Pattern-Based Approach to the Workflow Satisfiability Problem with User-Independent Constraints[1]

D. Karapetyan[a,b,*], A. J. Parkes[b], G. Gutin[c], A. Gagarin[d]

[a] *University of Essex, Colchester, UK*
[b] *University of Nottingham, Nottingham, UK*
[c] *Royal Holloway, University of London, Egham, UK*
[d] *Cardiff University, Cardiff, 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) 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 constraints (UI), 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 as they significantly extend the size of the problem that can be routinely solved.

We give a new view of the FPT nature of the WSP with UI constraints, showing that it decomposes 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 it also has only polynomial space complexity whereas the old algorithm takes memory exponential in k, which limits its application.

We also provide a new pseudo-boolean (PB) formulation of the WSP with UI constraints which exploits this new decomposition of the problem into two levels. Our experiments show that efficiency of solving this new PB formulation of the problem by a general purpose PB solver can be close to the bespoke FPT algorithm, which raises the potential of using general purpose solvers to tackle FPT problems efficiently. In this context, the WSP with UI constraints can be also considered as an extension of the hypergraph list colouring problem.

We also study the computational performance of various algorithms to complement the overly-pessimistic worst-case analysis that is usually done in FPT studies. To support this we extend studies of phase transition (PT) phenomena in the understanding of the average computational effort needed to solve decision problems. We investigate, for the first time, the phase transition properties of the WSP, under a model for generation of random instances, and show how the methods of the phase transition studies can be adjusted to FPT problems.

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

1. Introduction

An ongoing computational challenge within Artificial Intelligence (AI) has been the combinatorial explosion. In response, AI has developed, and continues to develop, many powerful techniques to address this challenge. One technique, originating from theoretical computer science but currently becoming

[1] A preliminary version of some portions of this paper was published in the proceedings of FAW 2015 [31].
[*] Corresponding author

Email addresses: dkarap@essex.ac.uk (D. Karapetyan), andrew.parkes@nottingham.ac.uk (A. J. Parkes), g.gutin@rhul.ac.uk (G. Gutin), gagarina@cardiff.ac.uk (A. Gagarin)

popular in AI, is that of the theory of fixed parameter tractable (FPT) algorithms, which is concerned with parametrisation of hard problems that can reveal new ways in which they are tractable (in the FPT sense). In this paper, we are linking together AI and FPT results and applying 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. [4, 16, 42, 44]. 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 – there is no 'flow' but the challenge is to plan by making feasible assignments for all the steps.) The WSP has important applications and has been extensively studied in the security research community [4, 6, 14, 15, 44].

However, the WSP is NP-complete, and despite the practical importance of the problem, it has been difficult to solve even some moderate-sized instances [11, 42]. 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, meaning 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 small parameter k is the number of steps as it was observed that in real instances the number k of steps is usually much smaller than the number n of users [44].

It has been shown [12] that the WSP is FPT if the set of constraints is restricted to the so-called *user-independent* (UI) constraints, i.e. constraints whose satisfaction does not depend on specific user identities. Then existing methods [11] achieve a runtime that is polynomial in n, and exponential only in k. Since in practice, we can generally assume, and exploit, that $k \ll n$, this is a very significant improvement over direct search.

(Constraints described in the WSP literature are relatively simple, so it was assumed in [12] that every constraint can be checked in polynomial time. We will use the same assumption in this paper. We will also assume that an authorisation test $s \in A(u)$ takes constant time, as we store authorisation lists in memory.)

This paper has three major, interacting components:

1. A highly effective procedural method PBT based on a two level decomposition of the problem, and a specialised backtracking search supported by heuristics and pruning.

2. A declarative method from a pseudo-boolean [8, 34] formulation 'PBPB' which also exploits the two level decomposition of the problem.

3. Experimental studies of the algorithm performances, but focussing on average case complexity, and supported by Phase Transition (PT) phenomena [7, 28, 10, 37] 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 that we call *patterns*. Patterns capture the decisions concerned with UI constraints but generally do not fix user assignment. 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: upper level corresponding to UI constraints, where the decisions can be encoded with a pattern, and lower level corresponding to user assignment. 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 polynomial algorithms. Thus, PBT has running time exponential in k only.

We will show that PBT is not only FPT but also has polynomial space usage.[2] 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 thousands, and so the problem is much larger than the parameter k would suggest. This is a significant achievement for an NP-complete problem, and also can be expected to be sufficient for practical-sized WSP instances.

In addition to the new efficient FPT algorithm PBT, we provide a new Pseudo-Boolean (PB) formulation 'PBPB' of the problem. Existing PB or integer programming encodings of the WSP with UI constraints [11, 44] 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). Since we deal with user-independent constraints, it is clearly not suitable to express solution methods for this type of problem in terms of direct branching only on these x-variables as they explicitly depend on the users. Therefore, in the PBPB formulation of the problem, we also use a set of M-variables:

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

Setting $M_{ij} = 1$ or $M_{ij} = 0$ roughly corresponds to how the patterns are used in PBT. 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 – 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 [35], e.g. see [22], where they are referred to as the 'colouring matrix'. By using a generic PB solver, SAT4J [34], we show that 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. This substantial progress in comparison to UDPB can be explained by the ability of PBPB to exploit the FPT nature of the problem.

Another interesting observation is that the WSP is basically the constraint satisfaction problem (CSP), where for each variable s (called a step in WSP terminology) we have an arbitrary unary constraint (called an authorisation) that assigns possible values (called users) for s. (It is important to remember that for the WSP we do not use the term 'constraint' for authorisations. So, when we define special types of constraints, we do not extend these types to authorisations, which remain arbitrary.) However, the approach in WSP is different: in CSP, the number of variables is usually 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 application of such CSPs. However, there are only a few studies that consider relevant cases. One of the most closely related ones is [25] 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 [25] 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.

Regarding the computational complexity, usually, FPT problems and algorithms have been studied from the perspective of worst case analysis. This is appropriate for initial studies, however, it is well-known that worst-case analysis can often be over-pessimistic (and sometimes wildly so). Hence, in terms of the study of the potential practical usages of proposed algorithms, it is vital to perform some study of the performance averaged over different instances. One approach to doing this is of course to set up a benchmark suite of real 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, we use a generator of artificial instances. However, in this case, we need to have a systematic way to decide on the parameters used in the generation of instances, and in a fashion that gives the best chance of obtaining a meaningful and reliable insight into the behaviour of different FPT algorithms.

[2] Note that, the complexity class FPT does not in itself directly restrict the space usage; the previous algorithms had a space usage that was also exponential in k, and this restricted their application to (roughly) $k \leq 20$.

With these motivations, in the second half of the paper we study the WSP from the perspective of phase transition or threshold phenomena. It has long been known that complex systems can exhibit threshold phenomena, see e.g. [7, 28, 10, 37, 43, 38, 1, 36, 24]. They are characterised by a sharp change, or phase transition (PT), in the properties of problem instances when a parameter in the instance generation is changed. An important discovery was that such thresholds are also associated with decision problems that are the most challenging for search: The PT is the source of hard instances of the associated decision problem. In the context of NP-complete problems, this means the average time complexity $t(n)$ is exponential 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, and so be substantially better, possibly even polynomial, for sufficiently under-constrained instances.

Testing on instances without 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 scaling within the phase transition region. (We also argue later that real instances are likely to be close the PT region.) In order to do this we will show how the generator we use for the WSP instances leads to behaviour that is expected from a phase transition. We will also show that empirical average case analysis of FPT algorithms is more difficult than a typical PT study, since FPT problems have not just one but several size parameters. Then a good study has to consider different regions of this multi-dimensional size space. To the best of our knowledge this 'empirical average case' has not previously been systematically organised for the case of FPT studies.

1.1. Contributions and highlights

The contributions of this paper are hence:

- A clear split of WSP with UI constraints into two levels by means of patterns, and a pattern-based search method, PBT, that effectively exploits this two-level split. Also an efficient implementation of PBT[3], with evidence that it successfully tackles much larger problems than previously possible. Moreover, it is the first WSP algorithm that is not only FPT but runs in polynomial space.
- A new Pseudo-Boolean (PB) encoding, PBPB, (Section 5), that captures the key ideas of the PBT method by introduction of auxiliary variables. This raises a new direction of research concerned with formulations of FPT problems that can be solved by general purpose solvers in FPT time and, thus, giving a new approach to prototyping or even developing solution methods for such problems.
- A study of the phase transition properties of a generator of WSP instances, giving evidence of it, and studying some standard properties expected from a phase transition. We also discuss how a standard methodology to predict analytically the point of the phase transition needs to be adjusted because of the FPT nature of the problem.
- An FPT-aware phase transition study of the empirical properties of the algorithms:
  - An approach to empirical average case study of FPT problems, based on the standard PT study adjusted to the multi-dimensional size space of an FPT problem. We suggest to measure scaling of algorithms along one-dimensional 'slices' through the size space. We raise and leave open an associated question of selecting other appropriate slices.
  - Empirical evidence that the scaling of the average runtime is indeed exponential only in k, consistent with the expectations from the worst-case FPT analysis.
  - Empirical evidence that away from the PT region, the complexity drops rapidly. This matches the usual "easy-hard-easy" pattern expected in phase transitions [10].

A contributions of this work is to show that the WSP problem with UI constraints is FPT due to a particular 'layer structure', and that this structure is compatible with, and complementary to, exploitation by standard AI search techniques, as well as logical representational schemes such as PB. Hence, enabling a mix of AI and FPT algorithmic and analysis techniques may give another tool to combat the combinatorial explosion challenges that arise within AI.

[3]The PBT implementation presented in this paper is a significant improvement over the algorithm introduced in the preliminary report [31]. The new version still fundamentally exploits the upper/lower decomposition and pattern based search (introduced in [31]) but employs numerous algorithmic enhancements demonstrating better worst-case time and space complexity, and is faster by about an order of magnitude according to our computational experiments.

1.2. Structure of the paper

Section 2 gives the needed background on the theory of FPT, and the WSP. Section 3 describes the notions of patterns central for the discussed algorithms. Section 4 gives the new PBT algorithm in detail and demonstrates that it is FPT. Section 5 gives the new PB formulation PBPB of the WSP. (Associated with this, Appendix A gives details of encoding of various UI constraints.) Section 6 gives a descriptive comparison of the workings of the PBT algorithm, the existing algorithm [11] which we call here Pattern User-Iterative (PUI), and the PBPB encoding.

Section 7 introduces the instance generator that we use for the experiments and shows how there is a phase transition, that has the expected properties of a standard phase transition. Some more technical aspects related to the PT are given in two appendices: Appendix B gives an analytical computation to estimate the location of the phase transition; Appendix C gives empirical evidence of freezing of variables around the PT region, i.e. of variables whose values are forced by the instance – a phenomenon expected from PT. Section 8 gives 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.

Finally, in Section 9 we discuss overall conclusions and potential for future work.

2. Background

In this section, and to help the paper be reasonably self-contained, we provide the needed background, and discussion of existing related work, for the topics of FPT and WSP.

2.1. The complexity class: Fixed Parameter Tractable (FPT)

The key idea is that there is a parameter k and once the value of k is fixed then the complexity becomes 'easy' and in particular fixed order polynomial. There are various equivalent definitions of what it means for a problem to be FPT, but a standard one is that it can be solved in time $f(k)n^{O(1)}$ for some computable function $f(k)$. The FPT literature also uses the notation "O*" to denote a version of big-Oh that suppresses polynomial factors (in the same fashion that the $\tilde{O}$ notation suppresses logarithmic factors), and so can write $f(k)n^{O(1)}$ as $O^*(f(k))$. Observe that the exponent of the n is a constant not dependent on k, and so the complexity is polynomial in n. There is a general expectation that a problem admitting an FPT algorithm is "easy" as very high-order polynomials are not so likely to occur. However, it is important to note that for a problem to be in FPT, it is not enough to just require that "at any fixed k the problem is polynomial", because this would also allow high order polynomials which would not be practical. (In fact, the associated class of problems solvable in time $O(n^{g(k)})$, where $g(k)$ is an arbitrary computable function of k only, is a larger class called **XP**.)

