Engineering Algorithms for Workflow Satisfiability Problem with User-Independent Constraints

D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones

Royal Holloway, University of London, UK
{D.Cohen,Jason.Crampton,Andrei.Gagarin,G.Gutin,M.E.L.Jones}@rhul.ac.uk

Abstract. The workflow satisfiability problem (WSP) is a planning problem. Certain sub-classes of this problem have been shown to be fixed-parameter tractable. In this paper we develop an implementation of an algorithm for WSP that has been shown, in our previous paper, to be fixed-parameter for user-independent constraints. In a set of computational experiments, we compare our algorithm to an encoding of the WSP into a pseudo-Boolean SAT problem solved by the well-known solver SAT4J. Our algorithm solves all instances of WSP generated in our experiments, unlike SAT4J, and it solves many instances faster than SAT4J. For lightly constrained instances, SAT4J usually outperforms our algorithm.

1 Introduction

It is increasingly common for organizations to computerize their business and management processes. The co-ordination of the steps that comprise a computerized business process is managed by a workflow management system. Typically, the execution of these steps will be triggered by a human user, or a software agent acting under the control of a human user, and the execution of each step will be restricted to some set of authorized users. In addition, we may wish to constrain the users who execute sets of steps (even if authorized). We may, for example, require that two particular steps are executed by two different users, in order to enforce some separation-of-duty requirement.

We assume the existence of a set U of users and model a workflow as a set S of steps; a set $\mathcal{A} = \{A(u) : u \in U\}$ of authorization lists, where $A(u) \subseteq S$ denotes the set of steps for which u is authorized; and a set C of (workflow) constraints. A constraint is a pair $c = (L, \Theta)$, where $L \subseteq S$ and Θ is a set of functions from L to U: L is the *scope* of the constraint and Θ specifies those assignments of steps in L to users in U that satisfy the constraint. Given a workflow $W = (S, U, \mathcal{A}, C)$, W is said to be *satisfiable* if there exists a function (called a *plan*) $\pi : S \to U$ such that

1. for all $s \in S$, $s \in A(\pi(s))$ (each step is allocated to an authorized user);
2. for all $(L, \Theta) \in C$, $\pi|_L = \theta|_L$ for some $\theta \in \Theta$ (every constraint is satisfied).

Evidently, it is possible to specify a workflow that is not satisfiable. Equally, an unsatisfiable workflow is of no practical use. Hence, it is important to be able to determine whether a workflow is satisfiable or not. We call this the *workflow satisfiability problem*. This problem has been studied extensively in the security community [2, 5, 16] and more recently as an interesting algorithmic problem [4, 7].

The workflow satisfiability problem (WSP) is known to be NP-hard [16] (and is easily shown to be NP-hard even if restricted to separation-of-duty constraints). Wang and Li [16] observed that, in practice, the number k of steps is usually significantly smaller than the number n of users and, thus, suggested to parameterize WSP by k. Wang and Li [16] showed that, in general, WSP is W[1]-hard, but it is *fixed-parameter tractable (FPT)* for certain classes of constraints (i.e., it can be solved in time $O^*(f(k))$, where f is an arbitrary function of k only and O^* suppresses not only constants, but also polynomial factors; we call algorithms with such running time *fixed-parameter*). For further terminology on parameterized algorithms and complexity, see monographs [8, 11, 14]. Crampton *et al.* [7] extended the FPT classes of [16] and obtained significantly faster algorithms.

Cohen *et al.* recently designed a generic WSP algorithm and proved that the algorithm is fixed-parameter for WSP restricted to the class of user-independent constraints [4]. Informally, a constraint c is *user-independent* if, given a 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. Almost all constraints studied in [7, 16] and other papers are user-independent (since the separation-of-duty constraints are user-independent, WSP restricted to the class of user-independent constraints is NP-hard).

