 2004) are UI. Most of the constraints studied in (Cohen et al., 2016; Crampton et al., 2013; Wang & Li, 2010) and other papers are also UI. Classical examples of UI constraints are the requirements that two steps are performed by either two different users (*separation-of-duty*), or the same user (*binding-of-duty*). More complex constraints such as at least-r and at-most-r can state that at least/at most r users are required to complete some sensitive set of steps.

2.3 Connection with Graph Colouring and Constraint Satisfaction

The WSP (with arbitrary constraints) can be seen as a constraint satisfaction problem (CSP), where the unary constraints are called authorisations. (In this paper, while WSP constraints are UI, authorisations are arbitrary.) However, WSP is not a typical CSP: in many applications of CSP, the number of variables is much larger than the number of values (size of the domains), whereas in the WSP viewed as a CSP, usually the number of steps (number of variables) is much smaller than the number of users (size of the domains).

To the best of our knowledge, WSP is the first real-world application of such CSPs, and there are only a few studies that consider relevant cases. One of the most closely related ones is (Fellows et al., 2011) which discusses the "all different" constraints, requiring that all the variables in the scope are assigned different values. From the WSP point of view, it is a special kind of UI constraint. Among other results, it is shown in (Fellows et al., 2011) that CSP with "all different" constraints parametrised by the number of variables admits FPT algorithms. However, our study considers a much more general class of constraints. Moreover, it is concerned with practical considerations such as practically efficient algorithms and average case empirical analysis.

We note here that the WSP with UI constraints can be considered as an extension of the hypergraph list colouring problem, where steps correspond to the hypergraph vertices, users to the colours, and user-step authorisations define colours lists. Each constraint (T_c, Θ_c) then defines a hyperedge connecting vertices T_c, but the logic of colouring a hyperedge can be arbitrarily sophisticated as long as it is colour-symmetric. This colour symmetry is exactly the requirement implied by UI constraints, while the general WSP does not restrict the colouring logic at all.

3. Patterns

In this section, we discuss the concept of patterns as these capture equivalence classes under the permutation of users and form a vital part of the PBT algorithm presented in the next section.

3.1 Equivalence Classes and Patterns

We define an *equivalence relation* on the set of all plans. We say that two plans $\pi : T \to U$ and $\pi' : T' \to U$ are *equivalent*, denoted by $\pi \approx \pi'$, if and only if $T = T'$, and $\pi(s) = \pi(t)$ if and only if $\pi'(s) = \pi'(t)$ for every $s, t \in T$. (This is a special case of an equivalence relation defined in (Cohen et al., 2014).) To handle equivalence classes of plans, we introduce a notion of pattern. Patterns were first introduced by Cohen et al. (2014) but we follow the definition of patterns from (Crampton et al., 2015). Let a *pattern* $\mathcal{P}$ (on T) be a partition of T into non-empty sets called *blocks*. A pattern prescribes groups (blocks) of steps to be assigned to the same user, and also requires that steps from different blocks are assigned to different users. In other words, a pattern $\mathcal{P}$ requires that $\pi(s) = \pi(t)$ if and only if $s, t \in B$ for some $B \in \mathcal{P}$. Directly from the definition, if $B_1 \neq B_2 \in \mathcal{P}$, then $\pi(B_1) \neq \pi(B_2)$, where $\pi(B)$ is the user assigned to every step within block B.

Patterns also provide a convenient way to test equivalence of plans. Let $\mathcal{P}(\pi)$ be the pattern describing the equivalence class of the plan π; it can be computed as $\mathcal{P}(\pi) = \{\pi^{-1}(u) : u \in U,\ \pi^{-1}(u) \neq \emptyset\}$. Then $\pi \approx \pi'$ if and only if $\mathcal{P}(\pi) = \mathcal{P}(\pi')$ (Cohen et al., 2014), see Figure 2 for an example.

π_1

Step	User
s_1	u_3
s_2	u_3
s_3	u_1
s_4	u_5
s_5	u_5

$\approx$

π_2

Step	User
s_1	u_3
s_2	u_3
s_3	u_6
s_4	u_5
s_5	u_5

$\mathcal{P}(\pi_1) = \{\{s_1, s_2\},\ \{s_3\}, \{s_4, s_5\}\}$ $\qquad$ $\mathcal{P}(\pi_2) = \{\{s_1, s_2\},\ \{s_3\}, \{s_4, s_5\}\}$

Figure 2: An example of two equivalent plans π_1 and π_2, eligible for the instance defined in Figure 1. The patterns $\mathcal{P}(\pi_1)$ and $\mathcal{P}(\pi_2)$ are equal. Note that π_1 is authorised while π_2 is not.

In the language of graph list-colouring, a plan is a partial colouring (since a plan is not necessarily complete), a block is a set of vertices which are required to have the same colour, and a pattern is a set of blocks with a requirement that all the blocks are assigned different colours ignoring the concrete assignment of colours. The idea of patterns is somewhat related to the Zykov's method (Dutton & Brigham, 1981) (for the graph colouring problem) as they both are designed to effectively exploit the colour symmetry of the problems. Although in WSP this "colour symmetry" is broken by the authorisation lists, we use a similar approach to satisfy constraints C which are symmetric in WSP with UI constraints.

We say that a pattern $\mathcal{P}$ is *eligible* if there exists an eligible plan π such that $\mathcal{P}(\pi) = \mathcal{P}$. Similarly, we say that a pattern $\mathcal{P}$ is *authorised* if there exists an authorised plan π such that $\mathcal{P}(\pi) = \mathcal{P}$. An eligible authorised pattern is called *valid.* If a pattern is defined on $T = S$, then it is called *complete*.

3.2 Finding an authorised plan within an equivalence class

In this section we will describe an algorithm for finding an authorised plan π such that $\mathcal{P}(\pi) = \mathcal{P}$ for a given pattern $\mathcal{P}$ or detecting that such a plan does not exist.

Definition 3.1. *For a given pattern $\mathcal{P}$, an* assignment graph $G(\mathcal{P})$ *is a bipartite graph $(V_1 \cup V_2, E)$, where $V_1 = \mathcal{P}$ (i.e. each vertex in V_1 represents a block in the partition $\mathcal{P}$), $V_2 = U$ and $(B, u) \in E$ if and only if $B \in \mathcal{P}$, $u \in U$ and $B \subseteq A(u)$.*

We can now formulate a necessary and sufficient condition for authorisation of a pattern.

Proposition 3.1. *A pattern $\mathcal{P}$ is authorised if and only if $G(\mathcal{P})$ has a matching covering every vertex in V_1.*

Proof of Proposition 3.1 is straightforward.

Proposition 3.1 implies that, to determine whether an eligible pattern $\mathcal{P}$ is valid, it is enough to construct the assignment graph $G(\mathcal{P})$ and find a maximum size matching in $G(\mathcal{P})$. It also provides an algorithm for converting a matching M of size $|\mathcal{P}|$ in $G(\mathcal{P})$ into a valid plan π such that $\mathcal{P}(\pi) = \mathcal{P}$. An example of how Proposition 3.1 can be applied to obtain a plan from a pattern is shown in Figure 3.

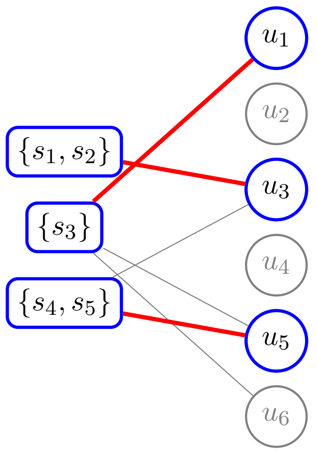
4. The New PBT Algorithm

In this section we first give the core PBT algorithm, and then some improvements, including branching heuristics that significantly improved the performance. Cohen et al. (2014) were the first to introduce a general theoretical approach based on patterns to solve WSP. In their next paper, Cohen et al. (2016) provided the first refined FPT algorithm and its implementation in the case of WSP with UI constraints; for a short discussion of the algorithm of (Cohen et al., 2016), see Section 6.1.

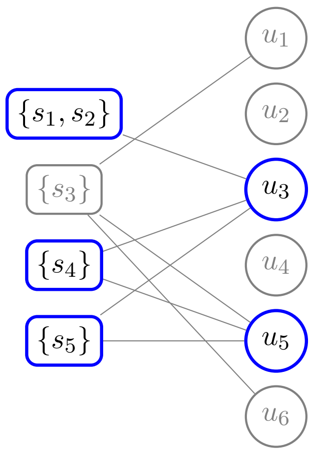
4.1 Pattern-Backtracking Algorithm (PBT)

We call our new method *Pattern-Backtracking* (PBT) as it uses the backtracking approach to systematically explore the search space of patterns. The key idea behind the PBT algorithm is that it is not necessary to search the space of plans; it is sufficient to search the space of patterns checking, for each eligible complete pattern, if it is authorised. (This idea was used in the preliminary version (Karapetyan et al., 2015) of this paper; a similar idea was also exploited by dos Santos et al. (2015, 2017) who however use different algorithmic techniques such as Petri nets and Datalog.)

The algorithm is FPT for parameter k, as the size of the space of patterns is a function of k only, and finding a valid plan within an equivalence class (or detecting that the equivalence class does not contain any valid plans) takes time polynomial in n. This separation helps to focus on the most important decisions first (as the search for eligible complete patterns is the hardest part of the problem).



(a) Satisfiable example. Every block is matched (red edges) to a user. The plan corresponding to this matching is shown on the right.



(b) Unsatisfiable example. Three blocks on the left (highlighted) can only match two users on the right (highlighted). Thus, at most two of these three blocks can be assigned distinct users.

Figure 3: Two examples of assignment graphs for eligible patterns $\mathcal{P}_1 = \{\{1,2\}, \{3\}, \{4,5\}\}$ (Figure (a)), and $\mathcal{P}_2 = \{\{1,2\}, \{3\}, \{4\}, \{5\}\}$ (Figure (b)). (The WSP instance is defined in Figure 1.) The matching in Figure (a) is of the maximum possible size $|\mathcal{P}_1| = 3$, and hence $\mathcal{P}_1$ is authorised. There exists no matching of size $|\mathcal{P}_2| = 4$, and hence $\mathcal{P}_2$ is unauthorised, i.e. there exists no valid plan within the corresponding equivalence class.

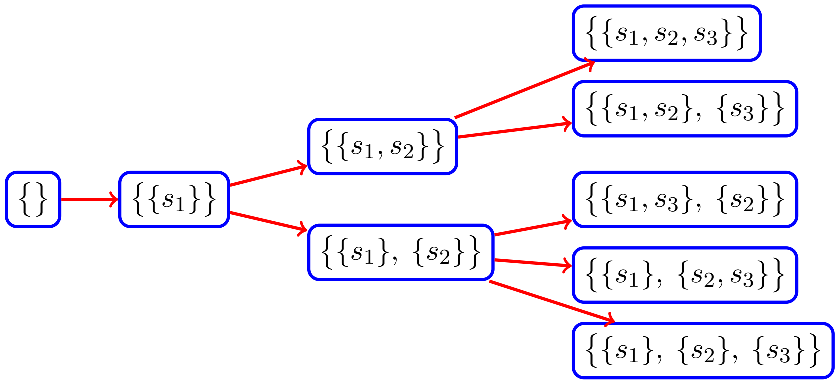

$\{\}$
$\{\{s_1\}\}$
$\{\{s_1, s_2\}\}$
$\{\{s_1\}, \{s_2\}\}$
$\{\{s_1, s_2, s_3\}\}$
$\{\{s_1, s_2\}, \{s_3\}\}$
$\{\{s_1, s_3\}, \{s_2\}\}$
$\{\{s_1\}, \{s_2, s_3\}\}$
$\{\{s_1\}, \{s_2\}, \{s_3\}\}$

Figure 4: Illustration of the backtracking mechanism within PBT.

The basic version of PBT is a DPLL-style backtracking algorithm traversing the space of patterns, see Figure 4. The search starts with an empty pattern $\mathcal{P}$, and then, at each forward iteration, the algorithm adds one step to $\mathcal{P}$. Let $S(\mathcal{P})$ denote the steps included in $\mathcal{P}$. The branching captures the simple notion that any step s not in $S(\mathcal{P})$ must either be placed in some block of $\mathcal{P}$, or else form a new block. These choices are disjoint and so it follows that the search tree is indeed a tree; the same pattern cannot be produced in different ways. Hence the tree can be searched using depth-first search (DFS). This is the key difference between PBT and the previous pattern-based WSP algorithm, PUI. PUI explores the space of patterns in a breadth-first way, in the worst case building and storing an exponential-in-k number of valid incomplete patterns. This leads to the space complexity of PUI also being exponential in k. For details see Section 6.1.

Pruning in the basic version of PBT is based on constraint violations only. Hence, any leaf node at level k in the basic version corresponds to an eligible complete pattern. Using Proposition 3.1, the algorithm verifies the authorisation of the pattern. Once an eligible authorised (i.e. valid) complete pattern is found, the algorithm terminates returning the corresponding valid complete plan.

In practice (the source code of our implementation of PBT is publicly available, see (Karapetyan, 2019)), we interleave the backtracking with authorisation tests. The real PBT uses an improved branch pruning that takes into account both constraints and authorisation. In particular, for every node of the search tree the algorithm verifies if the pattern is authorised, and if not, then that branch of the search tree is pruned.