2.2. The WSP

In the WSP, we are given a set U of *users*, a set S of *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 disregarded).

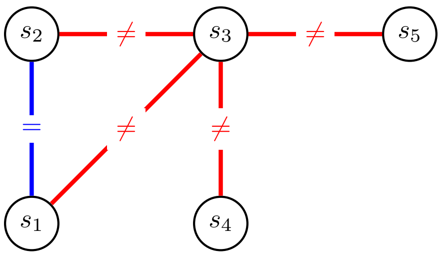
If $W = (S, U, \mathcal{A}, C)$ is the *workflow* and $T \subseteq S$ is a set of steps then we say that a function $\pi : T \to U$ is a *plan*. A plan is called

- *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 *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.

Consider an example of a WSP instance with UI constraints on Figure 1. In this particular instance it 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:

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



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 $n = 8$. 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.

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

$$\pi(s_1) = u_3,\ \pi(s_2) = u_3,\ \pi(s_3) = u_7,\ \pi(s_4) = u_2 : \text{ authorised, eligible, i.e. valid.} \tag{4}$$

The existence of a valid 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 [44]. 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 [44] introduced its parameterisation[4] by k. Algorithms for this parameterised problem were also studied in [12, 11, 16]. While in general the WSP is W[1]-hard [44] (this means the WSP, in general, is very unlikely to be FPT as it is widely believed that FPT$\neq$W[1] (similar to P$\neq$NP) [21]), the WSP restricted[5] to some practically important families of constraints is FPT [12, 16, 42, 44].

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 [2] are UI. Most of the constraints studied in [11, 16, 44] 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 can state that at least/at most/exactly r users are required to complete some sensitive set of steps (these constraints belong to the family of *counting* constraints), where r is usually small. Such constraints are called *at least-r*, *at most-r*, and *exactly-r*, respectively.

2.3. Analogy with Graph Colouring

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.

[4]We use terminology of the recent monograph [20] on parameterised algorithms and complexity.
[5]While we consider special families of constraints, we do not restrict authorisation lists.

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 $s, t \in T$.[6]

To handle equivalence classes of plans, we introduce a notion of pattern. Let a *pattern* $\mathcal{P}$ (on T) be a partition of T into non-empty sets called *blocks*.[7] A pattern prescribes groups (blocks) of steps to be assigned the same user, and requiring that steps from different blocks are assigned 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 $\mathcal{P}(\pi) = \mathcal{P}(\pi')$ [12], see Figure 2 for an example.

π_1

Step	User
s_1	u_3
s_2	u_3
s_3	u_7
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_1
s_5	u_1

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

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, a block is a set of vertices which are required to 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 [23] (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 still exploit similar approach to tackle constraints C which are symmetric in WSP with UI constraints.

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 $B \subseteq A(u)$.*

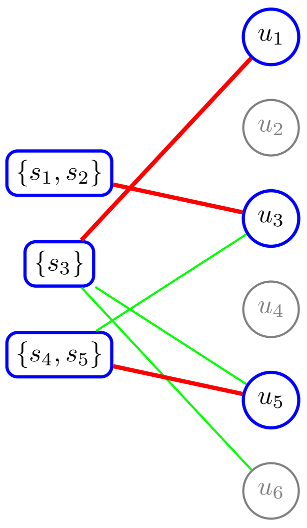
We can now formulate a necessary and sufficient condition for authorisation of a pattern.

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.*

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.

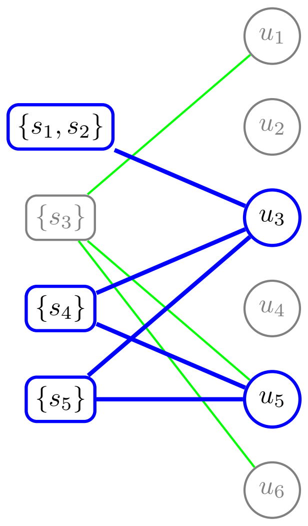
[6]This is a special case of an equivalence relation defined in [12].

[7]Patterns were first introduced in [12]; we follow the definition of patterns from [14].



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

(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.

Depending on the algorithm employed, it takes $O(n^{2.5})$ or $O(n^3)$ time to solve the maximum matching problem, where n is the number of vertices in the graph. However, the assignment graph is usually highly unbalanced since $|\mathcal{P}| \leq k$ and $|U| \gg k$. As the maximum size of M is at most $|\mathcal{P}|$ and the maximum length of an augmenting path[8] in G is $O(k)$, the time complexity of the Hungarian and Hopcroft-Karp methods are $O(k^3)$ and $O(k^{2.5})$, respectively [40]. In other words, finding an authorised plan within an equivalence class is not only polynomial in n but it is actually polynomial in a usually smaller k.

4. The New PBT Algorithm

In this section we first give the core PBT algorithm, and then give some improvements, including branching heuristics that significantly improved the performance.

4.1. Pattern-Backtracking Algorithm (PBT)

We call our new method *Pattern-Backtracking* (PBT) as it uses the backtracking approach to browse 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 there are any valid plans in its equivalence class.

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) significantly speeding up the algorithm.

The basic version of PBT is a 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).

[8]For a matching M in a graph G, a path P is called *M-augmenting* if the edges of P alternate between edges in M and edges not in M and, moreover, the first and last edges of P are not in M. [40].

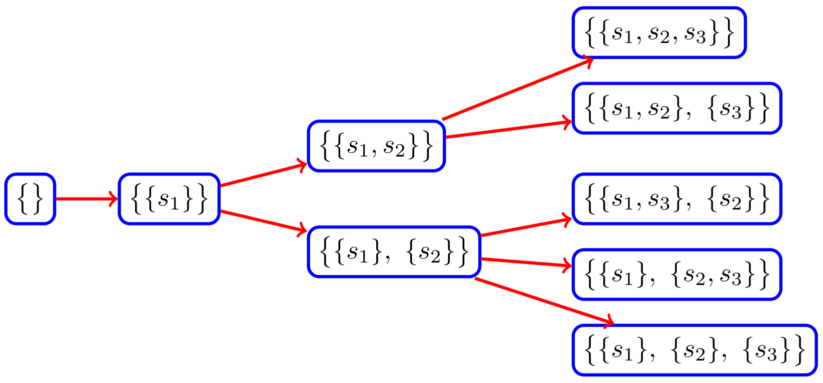

$\{\}$
$\{\{s_1\}\}$
$\{\{s_1, s_2\}\}$
$\{\{s_1\}, \{s_2\}\}$
$\{\{s_1, s_2, s_3\}\}$
$\{\{s_1, s_2\}, \{s_3\}\}$
$\{\{s_1, s_3\}, \{s_2\}\}$
$\{\{s_1\}, \{s_2, s_3\}\}$
$\{\{s_1\}, \{s_2\}, \{s_3\}\}$

Figure 4: Illustration of the backtracking mechanism within PBT.

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);
```

Pruning in the basic version 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, 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 attempts to find a feasible assignment of users and, if such an assignment does not exist, 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.

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 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

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);
```

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 [45]. Thus, we only need to try to find an M-augmenting path in G'. The augmenting algorithm requires time $O(|V(G')| + |E(G')|) = O(kn)$.

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 [5, 11, 42, 44]. 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 counting (i.e. 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 including 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 $T_c \subseteq S(\mathcal{P}')$. For example, an at-most-3 constraint with scope $\{s_1, s_2, \dots, s_5\}$ can be used to prune pattern $\{\{s_1\}, \{s_2\}, \{s_3\}, \{s_4\}\}$. Our implementation of PBT produces extensions 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 measure the potential branching in s for encouraging pruning of the search tree in the context of the current pattern $\mathcal{P}$.

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.
- $\rho_{\geq,\leq}(s) = |\{c \in C_{\leq}, \ c' \in C_{\geq} : \ |T_c \cap T_{c'}| \geq 3\}|$ is the number of pairs of constraints (at-least, at-most) 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 $\rho_{\neq}$, $\rho_{\neq,\leq}(s)$, $\rho{\geq}, {\leq}(s)$ do not depend on the state of the search and, hence, can be precalculated. Also note that $\rho^i_{\leq}(s, \mathcal{P})$ 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 automated parameter tuning. 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 of at-least and at-most constraints. This reflects our empirical observation (also confirmed analytically in Appendix B) that the at-least constraints are usually relatively weak in our instances and rarely help in pruning branches of the search tree.

We conducted direct 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 each level of the search tree reaches its maximum of 5.55 observed at the depth of 23. 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 observed at the depth of 7, from which point it almost monotonically reduces with the depth of the search.

5. Pseudo-Boolean Formulation

Firstly, we provide the old pseudo-Boolean formulation [11] to make the paper self-contained. We then show in Section 5.2 how the old formulation can be extended to exploit the FPT nature of the problem.

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 (recall, see Section 4.3, that our test instances include not-equals, at-least and at-most constraints).

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 $s \in S$.

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

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

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

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

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

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

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

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

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

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

Constraints (6) guarantee that exactly one user is assigned to each step. Constraint (7) implements 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 [22, 23, 35]. [9] 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 [22] 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 also 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.

[9] The matrix M can be thought of as a 'Merge matrix' as it controls whether or not two steps are effectively merged by being required to have the same user.

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

To define authorisations in (23), we still need the $x_{s,u}$ variables, which have 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 are sufficient that a solution of them corresponds to an eligible pattern. Hence (24)–(26) correspond to the upper level of 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 eligible for $C = \{c\}$ plans $\pi : T_c \to U$. 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 illustrate how Proposition 5.1 works, we give the following example. To require that $\mathcal{P} \neq \{\{s_1, s_2\}, \{s_3\}\}$, 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. This might be impractical if $|\overline{PAT}|$ is large. In Appendix A we give more compact approaches to formulate some standard UI constraints and also discuss why our encodings (25) and (26) are correct.

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 gives an asymptotic worst-case analysis of PBT and PUI. Section 6.3 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.

6.1. Branching Strategies

In this section, we use the language of patterns and of the PBPB encoding in order to compare and contrast the PBPB formulation with the new PBT algorithm and the previous FPT algorithm PUI. 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 as shown in Figure 5a and with the requirement that the steps in different blocks are assigned different users.[10] 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. PBT considers steps one at a time, and its branching corresponds to picking a block in the pattern which to extend (or creating a new block). Figure 6 illustrates the branching within PBT in terms of the options for extending the value assignments to the M matrix.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
s_1	**1**	**1**	**0**	**0**	**0**	**?**	**?**	**?**
s_2	**1**	**1**	**0**	**0**	**0**	**?**	**?**	**?**
s_3	**0**	**0**	**1**	**1**	**1**	**?**	**?**	**?**
s_4	**0**	**0**	**1**	**1**	**1**	**?**	**?**	**?**
s_5	**0**	**0**	**1**	**1**	**1**	**?**	**?**	**?**
s_6	**?**	**?**	**?**	**?**	**?**	**1**	**?**	**?**
s_7	**?**	**?**	**?**	**?**	**?**	**?**	**1**	**?**
s_8	**?**	**?**	**?**	**?**	**?**	**?**	**?**	**1**

(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.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
s_1	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
s_2	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
s_3	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
s_4	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
s_5	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
s_6	**0**	**0**	**0**	**0**	**0**	**1**	**?**	**?**
s_7	**0**	**0**	**0**	**0**	**0**	**?**	**1**	**?**
s_8	**0**	**0**	**0**	**0**	**0**	**?**	**?**	**1**

(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.

