Knowledge Representation and Reasoning

Coursework - Workflow Satisfiability Problem

Charlie Wilkins

psychw@nottingham.ac.uk
14322520

November 16, 2024

Contents

1 Introduction

The Workflow Satisfiability Problem (WSP) is defined by Crampton et. al [3] as "the problem of determining whether there exists an assignment of users to workflow steps that satisfies the policy", where the "policy" is a set of access-control rules dictating which users can be assigned to which steps. The WSP is notable for its ability to model real-world scenarios involving workflow scheduling [3]. It is known to be NP-Hard [4], and NP-Complete in the case that certain constraints are included (including the Separation-of-Duty constraint modelled here) [7].

OR-Tools is "an open source software suite for optimization"[1], developed by Google and implemented as a library in multiple programming languages including Python. Crucially, the library offers a module known as the CP-SAT Solver intended for use in Constraint Programming (CP). Given that the WSP has proven tractable to CP [1], the CP-SAT Solver is highly applicable in this case.

As such, this study focuses on the implementation of a solver for the WSP in Python using the CP-SAT Solver provided by the OR-Tools suite. Python was chosen due to existing knowledge of the library in this language, as well as due to the specification of the problem in this case requiring complex string manipulation to generate problem instances - Python is known to be an effective and well-documented language for string manipulation[2].

The study finds that a solver can be created using Python and CP-SAT that is effective at handling all five of the constraints specified in this case (see Formulation section) without any known flaws or errors, and is capable of determining whether the initially-found solution (if there is one) is unique. However, the system struggles with large problem instances, to the point of failing to terminate after half-an-hour or more in some cases. The cause of this is fairly well-understood (see Evaluation section), however it is a key limitation which would surely need to be fixed as a matter of priority were work on the system to continue.

2 Formulation

The specification of WSP in this case called for five constraints to be implemented. Here, we will discuss each of these constraints in turn, giving the mathematical formulation used as well as some acknowledgement of how it was implemented in the Python program.

The specification also called for the problem to be able to tell if the solution it had found (in the case that it had judged the problem instance *sat*) was unique or not. This will be discussed in the final part of this section.

At the heart of the solver are two sets of variables and an assignment:

- The set $S = \{s_1, s_2 ... s_n\}$ of steps or tasks to be assigned.
- The set $U = \{u_1, u_2 ... u_n\}$ of users to whom steps can be assigned.
- The assignment $\pi : S \to U$ of steps to users, e.g. $\pi(s_1) = u_1$ denotes that the step s_1 is assigned to the user u_1.[3] We can say that a problem instance is *sat* if there exists a valid assignment $\pi(s)$ for all $s \in S$.

With that defined, we can move on to discuss the formulation of each constraint.

2.1 Authorisation

An Authorisation constraint consists of a user $u \in U$ and an authorisation list $A \subseteq S$. It states that u is only permitted to perform tasks in A. This is implemented in the solver according to the following mathematical formula:

[1] Taken from Google's official page for the software here: https://developers.google.com/optimization

[2] See e.g. https://towardsdatascience.com/tips-for-string-manipulation-in-python-92b1fc3f4d9f

[3] Note that in the program, this is represented by the array $assignment[]$, e.g. $assignment[0] = 1$

$$\pi(s) \neq u \qquad u \in U, \forall s \notin A, A \subseteq S, s \in S \tag{1}$$

Quite simply, we add a constraint for every step s that is **not** in the authorisation list A stating that the user u cannot be assigned that step.

2.2 Binding of Duty

An instance of the Binding of Duty constraint supplies two steps s_1 and s_2, and states that they must be carried out by the same user. This is formulated simply, like so:

$$\pi(s_1) = \pi(s_2) \qquad s_1, s_2 \in S \tag{2}$$

In plain language, the assignments of s_1 and s_2 must be the same.

2.3 Separation of Duty

Separation of Duty is, essentially, the opposite to Binding of Duty: given two steps s_1 and s_2, they cannot be assigned to the same user:

$$\pi(s_1) \neq \pi(s_2) \qquad s_1, s_2 \in S \tag{3}$$

Again in plain language, the assignments of s_1 and s_2 must be different.

2.4 At-Most-k

At-Most-k is a constraint stating that for a set of steps $T \subseteq S$, the maximum amount of users assigned must be no more than k, in other words, the size of the "team" working on T can be no bigger than k. This can be expressed using set cardinatlity:

$$\begin{aligned} &uniqueUsers(T) <= k \\ &\qquad where\ uniqueUsers(T) = |removeDuplicates(X)| \\ &\qquad\qquad where\ X = \pi(t) \forall t \in T \qquad k \in \mathbb{N}, T \subseteq S \end{aligned} \tag{4}$$

In other words, $uniqueUsers()$ returns the amount of unique users in U assigned to a subset of S. This proved somewhat more difficult to implement in OR-Tools, due to the fact that most operations are not evaluated at "solve-time" i.e. the moment that the solver is called. For example, the code snippet...

$$len(set([1, 2, 3, 1]))$$

...which normally would return 3, will not behave similarly if called on CP-SAT variables.