The calling procedure for the PBT algorithm is shown in Algorithm 1, which in turn calls the recursive search function in Algorithm 2. The recursive function tries all possible extensions $\mathcal{P}'$ of the current pattern $\mathcal{P}$ with step $s \notin S(\mathcal{P})$. The step s is selected heuristically (line 4) using the empirically-tuned function $\rho(s, \mathcal{P})$, which indicates the importance of step s in narrowing down the search space. The implementation of $\rho(s, \mathcal{P})$ depends on the specific types of constraints involved in the instance and should reflect the intuition regarding the structure of the problem. See function (5) in Section 4.4 for a particular implementation of $\rho(s, \mathcal{P})$ for the types of constraints we used in our computational study.

Algorithm 1: Backtracking search initialisation (entry procedure of PBT)

```
   input : WSP instance $W = (S, U, \mathcal{A}, C)$
   output: Valid plan $\pi$ or UNSAT
1  Initialise $\mathcal{P}, G, M \leftarrow \emptyset$, $\pi \leftarrow$ Recursion($\mathcal{P}, G, M$);
2  return $\pi$ ($\pi$ may be UNSAT here);
```

Authorisation tests are implemented in lines 7 and 8 and discussed in more detail in Section 4.2. All the propagation and eligibility tests are implemented in line 5 and discussed in Section 4.3.

4.2 Authorisation-Based Pruning

Although it would be sufficient to check authorisations of complete patterns only, testing authorisations at each node allows us to prune branches if they contain no authorised plans. In doing so, it is not necessary to generate the assignment graph from scratch in every node:

- If $\mathcal{P}'$ is obtained from $\mathcal{P}$ by extending some block $B \in \mathcal{P}$, then G' can be obtained by removing all existing edges (B, u), $u \in U$, and adding edges (B', u) such that $B' \subseteq A(u)$, where B' is the extended block. One may note that the edge set (B', u), $u \in U$, is a subset of the edge set (B, u), $u \in U$; however, as we will explain below, we do not store the entire set of edges (B, u), $u \in U$, and hence cannot exploit this property.

- If $\mathcal{P}'$ is obtained from $\mathcal{P}$ by adding a new block $\{s\}$, then G' can be obtained by adding a new vertex $B = \{s\}$ to G and adding the edge (B, u), for each $u \in U$ such that $s \in A(u)$.

Similarly, it is possible to recover G from G'; hence, we can reuse the same data structure updating it in every node, with updates taking only $O(kn)$ time. It is also not necessary to compute the

Algorithm 2: Recursion($\mathcal{P}, G, M$) (recursive function for backtracking search)

```
input  : Pattern $\mathcal{P}$, authorisation graph $G = G(\mathcal{P})$ and a matching $M$ in $G$ of size $|\mathcal{P}|$
output: Valid plan or UNSAT if no valid plan exists in this branch of the search
1  if $S(\mathcal{P}) = S$ then
2      return plan $\pi$ defined by matching $M$;
3  else
4      Select an unassigned step $s \in S \setminus S(\mathcal{P})$ that maximises $\rho(s, \mathcal{P})$ (for details see
         Section 4.4);
5      Compute all the eligible patterns $X$ extending $\mathcal{P}$ with step $s$ (for details see Section 4.3);
6      foreach $\mathcal{P}' \in X$ do
7          Produce an assignment graph $G' = G(\mathcal{P}')$ (for details see Section 4.2);
8          if there exists a matching $M'$ of size $|\mathcal{P}'|$ in $G'$ then
9              $\pi \leftarrow$ Recursion($\mathcal{P}', G', M'$);
10             if $\pi \neq$ UNSAT then
11                 return $\pi$;
12 return UNSAT (for a particular branch of recursion; does not mean that the whole instance
       is unsat);
```

maximum matching M' in G' from scratch. We can obtain matching M' in G' from matching M in G in $O(kn)$ time. Indeed, if a new block is added, then a maximum matching of G' can have at most one edge more than M. If an existing block in G is extended, we can remove the old edge from M adjacent to the extended block, and then again a maximum matching can have at most one edge more than the updated M. By Berge's Augmenting Path Theorem, G' has a matching of size $|M|+1$ if and only if G' has an M-augmenting path (West, 2001). Thus, we only need to try to find an M-augmenting path in G'. The augmenting path algorithm requires $O(|V(G')| + |E(G')|) = O(kn)$ time.

In fact, we will never need more than $k \geq |\mathcal{P}|$ edges from a vertex $B \in \mathcal{P}$ of the assignment graph. Hence, when calculating the edge set for a $B \in \mathcal{P}$, we can terminate when reaching k edges. As a result, we only need $O(k^2)$ time to update/extend the matching M, but we still need $O(kn)$ time to update or extend the assignment graph.

Note that incremental maintenance of the matching M in every node of the search tree actually improves the worst-case time complexity compared to computation of M in the leaf nodes only, despite incremental maintenance being unable to take advantage of the Hopcroft-Karp method. Indeed, in the worst case (when the search tree is of its maximum size), each internal node of the tree has at least two children. Hence, the total number of nodes is at most twice the number of leaf nodes. Then the total time spent by PBT on authorisation validations is $O((kn + k^2)p)$, where p is the total number of complete patterns. Observe that validation of authorisations of complete patterns only, as in the basic version of PBT, would take $O((kn + k^{2.5})p)$ time.

When updating the assignment graph, we have to compute the edge set for a block $B \in \mathcal{P}$, and this takes $O(kn)$ time, i.e. relatively expensive due to the potentially large n. Since, during the search, we are likely to compute the edge set for many of the blocks B multiple times, we cache these edge sets for each block B. The number of blocks generated during the search (bounded by

2^k) can be prohibitive for caching all the edge sets, and, thus, we erase the cache every time it reaches 16 384 records (this constant was obtained by parameter tuning).

4.3 Eligibility-Based Pruning

While much of the implementation of PBT is generic enough to handle any UI constraints, some heuristics make use of our knowledge about the types of constraints present in our test instances. In particular, we assume that the instances include only not-equals, at-most and at-least constraints, as in (Basin et al., 2012; Cohen et al., 2016; Roy et al., 2015; Wang & Li, 2010). Note that all of the methods discussed in this paper make use of this information – which is a common practice when developing decision support systems. Hence, this makes our experiments more realistic and also helps to fairly compare all the approaches, as the old ones were already specialised.

Since all our constraints are only restricting the number of distinct users to be assigned to the scope, we implemented incremental maintenance of corresponding counters for each at-most and at-least constraint. The implementation of line 5 scans all the constraints with scopes that include step s and verifies which of the blocks in $\mathcal{P}$ can be extended with s without violating the constraint. Similarly, the algorithm considers creation of a new block $\{s\}$.

Note that pruning based on some constraint c does not always require that $T_c \subseteq S(\mathcal{P}')$. E.g., an at-most-3 constraint with scope $\{s_1, s_2, \dots, s_5\}$ can prune pattern $\{\{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}\}$. Our implementation of PBT produces only the extensions of the pattern that will not immediately break any constraint (disregarding authorisations).

4.4 Branching Heuristic

This section essentially describes line 4 of Algorithm 2. As standard in tree-based search, the selection of the branch variable, or step s in PBT, can make a big difference to the size of the search tree. The selection is performed by taking the step with the highest value of a score $\rho(s, \mathcal{P})$. The score is designed with the intention to encourage early pruning of the search tree.

In order to define components of the score function $\rho(s, \mathcal{P})$, we split the constraints C into the not-equals, $C_{\neq}$, the at-most $C_{\leq}$, and at-least $C_{\geq}$. Different constraints have different strengths in terms of the pruning of the search, and so are permitted to be counted with different weights. Specifically:

$$\rho(s, \mathcal{P}) = \alpha_{\neq}\rho_{\neq}(s) + \alpha_{\neq,\leq}\rho_{\neq,\leq}(s) + \alpha_{\geq,\leq}\rho_{\geq,\leq}(s) + \alpha^0_{\leq}\rho^0_{\leq}(s, \mathcal{P}) + \alpha^1_{\leq}\rho^1_{\leq}(s, \mathcal{P}) + \alpha^2_{\leq}\rho^2_{\leq}(s, \mathcal{P}), \tag{5}$$

where the α's are numeric weights, and

- $\rho_{\neq}(s) = |\{c : c \in C_{\neq}, s \in T_c\}|$ is the number of not-equals constraints c that cover step s. The more constraints cover a step, the more important it is in general.
- $\rho_{\neq,\leq}(s) = |\{c : c \in C_{\neq}, s \in T_c, \exists c' \in C_{\leq}, T_c \subseteq T_{c'}\}|$ is the count of the not-equals constraints c covering s and also covered by some at-most constraint c'. Not-equals and at-most constraints are in conflict, and their interaction is likely to further restrict the search.
- $\rho_{\geq,\leq}(s) = |\{c \in C_{\leq}, c' \in C_{\geq} : |T_c \cap T_{c'}| \geq 3, s \in T_c \cap T_{c'}\}|$ is the number of pairs of constraints (at-least, at-most) covering s with intersections of at least 3 steps. Large intersections of at-most and at-least constraints are rare but do significantly reduce the search space.

- $\rho^i_{\leq}(s, \mathcal{P}) = |\{c : c \in C_{\leq}, s \in T_c, |T_c \cap S(\mathcal{P})| = r - i\}|$, where r is the parameter of the at-most-r constraint, is the number of at-most-r constraints such that they can cover at most i new blocks of the pattern. For example, $i = 0$ means that r distinct users are already assigned to the scope T_c and, hence, the choice of users for s is limited to those r users.

We emphasise that these terms might seem 'quirky', but they are the result of extensive experimentation with many different ideas, and they represent the best choices found. Space precludes discussion of other possibilities that were tried but not found to be effective in our instances.

Note that $\rho_{\neq}(s)$, $\rho_{\neq,\leq}(s)$, $\rho_{\geq,\leq}(s)$ do not depend on the state of the search and can be precalculated. Also $\rho^i_{\leq}(s, \mathcal{P})$ can make use of the counters that we maintain to speed-up eligibility tests (see above).

The values of parameters $\alpha_{\neq}$, $\alpha_{\neq,\leq}$, $\alpha_{\geq,\leq}$, $\alpha^0_{\leq}$, $\alpha^1_{\leq}$ and $\alpha^2_{\leq}$ were selected empirically using a bespoke automated parameter tuning method. We found out that the algorithm is not very sensitive to the values of these parameters, and we settled at $\alpha_{\neq} = 3$, $\alpha_{\neq,\leq} = 4$, $\alpha_{\geq,\leq} = 2$, $\alpha^0_{\leq} = 40$, $\alpha^1_{\leq} = 4$ and $\alpha^2_{\leq} = 0$.

Note that the function does not account for at-least constraints except for rare cases of large intersection in at-least and at-most constraints. This reflects our empirical observation (also confirmed analytically in Appendix B, see (Karapetyan et al., 2019)) that the at-least constraints are usually relatively weak in our instances and rarely help in pruning branches of the search tree.

We conducted empirical studies of the branching factor and the depth of the search. Our results show significant reduction of both parameters when the branching heuristic was enabled, greatly reducing the size of the search trees. For example, at $k = 40$, with the branching heuristic disabled, the overall number of nodes in the search tree is about $5.7 \cdot 10^{10}$, of which around 90% were at the depths 23–28. The average branching factor at the depth 23 reaches its maximum of 5.55. With the branching heuristic enabled, the overall number of nodes is about $1.0 \cdot 10^6$, of which 90% are at the depths 16–27. The average branching factor reaches its maximum of 2.56 at the depth of 7.

4.5 Worst-Case Analysis of PBT

Recall that, in the worst case, the total number of patterns considered by the PBT algorithm is less than twice the number of complete patterns. Observe that the number of complete patterns equals the number of partitions of a set of size k, i.e. the k'th Bell number $\mathcal{B}_k$ which is $O(2^{k \log_2 k})$. Finally, observe that the PBT algorithm spends time $O(k^2 + kn)$ on each node of the search tree, assuming that checking of all relevant constraints takes $O(k^2)$ time. Thus, the time complexity of the PBT algorithm is $O(\mathcal{B}_k \cdot (k^2 + kn)) = O^*(2^{k \log_2 k})$. In fact, the running time $O^*(2^{k \log_2 k})$ is likely to be optimal. Indeed, Gutin and Wahlström (2016) proved that unless the Strong Exponential Time Hypothesis (Impagliazzo & Paturi, 2001) fails, which is generally considered quite unlikely, there is no $O^*(c^{k \log_2 k})$-time algorithm for WSP with UI constraints with any $c < 2$.

The PBT algorithm follows the depth-first search order and, hence, stores only one pattern at a time. It also maintains a subgraph of the assignment graph with only $O(k^2)$ edges. (See Section 4.2 where the upper bound k on the degrees of blocks in the matching graph is explained.) Hence, the space complexity of the algorithm is $O(k^2)$, i.e. smaller than the problem itself ($O(kn)$). Note that small space complexity is very good for reducing cache misses.

5. Pseudo-Boolean and Constraint Satisfaction Formulations

To make the paper self-contained, we provide in Section 5.1 the old pseudo-Boolean formulation (Cohen et al., 2016), and in Section 5.2 we show how it can be extended to exploit the FPT nature of the problem. We also give a CSP formulation of the problem in Section 5.3.