It is well known that the gap between traditional "pen and paper" algorithmics and actually implemented and computer-tested algorithms can be very wide [3, 13]. In this paper, we demonstrate that the algorithm of Cohen *et al.* is not merely of theoretical interest by developing an implementation that is able to outperform SAT4J in solving instances of WSP. In a set of computational experiments, we compare performance of our implementation with performance of the well-known pseudo-Boolean satisfiability solver SAT4J [12]. Unlike SAT4J, our implementation solves all instances of WSP generated in the experiments and usually solves the instances faster than SAT4J. However, for lightly-constrained instances, SAT4J usually outperforms our implementation.

The paper is organized as follows. In Section 2, we describe how WSP for the family of instances we are interested in can be formulated as a pseudo-Boolean SAT problem. We also describe our fixed-parameter algorithm and discuss its implementation. Section 3 describes test experiments we have conducted with synthetic data to see in which cases our implementation is more efficient and effective than SAT4J. Finally, Section 4 provides conclusions and discusses plans for future work.

2 Methods of Solving WSP

In this paper we seek to demonstrate that (i) our algorithm can be implemented in such a way that it solves certain instances of WSP in a reasonable amount of time; (ii) our implementation can have better performance than SAT4J on the same set of instances. In this section we describe concisely how WSP can be encoded as a pseudo-Boolean SAT problem, how our algorithm works, and the heuristic speed-ups that we have introduced in the implementation of our algorithm.

2.1 WSP as a Pseudo-Boolean SAT Problem

Pseudo-Boolean solvers are recognized as an efficient way to solve general constraint networks [12]. Due to the difficulty of acquiring real-world workflow instances, Wang and Li [16] used synthetic data in their experimental study. Wang and Li encoded WSP as a pseudo-Boolean SAT problem in order to use SAT4J to solve instances of WSP. Their work considered separation-of-duty constraints, henceforth called *not-equals constraints*, which may be specified as a pair (s, t) of steps; a plan π satisfies the constraint (s, t) if $\pi(s) \neq \pi(t)$. Wang and Li considered a number of other constraints in their work, which we do not use in our experimental work since they add no complexity to an instance of WSP. In the experiments of Wang and Li, SAT4J solved all generated instances, and each of them quite efficiently. We test SAT4J on a set of WSP instances of a different type than in [16]. By varying the relevant parameters, we can make our instances more difficult for SAT4J to solve, as we show in the next section.

For a step s, let $A(s) = \{u \in U : s \in A(u)\}$. For their encoding, Wang and Li introduced (0,1)-variables $x_{u,s}$ to represent those pairs (u, s) that are authorized by $\mathcal{A}$. That is, we define a variable $x_{u,s}$ if and only if $s \in A(u)$. The goal of the SAT solver is to find an assignment of values to these variables (representing a plan) where $x_{u,s} = 1$ if and only if u is assigned to step s. These variables are subject to the following constraints:

- for every step s, $\sum_{u \in A(s)} x_{u,s} = 1$ (each step is assigned to exactly one user);
- for each not-equals constraint (s, t) and user $u \in A(s) \cap A(t)$, $x_{u,s} + x_{u,t} \leqslant 1$ (no user is assigned to both s and t).

We use not-equals constraints and some relatively simple counting constraints: "at-most-3" and "at-least-3" with scopes of size 5. The former may be represented as a set $(T, \leqslant)$, where $T \subseteq S$, $|T| = 5$, and is satisfied by any plan that allocates no more than three users in total to the steps in T. The latter may be represented as $(T, \geqslant)$ and is satisfied by any plan that allocates at least three users to the steps in T. For convenience and by an abuse of notation, given a constraint c of the form $(T, \geq)$ or $(T, \leq)$, we will write $s \in c$ to denote that $s \in T$.

We encode "at-least-3" and "at-most-3" constraints as part of the pseudo-Boolean SAT instance in the following way. For each "at-least-3" constraint c

and user u, we introduce a (0,1)-variable $z_{u,c}$. The variables $z_{u,c}$ will be such that if $z_{u,c} = 1$ then u performs a step in c, and will be used to satisfy a lower bound on the number of users performing steps in c. For each "at-most-3" constraint c and user u, we introduce a (0,1)-variable $y_{u,c}$. The variables $y_{u,c}$ will be such that if u performs a step in c then $y_{u,c} = 1$, and will be used to satisfy an upper bound on the number of users performing steps in c.