As such, what is used in the program is a Boolean CP-SAT variable stating whether or not, for each user, they are assigned to a task in the given subset. Additionally to this, linking users to tasks in this way require "constraint channeling"[4], forcing the creation of more arbitrary variables.

The operation of the solver on this constraint is mathematically very similar to the formula given above, as since *true* Booleans in a set can be "counted" in Python, we are able to effectively "mock" the $uniqueUsers()$ function and arrive at the same value. However, the large amount of variables created by this method is likely one cause of the system's poor performance (see Evaluation section).

[4] https://developers.google.com/optimization/cp/channeling

2.5 One-Team

The One-Team constraint is the most complex of the constraints the solver implements. Provided is a set of steps followed by an arbitrary number of sets of users, known as "teams", with the constraint being that all the given steps must be carried out by members of the same team. In other words, given the steps s_1 and s_2, and the teams $\{u_1, u_2\}$ and $\{u_3, u_4\}$, $\pi(s_1) = u_3$ and $\pi(s_2) = u_4$ is a valid assignment, but $\pi(s_1) = u_1$ and $\pi(s_2) = u_3$ is not. Mathematically, this constraint can be formulated as follows:

$$\exists x' \in X.x' = true$$
$$where\ \forall x \in X, \forall T \in Ts, x = AllInTeam(S', T) \qquad S' \subseteq S, (\forall T \in Ts) \subseteq U, |Ts| = |X| \quad (5)$$

Simply, we create a Boolean variable x for each team, whose value is set by some predicate $AllInTeam()$, which returns true if all of a given set of steps are assigned to members of a given set of users. If at least one such x is true, then we can say that the specified steps have been assigned to one of the specified teams.

However, this constraint once again proves much harder to implement using OR-Tools than to specify mathematically. Once again, the issue can be found in the lack of possible operations in OR-Tools - Python provides the powerful *in* keyword to test if an element is in a list, so an easy solution might seem to run something like this:

$$\begin{aligned}
&allInTeams = []\\
&for\ team\ in\ teams:\\
&\quad stepsInTeam = [(step\ in\ team)\ for\ step\ in\ steps]\\
&\quad allInTeam = (stepsInTeam == len(steps))\\
&\quad allInTeams.append(allInTeam)
\end{aligned}$$

However, the *in* keyword cannot be used at solve-time with OR-Tools. As such, the implementation of the $AllInTeam$ predicate and the building of the X set once again required costly use of extra Boolean variables due to constraint channeling and a requirement to check individual tasks against individual team members.

2.6 Solution Uniqueness

The final operation that requires mathematical expression is not strictly a constraint, however it was an optional requirement of the solver. Namely, that the solver was able to determine whether or not a solution was unique. This determination is carried out by adding a new constraint to the solver:

$$\exists s.\pi(s) \neq \pi'(s) \qquad s \in S \quad (6)$$

Where π represents the original set of assignments, and π' represents an arbitrary new set. Thus, there must be at least one different assignment for π' to be valid and therefore for a different solution to exist. If the output after adding this constraint is *unsat*, then we know that the original solution was unique.

3 Implementation

As has been discussed elsewhere, the solver is implemented in Python using OR-Tools. OR-Tools requires Python3 specifically, so this must be installed. OR-Tools can then be installed via pip, and the program can be run normally, taking as a command-line argument a target file to solve.

The source code appears twice in the repository, once under the src/ directory and once in CourseworkTester/. When run from src/, it will locate the specified problem instance in the instances/ directory - this functionality is removed in the other directory for compatability with the provided testing script. For more details, view the included README.md file.

As explained in the above Section, the solver completes all the requirements listed, including the one to test whether a solution is unique. It fulfils this by re-running the solver, with an additional constraint that at least one assignment is different to the ones originally provided. Thus, if this proves *unsat*, a second solution does not exist.

There are no known bugs in the solver, and when run with the provided testing script no tests fail. However, its performance is far from perfect. The solver struggles to solve difficult instances, in some cases failing to terminate after half-an-hour or longer. The likely reasons for this will be discussed in the following Section.

4 Evaluation

As stated above, testing suggests that the logic of the program is free from bugs and flaws. As such, this section will focus primarily on the performance of the solver. The below table demonstrates the performance of the solver on the 19 provided Example Instances, and the results it produces:

Example	**Time (s)**	**Sat**	**Unique**
1	0.007	yes	no
2	0.008	no	N/A
3	0.007	yes	yes
4	0.008	no	N/A
5	0.020	yes	yes
6	0.018	no	N/A
7	0.014	yes	yes
8	0.012	no	N/A
9	0.170	yes	no
10	0.088	yes	no
11	2.094	yes	no
12	2.099	yes	no
13	0.604	no	N/A
14	0.023	no	N/A
15	0.036	no	N/A
16	>30m	N/A	N/A
17	>30m	N/A	N/A
18	>30m	N/A	N/A
19	>30m	N/A	N/A

This table, particularly the last four entries, demonstrates the severe performance issues in the solver. It is notable that none of these four contain One-Team constraints, but they do contain large amounts of At-Most-k rules. Large amounts of these are also found in examples 11 and 12, which also show notably poor performance compared to those around them. However, we are forced to assume also that large quantities of One-Team constraints would produce the same effect, due to the implementation of both of these constraints featuring the same limitation.