5.1 Old Pseudo-Boolean Formulation (UDPB)

The main decision variables in the old formulation are $x_{s,u}$, $s \in S$, $u \in U$, where user u is assigned to step s if and only if $x_{s,u} = 1$. Because $x_{s,u}$ directly assigns a step to a particular user, we call this formulation *User Driven PB* (UDPB). We also introduce auxiliary variables $y_{c,u}$ and $z_{c,u}$ for at-least and at-most constraints c, respectively. Variables $y_{c,u}$ and $z_{c,u}$ are used to bound the number of distinct users assigned to the constraints' scopes.

The old formulation is presented below in (6)–(15). We use notation $A^{-1}(s) = \{u \in U : s \in A(u)\}$ for the set of users authorised for step $s \in S$.

$$\sum_{u \in U} x_{s,u} = 1 \qquad \forall s \in S, \tag{6}$$

$$x_{s,u} = 0 \qquad \forall s \in S \text{ and } \forall u \in U \setminus A^{-1}(s), \tag{7}$$

$$x_{s_1,u} + x_{s_2,u} \le 1 \qquad \forall \text{ not-equals constraint with scope } \{s_1, s_2\} \text{ and } \forall u \in U, \tag{8}$$

$$y_{c,u} \ge x_{s,u} \qquad \forall \text{ at-most-}r \text{ constraints } c \text{ with scope } T_c,\ \forall s \in T_c \text{ and } \forall u \in U, \tag{9}$$

$$\sum_{u \in U} y_{c,u} \le r \qquad \forall \text{ at-most-}r \text{ constraint } c, \tag{10}$$

$$z_{c,u} \le \sum_{s \in T_c} x_{s,u} \qquad \forall \text{ at-least-}r \text{ constraint } c \text{ with scope } T_c, \text{ and } \forall u \in U, \tag{11}$$

$$\sum_{u \in U} z_{c,u} \ge r \qquad \forall \text{ at-least-}r \text{ constraint } c \tag{12}$$

$$y_{c,u} \in \{0,1\} \qquad \forall \text{ at-most-}r \text{ constraint } c \text{ and } \forall u \in U, \tag{13}$$

$$z_{c,u} \in \{0,1\} \qquad \forall \text{ at-least-}r \text{ constraint } c \text{ and } \forall u \in U, \tag{14}$$

$$x_{s,u} \in \{0,1\} \qquad \forall s \in S \text{ and } \forall u \in U. \tag{15}$$

Constraints (6) guarantee that exactly one user is assigned to each step. Constraints (7) implement authorisations. Constraints (8)–(12) define WSP constraints (recall that our test instances include only not-equals, at-least and at-most UI constraints, although the UDPB formulation can obviously encode any computable WSP constraints, whether UI or not).

5.2 New Pseudo-Boolean Formulation (PBPB)

A contribution of this paper is a new pseudo-Boolean formulation (16)–(28) exploiting the FPT nature of the problem. This formulation, which we call *Pattern Based PB* (PBPB), was inspired by formulations of the graph colouring problem (Dukanovic & Rendl, 2008; Dutton & Brigham, 1981; Lovász, 1979). In particular, steps s_1 and s_2 are assigned the same user if and only if $M_{s_1,s_2} = 1$ (we assume $M_{s,s} = 1$ for every $s \in S$). Such variables are not concerned with the identity of users and, thus, are more effective when handling UI constraints. This is the same idea as behind colour matrix in (Dukanovic & Rendl, 2008) which preserves the colour symmetry and encapsulates only the decisions that matter at the upper level of the search. However it extends such usage in two ways. Firstly, WSP with UI constraints has a richer set of constraints, defined on "hyperedges". Secondly, the matrix M is tightly integrated with the non-UI authorisations. Thus, we still use the x variables with the same meaning as in the UDPB formulation but complement them with the new variables M.

The formulation (16)–(28) is given for only the specific constraints (namely, at-most-3 and at-least-3 constraints with scope of size 5, and not-equals constraints); for formulations of other constraints, including general UI constraints, see Appendix A, see (Karapetyan et al., 2019).

$$
\begin{aligned}
& M_{s_1,s_2} = M_{s_2,s_1} && \forall s_1 \neq s_2 \in S, && (16)\\
& M_{s,s} = 1 && \forall s \in S, && (17)\\
& M_{s_1,s_2} \geq M_{s_1,s_3} + M_{s_2,s_3} - 1 && \forall s_1 \neq s_2 \neq s_3 \in S, && (18)\\
& M_{s_1,s_2} \leq M_{s_2,s_3} - M_{s_1,s_3} + 1 && \forall s_1 \neq s_2 \neq s_3 \in S, && (19)\\
& \sum_{u \in U} x_{s,u} = 1 && \forall s \in S, && (20)\\
& x_{s_1,u} - x_{s_2,u} \leq 1 - M_{s_1,s_2} && \forall s_1 \neq s_2 \in S \text{ and } \forall u \in U, && (21)\\
& x_{s_1,u} + x_{s_2,u} \leq 1 + M_{s_1,s_2} && \forall s_1 \neq s_2 \in S \text{ and } \forall u \in U, && (22)\\
& x_{s,u} = 0 && \forall s \in S \text{ and } \forall u \in U \setminus A^{-1}(s), && (23)\\
& M_{s_1,s_2} = 0 && \forall \text{ not-equals constraint with scope } \{s_1, s_2\}, && (24)\\
& \sum_{s_1 < s_2 \in T} M_{s_1,s_2} \geq 2 && \forall \text{ at-most-3 constraint with scope } T, |T| = 5, && (25)\\
& \sum_{s_1 < s_2 \in T} M_{s_1,s_2} \leq 3 && \forall \text{ at-least-3 constraint with scope } T, |T| = 5, && (26)\\
& M_{s_1,s_2} \in \{0,1\} && \forall s_1, s_2 \in S, && (27)\\
& x_{s,u} \in \{0,1\} && \forall s \in S \text{ and } \forall u \in U. && (28)
\end{aligned}
$$

The $x_{s,u}$ variables, used to define authorisations in (23), need to be linked to the M_{s_1,s_2} variables. In particular, if $M_{s_1,s_2} = 1$ then we require that $x_{s_1,u} = x_{s_2,u}$ for every $u \in U$, see (21), and if $M_{s_1,s_2} = 0$ then $x_{s_1,u} + x_{s_2,u} \leq 1$, i.e. at least one of $x_{s_1,u}$ and $x_{s_2,u}$ has to take value 0, see (22). To improve propagation, we formulate optional (transitive closure) constraints (18) and (19). These constraints are entailed by the link between the M and x variables in (20)–(22), but adding them increases the propagation avoiding the cost of extra reasoning involving the x variables.

Constraints (24) encode not-equals, (26) encode at-least-3 and (25) encode at-most-3 (these are the constraints present in our instances; for details see Section 7.1). It is useful that (24)–(26) involve only the M variables; together with (16)–(19) they guarantee that a solution corresponds to an eligible pattern. Hence, (24)–(26) correspond to the upper level of the search, i.e. the search over the space of patterns.

It is easy to observe that any constraint that is expressed only in terms of the M's is automatically UI, as it does not involve the x variables (users), and so cannot change with permutations of them. The following proposition states that the converse also applies.

Proposition 5.1. *On solving an instance of the WSP, the decision variables M are sufficient to encode any UI constraint.*

Proof. By definition, any WSP constraint $c = (T_c, \Theta_c)$ can be defined by the set Θ_c of all the plans $\pi : T_c \to U$ that are eligible for $C = \{c\}$. Moreover, if a constraint c is UI then $\pi \in \Theta_c$ implies that $\pi' \in \Theta_c$ for every $\pi' \approx \pi$. Then it follows that a UI constraint can be described by listing equivalence classes of plans or, equivalently, patterns on T_c.

Recall that a pattern can be uniquely described with the M variables; in particular, a pattern $\mathcal{P}$ can be described as $M_{s',s''} = 1$ for every $s', s'' \in B$, $B \in \mathcal{P}$, and $M_{s',s''} = 0$ for every $s' \in B'$, $s'' \in B''$ and $B' \neq B'' \in \mathcal{P}$. Then it is easy to exclude a pattern via a linear inequality expressed in variables M.

Let $\overline{PAT}$ be the list of all patterns on T_c disobeying a UI constraint $c = (T_c, \Theta_c)$. Then, in general, we can encode a UI constraint c with constraints (16), (17) and

$$\sum_{B\in\mathcal{P}} \sum_{s'<s''\in B} (1 - M_{s',s''}) + \sum_{B'\neq B''\in\mathcal{P}} \sum_{s'\in B'} \sum_{s''\in B'', s'<s''} M_{s',s''} \geq 1 \qquad \forall \mathcal{P} \in \overline{PAT}\,. \tag{29}$$

□

To see how Proposition 5.1 works, consider the following example. To require that $\mathcal{P} \neq \{\{s_1, s_2\}, \{s_3\}\}$ (which is a UI constraint), or, equally,

$$M \neq \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

one can write

$$(1 - M_{12}) + M_{13} + M_{23} \geq 1\,.$$

Note that to encode a UI constraint in this way, one may need numerous constraints to prohibit multiple patterns. In Appendix A, see (Karapetyan et al., 2019) we give more compact approaches to formulate some standard UI constraints and also discuss why our encodings (25) and (26) are correct.

5.3 CSP Formulation

As noted above, the WSP with UI constraints is effectively a Constraint Satisfaction Problem (CSP). There is a variable x_s for each step $s \in S$, with the domain $A^{-1}(s)$. Note that any CSP constraint that is not sensitive to specific values, but only there equality or not corresponds to a UI constraint, and vice versa.

The not-equals constraint with scope $T = \{i, j\}$ is encoded as

$$x_i \neq x_j \tag{30}$$

In order to encode an at-least-3 constraint with scope T of size 5, we request that there exists at least one subset of steps in T of cardinality 3 with all different values. Let $\mathcal{X}$ be the set of all subsets of T of cardinality 3. Let $\mathcal{X}_r \subset T$ be the rth element of $\mathcal{X}$. Then the constraint can be encoded using auxiliary Boolean variables v_r as follows:

$$\bigvee_{r=1}^{|\mathcal{X}|} v_r \tag{31}$$

$$v_r \implies x_i \neq x_j \qquad r = 1, 2, \ldots, |\mathcal{X}|, \quad i < j \in \mathcal{X}_r \tag{32}$$

$$v_r \in \{0, 1\} \qquad r = 1, 2, \ldots, |\mathcal{X}|. \tag{33}$$

In order to encode an at-most-3 constraint with scope T of size 5, we request that in every subset $Y \subset T$ of cardinality 4, there exists at least one pair of steps $i \neq j \in Y$ such that $x_i = x_j$. (Indeed, an at-least-3 out of 5 constraint with scope T is falsified if and only if any four steps in T are assigned all different values.) Let $\mathcal{Y}$ be the set of all subsets of T of cardinality 4. Then the constraint can be encoded as follows:

$$\bigvee_{i \neq j \in Y} x_i = x_j \qquad \forall Y \in \mathcal{Y} \tag{34}$$

6. Analysis of WSP Solution Approaches

In this section we analyse and compare the existing and new WSP solution approaches. In Section 6.1, we discuss different branching strategies and how they are linked to performance of WSP algorithms. Section 6.2 discusses properties of PBPB with respect to both the upper level search for eligible patterns and of the assignment problems that arise at the lower level. (For asymptotic worst-case analysis of PBT and PUI please see Section 4.5).

6.1 Branching Strategies

In this section, we use the language of patterns and of the PBPB encoding in order to discuss possible branching strategies in a WSP algorithm. The first key observation is that any pattern can be described with M_{ij} variables. The matrix of M variables corresponding to a complete pattern is exactly a permutation of block-ones-diagonal matrix, where a block in the matrix corresponds to a block of the pattern. A pattern as used within PBT is then a set of blocks (see Figure 5a) with the requirement that the steps in different blocks are assigned different users.(It is important to note that in such figures, purely for illustration, we are assuming that the rows/columns are permuted to reveal any block-diagonal structure; there is no implication that the steps are processed in a fixed order.) We will say that such an (incomplete) pattern is 'open' as the relation between the steps in the pattern and those not in the pattern is left as undetermined; for a step not in the pattern, the values of M are not yet fixed. The openness of the pattern corresponds to the open nature of the assignments within PBT. Figure 6 illustrates the branching within PBT in terms of the options for extending the value assignments to the M matrix.