PUI, the previously state-of-the-art FPT algorithm for the WSP with UI constraints [11], 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 $B \subseteq A(u_i) \setminus S(\mathcal{P})$. The algorithm guarantees that the new pattern is authorised by construction but needs to verify if the new pattern is eligible and, if so, whether it is already present in $\mathscr{P}$; indeed, PUI may generate the same pattern multiple times. Observe that, after completion of an iteration corresponding to user u_i, set $\mathscr{P}$ includes all the eligible patterns

[10] 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.

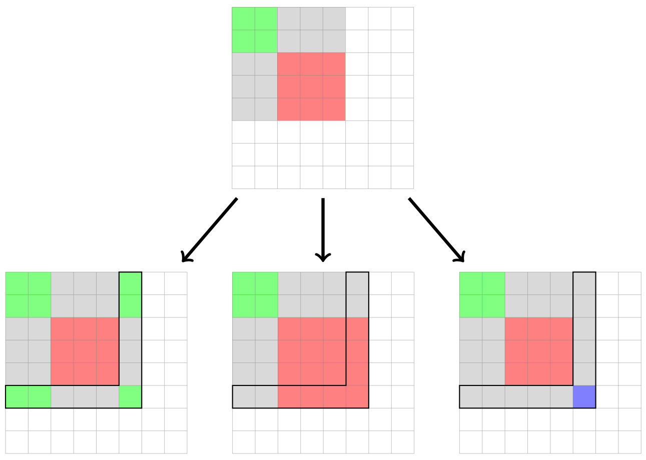


Figure 6: PBT branching. The parent pattern contains two blocks (green of size 2 and red of size 3). PBT extends the parent pattern by assigning a new step in three different ways: (left) extend the first block with the new step; (centre) extend the second block with the new step; (right) create a new 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.

that can be authorised by the already processed users u_1, u_2, …, u_i. PUI proceeds until it finds a valid complete pattern, or all the users $u \in U$ are tried (by which time $\mathscr{P}$ will contain all valid patterns).

Effectively, PUI branches on the x variables and uses patterns only to 'merge' equivalent branches of search. One may observe that the patterns in PUI have a slightly different interpretation compared to the 'open' patterns in PBT. Indeed, a block, once added to a pattern in PUI, will never be extended and in this sense it is 'closed', see Figure 5b.

Hence, the two key differences between PBT and PUI branching are:

Delayed vs. immediate user assignment: PBT implements delayed assignment of users, branching on user-independent M variables. In this sense, PBT is driven by the UI constraints. 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 UI-constraints-driven approach would usually be more effective. Since there is an efficient procedure to find an authorised plan realising a pattern, there is no difference whether we find a valid pattern or a valid plan. However, as we will show in Appendix B, a significant portion of patterns is authorised but usually only a few patterns are eligible; hence branching on M variables is, on average, likely to produce smaller search trees.

Open vs. closed patterns: PBT extends patterns step by step and, hence, allows a block of a pattern to be expanded (open patterns). PUI extends patterns user by user, with each user corresponding to a new pattern block. As the block cannot be expanded during the rest of the iterations, PUI searches in the space of closed patterns. 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.

It should be noted that 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 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 – a result that initially surprised us.

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. In certain cases, when user authorisations are tight, this may lead to superior strategies and hence is a strength of the general purpose solver approach.

6.2. *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.[11] 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, in a sense. Indeed, Gutin and Wahlström [27] proved that unless the Strong Exponential Time Hypothesis [29] fails, which is 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.[12] 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. Let us compare it to the space complexity of PUI, which was the previously state-of-the-art FPT algorithm for WSP with UI constraints [11]. For PUI the space complexity is exponential, $O(\mathcal{B}_{k+1}k)$, as PUI needs to store all the valid patterns produced during the search and the total number of (not necessarily complete) patterns is $\mathcal{B}_{k+1}$.[13]

6.3. *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.

[11] Assuming that validation of all relevant constraints takes $O(k^2)$ time.

[12] See Section 4.2 where the upper bound k on the degrees of blocks in the matching graph is explained.

[13] To count the number of patterns, let us temporarily add an extra element x to the set S of steps. Then there are $\mathcal{B}_{k+1}$ partitions of this extended set. Note that by removing the subset containing x from each of these partitions, we get a (not necessarily complete) unique pattern. Hence, the number of patterns is $\mathcal{B}_{k+1}$.

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.

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

$$\sum_{u \in U} x_{s,u} = 1 \qquad \forall s \in S, \tag{30}$$

$$x_{s,u} = 0 \qquad \forall s \in S \text{ and } \forall u \in U \setminus A^{-1}(s), \tag{31}$$

$$x_{s_1,u} = x_{s_2,u} \qquad \forall s_1 \neq s_2 \in B, B \in \mathcal{P} \text{ and } \forall u \in U, \tag{32}$$

$$x_{s_1,u} + x_{s_2,u} \leq 1 \qquad \forall B_1 \neq B_2 \in \mathcal{P},\ \forall s_1 \in B_1,\ \forall s_2 \in B_2 \text{ and } \forall u \in U, \tag{33}$$

$$x_{s,u} \in \{0,1\} \qquad \forall s \in S \text{ and } \forall u \in U. \tag{34}$$

This is a bipartite matching problem but with blocks of steps being assigned to a user. However, because of (32), when any one step in a block is assigned, some $x_{s,u} = 1$ then all the others in the block are also forced, by propagation, to the same user.

The following proposition shows this 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.[14]

Proposition 6.1. *The PB formulation (30)–(34) 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.*[15]

Proof The PB formulation (30)–(34) 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. □

The above 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. Furthermore, the 'potential' might well be achieved in practice, as branching on the M variables may well be preferred due to the branch-selection heuristics within standard solvers; see the argument at the end of Section 6.1, and also note that in the PBPB formulation each individual M variable tends to occur in more constraints than the individual x variables. Specifically, from the PB formulation each individual x_{su} variable will occur in just $O(k)$ constraints; whereas each individual $M_{s1,s2}$ variable will occur in $O(n)$ constraints. 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 ii is just the verification of a given witness).

[14]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. [41] but in practice are too difficult to control.

[15]The proposition is closely related to known methods in kernelisation [26]; however, due to the lack of space we do not want to use that here.

Proposition 6.2. *When the PB formulation (30)–(34) 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 (32) to limit the users permitted for the representative. Hence the problem (30)–(34) 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 [9] 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 an 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 (30)–(34) 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. [13, 19], it follows there is also a PB proof of (30)–(34). $\square$

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.

7. Instance Generator and Phase Transitions

Due to the difficulty of acquiring real-world WSP instances and to support extensive studies of the scaling of the runtimes, similar to other authors [11, 42, 44] we use the synthetic instance generator described in [11].

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 phase transition (PT), between the satisfiable and unsatisfiable regions. The overall 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. Following [11, 31, 42], we decided to focus this study on at-least-3 and at-most-3 constraints with a scope of 5, $|T_c| = 5$, as this seemed a reasonable constraint that may occur in practice.

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).

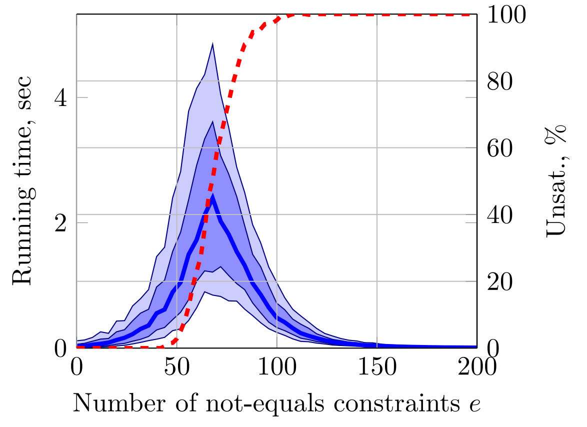
The generator, which we denote as $WIG(k, n, e, \gamma)$, is stochastic, but as usual can be made deterministic by also specifying a value for the random generator seed. For each user $u \in U$, it generates $A(u)$ such that the size of $A(u)$ is first selected uniformly from $\{1, 2, \dots, \lfloor 0.5k \rfloor\}$ 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 also γ at-least-3 constraints uniformly at random.

The PBT algorithm and the test instance generator are available for downloading [30].

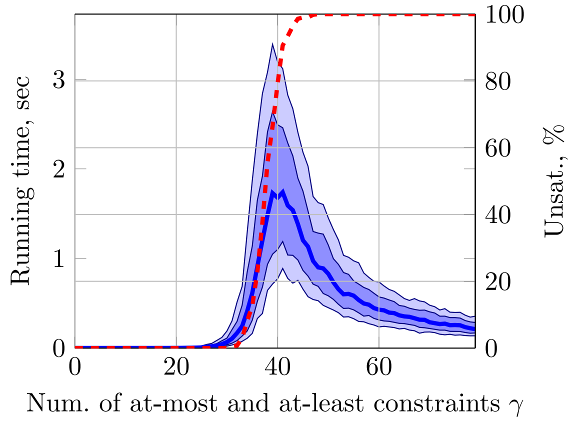
7.2. Thresholds from the Instance Generator

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 such phase transitions.

Figure 7 shows an example, at $k = 40$ and $n = 10k = 400$, 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 valid plans 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: Companies 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.



(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.

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 $\mathcal{WIG}(k, n, e_{50}, \gamma)$ instances satisfiable with 50% probability. The value of $e_{50}(k, n, \gamma)$ is obtained empirically for each k, n and γ.

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 we give an (approximate) computation in the style of an 'annealed estimate' [43] 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. Whether it gives a good estimate depends on the distribution of solutions between 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 with counting all the plans and estimating 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%. To obtain relatively accurate results, we have to estimate the number of eligible patterns and then the probability of a pattern being valid. This will give us the expected number of valid *patterns* which allows us to more accurately estimate the critical point e_{50}.

Lastly, to further support that the observed phenomena have properties expected of a phase transition, we conducted a set of experiments at the critical points and around them. In particular, we show in Appendix C the emergence of *forced variables* similar to [17], i.e. the decision variables with values forced by the instance, in the critical region. We observe that the phase transition 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

In this section, we empirically study the efficiency and scaling behaviour of the new PBT algorithm, and PBPB encoding, and compare with each other and with the existing solvers. Specifically, we compare the following WSP solvers:

PBT The algorithm proposed in this paper;

PUI The FPT algorithm proposed and evaluated in [11];

UDPB (Res) The old pseudo-Boolean SAT formulation of the problem (see Section 5.1) solved with SAT4J [34] 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 (CP) PBPB solved with SAT4J in the cutting planes proof system mode.

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 phase transition WSP instances in our computational study. In a standard (non-FPT) study of phase transitions, 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 phase transition 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.

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

2-d, 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.

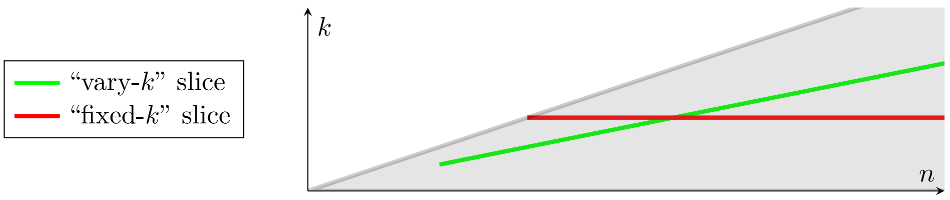


Figure 8: Schematic view of the two "slices" used in our computational study in order to cover the 2-d size space (k, n) used in order to investigate the empirical FPT properties. Such FPT studies should naturally focus on the lower right (grey) region in which k is small compared to n.