The variables are subject to the following constraints:

- for each "at-least-3" constraint c and user u: $z_{u,c} \leq \sum_{s \in c \cap A(u)} x_{u,s}$;
- for each "at-least-3" constraint c: $\sum_{u \in U} z_{u,c} \geqslant 3$.
- for each "at-most-3" constraint c, $s \in c$ and $u \in A(s)$: $x_{u,s} \leq y_{u,c}$;
- for each "at-most-3" constraint c: $\sum_{u \in U} y_{u,c} \leqslant 3$.

It is possible to prove the following assertion. The proof is omitted due to the space limit.

Lemma 1. *A WSP instance has a solution if and only if the corresponding pseudo-Boolean SAT problem has a solution.*

2.2 Fixed-Parameter Algorithm

We proceed on the basis of the assumption that the number k of steps is significantly smaller than the number n of users. Any function $\pi : T \to X$ with $T \subseteq S$ and $X \subseteq U$, is called a *partial plan*. We order the set of users and incrementally construct "patterns" for partial plans that violate no constraints for the first i users. At each iteration we assign a set of steps (which may be empty) to user $i + 1$. An assignment is *valid* if it violates no constraints and extends at least one existing pattern for users $1, \ldots, i$. Under certain conditions, we dynamically change the ordering of the remaining users.

Patterns are encodings of equivalence classes of partial plans and are used to reduce the number of partial plans the algorithm needs to consider. In particular, we define an equivalence relation on the set of all possible plans. This equivalence relation is determined by the particular set of constraints under consideration [4]. In the case of user-independent constraints, two partial plans $\pi : T \to X$ and $\pi' : T' \to X'$ are equivalent, denoted by $\pi \approx \pi'$, if and only if $T = T'$ and for all $s, t \in T$, $\pi(s) = \pi(t)$ if and only if $\pi'(s) = \pi'(t)$.

A pattern is a representation of an equivalence class, and there may be many possible patterns. In the case of $\approx$, we assume an ordering $s_1, \ldots, s_k$ of the set of steps. Then the encoding of an equivalence class for $\approx$ is given by $(T, (x_1, \ldots, x_k))$, where $T \subseteq S$ and, for some π in the equivalence class, we define

$$x_i = \begin{cases} 0 & \text{if } s_i \notin T, \\ x_j & \text{if } \pi(s_i) = \pi(s_j) \text{ and } j < i, \\ \max\{x_1, \ldots, x_{i-1}\} + 1 & \text{otherwise.} \end{cases}$$

We must ensure that there exist efficient algorithms for searching and inserting elements into the set of patterns. Cohen *et al.* [4] show that such algorithms exist

for user-independent constraints, essentially because the set of patterns admits a natural lexicographic order.

The overall complexity of the algorithm is determined by k, n, and the number w of equivalence classes for a pair (U_i, T), where U_i denotes the set of the first i users considered by the algorithm. Cohen *et al.* show that the algorithm has run-time $O^*(3^k w \log w)$ [4, Theorem 1]. Thus, we have an FPT algorithm when w is a function of k. For user-independent constraints, $w \leqslant B_k$, where B_k is the kth Bell number; $B_k = 2^{k \log k(1-o(1))}$ [1].

2.3 Implementing the Algorithm

In this section, we describe our algorithm in more detail, via a pseudo-code listing (Algorithm 1), focusing on the heuristic speed-ups we have introduced in our implementation. The algorithm iterates over the set of users constructing a set Π of patterns for valid partial plans (that is partial functions from S to U that violate no constraints or authorizations). For each user u, we attempt to find the set Π_u of valid plans that extend a pattern in Π and assign some non-empty subset of unassigned steps to u.