PUI, the previously state-of-the-art FPT algorithm for the WSP with UI constraints (Cohen et al., 2016), implements a different branching strategy. It iterates over the set of users U gradually building a set $\mathscr{P}$ of valid patterns. At an iteration corresponding to some user $u_i \in U$, PUI attempts to extend each $\mathcal{P} \in \mathscr{P}$ with exactly one new block B, trying each non-empty $B \subseteq A(u_i) \setminus S(\mathcal{P})$. PUI will never attempt to extend an existing block in a pattern, and in this sense it uses 'closed' patterns, see Figure 5b. While closed patterns support richer propagation, they also reduce the flexibility of search. Indeed, a single extension of a closed pattern inevitably fixes many more M variables than that of an open pattern, reducing the ability of the algorithm to focus on the most constrained parts of the problem. In general, the smaller the decisions are, the more flexibility the search has in ordering them and, hence, prioritising the most important ones. Thus, we suggest that open patterns are preferable to closed patterns.

$$\begin{array}{c|cccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \\
\hline
s_1 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{?} & \mathbf{?} & \mathbf{?} \\
s_2 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{?} & \mathbf{?} & \mathbf{?} \\
s_3 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{?} & \mathbf{?} & \mathbf{?} \\
s_4 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{?} & \mathbf{?} & \mathbf{?} \\
s_5 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{?} & \mathbf{?} & \mathbf{?} \\
s_6 & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{1} & \mathbf{?} & \mathbf{?} \\
s_7 & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{1} & \mathbf{?} \\
s_8 & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{?} & \mathbf{1}
\end{array}$$

(a) Open pattern. The pattern has two blocks, each of which can be extended with new steps. New steps can also be assigned to a new block.

$$\begin{array}{c|cccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \\
\hline
s_1 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
s_2 & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
s_3 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
s_4 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
s_5 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
s_6 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{?} & \mathbf{?} \\
s_7 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{?} & \mathbf{1} & \mathbf{?} \\
s_8 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{?} & \mathbf{?} & \mathbf{1}
\end{array}$$

(b) Closed pattern. Both blocks are closed, i.e. the algorithm cannot add any other steps to these two blocks; it can only create new blocks.

Figure 5: Open vs. closed pattern in terms of M matrix. Full M matrix is shown for clarity although it is symmetric by definition. Question marks show undecided variables. Grey cells (with zeros) are variables fixed at 0; green and red cells (with ones) are variables fixed at 1.

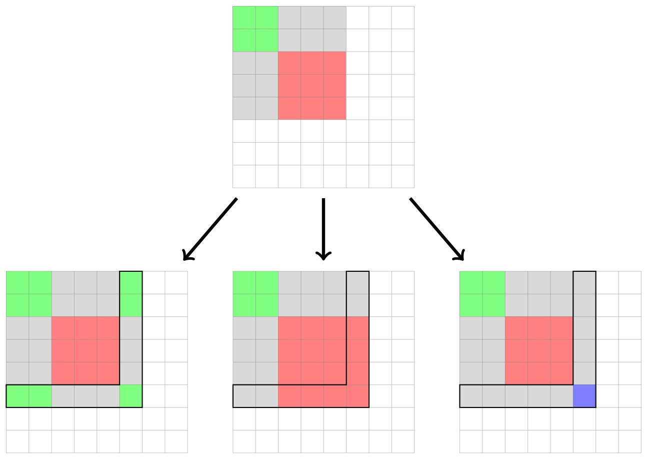


Figure 6: PBT branching. The parent pattern contains two blocks (green of size 2 and red of size 3). PBT extends the parent pattern by assigning a new step in three different ways: (left) extend the first block with the new step; (centre) extend the second block with the new step; (right) create a new block consisting of the new step only. A black frame encloses the the variables fixed in each of the branches. Note that the branching step can be chosen arbitrarily; PBT uses a heuristic to select the branching step.

Storing the set $\mathscr{P}$ of valid patterns also makes the space complexity of PUI exponential in k. Consider an instance with $n = k$ and no constraints. Let $s \in S$ be some step, and let all the users in this instance be authorised to all the steps except for s, i.e. $A(u) = S \setminus \{s\}$ for every $u \in U$. Clearly, such an instance is unsatisfiable. To establish this, PUI will generate and store all incomplete valid patterns. Observe that any partition of $S' \subseteq (S \setminus \{s\})$ is a valid pattern in this instance. To count them, consider the set of all partitions of S. There are exactly $\mathcal{B}_k$ of such partitions. Now, in each of these partitions, remove the block that includes s. This will give us a set of all partial partitions of

S that do not cover s. This set exactly corresponds to the set of all the incomplete valid patterns in our problem instance. Hence, to solve this instance, PUI will populate $\mathscr{P}$ with exactly $\mathcal{B}_k$ patterns. Hence, the space complexity of PUI is $\Omega^*(B_k)$, where Ω^* hides polynomial factors similarly to the O^* notation.

Another aspect in which PBT is different to PUI is that PBT implements delayed assignment of users, branching on user-independent M variables. Conversely, PUI branches on the x variables, i.e. fixes the assignment of users while branching, achieving the FPT running time by merging equivalent branches. We argue that delayed user assignment would usually be more effective. As we show in Appendix A, see (Karapetyan et al., 2019), a significant portion of patterns are authorised but usually only a few patterns are eligible; hence branching on the M variables is likely to produce smaller search trees.

It is possible to implement an algorithm with closed patterns and delayed user assignment. Our attempt to implement such a 'closed-pattern-based PBT', however, resulted in an algorithm significantly slower than PBT (although faster than PUI).

It is also possible to implement an open-pattern style search with immediate user assignment. Indeed, a general purpose PB solver is likely to exploit this strategy when solving UDPB formulation. However, assignment of some (but not all) x_{su} for a $u \in U$ would mean that subsequent pruning of the branch does not guarantee that the corresponding open pattern is invalid; indeed, a different assignment of x_{su}, $s \in S$, may produce a valid plan. Hence, patterns could not be used to merge branches of the search, and the algorithm would not be FPT. For FPT algorithms with immediate user assignment, it is vital, like in PUI, to try all the authorised combinations of x_{su} for a given $u \in U$ before proceeding to the next u; by following this strict sequence, the algorithm guarantees that it finds all the eligible patterns authorisable by processed users. For the same reason, PUI could not be implemented as a DFS algorithm.

An important observation is that FPT algorithms with immediate user assignments (i.e. PUI) are forced to order the branching variables by user whereas algorithms with delayed user assignments and open patterns (i.e. PBT) order the branching variables by steps. In a problem with a relatively small number of steps and a large number of users, it is more likely that the users are relatively uniform compared to the steps, and hence the search is more sensitive to the order of steps than to the order of users. In other words, branching heuristics in PBT-like algorithms are expected to be more effective compared to branching heuristics in PUI-like algorithms.

Now consider the combination of the PBPB encoding with a DPLL-based (Davis, Logemann, & Loveland, 1962) PB solver such as SAT4J. Internally, a PB solver on PBPB will need to be making branching decisions. This is generally done so as to prefer branching on variables that propagate to entail values for other variables, and given the central nature of the M variables, it seems reasonable they would be favoured as branch variables. As pointed out above, a complete assignment to the variables M_{s_1,s_2}, $s_1, s_2 \in S$, and satisfying the constraints (16)–(19), uniquely defines a complete pattern. A PB solver will be handling partial assignments to the M variables, but it is still reasonable to ask if they are structured like open or closed patterns. To address this, consider the effects of the transitivity constraints (18) and (19). If two M variables sharing a step, e.g. M_{12} and M_{23}, are set to 1, then (19) immediately forces a propagation, $M_{13} = 1$, and similarly for (18), so if block-diagonals of 1's happen to overlap then will form into a larger block of 1's. Hence there is a tendency to complete the blocks in the M matrix, but there will be no reason to close them. This will tend to drive the partial M-assignments to have a structure close to open patterns. We hence expect that a standard (DPLL-based) PB solver could work on the PBPB formulation in a similar fashion of using open patterns and then extending them. So we expect that the behaviour of the PB solver with PBPB will be more similar to PBT than to PUI; we will see evidence for this in Section 8.

We also note here, that, unlike the bespoke PBT and PUI algorithms, PBPB solvers have the flexibility to arbitrarily alternate between branching on M's and x's. When user authorisations are tight, this may lead to superior strategies and hence is a potential strength of the general-purpose solver approach.

6.2 Properties of the New Pseudo-Boolean Formulation PBPB

In this section, we show that the PBPB formulation can also potentially admit FPT running time and polynomial space complexity. The discussion breaks into the upper level search on the M-variables for eligible patterns, and the subsequent lower level matching problems arising from the x variables in the context of a pattern.

For the upper level, there are $O(k^2)$ of the M variables, and so a tree search in PB would have the potential to fully instantiate these before handling the user assignments via the x variables (which is not an unreasonable assumption, see Section 6.1), using a tree of worst-case size $2^{O(k^2)}$. This is FPT, and so whether or not PBPB has a potential to be FPT as a whole depends on the complexity of the user assignments once a complete pattern is reached.

When all the M's are instantiated, the PBPB formulation (16)–(28) reduces to the following:

$$\sum_{u \in U} x_{s,u} = 1 \qquad \forall s \in S, \tag{35}$$

$$x_{s,u} = 0 \qquad \forall s \in S \text{ and } \forall u \in U \setminus A^{-1}(s), \tag{36}$$

$$x_{s_1,u} = x_{s_2,u} \qquad \forall s_1 \neq s_2 \in B, B \in \mathcal{P} \text{ and } \forall u \in U, \tag{37}$$

$$x_{s_1,u} + x_{s_2,u} \leq 1 \qquad \forall B_1 \neq B_2 \in \mathcal{P},\ \forall s_1 \in B_1,\ \forall s_2 \in B_2 \text{ and } \forall u \in U, \tag{38}$$

$$x_{s,u} \in \{0,1\} \qquad \forall s \in S \text{ and } \forall u \in U. \tag{39}$$

This is a bipartite matching problem but with blocks of steps being assigned to a user. However, because of (37), when any one step in a block is assigned (i.e. some $x_{s,u} = 1$) then all the other steps in the block are also forced, by propagation, to the same user.

While we know algorithms to solve the matching problem in polynomial time, a general-purpose PB solver in a mode that essentially only uses SAT-style resolution cannot solve it efficiently. The Pigeon Hole Problem, seeking to assign n entities to $n-1$ holes and which is a special case of the Bipartite Matching Problem, is known to be exponentially hard for DPLL solvers (Haken, 1985) that only use SAT representations; but polynomial size proofs are possible for PB, and so this explains why we encode it using PB. The following proposition shows the formulation (35)–(39) can be solved in FPT time by a general purpose PB solver, using standard tree-search methods (branching and propagation), but not introducing new variables during the search process. It is important to make this assumption because search or proof methods that are allowed to introduce new variables have the potential to be a lot more powerful, e.g. (Razborov, 2002), but such methods are currently too difficult to control in practice.

Proposition 6.1. *The PB formulation (35)–(39) can be solved by tree search and propagation (without the introduction of new variables), in polynomial space, and in time exponential in $|\mathcal{P}|$ only.*[2]

2. The proposition is closely related to known methods in kernelisation (Gutin et al., 2015); however, due to the lack of space we do not want to pursue that here.

Proof. The PB formulation (35)–(39) corresponds to the standard bipartite matching problem on a graph with the vertices of one partition consisting of the blocks of $\mathcal{P}$ and the other partition vertices are the users U. Observe that if the degree (number of authorised users) of a block $B \in \mathcal{P}$ is greater than the number of blocks in $|\mathcal{P}|$, then no set of choices for the other blocks can remove all the options for that block. Hence, all vertices $B \in \mathcal{P}$ of degree $|\mathcal{P}|$ or above can be delayed until last in the search tree: if the search does reach them, then they can be given arbitrary values and so will never lead to backtracking. (An example occurs in Figure 3a in which the block $\{s_3\}$ has 3 authorised users; so its assignment can be delayed until after the other 2 blocks.) Variables within a block are all constrained to be equal, hence, eventually one of them will be picked as the branch variable; at this time the propagation will give values to all the others. The other members of a block hence will no longer be candidate branch variables, and they will not contribute to the size of the search tree. Hence in the backtracking portion of the branching, branching factor of the search will be limited by $|\mathcal{P}| - 1$, and the depth of the search by $|\mathcal{P}|$. Hence the search tree size is $O((|\mathcal{P}| - 1)^{|\mathcal{P}|})$, and the depth is polynomial. □

This proposition is sufficient to show that a PB solver based on standard branch-and-propagate methods has the potential to solve the PBPB formulation in FPT time. However, Proposition 6.1 effectively shows that an unbalanced bipartite matching problem (with parts of size $O(k)$ and $O(n)$, respectively) can be solved by a PB solver in time polynomial in n and exponential in k, whereas we know that the Hungarian method is polynomial in both n and k. Although we have not observed difficult matching problems in our experiments with WSP algorithms, it is still natural to discuss the worst case, and consider what are the potential limitations of the PB approach. For this we will switch to a "proof theory" perspective and ask about the sizes of the proofs of unsatisfiability available within the PB representation (note that a proof of satisfiability of the matching problem is trivial in the sense that it is just the verification of a given witness).