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 [11, 31], 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, however the conclusions are very similar.

"fixed-k" Use a constant value of k and simply 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.

The intent of the PT study is that at each selected (k, n) the remaining parameters e and γ of the instance generator will be selected so as to generate instances that have 50% chance of being satisfiable. However, then the constraint space (e, γ) is also two dimensional, and we again need to reduce this space. In this paper, we make the choice that we constrain to $\gamma = k$ – this is somewhat ad hoc, but is designed to be simple, and again we believe it is justified (post hoc) by the results which we will see do indeed give useful insight into the behaviours of the different algorithms. Finally, as mentioned in Section 7, at the given values of (k, n, γ) the value of e is selected to be $e_{50}(k, n, \gamma)$, the value giving a 50% chance of the instance being satisfiable – the actual value of e_{50} is determined empirically. Values of e_{50} obtained and used in our study are freely available to facilitate future work [30].

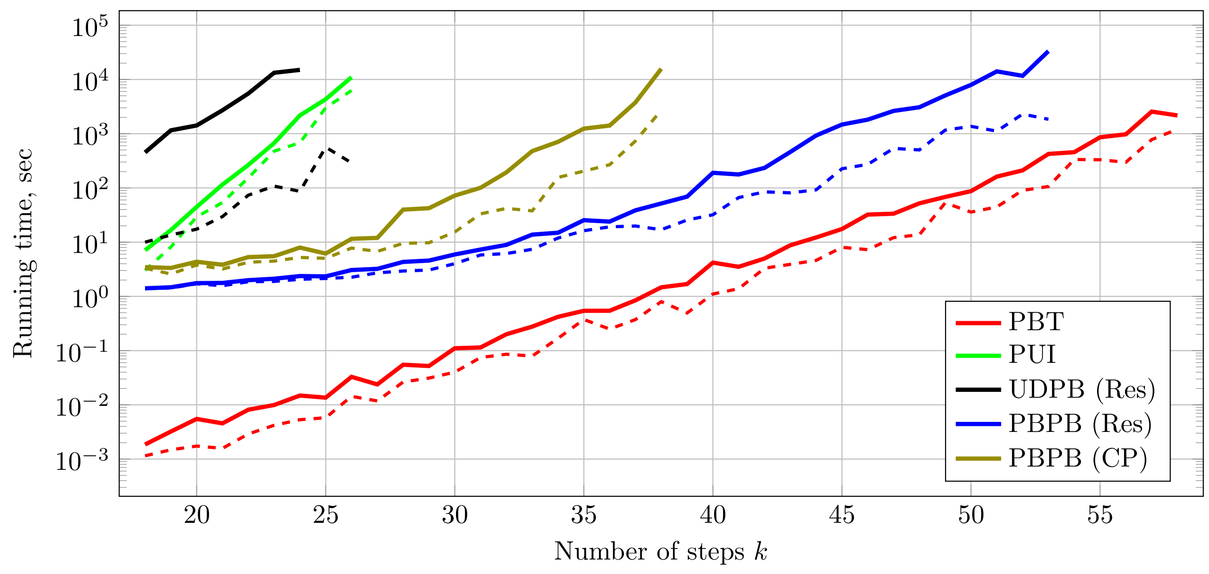
8.2. Performance Comparison: Slice "vary-k"

To compare performance of various WSP algorithms, for each value of k we generated 100 instances using $\mathit{WIG}(k, 10k, e_{50}, k)$. The empirical number of not-equals constraints e_{50} for each k needed to obtain PT instances is shown on Figure 9c.[16] In fact, determining the parameters needed to give the crossover point was a significant computational effort by itself, using multiple runs of PBT – a task that would not have been practical without the improved effectiveness of PBT.

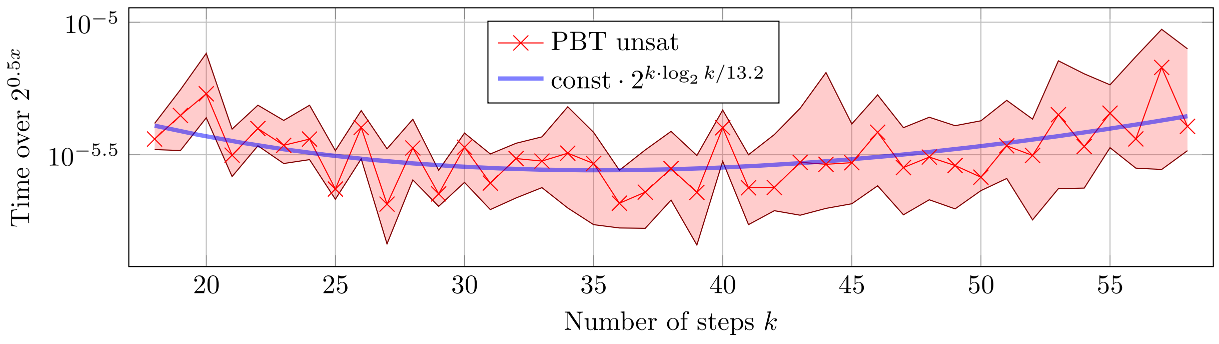
We then simply solved each instance with each of the algorithms and in Figure 9 we report the median running time. We report separate times for the satisfiable and unsatisfiable instances. Note that the unsatisfiable ones are generally harder, and for PBT are not influenced by the heuristics used to select the order in which search tree branches are explored. 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.

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

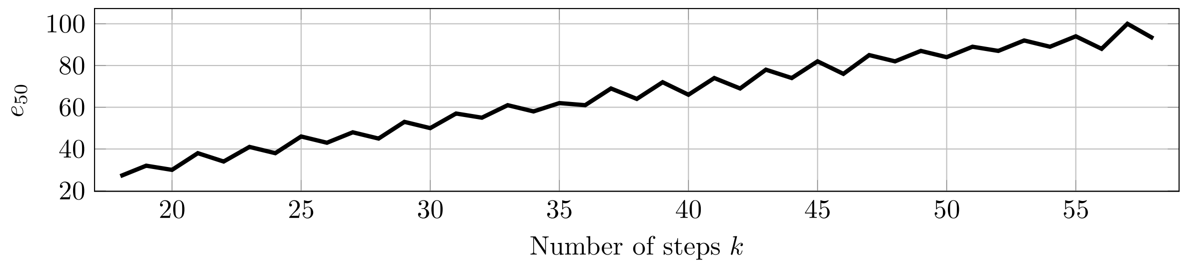
[16] One can observe the 'zig-zag' shape of the curve in that the values corresponding to odd k's are greater than the values corresponding to even k's. This is simply because the size of each authorisation list is randomly drawn by our instance generator from $[1, \lfloor 0.5k - 0.5 \rfloor]$. As a 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.



(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, allowing a better view. Any exponential function would be a straight line in this plot, however empirical running time appears to be curved. The blue line, corresponding to const $\cdot 2^{k \cdot \log_2 k/13.2}$, appears to be a relatively accurate approximation of the running times, demonstrating that scaling of PBT running time closely follows B_k, with a speed-up 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. $WIG(k, 10k, e_{50}, k)$.

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 currently not yet possible, but generally the expectation, based on experience, is that heuristics will improve the coefficients in the exponents but retain the form. Given the form of the scaling from the worst-case analysis in Section 6.2 is hence reasonable to compare the empirical scaling to $2^{(k \log_2 k)/b}$ for some empirically determined constant b. To study this further, we scaled the PBT running time by 2^{ak} to reduce its variation, and plotted it, along with the [35–65] percentile range, on a logarithmic scale, see Figure 9b. In such a plot, any function of the 2^{ak} form would appear as a straight line. One can see that the PBT running time clearly curves up. In fact, a good fit to it is a function $2^{k \cdot \log_2 k/13.2}$, confirming our assumption and indicating the effectiveness of the branching heuristics and pruning. However, note that such heuristics are not in themselves needed for the algorithm to be FPT; this reinforces our point that even after an FPT algorithm is produced for a problem, there is likely to be a significant scope for heuristic improvements, and furthermore that such heuristic improvements can be made without losing the FPT guarantees.

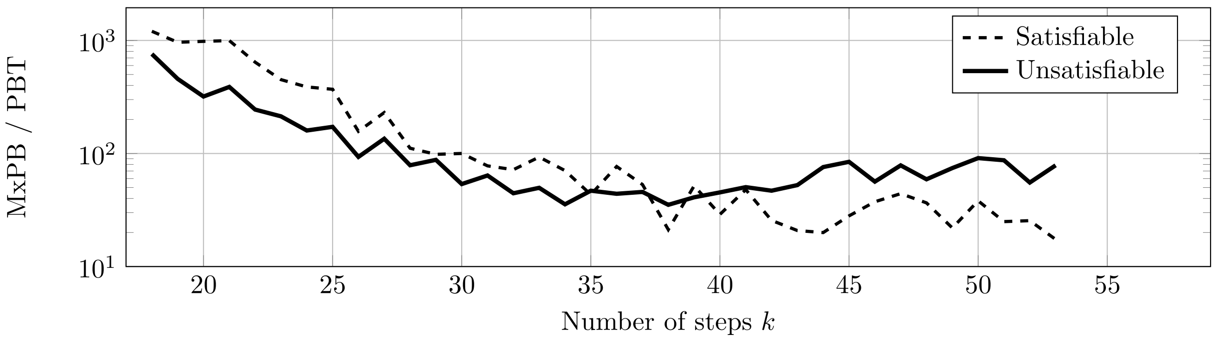
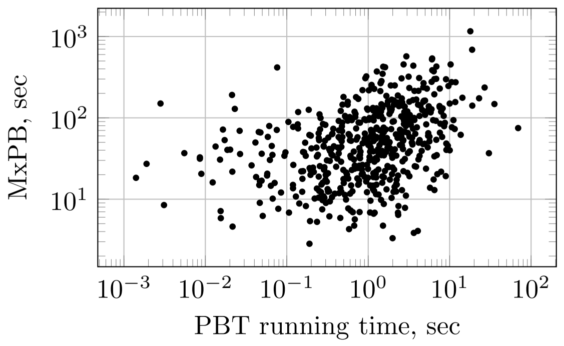
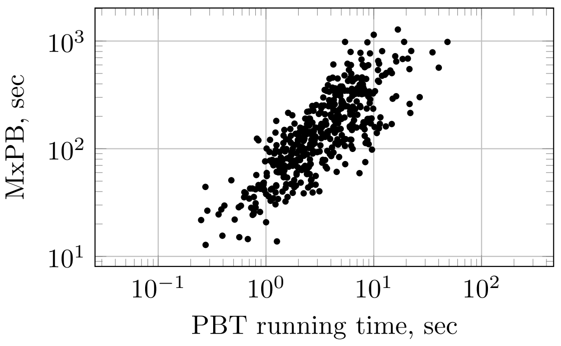


Figure 10: Comparison of MxPB and PBT performance. The vertical coordinate is the ratio between the median running time of MxSAT4J and median running time of PBT.



(a) Satisfiable instances



(b) Unsatisfiable instances

Figure 11: Correlation between PBT and MxPB 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 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 algorithm, 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.