We assume a fixed ordering of the elements of S and that the users are ordered initially by the cardinality of their respective authorization lists (with ties broken according to the lexicographic ordering of the authorized steps). The ordering we impose on the set of users allows us to introduce a heuristic speed-up based on the idea of a useless user. A user v is *useless* if there exists a user u such that $A(v) \subseteq A(u)$, u has already been processed, and $\Pi_u = \emptyset$. Each iteration of the algorithm considers assigning steps to a particular user u and constructs a set Π_u of extended plans that includes this user. If, having examined all patterns in Π, we have $\Pi_u = \emptyset$, then there is no subset of steps (for which u is authorized) that can be added to Π without violating some constraint. Since all constraints are user-independent, we may now remove all useless users from the list of remaining users.

Much of the work of the algorithm is done in line 8. The time taken to check the validity of a plan can be reduced by considering the constraints in the following order: (1) not-equals constraints; (2) at-most-3 constraints; (3) at-least-3 constraints. The intuition underlying this design choice is that we should consider constraints that violate the most plans first. In line 8 we consider a plan with a prescribed pattern and test whether its extension (by the assignment of steps in T' to user u) is valid. In line 12 we have to compute the pattern for the extended plan and add it to the list Π_u of extended plans. As we noted in the previous section, results by Cohen *et al.* [4] assert that these subroutines can be computed efficiently.

Finally, we are able to propagate information about the current state of any at-most-3 constraints. Suppose ℓ steps from the 5 steps in an at-most-3 constraint $(T, \leqslant)$ have been assigned to two distinct users, where $\ell \in \{2, 3\}$. Then the remaining $5 - \ell$ steps must be assigned to a single user. Hence we may discard the pattern immediately if there is no user that is authorized for all the $5 - \ell$ remaining steps in T. Similarly, if there are any not-equals constraints defined on

Algorithm 1: Algorithm for WSP with user-independent constraints

```
1  begin
2      Initialize the set Π of patterns to the zero-pattern (∅, (0, ..., 0))
3      foreach u ∈ U do
4          Initialize Π_u = ∅;
5          foreach pattern p = (T, (x_1, ..., x_k)) in Π do
6              T_u ← A(u) \ T;
7              foreach ∅ ≠ T' ⊆ T_u do
8                  if plan π ∪ (T' → u) is valid (where π is a plan with pattern p)
                   then
9                      if T ∪ T' = S then
10                         return SATISFIABLE and π ∪ (T' → u);
11                     end
12                     Compute the pattern for π ∪ (T' → u) and add it to Π_u if
                       not present in Π;
13                 end
14             end
15         end
16         if Π_u = ∅ then
17             Remove useless users
18         else
19             Π ← Π ∪ Π_u
20         end
21         Re-order the list of remaining users according to propagation of
           at-most-3 constraints;
22     end
23     return UNSATISFIABLE;
24 end
```

any pair of the $5 - \ell$ remaining steps in T, then we know the pattern (encoding a partial plan) cannot possibly generate a total valid plan, and we may discard it at that point. Moreover, in an effort to determine whether the constraint can be satisfied as quickly as possible, we identify users who are authorized for all the $5 - \ell$ remaining steps in T and move one such user to the beginning of the list of remaining users at the end of each iteration accordingly (line 21). This determines a dynamic ordering of users.

3 Experiments

We used C++ to write an implementation of our algorithm for WSP with user-independent constraints. We then generated a number of instances of WSP and compared the performance of our implementation with that of SAT4J when solving those instances. All our experiments used a MacBook Pro computer

having a 2.6 GHz Intel Core i5 processor, 8 GB 1600 MHz DDR3 RAM[1] and running Mac OS X 10.9.2.

3.1 Testbed

Based on what might be expected in practice, we used values of $k = 16, 20, 24$, and set $n = 10k$. The number c_1 of at-most-3 constraints plus the number c_2 of at-least-3 constraints was set equal to $20, 40, 60$, and 80 (for $k = 24$, we did not consider $c_1 + c_2 = 20$ as the corresponding instances are normally easy to solve by SAT4J). We varied the "not-equals constraint density," i.e. the number of not-equals constraints as a percentage of $\binom{k}{2}$ (the maximum possible number), by using values in the set $\{10, 20, 30\}$. We also assumed that every user was authorized for at least one step but no more than $k/2$ steps; that is, $1 \leqslant |A(u)| \leqslant k/2$. While not-equals constraints and authorizations were generated for each instance separately, the at-most-3 constraints and at-least-3 constraints were kept the same for the corresponding three different values of densities of not-equal constraints and the same value of k.

