CNF Instances from the Software Package Installation Problem*

Josep Argelich[1] and Inês Lynce[2]

[1] INESC-ID
Rua Alves Redol 9, Lisboa, Portugal
josep@sat.inesc-id.pt

[2] INESC-ID and IST/TU Lisbon
Rua Alves Redol 9, Lisboa, Portugal
ines@sat.inesc-id.pt

Abstract

The complexity of software package installations poses a number of key challenges. Very often it is not possible to install all the desired packages and a variety of solutions can be thought of. Recent work has suggested the optimization of a user-provided objective function. The present work aims at characterizing the CNF instances resulting from the software package installation problem. Starting from a basic configuration, different installations have been generated by adding a subset of the most popular packages. We have studied the hardness of satisfiable and unsatisfiable instances, as well as the unsatisfiable cores of the unsatisfiable instances and the maximum number of packages that can be installed when the instance is unsatisfiable. The obtained results shed additional light on optimization criteria to solve installation conflicts.

1 Introduction

A complex software system is made of components which are related either implicit or explicitly. Free and Open Source Software (FOSS) distributions are an example of such systems, which are developed by distinct individuals or entities who share their work through fast and reliable connections. FOSS represents the most revolutionary paradigm in software engineering that is opposed to traditional systems which have a centralised and closed development.

FOSS distributions are usually based on deployment units known as packages. The relationships between these packages are called dependencies and are expected to be handled in a consistent and efficient way. The management of package dependencies can be seen either from the server/distributor side or from the client side. In the former, the main goal is to assure

*This research is partially funded by European project Mancoosi (FP7-ICT-214898) and by FCT project Bsolo (PTDC/EIA/76572/2006).

Proceedings of the 15th International RCRA workshop (RCRA 2008):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Udine, Italy, 12–13 December 2008

that package repositories are consistent, and eventual errors and inconsistencies are handled in an effective, fast and automatic way. In the latter, the main goal is to install new packages (and its dependencies) on a system of already installed packages.

Previous work in the area has applied Boolean satisfiability (SAT) based tools to ensure the consistency of repositories as well as to solve consistently package installation. In the context of the EDOS project [3], researchers have introduced SAT-based tools to support distribution editors [11]. These new tools are automatic and ensure completeness, which makes them more reliable than ad-hoc and manual tools. In addition, the OPIUM tool [13] is also a complete SAT-based tool[1] that optimizes a user provided objective function (such as smaller packages should be preferred to larger ones).

This paper describes research work developed in the context of the Mancoosi project [4]. Mancoosi is an European research project in the Seventh Research Framework Programme (FP7) of the European Commission. While the predecessor project EDOS had focused on tools for the distribution editor, the Mancoosi project aims at developing tools for the system administrator by (i) developing mechanisms that provide rollbacks of failed upgrade attempts, allowing the system administrator to revert the system to the state before the upgrade and (ii) developing better algorithms and tools to plan upgrade paths based on various information sources about software packages and on optimization criteria.

Our work concerns the installation of new packages on a system of already installed packages. Although this work focus on a problem already studied in the past [13], the objectives are different. We argue that the instances of this problem have not yet been properly studied, and this should be done regardless the optimization criterion provided by the user. Moreover, studying such instances can provide important information to drive future research work.

This paper is organised as follows. Next section introduces the software installability problem, and is followed by the description of the encoding of this problem into conjunctive normal form (CNF). Next, the experimental evaluation section analyses the generated CNF instances with respect to its satisfiability. Unsatisfiable instances are then further analysed in terms of unsatisfiable subformulas and maximum satisfiability. Finally the paper concludes.

[1] Actually, OPIUM is based on Pseudo Boolean Optimization (PBO). PBO is a SAT based technology that expresses constraints over Boolean variables as linear inequalities with integer coefficients that are extended with an optimization function.

2 The software installability problem

The software installability problem can be naturally encoded as a SAT problem where problem instances are encoded using the CNF format. A CNF formula is a conjunction of clauses, where a clause is a disjunction of literals and a literal is either a Boolean variable (a positive literal) or its negation (a negative literal). The SAT problem consists in finding whether there is an assignment to the Boolean variables such that the formula is satisfied. For a formula to be satisfied all of its clauses must be satisfied. For a clause to be satisfied at least one of its positive literals must be assigned to true or alternatively at least one of its negative literals must be assigned to false.