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; our attempts to implement one gave a relative improvement of the algorithm's performance on satisfiable instances but the overheads were comparable to the gain 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 resolution proof system (PBPB (Res)) performs clearly better than cutting planes (PBPB (CP)) on the 'vary-k' slice. However, this may change with a higher ration between n and k, as evident from Appendix D, where PBPB (CP) is more competitive, and also from Section 8.3, where PBPB (CP) outperforms PBPB (Res) at high n to k ratio. It would be interesting to develop PB solvers that would automatically get the best of these two regimes; and so the WSP could be a useful benchmark.

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 patters 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 $n \gg k$.

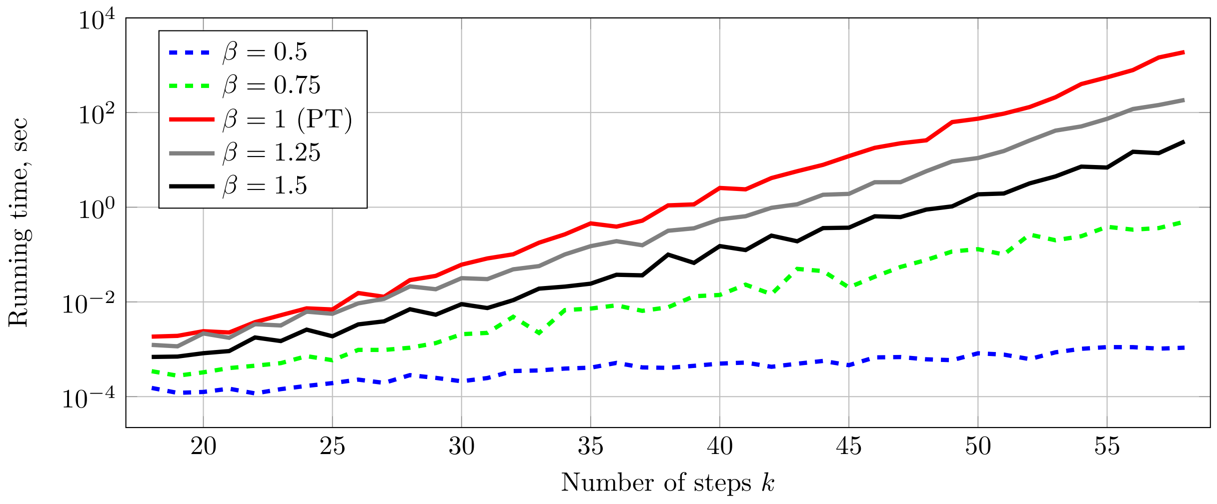


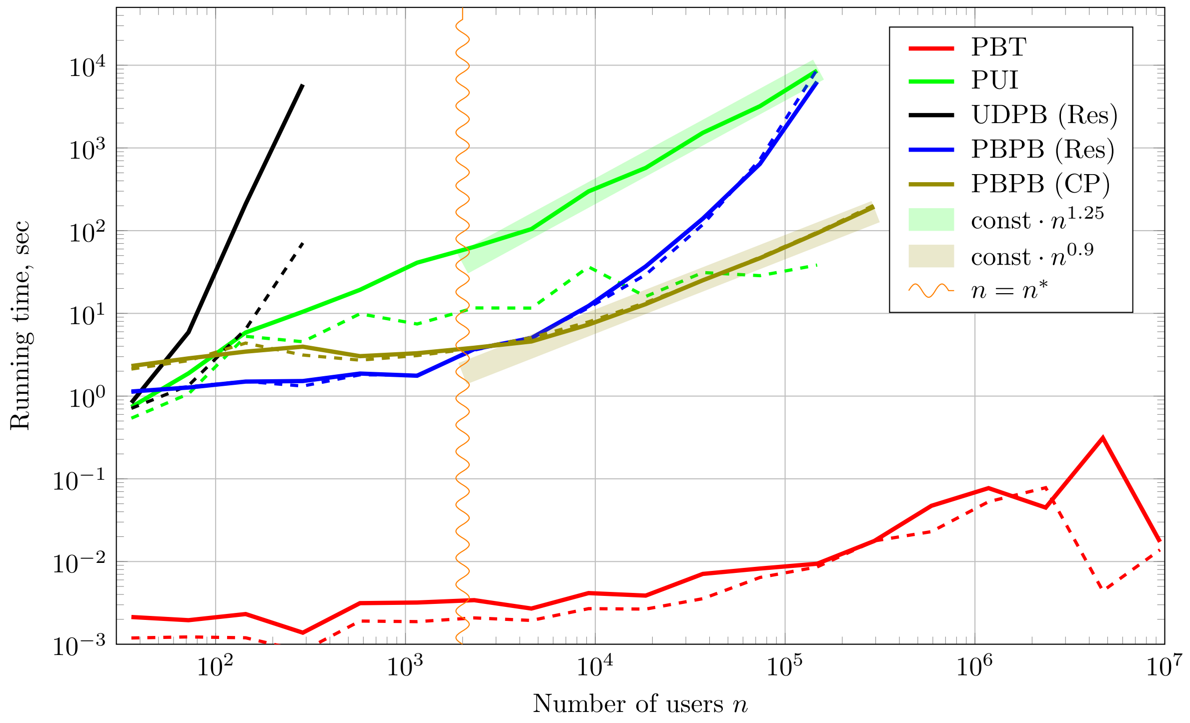
Figure 12: Performance of PBT on instances outside the PT region (obtained by changing β).

Although we argued that good experiments should seek instances in phase transition region of parameter space, it is still interesting to verify the performance of our approach, implemented in 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 phase transition, $\beta = 1.0$. It shows the classic so-called "easy-hard-easy" behaviour. Below the phase transition ($\beta < 1$) the runtimes and scaling are very much better than at the phase transition; that is most of the instances are under-constrained, having many solutions, and so solving will terminate early.[17] Above the phase transition ($\beta > 1$) most of the instances are unsatisfiable, but the pruning will increase, and this is reflected in the improved scaling.

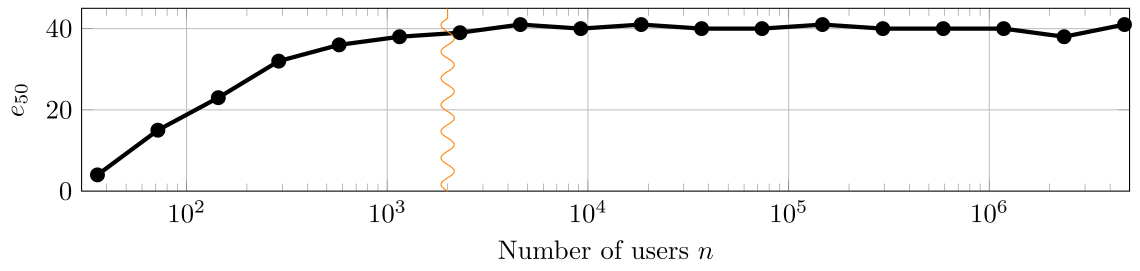
[17]A natural question is whether, at small enough β the scaling improves to being polynomial, however, we do not study that question here.

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, besides looking at the effect of increasing k, it is not only of theoretical interest, but also it is practically important to evaluate scalability of the approaches with regards to the number of users n.



(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. $WIG(18, n, e_{50}, 18)$.

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 simply selected to investigate behaviour of all the algorithms.

Figure 13b shows how the value of $e = e_{50}(18, n, 18)$ varies with n so as to remain at the phase transition 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 phase transition 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 on 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 is natural as the algorithm works through the users one at a time, and when 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 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 on n. However the real user ordering heuristic does not always pick the "right" users; as a result the running time mildly increases with n. That is, the PUI's running time matches our expectations.

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 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 scalability in this experiment, solving instances of size about 10^7 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 (CP) shows very good performance, demonstrating sub-linear scaling (roughly $n^{0.9}$), and outperforming PBPB (Res) on large n's. It is natural to hypothesize that this is because at large n the matching problem becomes ever more important, making the cutting planes proof system more suitable. Although, as discussed in Section 6.3, there is the potential of PB solutions in FPT time even with the resolution proof system, it seems quite possible that the standard PB solver in resolution proof system mode will not be able to find them, and so SAT4J (Res) could not fully exploit the FPT nature of the problem. On the other hand, the natural expectation, and confirmed by separate experiments, is that at large n, the matching problem will be very easy as there is little conflict between users, and simple propagation is usually sufficient to solve the problem. However, the cause for the relatively poor performance of PBPB (Res) at large n requires future investigation.

9. Conclusion

The Workflow Satisfiability Problem (WSP), with User Independent constraints, is of practical importance as it concerns a common problem in large organisation of the assignment of k tasks or 'steps' to a pool of n workers or 'users' in a fashion so as to satisfy a large variety of constraints. Furthermore, it can be considered as a powerful extension of hypergraph list colouring and, thus, may find other applications. However, current exact solution methods were only capable of solving cases of up to about $k \approx 20$ and this limited practical applications. In this paper, we have provided a new algorithm, PBT, and also a new Pseudo-Boolean encoding PBPB, that perform many orders of magnitude better, and allow solving much larger instances ($k \approx 50$ even at the hard PT).

The essential idea underlying PBT, and also inspiring PBPB, is that it is performing search on two levels:

- 'upper level': solving the UI constraints, by branching as to whether or not steps are assigned to the same user or to different users.
- 'lower level': solving the authorisation constraints, and so performing the assignment of users to steps.

The upper level is a heuristic tree-based search over the k steps, and so to take 'exponential' time commensurate with the number of patterns, $2^{\Theta(k \log_2 k)}$. The lower level is a bipartite matching problem and so solvable in polynomial time. The resulting complexity is $O(f(k)p(n, k))$ and so is FPT.

Although the WSP had been shown to be FPT, the previous FPT algorithm PUI was of limited practicality because of excessive, super-polynomial, memory usage. The new PBT algorithm is also FPT by construction, but because of the tree search it is also polynomial in memory usage, greatly extending the practical usability. Although PUI did have a form of the upper/lower split, it was not fully exploited; the PBT and PBPB gain their advantage by raising the upper/lower split to be the primary driving force, and appropriately ordering the branching decisions. 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 phase transitions arising from a generator of WSP instances. We found strong evidence of phase transition phenomena in the same fashion as previous extensive studies within graph theory and AI.

The immediate result of showing NP-hardness can be characterised (roughly) as simple broad-brush statement about the worst case such as "exponential in n". The point of work on PTs is that this broad view can be refined to be more specific about which instances show such bad behaviour, and also what the exponential is in practice. FPT adds a refinement that it now means "exponential in k, but polynomial in n".

As a result, FPT implies that the 'size space' of the problem is at least two dimensional, and extended methods are required to study empirical scaling of FPT algorithms. This paper gives a novel combination of FPT and PT and the use of multiple slices for a thorough empirical study of scaling of algorithms. In particular, we have chosen the "vary-k" slice to study the algorithms' scalability in terms of k, and the "fixed-k" slice to test the practical FPT properties of the algorithms. The "vary-k" slice revealed that PBT has scaling similar to that of PBPB, and that both algorithms demonstrate scaling having roughly a 13-fold reduction in the exponent compared to that predicted by the worst-case time complexity, significantly extending the range of practically solvable WSP instances. Note that all the old algorithms showed significantly worse scaling. The "vary-n" slice confirmed very good scaling (at least within reasonable values of n) of all the FPT algorithms.