Proposition 6.2. *When the PB formulation (35)–(39) is unsatisfiable, then there is a PB proof of that unsatisfiability, without introducing new variables, and that is polynomial in both $|\mathcal{P}|$ and n.*

Proof. Observe that we can take an arbitrary representative, a step, from within each block of the pattern $\mathcal{P}$ and use propagation through (37) to limit the users permitted for the representative. Hence the problem (35)–(39) is precisely the matching of the selected representative of each block to a permitted user. The Proposition 6.2 follows from the Hall's marriage theorem (Cameron, 1994); a matching problem on a bipartite graph $G = (L \cup R, E)$ has a complete matching of the vertices of the partition L, if and only if it is true that for all subsets L' of L, there are at least $|L'|$ elements in R that may match with some vertex in L'. In WSP language this basically means that the matching problem is unsatisfiable if and only if there is some subset $\mathcal{B}$ of blocks, for which the corresponding set of candidate users is smaller than $|\mathcal{B}|$; for example, see Figure 3b. Such a subset is a constrained form of the pigeon-hole problem (PHP), stating that $|\mathcal{B}|$ blocks cannot be assigned to fewer than $|\mathcal{B}|$ users, and restricting equations (35)–(39) to such a subset from the marriage theorem leads to a PB encoding of the PHP. (There are also extra constraints to remove unauthorised assignments within the PHP, but these are not needed, as the counting already suffices.) Since the PHP is known to have a polynomial size PB proof without the use of new variables, see e.g. (Cook et al., 1987; Dixon et al., 2004), it follows there is also a PB proof of (35)–(39). □

Note that Propositions 6.1 and 6.2 are quite different in that the first one is discussing the process of a solution, whereas the second one is about a witness to unsatisfiability but not the time to find it. Nevertheless we conclude that a PB solver might, at least, be expected to use tree search to solve the assignment problem in time exponential in k, but also have the potential to be able to solve it in time polynomial in k.

We further note that our separate experiments confirmed that at large n the matching problem occurring at the lower level of the algorithm are typically very easy as there is little conflict between users, and simple propagation is usually sufficient to solve the problem.

7. Instance Generator and Phase Transitions

It is preferred to use real-world instances in computational studies. However, there are only very few real-world instances of WSP available publicly, and those are of size too small to be of interest in our computational experiments, see e.g. (Bertolissi et al., 2018). We have considered the instances appearing in practice from (Bertolissi et al., 2018; dos Santos et al., 2015, 2017), namely, TRW (Trip Request Workflow), ITIL (IT Financial Reporting), and ISO (Budgeting for Quality Management) having 5, 7, 9 steps (tasks) and 5, 2, 3 not-equals constraints, respectively. More details and description of user-step authorisations for 3 satisfiable and 3 unsatisfiable versions of these instances can be found in Section 5 of (dos Santos et al., 2017). Our new solver PBT correctly solved each of the six instances in less than 0.001 sec. If some businesses do not use more and more complex constraints because of computational complexity issues, PBT provides a good opportunity to consider larger and more involved workflow scenarios.

Therefore, due to the difficulty of acquiring real-world instances with an appropriate range of sizes, to support extensive studies of the scaling of the runtimes, similarly to other authors (Cohen et al., 2016; Roy et al., 2015; Wang & Li, 2010), we use the synthetic instance generator described in (Cohen et al., 2016) and available for downloading (Karapetyan, 2019). In this section, we first present the generator of the WSP instances. We then empirically study the probability of satisfiability of the instances as we vary the generator parameters. We give evidence for PT, between the satisfiable and unsatisfiable regions. The point is that the resulting instances from the PT region can be expected to be a good test of the effectiveness of solution algorithms.

7.1 The Instance Generator

Three families of UI constraints are used: *not-equals* (also called *separation-of-duty*), *at-most-r* and *at-least-r* constraints. A not-equals constraint with scope $\{s, t\}$ is satisfied by a complete plan π if and only if $\pi(s) \neq \pi(t)$. An at-most-r constraint c with scope T_c is satisfied if and only if $|\pi(T_c)| \leq r$. Similarly, an at-least-r constraint c with scope T_c is satisfied if and only if $|\pi(T_c)| \geq r$. We do not explicitly consider the widely used binding-of-duty constraints, that require two steps to be assigned to one user, as those can be trivially eliminated during preprocessing. While the binding-of-duty and separation-of-duty constraints provide the basic modelling capabilities, the at-most-r and at-least-r constraints impose more general "confidentiality" and "diversity" requirements on the workflow, which can be important in some business environments. We decided to focus this study on at-least-3 and at-most-3 constraints with a scope of 5, $|T_c| = 5$ for the following reason. Cohen et al. (Cohen et al., 2016) performed several computational experiments with the at-least-r and at-most-r constraints for various values of r and various sizes of scope T. The constraints at-least-3 and at-most-3 with scope of size 5 were selected in (Cohen et al., 2016) as they appear to be in the practical range of the parameters r and $|T|$ and the values $r = 3$ and $|T| = 5$ often lead to computationally challenging WSP instances. For more details, see Section 5.2 of (Cohen et al., 2016).

The specific stochastic WSP Instance Generator takes as input four parameters,

1. k, the number of steps;
2. n, the number of users;
3. e, the number of not-equals constraints;
4. γ, the number of at-most-3 and also the number of at-least-3 constraints (all with scope 5).

The generator, which we denote as $WIG(k, n, e, \gamma)$, is stochastic, but as usual can be made deterministic by also specifying a value for the random generator seed. For each user $u \in U$, it generates $A(u)$ such that the size of $A(u)$ is first selected uniformly from $\{1, 2, \ldots, \lfloor 0.5k \rfloor\}$ at random and then the set $A(S)$ itself is selected randomly and uniformly from S with no repetitions. This results in each step having $n/4$ authorised (random) users on average. The generator also produces e distinct not-equals, γ at-most-3 constraints, and γ at-least-3 constraints, all uniformly at random.

Note that in general, WSP instances could have some symmetries, e.g. different steps, or different users are interchangeable, potentially, making search processes more inefficient and needing the usage of symmetry breaking methods (e.g. see (Crawford et al., 1996)). However, our benchmark instances have negligible amounts of such symmetry (both by theory and also by direct evaluation) and so we do not consider it here.

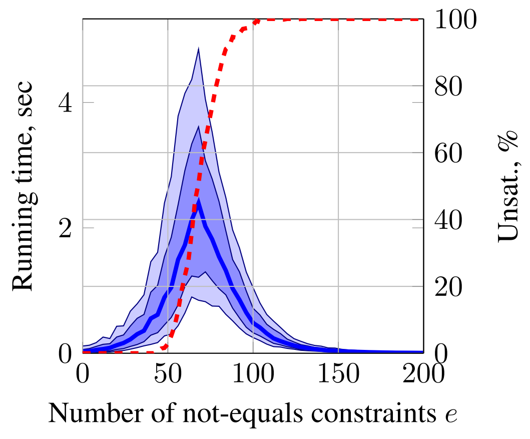
The PBT algorithm, conversion routines, test instances with solutions and the test instance generator are available for downloading (Karapetyan, 2019).

7.2 Thresholds from the Instance Generator

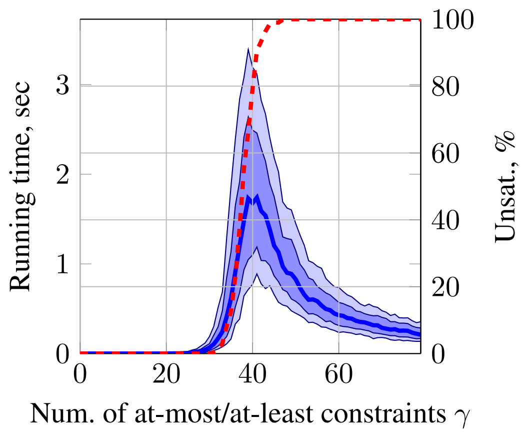
In this section, we focus on experiments to study the dependency of instance properties on the parameters of the instance generator, and show that the WSP instances we use exhibit the classic properties expected of PTs.

Figure 7 shows an example, at $k = 40$ and $n = 400$, of the running time of PBT and the percentage of unsatisfiable (unsat) instances as they change with variation of parameters e (Figure 7a) and γ (Figure 7b). As one could expect, the number of unsat instances grows with the number of constraints. It can also be observed that the hardest instances are those near the 50% level of the unsat curve. Following standard arguments, under-constrained instances have a lot of valid plans which makes them relatively easy. Unsatisfiability of the oversubscribed instances can be proved relatively quickly due to heavy pruning of the branches. However, instances around the 50% unsat level are likely to have none to few valid plans making it hard to find them or prove unsatisfiability. Note that one can argue that real-world instances in many cases are likely to be in the region of 50% unsatisfiable: Organisations are likely to be constraining their workflows up to the point when the workflows become unsatisfiable, or start with unsatisfiable workflows and gradually relax the constraints until obtaining satisfiable problems.

In the comparison of algorithms we use 'critical' instances with $\gamma = k$ and $e = e_{50}$, where $e_{50} = e_{50}(k, n, \gamma)$ makes $WIG(k, n, e_{50}, \gamma)$ instances satisfiable with 50% probability. The values of $e_{50}(k, n, \gamma)$ are obtained empirically for each k, n and γ and are freely available to facilitate future work (Karapetyan, 2019).



(a) The number γ of at-most and at-least constraints is fixed at $\gamma = k$, and the number e of not-equals constraints is varied.



(b) The number e of not-equals constraints is fixed at $e = 78$ (corresponding to 10% of all available choices), and the value of γ is varied.

Figure 7: At $(k, n) = (40, 400)$, the running times of PBT (blue) and percentage of unsatisfiable instances (red) for various values of the instance generator parameters. The blue shades show the [35%–65%] (deep blue) and [25%–75%] (lighter blue) percentiles of the runtimes.

Although we do not use it directly, it is also possible to estimate e_{50} analytically based on the instance generator properties. In Appendix B, see (Karapetyan et al., 2019), we give an (approximate) computation in the style of an 'annealed estimate' (Selman & Kirkpatrick, 1996) of the average number of solutions given k, n, e and γ; with the intent to use it to obtain an approximate value of $e_{50}(k, n, \gamma)$. The annealed estimate seeks the average number of solutions over all instances, and so gives an upper bound on the PT – because once the average drops below 0.5 then at least half the instances must be unsatisfiable. (A similar concept is used in the constrainedness parameter (Gent et al., 1996) which is expressed via the average number of feasible solutions in an ensemble of problem instances.) The main novelty of our analysis is that it accounts for the unevenness of distribution of solutions arising from the two-level nature of the problem. In particular, we observed that a straightforward strategy of direct estimation of the probability of a single plan being valid does not give a tight estimate in our case. Indeed, there might be millions of authorised plans per pattern but at the same time the expected number of eligible patterns can be well below one. In that case, most of the instances will have no valid plans at all but some very rare instances will have millions of valid plans. Since the straightforward estimation strategy gives the average number of valid plans per instance, its result is likely to be a significant over-estimate of the position of the PT. In order to estimate the critical point e_{50} more accurately, we have to ask a different question: we need to know when the probability of an instance to have at least one valid plan is 50%. For this, we have to estimate the number of eligible patterns and then the probability of a pattern being authorised. This will give us the expected number of valid *patterns* which yields a more accurate estimate of the critical point e_{50}.

Lastly, to further support that the observed phenomena have properties expected of a PT, we conducted a set of experiments at the critical points and around them. In particular, we show in Appendix C, see (Karapetyan et al., 2019) the emergence of *forced variables* similar to (Culberson

& Gent, 2001), i.e. the decision variables with values forced by the instance, in the critical region. We observe that the PT coincides with a rapid growth of the number of M variables forced to be either 0 or 1, effectively corresponding to forced (not included explicitly) not-equals or "equals" constraints, respectively.

8. Computational Experiments

Specifically, we empirically study and compare the following WSP solvers:

PBT The algorithm proposed in this paper;

PUI The FPT algorithm proposed and evaluated in (Cohen et al., 2016);

UDPB (Res) The old pseudo-Boolean SAT formulation of the problem (see Section 5.1) solved with SAT4J (Le Berre & Parrain, 2010) in the resolution proof system mode. We also attempted to use the cutting planes mode to solve UDPB but the performance was prohibitively poor for running the experiments.

PBPB (Res) The new pseudo-Boolean SAT formulation, PBPB, of the problem (see Section 5.2) solved with SAT4J in the resolution proof system mode.

PBPB (CutP) PBPB solved with SAT4J in the cutting planes proof system mode.

CSP (CP-SAT) Our CSP formulation solved with CP-SAT, the latest constraint solver from OR-Tools.

The PBT algorithm is implemented in C#, and the PUI algorithm is implemented in C++. Our test machine is based on two Intel Xeon CPU E5-2630 v2 (2.6 GHz) and has 32 GB RAM installed. Hyper-threading is enabled, but we never run more than one experiment per physical CPU core concurrently, and concurrency is not exploited in any of the tested solution methods.

8.1 Slices in Studying Algorithm Scaling

As discussed in Section 7, we focus on the PT WSP instances in our computational study. In a standard (non-FPT) study of PTs, this is generally straightforward – at least conceptually, though potentially quite computationally challenging. For example, consider standard Random-3SAT; the size of the problem is indicated by the number, n, of propositional variables. For each value of n, the number of clauses c is selected so that the instances have a 50% probability of being satisfiable, which we might write as "set $c = c_{50}$". Then, to study the algorithm's complexity, one tests it on these PT instances.

However, a key aspect of FPT is that, in addition to a main problem size parameter n, it also has some other parameter k which is closely involved in the problem complexity. One will generally wish to study the algorithm's scalability in terms of both k and n. We call n and k *size parameters* following the observation that they control the size of the space of WSP solutions. Consequently, we say that (k, n) is the *size space*. The remaining parameters e and γ are then *constraint parameters* as they control the number of constraints, and they are chosen such that the instances have 50% chance of being satisfiable as discussed in Section 7.2.

Since (k, n) is two dimensional, studying the performance over the whole size space is computationally expensive and also difficult to analyse. Accordingly, in this paper, for simplicity and clarity we study the scaling along one-dimensional subspaces of (k, n), which we will refer to as *slices*. Since the size space is two-dimensional, we need to study the scaling in at least two independent (not necessarily orthogonal) directions; or along two independent one-dimensional "slices". While studying the FPT properties, it is natural that such slices should also tend to focus on the regions in which k is small compared to n.