This limitation is simple to summarise: number of variables. Due to the above-mentioned constraint-channeling requirement for "tying" variables together in OR-Tools, the Python implementations of the mathematical formulations used for the constraints both generate huge numbers of Boolean variables, which are required to test relations between tasks and users. The

fact that this high variable count is the cause of the performance issues can be observed in other ways: the program shows notable slowdown even before solving the instance when constructing the rules for At-Most-k and One-Team constraints, and it also uses an abnormally high amount of RAM for a simple Python script - as much as 1.5 gigabytes.

Potential alleviations of this issue can be found in the next Section, so to close this one we will summarise the performance of the solver as follows: it is mathematically an effective and correct solver, but suffers from severe performance issues caused by a limitation of the chosen library requiring the generation of excessive amounts of variables.

5 Conclusions & Future Work

This section will feature a brief summary of the above Report, highlighting the key facts about the solver and the way in which it was built. From these conclusions, we will then be able to discuss several key options for "next steps" that could be taken were development on the solver to continue.

5.1 Conclusions

The solver that has been developed is implemented in Python, using Google's OR-Tools suite to provide a basis for Constraint Programming. Constraint Programming is a proven solution for the Workflow Satisfiability problem [1] [5], and is in general a useful tool for problems of this type as it allows precise mathematical formulation of the means by which solutions are reached [6].

We have discussed the formulation in this case, for five constraints as well as the extra requirement of determining whether a found solution is unique. We have seen that the solver is mathematically correct, however suffers from severe performance issues. We have further discussed the cause of these issues, which is likely the high count of variables required by OR-Tools to model the mathematical formulation we have developed.

As such, any following work on the solver must focus on solving these performance issues. Discussion of some potential solutions can be found below.

5.2 Future Work

In this final part of the Report, we will discuss a number of "next steps" for the solver, with a particular focus on potential solutions to the aforementioned performance issues.

The first and most obvious solution would be to change the mathematical formulation to one which, in OR-Tools, requires fewer variables. There are several likely avenues for different formulations. but a particularly likely one is found in the fact that the WSP is known to be Fixed-Parameter Tractable (FPT) [2]. Put simply, this means that, for certain constraints or families of constraints, the complexity of the problem can be tied to a single Fixed Paramter which the solution is built around. In the case of the WSP, this fixed parameter is generally taken to be the number of tasks [2]. With this in mind, a more sophisticated mathematical formulation might well be found based on the principles of FPT.

A more prosaic "programmer's solution" also naturally presents itself. This is the introduction of multi-threading into the system. When running the solver on a difficult instance, it is easy to observe a single CPU core running at 100% capacity while others sit idle. Multi-threading therefore seems an obvious fix, or at least an obvious improvement. In OR-Tools, a CP-SAT Solver can easily be set to run in a multi-threaded mode, by setting the number of "workers" spawned by the solver. Each worker starts in a different location in the search space, meaning that each one has more-or-less the same chance of finding a solution[5]. This is a somewhat naive,

[5]See discussion by OR-Tools dev Laurent Perron here: https://groups.google.com/g/or-tools-discuss/c/i3TF_Szuz_k

non-deterministic approach to multi-threading, but would still likely provide some performance improvement. However, given a brief, non-formal test, multi-threading the solver on a difficult problem instance still failed to produce a solution within half-an-hour, and also caused all four activated CPU cores to run at 100% capacity. Additionally, it slightly increased the RAM used by the program. As such, given the known issues in the program, this is likely only a small part of any potential solution.

Once the performance issues had been solved, possibly with reference to the above discussion, there are several other ways that the solver could be improved, but the most obvious would be the addition of new constraints. Examples of other constraints occur throughout the literature (see e.g. [2] [5]), and integrating these into the solver would provide a more sophisticated model of the real-world scenarios the WSP aims to solve.

6 Bibliography

References

[1] T. Benoist, E. Gaudin, and B. Rottembourg. Constraint programming contribution to benders decomposition: a case study. In *8th International Conference on Principles and Practice of Constraint Programming*, 2002.

[2] D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Iterative plan construction for the workflow satisfiability problem. *Journal of Artificial Intelligence Research*, 2014.

[3] J. Crampton, A. Gagarin, and G. Gutin. On the workflow satisfiability problem with class-independent constraints for hierarchical organizations. *ACM Transactions on Privacy and Security*, 2016.

[4] J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Algorithms for the workflow satisfiability problem engineered for counting constraints. *Journal of Combinatorial Optimization*, 2016.

[5] D. Karapetyan, A. J. Parkes, G. Gutin, and A. Gagarin. Pattern-based approach to the workflow satisfiability problem with user-independent constraints. *Journal of Artificial Intelligence Research*, 2019.

[6] P. Kotecha, M. Bhushan, R. Gudi, S. Narasimhan, and R. Rengaswamy. Constraint programming based input signal design for system identification. In *11th International Symposium on Process Systems Engineering*. 2012.

[7] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Transactions on Information and System Security*, 2010.