We adopt the following convention to label our test instances in Tables 1 and 3: $c_1.c_2.d$ denotes an instance with c_1 at-most-3 constraints, c_2 at-least-3 constraints and constraint density d. Informally, we would expect the difficulty of solving instances $c_1.c_2.d$ for fixed k, c_1 and c_2 would increase as d increases (as the problem becomes "more constrained"). Similarly, at-most-3 constraints are more difficult to satisfy than at-least-3 constraints, so, for fixed k and d, instances $c_1.c_2.d$ will become harder to solve as c_1 increases. We would also expect that the time taken to solve an instance would depend on whether the instance is satisfiable or not, with unsatisfiable instances requiring all possible plans to be examined.

Assuming the number c_2 of at-least-3 constraints does not influence satisfiability too much ($c_2 \leq 80$), we varied c_1 trying to generate WSP borderline instances, i.e. instances which have a good chance of being both satisfiable and unsatisfiable. Some instances we generated are lightly constrained, i.e. have a relatively large number of valid plans, while others are highly constrained, i.e. unsatisfiable or have a relatively small number of valid plans. The minimum and maximum values of c_1 used in the experiments to generate Tables 1–4 correspond to instances which we view as mainly borderline. In other words, we started with instances that are experimentally not clearly lightly constrained and stopped at instances which are likely to be highly constrained as the corresponding three instances for the same values of k, c_1, c_2 are unsatisfiable.

Counting constraints were generated by first enumerating all 5-element subsets of S using an algorithm from Reingold *et al.* [15]. We then used Durstenfeld's version of the Fisher-Yates random shuffle algorithm [9, 10] to select independently at random c_1 constraint sets for at-most-3 constraints and c_2 constraint sets for at-least-3 constraints, respectively. The random shuffle algorithm was

[1] Our computer is more powerful than the one used by Wang and Li [16].

also used to select steps for which each user is authorized (the list of authorization sets was generated randomly subject to the cardinality constraints above). Finally, the random shuffle was used to select steps for each not-equals constraint.

3.2 Results

In our experiments we compare the run-times and performance of SAT4J and our algorithm. For the number k of steps equal to 20 and 24, we provide Tables 1 and 3, respectively, which give detailed results of our experiments. We record whether an instance was solved, and the response if it was solved. Thus, we report 'Y', 'N', or '?', indicating, respectively, a satisfiable instance, an unsatisfiable instance, or an instance for which the algorithm terminated without reaching a decision. Note that our algorithm reached a conclusive decision ('Y' or 'N') in all cases, whereas SAT4J failed to reach such a decision for some instances, typically because the machine ran out of memory. For Algorithm 1, we record the number of patterns generated before a valid plan was obtained or the instance was recognized as unsatisfiable, as well as the number of users considered before the algorithm terminated. We also record the time taken for the algorithms to run on each instance.

For lightly constrained instances, SAT4J performs better than our algorithm. This is to be expected, because many of the (large number of) potential patterns are valid. Thus, the number of possible patterns explored by our iterative algorithm is rather large, even when the number of users required to construct a valid plan is relatively small. In contrast, SAT4J simply has to find a satisfying assignment for (all) the variables. The tables also exhibit the expected correlation between the running time of our algorithm and two numbers: the number of patterns generated by the algorithm and the number of users considered, which, in turn, is related to the number of constraints and constraint density.