One of the long-standing goals of AI has been to have general-purpose declarative representation which allows problems to be encoded and then solved by a general purpose solver, and of course this was one of the motivations for the PBPB formulation. Naturally it is then of interest as to whether such general-purpose representations for FPT problems also allow FPT algorithms. Ideally, there ought to be representations, with which off-the-shelf general solvers result in similar scaling to the direct implementations. We did observe such good behaviour of the PBPB+SAT4J (in resolution proof system mode) combination 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. Although, the constant was by one or two orders of magnitude worse, as might be expected, it indicated that the solver was able to determine a good search strategy. Furthermore, one may expect some additional efficiencies of PBPB when applied to real instances, as an off-the-shelf solver is more likely to be able to exploit associated structures. However, behaviour of PBPB solved in the default resolution proof system mode was significantly worse on the "fixed-k" slice compared to PBT. In such a slice, the PBPB (Res) solver demonstrated, apparently, exponential scaling despite we have shown that a simple tree-based solver is capable of exhibiting FPT time, i.e. polynomial scaling in "fixed-k" slice. This indicates that there is a room for improvement of current PB solvers, but that they have the potential to compete with bespoke solvers even on such two-level/FPT problems.

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 improve the branching heuristics (possibly by exploiting machine learning for adaptive search) and also find branch ordering selections that give a net gain on the satisfiable instances. It could also be modified to exploit parallel methods and multi-core machines. A particular 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 phase transition 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 studies 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 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 the general interest of the AI community in analysis of a widening range of complexity classes (e.g. [3]) (including studies of FPT problems, see e.g. [18, 33]) 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. This research was partially supported by EPSRC grants EP/H000968/1 (for DK and AP) and EP/K005162/1 (for AG and GG). The source codes of the instance generator and PBT, as well as the experimental data, are publicly available [30].

References

References

[1] Dimitris Achlioptas. Lower bounds for random 3-SAT via differential equations. *Theoretical Computer Science*, 265(12):159 – 185, 2001. Phase Transitions in Combinatorial Problems.

[2] American National Standards Institute. *ANSI INCITS 359-2004 for Role Based Access Control*, 2004.

[3] Delbert D. Bailey, Vctor Dalmau, and Phokion G. Kolaitis. Phase transitions of PP-complete satisfiability problems. *Discrete Applied Mathematics*, 155(12):1627 – 1639, 2007. {SAT} 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[4] D. A. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security and business objectives. *J. Comput. Security*, 22(5):661–698, 2014.

[5] David Basin, Samuel J. Burri, and Günter Karjoth. Optimal workflow-aware authorizations. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 93–102, New York, NY, USA, 2012. ACM.

[6] Elisa Bertino, Elena Ferrari, and Vijayalakshmi Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur.*, 2(1):65–104, 1999.

[7] B. Bollobas. *Random Graphs.* Academic Press, London, England, 1985.

[8] Endre Boros and Peter L. Hammer. Pseudo-boolean optimization. *Discrete Appl. Math.*, 123(1-3):155–225, November 2002.

[9] Peter J. Cameron. *Combinatorics: Topics, Techniques, Algorithms.* Cambridge University Press, 1994.

[10] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91, Sidney, Australia*, pages 331–337, 1991.

[11] D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Algorithms for the workflow satisfiability problem engineered for counting constraints. *J. Comb. Optim.*, 32(1):3–24, 2016. (DOI: 10.1007/s10878-015-9877-7).

[12] David Cohen, Jason Crampton, Andrei Gagarin, Gregory Gutin, and Mark Jones. Iterative plan construction for the workflow satisfiability problem. *J. Artif. Intel. Res.*, 51:555–577, 2014.

[13] W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25 – 38, 1987.

[14] J. Crampton, G. Gutin, and D. Karapetyan. Valued workflow satisfiability problem. In *ACM SACMAT*, pages 3–13, 2015.

[15] Jason Crampton. A reference monitor for workflow systems with constrained task execution. In E. Ferrari and G.-J. Ahn, editors, *SACMAT*, pages 38–47. ACM, 2005.

[16] Jason Crampton, Gregory Gutin, and Anders Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur.*, 16(1):4, 2013.

[17] Joseph Culberson and Ian Gent. Frozen development in graph coloring. *Theoretical Computer Science*, 265(12):227 – 264, 2001.

[18] Ronald de Haan, Martin Kronegger, and Andreas Pfandler. Fixed-parameter tractable reductions to SAT for planning. In *Proceedings of IJCAI*, IJCAI-2015, pages 1368–1373, 2015.

[19] Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing boolean satisfiability I: Background and survey of existing work. *J. Artif. Int. Res.*, 21(1):193–243, February 2004.

[20] R.G. Downey and M.R. Fellows. *Foundations of Parameterized Complexity.* Springer, 2013.

[21] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer Verlag, 1999.

[22] Igor Dukanovic and Franz Rendl. A semidefinite programming-based heuristic for graph coloring. *Discrete Applied Mathematics*, 156(2):180 – 189, 2008. Computational Methods for Graph Coloring and it's Generalizations.

[23] Ronald D. Dutton and Robert C. Brigham. A new graph colouring algorithm. *Comput. J.*, 24(1):85–86, 1981.

[24] Yun Fan and Jing Shen. On the phase transitions of random k-constraint satisfaction problems. *Artificial Intelligence*, 175(34):914 – 927, 2011.

[25] M. R. Fellows, T. Friedrich, D. Hermelin, N. Narodytska, and F. A. Rosamond. Constraint satisfaction problems: Convexity makes all different constraints tractable. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI11)*, pages 522–527, 2011.

[26] G Gutin, S Kratsch, and M Wahlstrom. Polynomial kernels and user reductions for the workflow satisfiability problem. *Algorithmica*, 2015. doi: 10.1007/s00453-015-9986-9.

[27] G. Gutin and M. Wahlström. Tight lower bounds for the workflow satisfiability problem based on the strong exponential time hypothesis. *Inf. Process. Lett.*, 116(3):223–226, 2016.

[28] Bernardo A. Huberman and Tad Hogg. Phase transitions in artificial intelligence systems. *Artif. Intell.*, 33(2):155–171, 1987.

[29] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[30] D. Karapetyan. Source codes of the pattern backtracking algorithm, the instance generator and the converter of instances into pbpb and udpb formulations. `http://csee.essex.ac.uk/staff/dkarap/?page=publications&key=WSP-AI`.

[31] D. Karapetyan, A. Gagarin, and G. Gutin. Pattern backtracking algorithm for the workflow satisfiability problem. In *Frontiers in Algorithmics 2015*, volume 9130 of *Lect. Notes Comput. Sci.*, pages 138–149. Springer, 2015.

[32] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, AAAI'05, pages 1368–1373. AAAI Press, 2005.

[33] Martin Kronegger, Andreas Pfandler, and Reinhard Pichler. Parameterized complexity of optimal planning: A detailed map. In *Proceedings of IJCAI*, IJCAI-2013, 2013.

[34] Daniel Le Berre and Anne Parrain. The SAT4J library, release 2.2. *J. Satisf. Bool. Model. Comput.*, 7:59–64, 2010.

[35] L. Lovasz. On the Shannon capacity of a graph. *IEEE Trans. Inf. Theor.*, 25(1):1–7, September 2006.

[36] Marc Mézard, Thierry Mora, and Riccardo Zecchina. Clustering of solutions in the random satisfiability problem. *Physical Review Letters*, 94(19):197205, 2005.

[37] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[38] Remi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, July 1999.

[39] Andrew J Parkes. Clustering at the phase transition. *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 340–346, 1997.

[40] J. Orlin R. Ahuja, T. Magnanti. *Network Flows.* Prentice Hall, 1993.

[41] Alexander A. Razborov. *Developments in Language Theory: 5th International Conference, DLT 2001 Wien, Austria, July 16–21, 2001 Revised Papers*, chapter Proof Complexity of Pigeonhole Principles, pages 100–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[42] Arindam Roy, Shamik Sural, Arun Kumar Majumdar, Jaideep Vaidya, and Vijayalakshmi Atluri. Minimizing organizational user requirement while meeting security constraints. *ACM Trans. Manage. Inf. Syst.*, 6(3):12:1–12:25, September 2015.

[43] Bart Selman and Scott Kirkpatrick. Critical behavior in the computation cost of satisfiability testing. *Artif. Intell.*, 81:273–295, 1996.

[44] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4):40, 2010.

[45] D.B. West. *Introduction to Graph Theory.* Prentice Hall, 2 edition, 2001.

Appendix A. Encoding Constraints

As noted in Section 5.2, any UI constraint can be formulated in terms of M variables only. However, the straightforward approach adopting a list of all obeying or disobeying patterns may result into encoding of exponential size in the size of the constraint scope. Here we show some more compact encodings of several standard UI constraints. Let c be the constraint to be encoded and $T_c = \{s_1, s_2, \dots, s_q\}$.

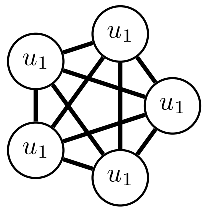
Appendix A.1. Easy Cases

For completeness, below we give a list of constraints that are easy to compactly encode with the M variables.

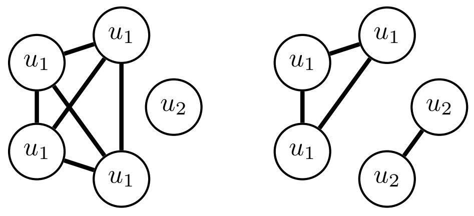
- Not-equals, also called separation of duty: $M_{s_1,s_2} = 0$.
- Equals, also called binding of duty: $M_{s_1,s_2} = 1$.
- All-different, or at-least-q-out-of-q: $M_{s_i,s_j} = 0$ for every $1 \le i < j \le q$.
- Not-all-different, or at-most-$(q-1)$-out-of-q: $\sum_{i=1}^{q-1} \sum_{j=i+1}^{q} M_{s_i,s_j} \ge 1$.

Appendix A.2. Counting Constraints with Scope Size up to Five

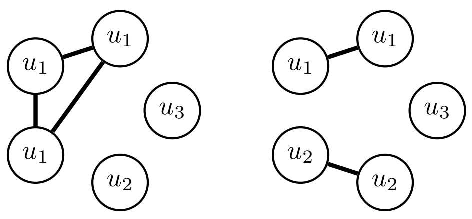
Let $G = (T_c, E)$ be a graph with vertex set T_c and edges $E = \{(s_i, s_j) : M_{s_i,s_j} = 1\}$. Observe that G uniquely represents a pattern on step set T_c as defined by appropriate variables M's; it consists of cliques only, with each clique corresponding to a block of the pattern. Recall that counting constraints are restricting the number of distinct users to be assigned to the scope; thus they are step-symmetric, i.e. satisfiability of a single counting constrain does not depend on the permutation of steps in its scope. In particular, the number of users assigned to the scope T_c is exactly the number of cliques in G, and this number of cliques can often be determined by simple counting of edges in G. Figure A.14 shows all patterns (subject to step permutations) on scope of size 5 and gives the possible number of edges.



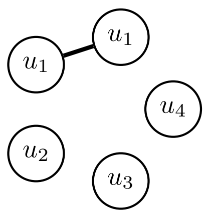
(a) One user: 10 edges



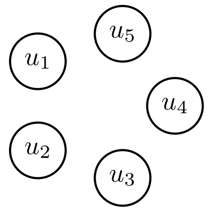
(b) Two distinct users: 4 or 6 edges



(c) Three distinct users: 2 or 3 edges



(d) Four distinct users: 1 edge



(e) Five distinct users: 0 edges

Figure A.14: Graphs G illustrating of the user assignments within a scope of five steps. There are $B_5 = 52$ patterns with scope of size 5, however those differing only by a permutation of the steps are not given as they do not change the number of edges.