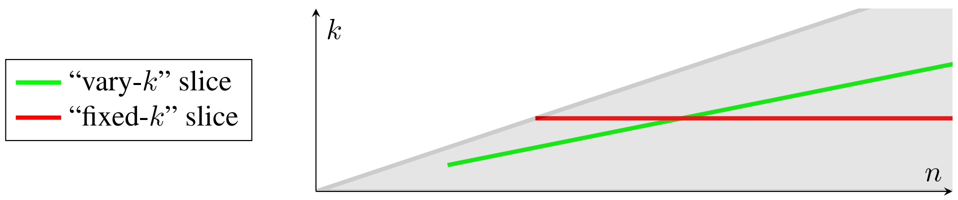


Figure 8: Schematic view of the two "slices" used in our computational study in order to cover the two-dimensional size space (k, n) used in order to investigate the empirical FPT properties. Such FPT studies should naturally focus on the lower right (grey) region in which k is small compared to n.

Many options of choosing the slices are possible, including non-linear slices, however in this paper we will just use two linear slices schematically illustrated in Figure 8, as they give a good and useful insight into the behaviour:

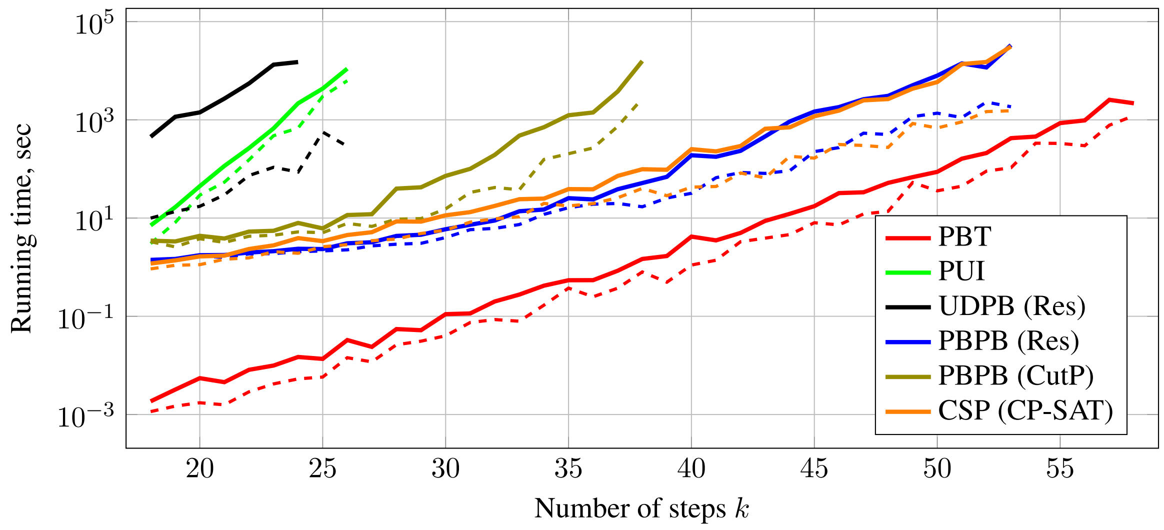
"vary-k" Vary the value of k, but the value of n is given as a specified function of k. In this paper, following (Cohen et al., 2016; Karapetyan et al., 2015), we use the choice $n = 10k$. This gives a simple and clean way of keeping k to be 'small' compared to n. We also report the experimental results for the $n = 100k$ slice in Appendix D, see (Karapetyan et al., 2019), however the conclusions are somewhat similar.

"fixed-k" Use a constant value of k and vary n. This is a natural slice for a test of FPT performance; recall that the worst-case time complexity grows polynomially with n at a fixed k, and one can expect the algorithms to demonstrate good scalability in this slice.

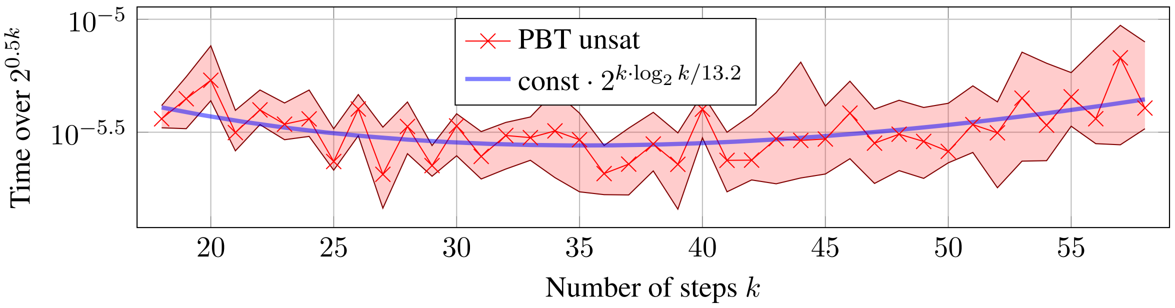
8.2 Performance Comparison: Slice "vary-k"

To compare performance of various WSP algorithms, for each value of k we generated 100 instances using $WIG(k, 10k, e_{50}, k)$. The empirical number of not-equals constraints e_{50} for each k needed to obtain PT instances is shown on Figure 9c. One can observe the 'zig-zag' shape of the curve in that the values corresponding to odd k's are greater than the values corresponding to even k's. This minor artefact arises simply because the size of each authorisation list is randomly drawn by our instance generator from $[1, \lfloor 0.5k - 0.5 \rfloor]$. As a result, the average number of authorisations in an instance with $k = 2i - 1$ is equal to that in an instance with $k = 2i$, $i \in \mathbb{N}$. This makes the authorisations in 'even' instances slightly more constrained compared to 'odd' instances, which is then compensated by reduced number e of not-equals constraints.

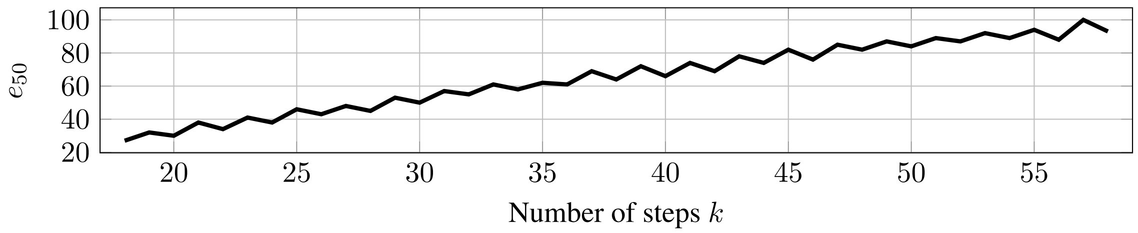
We then solved each instance with each of the algorithms and in Figure 9 we report the median running time. We report separate times for the satisfiable and unsatisfiable instances. The immediate observations are that all the algorithms demonstrate roughly exponential growth of the running time, but that the performances of all the methods differ widely.



(a) The solid lines correspond to unsatisfiable instances, and dashed to satisfiable instances.



(b) PBT unsat running time with the [35–65] percentile range (shaded) as an indication of a confidence interval on the medians. The vertical axis is rescaled to better show the fit. Any exponential function would be a straight line in this plot, however empirical running time appears to be curved. The blue line appears to be an accurate approximation of the running times, demonstrating that scaling of PBT running time closely follows B_k, with a speed-up power factor 13.2.



(c) The number e_{50} of not-equals constraints at phase transition.

Figure 9: Evaluation of algorithms' performance along the "vary-k" slice, i.e. $WIG(k, 10k, e_{50}, k)$.

Crucially, the new PBPB (Res) and PBT (Res) both drastically outperform the previous UDPB (Res) and PUI, showing lower growth rate and also being significantly faster even on small instances. We first look at the scaling of the best performing PBT, on the unsatisfiable instances as the discussion of scaling of satisfiable instances can be obscured by finding solutions early in the search tree. Although it is not immediately obvious, the scaling of the PBT on unsatisfiable instances in Figure 9 is slightly super-exponential; the empirical curve bends slight upwards and so 2^{ak}, with a constant a, does not give a convincing fit. Deep analysis of the average case effects of heuristic improvements in such tree-based search is not yet possible, but generally the expectation, based on experience, is that heuristics will improve the coefficients in the exponents but retain the form. Recalling from Section 4.5, that the number of patterns scales as $2^{\Theta(k \log k)}$, it is reasonable to compare the empirical scaling to $2^{(k \log_2 k)/b}$ for some empirically determined constant b. In Figure 9b we show that a good fit is a function $2^{k \cdot \log_2 k / 13.2}$, confirming our assumption and indicating the effectiveness of the branching heuristics and pruning.

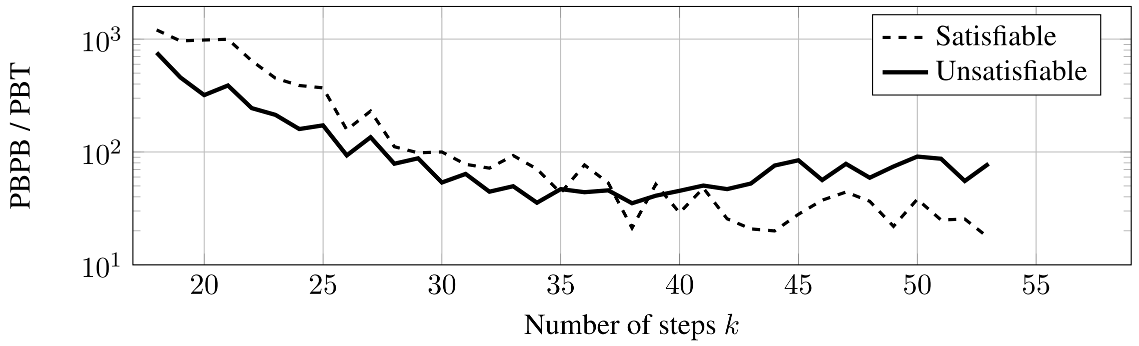
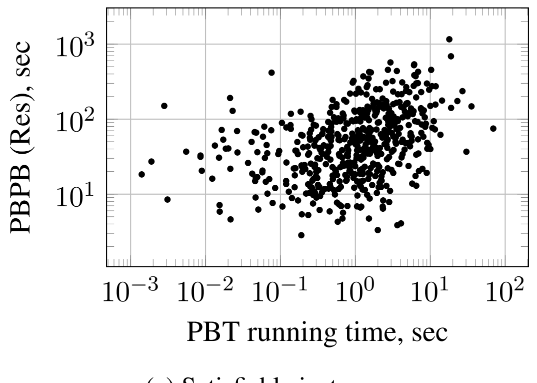
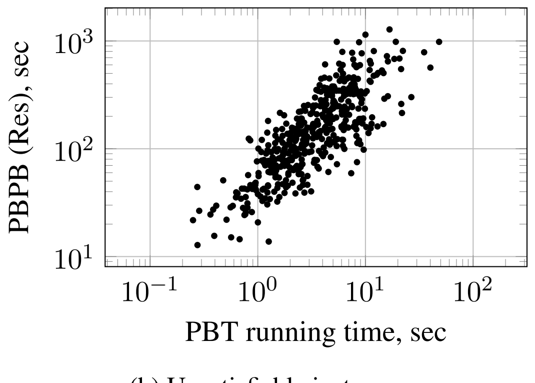


Figure 10: Comparison of PBPB and PBT performance. The vertical coordinate is the ratio between the median running time of PBPB (Res) and median running time of PBT.



(a) Satisfiable instances



(b) Unsatisfiable instances

Figure 11: Correlation between PBT and PBPB (Res) running times on instances of size $k = 40$. 1000 instances used in this experiment (about 500 satisfiable and 500 unsatisfiable).

The next observation is that PBT is faster than PBPB (Res) by one to two orders of magnitude, but that the scaling behaviours are similar. A more accurate comparison can be done by inspecting Figure 10. While the limited range of k and noise do not allow us to convincingly make any definite conclusions, we hypothesise that the ratio of the runtimes between PBT and PBPB (Res) is close to a constant for large k. Also, PBPB (Res) is relatively slow on small instances, presumably because of an expensive initialisation/preprocessing, normal for an off-the-shelf solver, but this does not affect the method's scalability. The similarity of the scaling on larger instances ($k > 35$) supports our hypothesis from Section 6.1 that the search processes of PBPB (Res) and PBT could well be similar.

It is also clear that PBPB (Res) and CSP (CP-SAT) show (remarkably) similar performance, despite CSP (CP-SAT) not explicitly using patterns. In this paper, we treat the PBPB and CSP solvers in an "off-the-shelf blackbox" fashion and do not study their internal workings. However, CP-SAT is built on top of a SAT solver, and so we speculate that the search in the SAT solver effectively mimics a pattern-based reasoning. This could be directly through the the v_k variables. Alternatively, conversion to SAT could introduce a new variable for each term or predicate that occurs in the constraints, including the terms $x_i = x_j$ and $x_i \neq x_j$ in (30), (32) and (34) – such Boolean variables would play a similar role to the pattern variables M_{ij}.