However, the situation is rather different for highly constrained instances, whether they are satisfiable or not. For such instances, SAT4J will have to consider very many possible valuations for the variables and the running times increase dramatically as a consequence. In contrast, our algorithm has to consider far fewer patterns and this more than offsets the fact that we may have to consider every user (for those cases that are unsatisfiable). Table 2 shows the summary statistics for the running times in Table 1 (to two decimal places). The statistics are based on running times for instances where both algorithms were able to return conclusive decisions. Note that the average time taken by our algorithm for satisfiable and unsatisfiable instances is of a similar order of magnitude; the same cannot be said for SAT4J. Note also the variances of the running times for the two algorithms, indicating that the running time of SAT4J varies quite significantly between instances, unlike our algorithm.

In Table 4, we summarize the results of our algorithm and SAT4J for all three values of k. As k increases, SAT4J fails more frequently, and was unable to reach a conclusive decision for over half the instances when $k = 24$. This is unsurprising, given that the number of variables will grow quadratically as k and n (which equals $10k$) increase. In Table 4 we report the average run-times

for those instances in which both algorithms were able to reach a conclusive decision. We also report (in brackets) the average run-time of our algorithm over all instances. For $k = 16$, the average run-time of our algorithm was two orders of magnitude better than that of SAT4J. As for the other values of k, our algorithm was much faster than SAT4J for unsatisfiable instances. However, for satisfiable instances, the picture was often more favourable towards SAT4J. Overall, for larger values of k, the average run-time advantage of our algorithm over SAT4J decreases, but the relative number of instances solved by SAT4J decreases as well.

It is interesting to note the way in which the mean running time $\hat{t}$ varies with the number of steps. In particular, $\hat{t}$ for our algorithm grows exponentially with k (with a strong correlation between k and $\log \hat{t}$), which is consistent with the theoretical running time of our algorithm ($O^*(2^{k \log k})$). The running time of SAT4J is also dependent on k, with a strong correlation between k and $\log \hat{t}$, which is consistent with the fact that there are $O(n^k)$ possible plans to consider. However, it is clear that the running time of SAT4J is much more dependent on the number of variables (determined by the number of users, authorizations, and constraints), than it is on k, unlike the running time of our algorithm.

4 Concluding Remarks

In this paper, we describe the implementation of a fixed-parameter algorithm designed to solve a specific hard problem known as the workflow satisfiability problem (WSP) for user-independent constraints. In theory, there exists an algorithm that can solve WSP for user-independent constraints in time $O^*(2^{k \log k})$ in the worst case. However, WSP is a practical problem with applications in the design of workflows and the design of access control mechanisms for workflow systems [6]. Thus, it is essential to demonstrate that theoretical advantages can be transformed into practical computation advantages by concrete implementations.

Accordingly, we have developed an implementation using our algorithm as a starting point. In developing the implementation, it became apparent that several application-specific heuristic improvements could be made. In particular, we developed specific types of propagation and pruning techniques for counting constraints.

We compared the performance of our algorithm with that of SAT4J—an "off-the-shelf" SAT solver. In order to perform this comparison, we extended Wang and Li's encoding of WSP as a pseudo-Boolean satisfiability problem. The results of our experiments suggest that our algorithm does, indeed, have an advantage over SAT4J when solving WSP, although this advantage does not extend to lightly constrained instances of the problem. The results also suggest that those advantages could be attributable to the structure of our algorithm, with its focus on the small parameter (in this case the number of workflow steps).

We plan to continue working on algorithm engineering for WSP. In particular, we plan to continue developing ideas presented in this paper and in [4] to develop

an efficient implementation of a modified version of our algorithm. We hope to obtain a more efficient implementation than the one presented in this paper. We also plan to try different experimental setups. For example, in this paper, we have used a uniform random distribution of authorizations to users with an upper bound at 50% of the number of steps for which any one user can be authorized. In some practical situations, a few users are authorized for many more steps than others. We have only considered counting constraints, rather than a range of user-independent constraints. In some ways, imposing these constraints enables us to make meaningful comparisons between the two different algorithms, but we would still like to undertake more extensive testing to confirm the initial results that we have obtained for this particular family of instances of WSP.

Acknowledgment. This research was supported by an EPSRC grant EP/K005162/1.