	# distinct users					
$\lvert T_c\rvert = q$	1	2	3	4	5	6
2	1	0	–	–	–	–
3	3	1	0	–	–	–
4	6	2–3	1	0	–	–
5	10	4–6	2–3	1	0	–
6	15	6–10	3–6	2–3	1	0

Table A.1: This table gives the bounds $\underline{\sigma_{q,r}} \le |E| \le \overline{\sigma_{q,r}}$ for the number $|E|$ of edges in a graph G for each scope size q and number of distinct users. One can observe that counting the number of edges is sufficient to define counting constraints with scope size up to 5.

Table A.1 gives the possible number of edges in G for various scope sizes q and numbers of distinct assigned users. One can see that for any $q \le 5$ the ranges do not overlap, indicating that it is possible to determine the number of distinct users assigned to T_c just by counting edges in G. However, at $q = 6$, some of the ranges overlap, and then not every counting constraint with $q \ge 6$ can be encoded in this way.

For $q \le 5$ and $1 \le r \le q$, let $\underline{\sigma}_{q,r}$ and $\overline{\sigma}_{q,r}$ be the lower and upper bounds, respectively, on the number of edges in G with q nodes and exactly r cliques. Then we can formulate an at-most-r constraint as

$$\sum_{i=1}^{q-1} \sum_{j=i+1}^{q} M_{s_i,s_j} \ge \underline{\sigma}_{q,r}, \tag{A.1}$$

and an at-least-r constraint as

$$\sum_{i=1}^{|T|-1} \sum_{j=i+1}^{|T|} M_{s_i,s_j} \leq \overline{\sigma}_{q,r}. \tag{A.2}$$

Encoding of counting constraints with scope sizes above five is considered in the next section.

Appendix A.3. Other Constraints

Here we give some other encodings that, among others, allow us to formulate counting constraints with scope size above 5 and some other standard UI constraints.

Observe that existence of at least t cliques in G (see Appendix A.2) means that there is at least one independent set $T' \subset T_c$ of size t. Thus, to encode an at-most-r constraint, we can request that there is no independent set of size $r+1$:

$$\sum_{s'<s'' \in T'} M_{s',s''} \geq 1 \text{ for every } T' \subset T_c,\ |T'| = r+1. \tag{A.3}$$

This encoding requires $\binom{q}{r+1}$ constraints and no new variables.

Now consider a permutation σ of T_c, and let $\sigma(q+1) = \sigma(1)$. Observe that

$$|\{i:\ i \in \{1,2,\dots,q\} \text{ and } (\sigma(i), \sigma(i+1)) \notin E\}| \geq t,$$

where t is the number of cliques in G. Hence, the following encodes an at-least-r constraint:

$$\sum_{i=1}^{q} (1 - M_{s_{\pi(i)}, s_{\pi(i+1)}}) \geq r \text{ for every permutation } \pi \text{ of } T_c. \tag{A.4}$$

Some of the constraints will be identical due to the step symmetry; as a result, we will need $(q-1)!/2$ constraints, and no new variables.

We also propose more compact encodings that involve creation of new variables. Let G_i be a subgraph of G induced by a node set $\{s_1, s_2, \dots, s_i\}$; observe that G_i also consists of cliques only. Let t_i be the difference in the number of cliques between G_i and G_{i-1} for $i = 2, 3, \dots, q$, and $t_1 = 1$. Then the number of cliques in $G = G_q$ can be computed as $\sum_{i=1}^{q} t_i$.

Using variables t, we can encode an at-least-r constraint with an arbitrary scope as follows:

$$t_1 = 1, \tag{A.5}$$

$$t_i \leq 1 - M_{s_j,s_i}\ \forall j < i,\ i = 2, 3, \dots, q \text{ and} \tag{A.6}$$

$$\sum_{i=1}^{q} t_i \geq r. \tag{A.7}$$

This encoding takes $O(q^2)$ constraints and $O(q)$ new variables.

Similarly we can encode an at-most-r constraint with an arbitrary scope:

$$t_1 = 1, \tag{A.8}$$

$$t_i \geq \sum_{j=1}^{i-1} (1 - M_{s_j,s_i}) - (i-2),\ i = 2, 3, \dots, q \text{ and} \tag{A.9}$$

$$\sum_{i=1}^{q} t_i \leq r. \tag{A.10}$$

This encoding takes $O(q)$ constraints and $O(q)$ new variables.

Another standard UI constraint is the *generalised threshold constraint* (t_l, t_r, T_c) which restricts the number of steps assigned to a user involved in execution of steps T_c [16]. More formally, each user assigned to at least one step $s \in T_c$ is required to execute between t_l and t_r steps in T_c. Observe that this constraint can be enforced by restricting the size of cliques in G between t_l and t_r:

$$t_l \leq \sum_{j=1}^{q} M_{s_i,s_j} \leq t_r \text{ for } s_i \in T_c\,. \tag{A.11}$$

Appendix B. Estimation of the Critical Point

Here we give an (approximate) computation in the style of an 'annealed estimate' of the average number of solutions given k, n, γ and e; with the intent to use it to obtain indications of the location of the phase transition points. Since many of the probabilities depend on the number b of blocks in the solution, we first compute the estimate for a given b and then aggregate the results.

The number of patterns with b blocks is exactly the Stirling number $c(k, b)$ of the second kind, and there are exactly $P(n, b) = \frac{n!}{(n-b)!}$ plans implementing a pattern with b blocks. Consider a scope T of size $|T| = q$. Let $p(q, r)$ be the probability of that T intersect with exactly r distinct pattern blocks. There are b^q ways to assign blocks within T, and $c(q, r) \cdot P(b, r)$ ways to assign exactly r distinct blocks. Hence, $p(q, r) = \frac{c(q,r) \cdot P(b,r)}{b^q}$. Then the probability that a random pattern (or plan, which is the same in this context) satisfies an at-most-r constraint is $\sum_{r'=1}^{r} p(q, r')$. The probability that a random pattern satisfies an at-least-r constraint is $1 - \sum_{r'=1}^{r-1} p(q, r')$. Not-equals is a special case of at-least-r with $q = 2$ and $r = 2$; hence, the probability that a not-equals hits a random pattern is $1 - p(2, 1) = 1 - \frac{1}{b}$ as one could predict. We conclude that the number $N^{\text{elig}}_{\text{pat}}(b)$ of eligible patterns with b blocks is on average

$$N^{\text{elig}}_{\text{pat}}(b) = c(k, b) \cdot \frac{1}{b^e} \cdot \left(\left(\sum_{r'=1}^{r} p(q, r') \right) \cdot \left(1 - \sum_{r'=1}^{r-1} p(q, r') \right) \right)^{\gamma} . \tag{B.1}$$

$$N^{\text{elig}}_{\text{pat}} = \sum_{b} N^{\text{elig}}_{\text{pat}}(b) \tag{B.2}$$

When B.2 gives $N^{\text{elig}}_{\text{pat}} < 0.5$ then at least half the instances have no eligible pattern and so must be unsatisfiable, and so this gives an upper bound on the location of the PT. However, it is relatively weak, and so also need, unsurprisingly, to also take account of the authorisations.

The probability of a random plan being authorised is $p^{\text{auth}}(b) = (1/4)^k$ as the probability of a single step being authorised is $1/4$. Then the number of valid plans, on average, is

$$N^{\text{valid}}_{\text{plans}} = \sum_{b=3}^{k} N^{\text{elig}}_{\text{pat}}(b) \cdot P(n, b) \cdot p^{\text{auth}}(b) . \tag{B.3}$$

(Note that there are no eligible patterns with $b < 3$ due to at-least-3 constraints.) We report the number $N^{\text{elig}}_{\text{pat}}(b)$ of eligible patterns, number $P(n, b) \cdot p^{\text{auth}}(b)$ of authorised plans per pattern and the average number $N^{\text{valid}}_{\text{plans}}$ of valid plans in Table B.2. Observe that $N^{\text{valid}}_{\text{plans}}$ is often well above 1, especially for large b, but this is mainly because of the huge number of authorised plans per pattern; the expected number of eligible patterns for large b is negligible. This suggests that the distribution of $N^{\text{valid}}_{\text{plans}}$ is highly multimodal; averaged over the instances of the given parameters, the number of valid plans will be large but the majority of instances will have no valid plans at all. Nonetheless, B.3 can be used to give an upper bound on the location of the PT because when the average number drops to 0.5, then, independently of the distribution, at least one-half of the instances must be unsatisfiable.

Thus, the average number of valid plans might not be a practical indicator of the probability of instance satisfiability; to establish the phase transition parameters, we suggest a different technique. In particular, we exploit the two-layer nature of the problem. We estimate the probability $p^{\text{auth}}_{\text{pat}}(b)$ that a pattern with b blocks is authorised, i.e. there exists at least one plan realising it, and then use it to compute the probability $p^{\text{sat}}(b)$ that there exists at least one valid plan.

To obtain a rough estimate of $p^{\text{auth}}_{\text{pat}}(b)$ (which is then not an upper bound), we assume that all the blocks in the pattern are of the same size k/b. This might be justified as reasonable because

1. There are more patterns with roughly equal size blocks – by standard counting;
2. Authorisations are harder to satisfy for larger blocks and the increase is likely be super-linear in the block size – and this will tend to promote solutions that have a fair distribution of block sizes.

The probability of a single block being authorised by a random user is $(1/4)^{k/b}$, and the probability that there is at least one user authorised to a given block is $1 - \left(1 - \left(\frac{1}{4}\right)^{k/b}\right)^n$. To simplify calculations,

	Prob. per constraint			Number of			Probability of	
	$\neq$	$\leq$	$\geq$	eligible patterns	auth. plans per pattern	valid plans	pattern is auth.	valid pat. exists
b				$N^{\text{elig}}_{\text{pat}}(b)$	$P(b,r)$	$N^{\text{valid}}_{\text{plans}}$	$p^{\text{auth}}_{\text{pat}}(b)$	$p^{\text{sat}}(b)$
3	$6.7 \cdot 10^{-1}$	$1.0 \cdot 10^{0}$	$6.2 \cdot 10^{-1}$	$2.8 \cdot 10^{-2}$	$2.3 \cdot 10^{-11}$	$6.5 \cdot 10^{-13}$	$2.3 \cdot 10^{-11}$	$6.5 \cdot 10^{-13}$
4	$7.5 \cdot 10^{-1}$	$7.7 \cdot 10^{-1}$	$8.2 \cdot 10^{-1}$	$2.4 \cdot 10^{4}$	$6.9 \cdot 10^{-9}$	$1.6 \cdot 10^{-4}$	$6.9 \cdot 10^{-9}$	$1.6 \cdot 10^{-4}$
5	$8.0 \cdot 10^{-1}$	$5.8 \cdot 10^{-1}$	$9.0 \cdot 10^{-1}$	$3.6 \cdot 10^{5}$	$2.0 \cdot 10^{-6}$	$7.3 \cdot 10^{-1}$	$1.8 \cdot 10^{-6}$	$4.7 \cdot 10^{-1}$
6	$8.3 \cdot 10^{-1}$	$4.4 \cdot 10^{-1}$	$9.4 \cdot 10^{-1}$	$1.5 \cdot 10^{5}$	$6.0 \cdot 10^{-4}$	$8.8 \cdot 10^{1}$	$2.7 \cdot 10^{-4}$	$1.0 \cdot 10^{0}$
7	$8.6 \cdot 10^{-1}$	$3.5 \cdot 10^{-1}$	$9.6 \cdot 10^{-1}$	$1.3 \cdot 10^{4}$	$1.8 \cdot 10^{-1}$	$2.2 \cdot 10^{3}$	$1.4 \cdot 10^{-2}$	$1.0 \cdot 10^{0}$
8	$8.8 \cdot 10^{-1}$	$2.8 \cdot 10^{-1}$	$9.7 \cdot 10^{-1}$	$5.0 \cdot 10^{2}$	$5.2 \cdot 10^{1}$	$2.6 \cdot 10^{4}$	$1.9 \cdot 10^{-1}$	$1.0 \cdot 10^{0}$
9	$8.9 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	$9.8 \cdot 10^{-1}$	$1.3 \cdot 10^{1}$	$1.5 \cdot 10^{4}$	$1.9 \cdot 10^{5}$	$6.2 \cdot 10^{-1}$	$1.0 \cdot 10^{0}$
10	$9.0 \cdot 10^{-1}$	$1.9 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	$4.4 \cdot 10^{6}$	$1.1 \cdot 10^{6}$	$9.1 \cdot 10^{-1}$	$2.2 \cdot 10^{-1}$
11	$9.1 \cdot 10^{-1}$	$1.6 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$3.7 \cdot 10^{-3}$	$1.3 \cdot 10^{9}$	$4.8 \cdot 10^{6}$	$9.9 \cdot 10^{-1}$	$3.7 \cdot 10^{-3}$
12	$9.2 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	$9.9 \cdot 10^{-1}$	$5.0 \cdot 10^{-5}$	$3.7 \cdot 10^{11}$	$1.8 \cdot 10^{7}$	$1.0 \cdot 10^{0}$	$5.0 \cdot 10^{-5}$
				…				
30	$9.7 \cdot 10^{-1}$	$2.6 \cdot 10^{-2}$	$1.0 \cdot 10^{0}$	$3.2 \cdot 10^{-49}$	$4.0 \cdot 10^{55}$	$1.3 \cdot 10^{7}$	$1.0 \cdot 10^{0}$	$3.2 \cdot 10^{-49}$