We also observe that the ratio between the solution time of unsatisfiable and satisfiable instances steadily grows for PBPB (Res), while it stays roughly constant for PBT. This may be explained by the fact that the PB solver is likely to employ some heuristics for ordering search branches; these heuristics generally improve the running time of the solver on satisfiable instances while leaving its performance on unsatisfiable instances intact. PBT does not currently have any such heuristic; in our attempt to implement one, the gain was comparable to the overheads, and, thus, we dropped the branch ordering heuristic in our final implementation of PBT.

We also directly investigated whether the running times of PBPB (Res) correlate with the running times of PBT, see Figure 11. On satisfiable instances the correlation is relatively weak which is natural as the running time depends on the branching decisions which differ in the two algorithms. On unsatisfiable instances the correlation is much stronger which again shows that, although the individual branching decisions of the two algorithms may be different, the effectiveness of the heuristics is comparable.

The gap between the running times on satisfiable and unsatisfiable instances is relatively small (within one order of magnitude) for all the solvers except for UDPB (Res). We hypothesise that this difference is due to the inherent symmetry of the UDPB formulation, meaning that there are many valid plans and there is a high probability of finding one well before exhausting the entire search tree. In contrast, the number of valid patterns in a PT instance is likely to be small, and one is likely to be found only after searching a significant part of the search tree. Note that PBPB (Res) is still superior to UDPB (Res) on satisfiable instances, as the plans search tree is much larger than the patterns search tree when $n \gg k$.

Although we argued that good experiments should seek instances in the PT region of parameter space, it is still interesting to verify the performance of PBT on under- and over-constrained instances. To build such instances in a consistent way, we define a new parameter β, and study instances $WIG(k, 10k, \beta e_{50}(k, 10k, k), \beta k)$, i.e. the PT instances with the number of constraints scaled by β. Figure 12 shows how the scaling changes as we move away from the PT, $\beta = 1.0$. It shows the classic so-called "easy-hard-easy" behaviour. Below the PT ($\beta < 1$) the runtimes and scaling are much better than at the PT; that is most of the instances have many solutions, and so solving will terminate early. Above the PT ($\beta > 1$) most of the instances are unsatisfiable, but the pruning will increase, and this is reflected in the improved scaling.

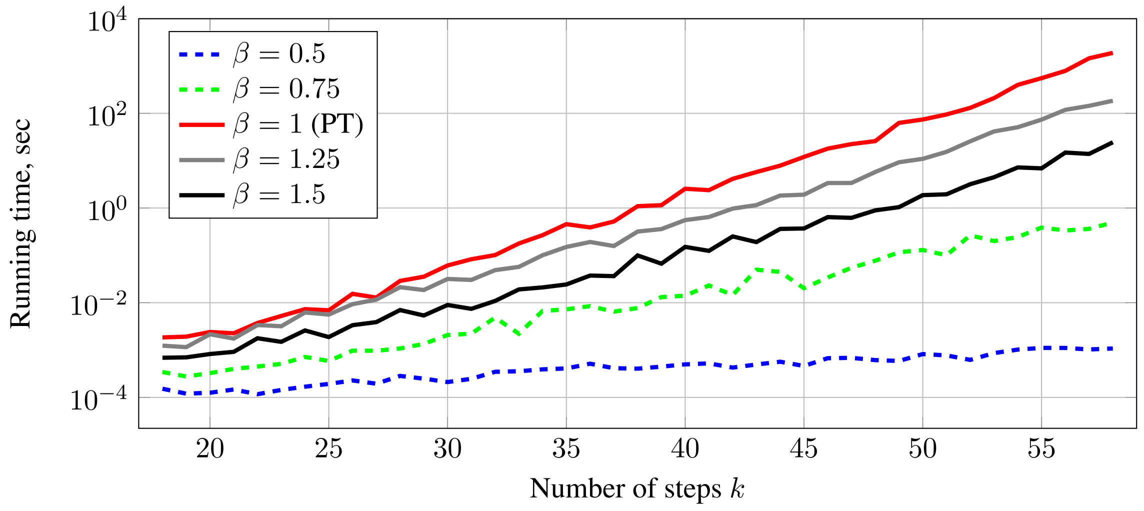


Figure 12: Performance of PBT on instances outside the PT region (obtained by changing parameter β).

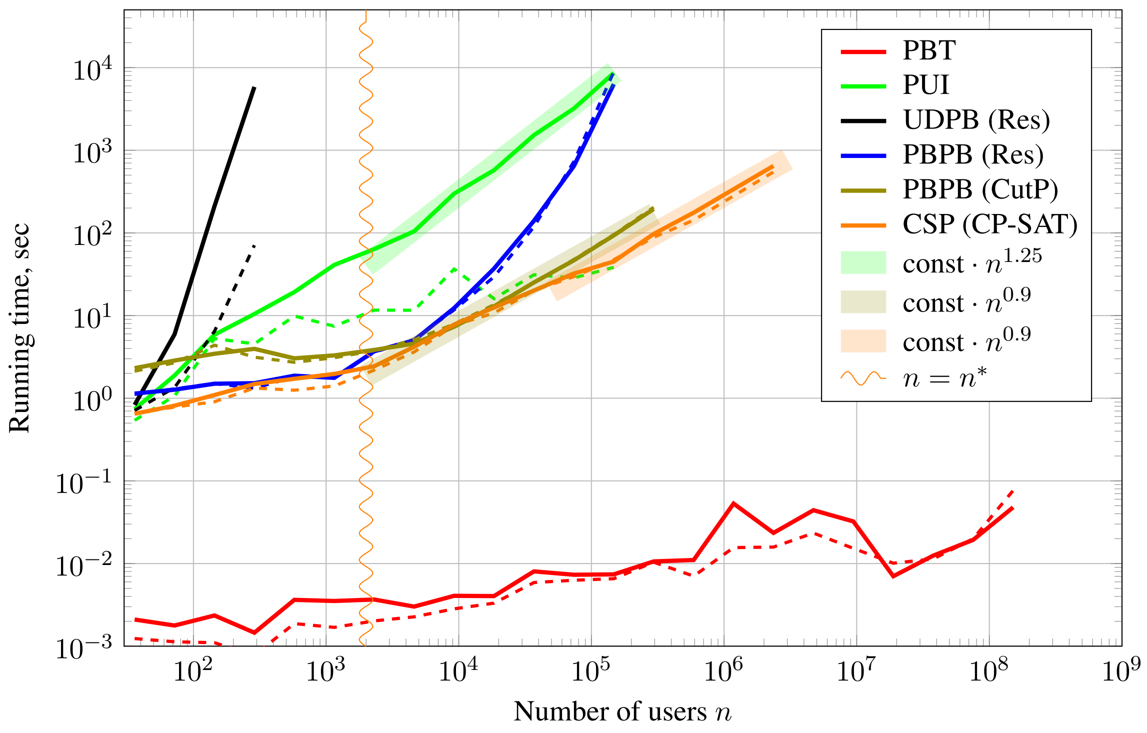
8.3 Performance Comparison: Slice "fixed-k"

All the FPT algorithms discussed in this paper are designed to solve large WSP with relatively small number k of steps. This reflects the fact that in large organisations there might be thousands of users with only tens of steps in an instance. Thus it is of both theoretical and practical importance to evaluate scalability of the approaches with regards to the number of users n.

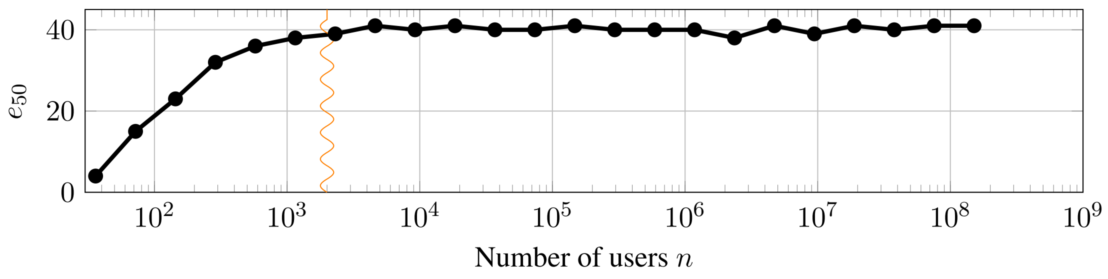
Figure 13 reports on a set of experiments with $WIG(18, n, e_{50}, 18)$ instances. Note that a relatively small value of $k = 18$ was chosen to allow even the slowest algorithms to terminate in reasonable time. The (impractically large) maximum value of n was selected to investigate behaviour of the algorithms.

Figure 13b shows how the value of $e = e_{50}(18, n, 18)$ varies with n so as to remain at the PT region. The corresponding results for the runtimes of the different algorithms are given in Figure 13a. Although all the instances at all values of n are selected from a PT region, it seems that there are two regions above and below a dividing value of $n^* \approx 2000$. For $n < n^*$ the number of not-equals constraints in the graph is increasing with n, but for $n > n^*$ the number of the not-equals constraints in the graph is roughly constant, and the properties of the instances are practically independent of n.

The simplest behaviour to interpret is that of PUI on unsatisfiable instances, which exhibits a scaling that is approximately proportional to $n^{1.25}$. This slightly superlinear scaling is natural as the algorithm works through the users one at a time, and if the instance is highly constrained by the not-equals constraints then the work per user may well become roughly constant, with only a mild accumulation of new patterns and, hence, increase of the patterns pool and associated runtime costs. On satisfiable instances PUI has a potential to solve the problem after $O(k)$ iterations, i.e. with a perfect user ordering heuristic its running time could theoretically be independent of n. However, the real user ordering heuristic does not always pick the "right" users and the running time mildly increases with n, matching our expectations.



(a) Median runtimes for the different solvers, as a function of n. Solid lines show running times for unsat instances and dashed lines for sat instances.



(b) Number e of not-equals constraints as a function of n selected so as to be 50% satisfiable.

Figure 13: The "fixed-k" slice for $k = 18$, i.e. *WIG*$(18, n, e_{50}, 18)$.

The running time of PBT shows little dependency on n. Observe that the upper level of the search algorithm in PBT does not depend on n. Due to the heuristic described in Section 4.2, the size of the assignment graph (the lower level) is bounded by k^2, i.e. does not generally grow with n. Hence, only the generation of the assignment graph depends on n in PBT. However, the larger the value of n, the less likely that full scans of the list of users are needed. As a result, PBT demonstrates sub-linear scaling in this experiment, solving instances with more than 10^8 users in under a second. Even if not directly practical in case of WSP, this result shows that a careful design and implementation of an FPT algorithm has a potential to routinely address very large problems.

In contrast to the experiments along the "vary-k" slice, the PBPB (Res) scaling on the "fixed-k" slice is not similar to that of PBT, and is possibly worse than polynomial (as the slope increases on the log-log plot). On the other hand, the PBPB (CutP) and CSP (CP-SAT), both show very good performance, demonstrating sub-linear scaling (roughly $n^{0.9}$), and outperforming PBPB (Res) on large n's. For PBPB (CutP), it is natural to hypothesise that this is because at large n the matching problem becomes even more important, making the cutting planes proof system more suitable. For CSP (CP-SAT), we can hypothesise that, unlike the SAT4J solver, the OR Tools SAT solver efficiently discovers strategies similar to the one proposed in Proposition 6.2, and thus solves the problem in FPT time. This behaviour is interesting, and good news for SAT solvers, but requires future investigation as there might be potential for significant improvement to enhance the solver's performance using the fact that the problem is FPT.

We note here that the off-the-shelf solvers (PBPB (Res), PBPB (CutP) and CSP (CP-SAT) all compete with each other in the 'vary-k' and 'fixed-k' slices (see also Appendix D, see (Karapetyan et al., 2019) for the $n = 100k$ slice), however none of them is a universal winner. It is clear though that, as expected, the resolution proof system performs better than the cutting planes resolution system on the 'vary-k' slice.

9. Conclusion

In this paper, we studied the Workflow Satisfiability Problem (WSP), with User Independent (UI) constraints, which admits FPT algorithms. Employing a new view of the FPT nature of the problem and the AI methods, we designed a new FPT algorithm, PBT, that significantly outperforms the previous state-of-the-art algorithm, extending the size of instances that can be reasonably tackled from $k \approx 20$ to $k \approx 50$. This is partly due to the fact that PBT is the first FPT algorithm for WSP with UI constraints that has polynomial space complexity. We believe that an important lesson is that having found an FPT algorithm for a problem should be just a starting point for designing a practical algorithm as there are still likely to be many opportunities for significant improvement via the repertoire of intelligent heuristic search mechanisms.

The direct study of algorithms for the WSP was also complemented with a study of PTs arising from a generator of WSP instances. We found strong evidence of PT phenomena in the same fashion as previous extensive studies within graph theory and AI.

While typical PT studies deal with a single size parameter, FPT implies that there are two 'size parameters', both playing important role in the complexity of the problem. This paper gives a novel combination of studying FPT and PT and the use of multiple 'slices' of the size space for a thorough empirical study of scaling of algorithms.

A common approach to solving decision problems in practical computing is to formulate them using a general-purpose declarative language and then using off-the-shelf solvers. Naturally, one may be interested in representations, with which off-the-shelf general solvers result in similar scaling to the direct implementations of FPT algorithms. We did observe such good behaviour of our new PBPB formulation solved by SAT4J (in resolution proof system mode) on the "vary-k" slice, in which the scaling was roughly as good as the PBT solver, and a lot better than that of the previous solvers. A very similar performance was observed from a CSP solver even though our CSP formulation did not explicitly exploit our understanding of the FPT nature of the problem. We expect that the internal mechanism of handling our CSP formulation by CP-SAT in fact leads to an internal search process similar to pattern-based methods of PBPB and the overall winner PBT.