References

1. Berend, D., Tassa, T.: Improved bounds on Bell numbers and on moments of sums of random variables. Probability and Mathematical Statistics 30(2), 185–205 (2010)
2. Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur. 2(1), 65–104 (1999)
3. Chimani, M., Klein, K.: Algorithm engineering: Concepts and practice. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds.) Experimental methods for the analysis of optimization algorithms, pp. 131–158 (2010)
4. Cohen, D., Crampton, J., Gagarin, A., Gutin, G., Jones, M.: Pattern-based plan construction for the workflow satisfiability problem. CoRR abs/1306.3649 (2013)
5. Crampton, J.: A reference monitor for workflow systems with constrained task execution. In: Ferrari, E., Ahn, G.J. (eds.) SACMAT. pp. 38–47. ACM (2005)
6. Crampton, J., Gutin, G.: Constraint expressions and workflow satisfiability. In: Conti, M., Vaidya, J., Schaad, A. (eds.) SACMAT. pp. 73–84. ACM (2013)
7. Crampton, J., Gutin, G., Yeo, A.: On the parameterized complexity and kernelization of the workflow satisfiability problem. ACM Trans. Inf. Syst. Secur. 16(1), 4 (2013)
8. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer Verlag (1999)
9. Durstenfeld, R.: Algorithm 235: Random permutation. Communications of the ACM 7(7), 420 (1964)
10. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Oliver and Boyd, third edn. (1948)
11. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer Verlag (2006)
12. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2. J. Satisf. Bool. Model. Comput. 7, 59–64 (2010)
13. Myrvold, W., Kocay, W.: Errors in graph embedding algorithms. J. Comput. Syst. Sci. 77(2), 430–438 (2011)
14. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford U. Press (2006)
15. Reingold, E.M., Nievergelt, J., Deo, N.: Combinatorial algorithms: Theory and practice. Prentice Hall (1977)
16. Wang, Q., Li, N.: Satisfiability and resiliency in workflow authorization systems. ACM Trans. Inf. Syst. Secur. 13(4), 40 (2010)

Table 1. Experimental test results for $k = 20$

	SAT4J		**Algorithm 1**			
Instance	Output	CPU Time (s)	Output	CPU Time (s)	Users	Patterns
15.5.10	Y	0.974	Y	43.853	6	2,204,316
15.5.20	Y	3.888	Y	26.983	16	208,816
15.5.30	N	1,624.726	N	106.959	200	151,646
20.0.10	Y	1.022	Y	23.777	12	244,494
20.0.20	Y	354.240	Y	25.333	64	34,326
20.0.30	N	987.137	N	15.332	200	12,911
15.25.10	Y	1.092	Y	26.167	8	870,082
15.25.20	Y	12.008	Y	68.890	35	257,808
15.25.30	N	1,886.133	N	75.804	200	112,561
20.20.10	Y	3.997	Y	63.146	33	232,886
20.20.20	N	3,014.967	N	38.623	200	34,310
20.20.30	N	178.208	N	12.091	200	6,093
25.15.10	Y	13.425	Y	79.267	34	153,833
25.15.20	N	1,611.904	N	27.582	200	19,332
25.15.30	N	3,157.095	N	16.070	200	8,116
30.10.10	Y	1.244	Y	20.399	21	63,540
30.10.20	N	2,002.985	N	19.225	200	11,279
30.10.30	N	2,413.049	N	8.393	200	3,416
35.5.10	Y	11.363	Y	23.035	32	20,381
35.5.20	?	2,406.241	N	11.771	200	6,363
35.5.30	N	2,061.416	N	3.409	200	2,107
40.0.10	N	2,734.843	N	28.124	200	13,582
40.0.20	?	3,720.915	N	6.543	200	3,570
40.0.30	N	844.309	N	3.712	200	1,483
15.45.10	Y	1.039	Y	41.948	10	1,201,221
15.45.20	Y	19.302	Y	109.774	31	512,416
15.45.30	Y	97.056	Y	2.123	10	25,433
20.40.10	Y	218.154	Y	58.158	26	209,141
20.40.20	?	3,290.306	N	46.538	200	47,382
20.40.30	N	777.610	N	12.490	200	10,165
25.35.10	Y	5.075	Y	37.573	15	119,829
25.35.20	N	1,984.297	N	30.570	200	26,063
25.35.30	N	3,710.115	N	19.508	200	10,332
30.30.10	Y	317.208	Y	52.735	50	57,123
30.30.20	?	3,514.502	N	14.979	200	8,148
30.30.30	N	489.492	N	7.217	200	5,088
35.25.10	?	7,099.028	N	24.928	200	13,493
35.25.20	N	2,293.669	N	6.365	200	4,072
35.25.30	N	783.961	N	4.229	200	1,577
15.65.10	Y	2.296	Y	35.941	6	2,305,213
15.65.20	Y	13.504	Y	40.761	12	770,002
15.65.30	N	2,351.671	N	156.593	200	274,532
20.60.10	Y	68.804	Y	51.024	27	209,564
20.60.20	?	3,380.584	N	82.120	200	94,197
20.60.30	N	1,438.178	N	13.085	200	10,736
25.55.10	?	3,197.404	N	183.198	200	124,643
25.55.20	N	630.904	N	17.206	200	11,102
25.55.30	N	600.167	N	7.010	200	4,896