Table B.2: Computational analysis of random WSP instances; $k = 30$, $\gamma = k$, $e = e_{\text{PT}} = 50$, $n = 10k$. b is the number of blocks in the solution. Columns 2 to 4 show probabilities of a random plan with the given number of blocks satisfying corresponding constraint. Using the estimates of the number of eligible patterns and authorised plans per pattern, we compute the average number of valid plans. We also estimate the probability of existence of at least one valid pattern.

we also relax the requirement that distinct blocks need to be assigned to distinct users, and conclude that

$$p^{\text{auth}}_{\text{pat}}(b) = \left(1 - \left[1 - \left(\frac{1}{4}\right)^{k/b}\right]^{n}\right)^{b}. \tag{B.4}$$

Using (B.4) we compute $p^{\text{sat}}(b)$ as follows:

$$p^{\text{sat}}(b) = \begin{cases} N^{\text{elig}}_{\text{pat}}(b) \cdot p^{\text{auth}}(b) & \text{if } N^{\text{elig}}_{\text{pat}}(b) < 1, \\ 1 - \left(1 - p^{\text{auth}}(b)\right)^{N^{\text{elig}}_{\text{pat}}(b)} & \text{if } N^{\text{elig}}_{\text{pat}}(b) \geq 1. \end{cases} \tag{B.5}$$

Then the probability p^{sat} of existence of at least one valid pattern is $1 - \prod_{b=3}^{k} 1 - p^{\text{sat}}(b)$, and the phase transition region can be established by finding parameters, using B.5, leading to $p^{\text{sat}} = 0.5$.

Observe that $p^{\text{sat}}(b)$ is very low at high b (see Table B.2) reflecting the fact that there is a very low probability of existence of a valid plan with many blocks. In fact, Table B.2 shows that the number of blocks in a valid pattern is tightly bounded – hence we expect the number of users in a valid plan being usually forced by the instance.

Moreover, we can see that the total number of eligible patterns is relatively low at phase transition and, hence, the complexity of the problem is driven by constraints and not authorisations. This is reflected in our overall strategy focusing on constraints (by means of patterns) and considering authorisations as a secondary component.

Another interesting observation is that the at-least-3 constraints are relatively weak while at-most-3 constraints significantly affect the probability of a pattern being eligible. This fact is exploited by our branching heuristic, see Section 4.4.

Our experiments show that the estimate of $p^{\text{sat}}(b)$ is relatively accurate; for example, for $k = 30$ it predicts phase transition at $\beta = 1.17$ (for the definition of β see Section 8.2), and for $k = 50$ at $\beta = 1.02$, see Figure B.15. Hence, our formulas can be used to quickly predict if certain parameters are likely to result in sat or in unsat instances.

Appendix C. Emergence of Forced Variables

A particularly interesting aspect of phase transitions is the emergence of forced variables in the critical region. That is, variables that must have some particular value in all solutions – and so are entailed by the system. This has been extensively studied in the context of Random 3SAT and Graph Colouring Problem (e.g. [17, 32, 39, 36]). In the context of standard graph colouring, the permutation symmetry

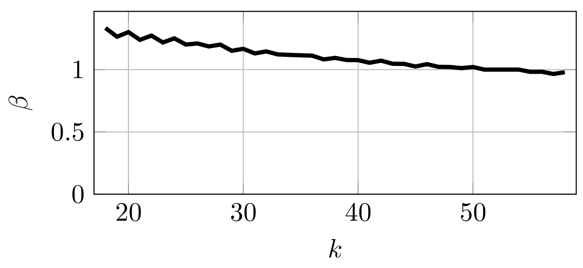


Figure B.15: Predicted values of β by our improved annealed estimate method. Recall that, by definition, the correct value of β is 1 for any k.

means no vertex can be forced to have a particular colour. Hence, the forcing is instead considered in terms of whether different vertices are forced to have the same colour, or else forced to have different colours. In other words, an instance might imply separation-of-duty and/or binding-of-duty constraints that are not explicitly listed in its description. This directly corresponds to whether M variables become forced.

We have performed experiments to determine this empirically in the WSP instances in the region of the phase transition. Determining whether a particular value of M is forced can be done directly by adding the negation to the instance and then testing for unsatisfiability.

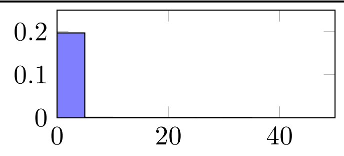
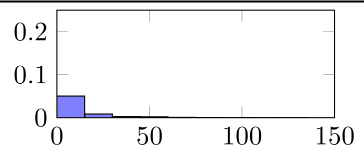
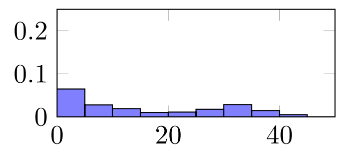
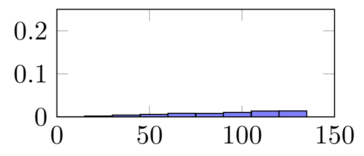
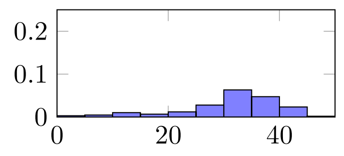
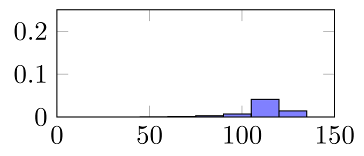
β	Avg. $=$	Avg. $\neq$	$=$	$\neq$
0.8	0	12		
1.0	16	91		
1.2	31	112		

Table C.3: Forced constraints. In this experiment, $k = 20$ and the total number of M-variables is 190 (recall that $M_{ij} = M_{ji}$; we count such a pair as one variable). Constraints explicitly defined by the instance are not counted as forced.

The results for the WSP are summarised in Table C.3. Note that there is a difference between a forced binding-of-duty (equals) constraint, denoted '$=$', and a forced separation-of-duty (not-equals) constraint, denoted '$\neq$'. Although a forced not-equals is more likely as there are usually more zeros than ones in the M matrix, and as $M_{s_1,s_2} = 0$ is a weaker decision than $M_{s_1,s_2} = 1$, we still observe quite a lot of forced equals constraints. We also see that as we move through the phase transition region from slightly under-constrained to slightly over-constrained, the number of forced M variables increases rapidly. Similar behaviour has been empirically seen in Random 3SAT (see e.g. [39]). This is important in that it again gives evidence that the WSP instances are behaving like a phase transition would be expected to behave, and so can be expected to be a good and effective test of algorithms for the WSP.

Note that we have only studied the freezing of the M variables, but due to the user authorisations (or list-colouring) it is quite possible that other notions of freezing also emerge. We also believe it further supports that the PT is worthy of study in its own right.

Another observation is that the information on forced variables (or constraints) could actually be useful to the users of the WSP decision support system. Indeed, knowing which of the constraints are forced might help the user to understand the instance and change it when necessary. A fast effective solver such as PBT can be used to produce such information, as well as specific solutions.

Appendix D. A supplementary "vary-k" slice

Figure D.16 shows how the solvers perform on the $n = 100k$ slice (see Figure 8).

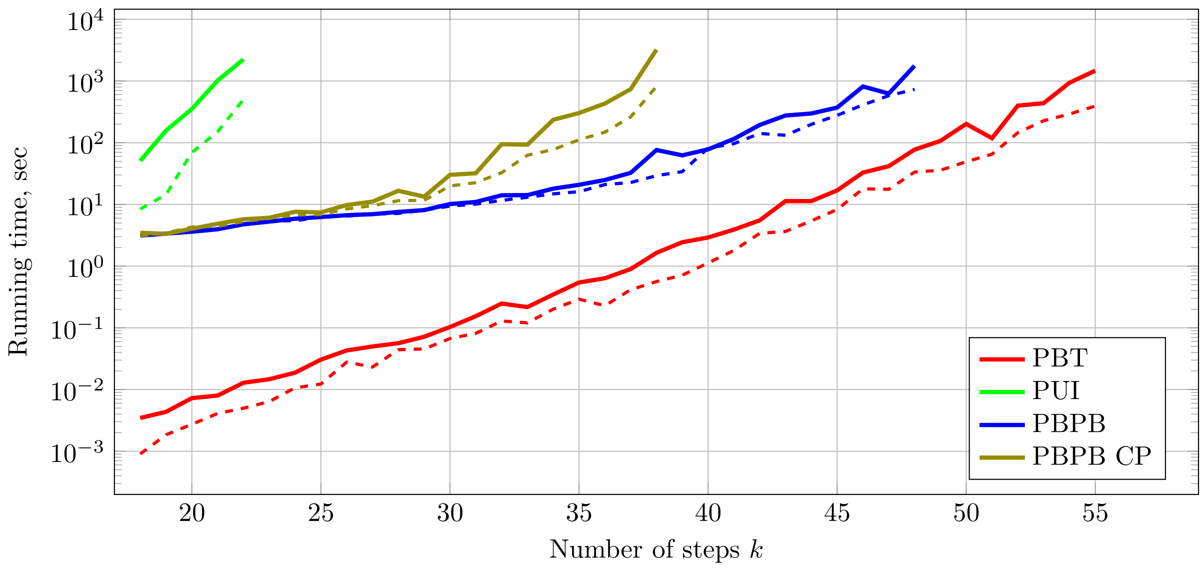


Figure D.16: Performance of the solvers on the $n = 100k$ slice.

The relation between the solvers' performance is similar to that on the 'vary-k' slice ($n = 10k$). The PBPB (CP) performance is slightly better compared to the PBPB (Res) performance, which is consistent with our observations that the cutting planes proof system is more appropriate when $n \gg k$. The poor performance of UDPB (Res) made it impractical to include it in this experiment; this is due to the non-FPT scaling of UDPB.