Although the running time of the general-purpose solvers was by one or two orders of magnitude worse than that of PBT, as might be expected, it indicated that the solvers were able to determine a good search strategy. Furthermore, one may expect some additional efficiencies of the general purpose solvers when applied to real instances, as an off-the-shelf solver is more likely to be able to exploit associated structures.

The behaviour of these off-the-shelf solvers was not as good in the "fixed-k" slice experiments compared to PBT. The PBPB (Res) solver demonstrated, apparently, exponential scaling despite us showing in Proposition 6.1 that a simple tree-based solver is capable of exhibiting FPT time, i.e. polynomial scaling in "fixed-k" slice. Also, the memory usage by the general-purpose solver was a bottleneck; e.g. CSP (CP-SAT) consumed around 1.5-2 GB of RAM at $k = 18$ and $n \approx 2.4 \cdot 10^6$. In comparison, for such a problem, PBT takes around 1 MB of RAM on top of the size of instance data itself (about 20 MB). This indicates that there is a good potential for improvement of current off-the-shelf solvers to enable them to take advantage of FPT properties.

9.1 Future Directions

A natural direction for future research is to further improve the performance of the PBT algorithm. One can investigate improving the pruning from the authorisations by adding extra lookahead; further improving the branching heuristics (possibly by exploiting machine learning for adaptive search) and also finding branch ordering selections that give a net gain on the satisfiable instances. A particularly important direction arises from noting that PB solvers on the PBPB encoding are likely to be benefiting from learning and storing of no-goods (entailed constraints). It would be interesting to consider how PBT could be enhanced with such no-good learning, and in such a way that is compatible with FPT – enhancing the FPT-driven two-layer nature of the solution, rather than breaking it. The PT properties of the set of generated instances should also be mapped in more detail, along with consideration of a wider range of UI constraints. Such an enhanced study of the PT properties should be also exploited for further evaluation of proposed algorithms.

Although the combination of PBPB and SAT4J in the resolution proof system mode worked well, there was evidence from the "fixed-k" that in some regions of the space of instances it was performing less well. In particular, scaling with n seemed to be non-polynomial, and this deserves further investigation. Possibly, general PB solution methods need different branching heuristics, or could be extended to better exploit the matching problem (equivalently, list colouring of a clique) that arises as a vital sub-problem when using the PBPB formulation. We emphasise that here we do not attempt to systematically cover 'all encodings', but to take some representative and natural ones and study their behaviour from an FPT perspective. However, we believe the methodology and initial results give a foundation for a future systematic study of combinations of encodings and solvers.

We propose this study as a contribution towards ensuring that general purpose solvers are appropriately effective on FPT problems; and give an example of developing a formulation that enables solvers to exploit the inherent FPT properties of the problem. A future challenge in AI may be to study how a general-purpose solver can automatically discover such formulations.

An important outcome is that the combination of decomposition and FPT ideas leads to new highly-effective algorithm, and then combining FPT with PT ideas give a powerful framework for empirical study. Our PT study of WSP also revealed interesting challenges in empirical average time complexity studies for FPT problems. We proposed using multiple slices through the size space while adjusting other parameters of the instance generator to stay in the PT region. However, it is still an open question how to best select these slices, or indeed how to do a more integrated study of the effects of the multiple parameters. Considering the general interest of the AI community in analysis of a widening range of complexity classes (e.g. (Bailey et al., 2007)) (including studies of FPT problems, see e.g. (De Haan et al., 2015; Kronegger et al., 2013)) and understanding of practical implications of these complexity classes, we believe that further development of the study of interactions between FPT and PT offers the potential for deeper insight into computational challenges arising in AI.

Acknowledgements We would like to thank the reviewers for the valuable comments that helped us to improve the paper. This research was partially supported by EPSRC grants EP/H000968/1 (for DK and AP), EP/K005162/1 (for AG and GG), and a Leverhulme Trust grant (GG). The source codes of the instance generator and PBT, as well as the test instances, solutions, translation routines and the experimental data, are publicly available (Karapetyan, 2019).

References

American National Standards Institute (2004). American national standard for information technology – Role Based Access Control (ANSI INCITS 359-2004)..

Bailey, D. D., Dalmau, V., & Kolaitis, P. G. (2007). Phase transitions of PP-complete satisfiability problems. *Discrete Applied Mathematics*, *155*(12), 1627 – 1639.

Basin, D., Burri, S. J., & Karjoth, G. (2012). Optimal workflow-aware authorizations. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT'12, pp. 93–102, New York, NY, USA. ACM.

Basin, D., Burri, S. J., & Karjoth, G. (2014). Obstruction-free authorization enforcement: Aligning security and business objectives. *Journal of Computer Security*, *22*(5), 661–698.

Bertino, E., Ferrari, E., & Atluri, V. (1999). The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, *2*(1), 65–104.

Bertolissi, C., dos Santos, D. R., & Ranise, S. (2018). Solving multi-objective workflow satisfiability problems with optimization modulo theories techniques. In *Proceedings of the 23d ACM Symposium on Access Control Models and Technologies*, SACMAT '18, pp. 117–128, New York, NY, USA. ACM.

Bollobas, B. (1985). *Random Graphs*. Academic Press, London, England.

Boros, E., & Hammer, P. L. (2002). Pseudo-boolean optimization. *Discrete Applied Mathematics*, *123*(1-3), 155–225.

Cameron, P. J. (1994). *Combinatorics: Topics, Techniques, Algorithms*. Cambridge University Press.

Cheeseman, P., Kanefsky, B., & Taylor, W. M. (1991). Where the Really Hard Problems Are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, IJCAI'91, pp. 331–337.

Cohen, D., Crampton, J., Gagarin, A., Gutin, G., & Jones, M. (2014). Iterative plan construction for the workflow satisfiability problem. *Journal of Artificial Intelligence Research*, *51*, 555–577.

Cohen, D., Crampton, J., Gagarin, A., Gutin, G., & Jones, M. (2016). Algorithms for the workflow satisfiability problem engineered for counting constraints. *Journal of Combinatorial Optimization*, *32*(1), 3–24.

Cook, W., Coullard, C. R., & Turán, G. (1987). On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, *18*(1), 25–38.

Crampton, J. (2005). A reference monitor for workflow systems with constrained task execution. In Ferrari, E., & Ahn, G.-J. (Eds.), *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, SACMAT'05, pp. 38–47. ACM.

Crampton, J., Gutin, G., & Karapetyan, D. (2015). Valued workflow satisfiability problem. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, SACMAT'15, pp. 3–13.

Crampton, J., Gutin, G., & Yeo, A. (2013). On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Transactions on Information and System Security*, *16*(1), 4:1–4:31.

Crampton, J., Gutin, G. Z., Karapetyan, D., & Watrigant, R. (2017a). The bi-objective workflow satisfiability problem and workflow resiliency. *Journal of Computer Security*, *25*(1), 83–115.

Crampton, J., Gutin, G. Z., & Watrigant, R. (2017b). On the satisfiability of workflows with release points. In Bertino, E., Sandhu, R., & Weippl, E. R. (Eds.), *Proceedings of the 22nd ACM Symposium on Access Control Models and Technologies,* SACMAT'17, pp. 207–217. ACM.

Crawford, J. M., Ginsberg, M. L., Luks, E. M., & Roy, A. (1996). Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, KR'96, pp. 148–159, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Culberson, J., & Gent, I. (2001). Frozen development in graph coloring. *Theoretical Computer Science*, *265*(1-2), 227–264.

Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, *5*(7), 394–397.

De Haan, R., Kronegger, M., & Pfandler, A. (2015). Fixed-parameter tractable reductions to SAT for planning. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pp. 2897–2903. AAAI Press.

Dixon, H. E., Ginsberg, M. L., & Parkes, A. J. (2004). Generalizing boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, *21*, 193–243.

dos Santos, D. R., Ranise, S., Compagna, L., & Ponta, S. E. (2017). Automatically finding execution scenarios to deploy security-sensitive workflows. *Journal of Computer Security*, *25*(3), 255–282.

dos Santos, D. R., Ranise, S., Compagna, L., & Ponta, S. E. (2015). Assisting the deployment of security-sensitive workflows by finding execution scenarios. In Samarati, P. (Ed.), *Data and Applications Security and Privacy XXIX, Proceedings of DBSec 2015*, Vol. 9149 of *Lecture Notes in Computer Science*, pp. 85–100. Springer.

Downey, R., & Fellows, M. R. (2013). *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer-Verlag, London.

Dukanovic, I., & Rendl, F. (2008). A semidefinite programming-based heuristic for graph coloring. *Discrete Applied Mathematics*, *156*(2), 180 – 189.

Dutton, R. D., & Brigham, R. C. (1981). A new graph colouring algorithm. *The Computer Journal*, *24*(1), 85–86.

Fellows, M. R., Friedrich, T., Hermelin, D., Narodytska, N., & Rosamond, F. A. (2011). Constraint satisfaction problems: Convexity makes AllDifferent constraints tractable. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, IJCAI'11, pp. 522–527.

Gent, I. P., MacIntyre, E., Prosser, P., & Walsh, T. (1996). The constrainedness of search. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference,* Vol. 1, AAAI/IAAI'96, pp. 246–252.

Gutin, G., Kratsch, S., & Wahlström, M. (2015). Polynomial kernels and user reductions for the workflow satisfiability problem. *Algorithmica*, *75*(2), 383–402.

Gutin, G., & Wahlström, M. (2016). Tight lower bounds for the workflow satisfiability problem based on the strong exponential time hypothesis. *Information Processing Letters*, *116*(3), 223–226.

Haken, A. (1985). The intractability of resolution. *Theoretical Computer Science*, *39*, 297 – 308.

Huberman, B. A., & Hogg, T. (1987). Phase transitions in artificial intelligence systems. *Artificial Intelligence*, *33*(2), 155–171.

Impagliazzo, R., & Paturi, R. (2001). On the complexity of k-SAT. *Journal of Computer and System Sciences*, *62*(2), 367–375.

Karapetyan, D. (2019). Source codes of the Pattern Backtracking algorithm, the instance generator, the instances used in the paper, corresponding solutions and the converter of instances into PBPB, UDPB and CSP formulations.. `https://dx.doi.org/10.5526/ERDR-00000114`, retrieved 22 July 2019.

Karapetyan, D., Gagarin, A., & Gutin, G. (2015). Pattern backtracking algorithm for the workflow satisfiability problem with user-independent constraints. In Wang, J., & Yap, C. (Eds.), *Frontiers in Algorithmics,* FAW 2015, Vol. 9130 of *Lecture Notes in Computer Science*, pp. 138–149. Springer.

Karapetyan, D., Parkes, A. J., Gutin, G., & Gagarin, A. (2019). Pattern-based approach to the workflow satisfiability problem with user-independent constraints. *CoRR*, *abs/1604.05636v3*.

Kronegger, M., Pfandler, A., & Pichler, R. (2013). Parameterized complexity of optimal planning: A detailed map. In *Proceedings of the 23d International Joint Conference on Artificial Intelligence*, IJCAI'13, pp. 954–961.

Le Berre, D., & Parrain, A. (2010). The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*, 59–64.

Lovász, L. (1979). On the Shannon capacity of a graph. *IEEE Transactions on Information Theory*, *25*(1), 1–7.

Mézard, M., Mora, T., & Zecchina, R. (2005). Clustering of solutions in the random satisfiability problem. *Physical Review Letters*, *94*(19), 197205:1–197205:4.

Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and easy distributions of SAT problems. In Rosenbloom, P., & Szolovits, P. (Eds.), *Proceedings of the 10th National Conference on Artificial Intelligence*, AAAI'92, pp. 459–465, Menlo Park, California. AAAI Press.

Ordyniak, S., & Szeider, S. (2013). Parameterized complexity results for exact bayesian network structure learning. *Journal of Artificial Intelligence Research*, *46*, 263–302.

Razborov, A. A. (2002). Proof complexity of pigeonhole principles. In Kuich, W., Rozenberg, G., & Salomaa, A. (Eds.), *Developments in Language Theory,* DLT 2001, Vol. 2295 of *Lecture Notes in Computer Science*, pp. 100–116, Berlin, Heidelberg. Springer.

Roy, A., Sural, S., Majumdar, A. K., Vaidya, J., & Atluri, V. (2015). Minimizing organizational user requirement while meeting security constraints. *ACM Transactions on Management Information Systems*, *6*(3), 12:1–12:25.

Selman, B., & Kirkpatrick, S. (1996). Critical behavior in the computational cost of satisfiability testing. *Artificial Intelligence*, *81*(1-2), 273–295.

Stuckey, P. J. (2018). MiniZinc Challenge 2018.. `https://www.minizinc.org/challenge2018/challenge_results2018.pdf`, retrieved 20 July 2019.

Wang, Q., & Li, N. (2010). Satisfiability and resiliency in workflow authorization systems. *ACM Transactions on Information and System Security*, *13*(4), 40:1–40:35.

West, D. B. (2001). *Introduction to Graph Theory* (2nd edition). Prentice-Hall.