Table 2. Summary statistics for $k = 20$

	SAT4J		**Algorithm 1**	
Output	Mean	Variance	Mean	Variance
Satisfiable	60.30	11570.72	43.73	585.10
Unsatisfiable	1708.04	896901.01	28.62	1360.75

Table 3. Experimental test results for $k = 24$

	SAT4J		**Algorithm 1**			
Instance	Output	CPU Time (s)	Output	CPU Time (s)	Users	Patterns
30.10.10	Y	15.88	Y	1,596.07	35	1,351,463
30.10.20	?	2,044.95	N	295.33	240	83,153
30.10.30	N	1,406.89	N	122.50	240	23,201
35.5.10	?	2,867.81	N	2,651.70	240	682,217
35.5.20	?	2,416.26	N	326.75	240	66,962
35.5.30	N	133.53	N	74.61	240	12,296
40.0.10	Y	42.57	Y	639.62	33	472,122
40.0.20	?	2,172.18	N	233.77	240	40,080
40.0.30	N	989.68	N	44.69	240	7,646
30.30.10	Y	4.11	Y	2,666.69	41	2,505,089
30.30.20	?	2,487.96	N	380.47	240	96,073
30.30.30	?	2,506.33	N	119.31	240	22,237
35.25.10	Y	277.69	Y	842.69	43	493,585
35.25.20	?	3,307.86	N	369.22	240	78,041
35.25.30	?	2,782.98	N	85.79	240	14,700
40.20.10	?	2,610.55	N	1,878.97	240	394,284
40.20.20	?	2,596.38	N	283.73	240	47,707
40.20.30	N	3,269.74	N	65.44	240	10,521
30.50.10	?	4,298.72	Y	7,151.83	99	2,582,895
30.50.20	N	197.10	N	604.01	240	175,348
30.50.30	N	1,004.96	N	194.36	240	36,095
35.45.10	?	3,911.16	N	1,106.80	240	243,817
35.45.20	?	3,141.84	N	251.76	240	46,863
35.45.30	?	2,642.05	N	74.00	240	13,959

Table 4. Test results for $k \in \{16, 20, 24\}$

				SAT4J		**Algorithm 1**	
Steps k	Min c_1	Sums $c_1 + c_2$	Output	Number	Mean Time	Number	Mean Time
16	5, 10	20, 40, 60, 80	Sat	38	3.98	38	1.27
			Unsat	28	408.20	28	0.77
20	15	20, 40, 60, 80	Sat	19	60.30	19	43.73
			Unsat	22	1,708.04	29	28.62 (34.47)
			Unknown	7		0	
24	30	40, 60, 80	Sat	4	85.06	5	1,436.27 (2,579.38)
			Unsat	6	1,166.98	19	184.27 (482.27)
			Unknown	14		0	