The software installability problem may be easily encoded into a CNF formula, where the Boolean variables determine whether a package is installed.

The constraints of a package are defined by its constraints, also called metadata, and the constraints of a software distribution are defined by the constraints of all the packages in the software distribution. Basically, we have two types of constraints between packages: *dependencies* and *conflicts*. The *dependencies* of a package p are the packages needed by p in order to make it work properly. Therefore, we have to satisfy all the dependencies of p if we want to install package p in our system. The *conflicts* of a package p are the packages conflicting with p, and cannot be installed in the system at the same time. Therefore, conflicting packages may not be present in the system in order to install p.

The constraints of each package can be defined by a tuple (p, D, C), where p is the package, D are the dependencies of p, and C are the conflicts of p. D is a set of *dependency clauses*. Each *dependency clause* is a disjunction of packages. A dependency clause is satisfied if at least one of its packages is installed in the system, and the dependencies D are satisfied if all the dependency clauses are satisfied. C is a set of packages conflicting with p. The conflicts C are satisfied if none of the packages in C is installed in the system. Figure 1 shows a real example of a package constraints.

```
Package: 915resolution
Architecture: i386
Version: 0.5.2-9
Depends: libc6 (>= 2.3.6-6), vbetool (>= 0.6.1)
Conflicts: 855resolution
```

Figure 1: Metadata for package `915resolution`.

3 CNF encoding for the software installability problem

A *software distribution* is a set of package constraints. We can represent the package constraints of a software distribution as a Boolean CNF formula as follows:

- Each package of a software distribution with n packages is represented by a Boolean variable. The set of variables is therefore $\{x_1, x_2, \ldots, x_n\}$, where each variable x_i with $1 \leq i \leq n$ represents a package p_i. If x_i is assigned to *true*, it means that the package p_i is installed in the system; if x_i is assigned to *false*, it means that the package p_i is not installed.
- For each package constraints (p, D, C) in the distribution, we can encode its dependencies using one Boolean clause for each dependency clause in D. A dependency clause c for package p_i, that corresponds to the Boolean variable x_i with $1 \leq i \leq n$, is encoded with the following Boolean clause: $\neg x_i \vee c$.
- For each package constraints (p, D, C) in the distribution, we can encode its conflicts using one Boolean clause for each package in C. A conflict between a package p_i and package $p_j \in C$ with $1 \leq i, j \leq n$ is encoded with the following Boolean clause: $\neg x_i \vee \neg x_j$.

Example 1 Given a set of package constraints $S = \{(p_1, \{p_2, p_5 \vee p_6\}, \emptyset), (p_2, \emptyset, \{p_3\}), (p_3, \{p_4\}, \{p_1\}), (p_4, \emptyset, \{p_5, p_6\})\}$, its encoded CNF instance is the following:

$$\begin{gathered}
\neg x_1 \vee x_2 \\
\neg x_1 \vee x_5 \vee x_6 \\
\neg x_2 \vee \neg x_3 \\
\neg x_3 \vee x_4 \\
\neg x_3 \vee \neg x_1 \\
\neg x_4 \vee \neg x_5 \\
\neg x_4 \vee \neg x_6
\end{gathered}$$

Example 1 shows the CNF encoding for a package distribution represented by the set of package constraints S. Note that this formula can be easily satisfied assigning all the propositional variables to *false* (nothing is installed), as all the clauses of the formula have at least one negative literal.

4 Experimental evaluation

In order to make the experimental evaluation as real as possible, we used a well known FOSS distribution as a source of packages metadata: the Debian

distribution. Although using the instances from [13] would be desirable, these instances were generated by a company and are not publicly available.

4.1 The Debian distribution

Debian [2] is a FOSS distribution based on GNU/Linux. The number of packages in Debian has grown considerably in the recent years, and nowadays its main/stable repository reaches more than 17,000 packages.

The Debian packages metadata for all its versions and architectures since March 13th of 2005 are available at *http://snapshot.debian.net*. For the experimental evaluation, we used a package metadata dated as of May 1st of 2008 from a stable version of Debian[2].

4.2 CNF instances

In order to generate CNF instances for the installability problem on a basic installation, we first need to extract the dependencies and conflicts for each package in the Debian distribution. The clauses that encode the constraints of a Debian distribution can be extracted from its metadata using the Ceve parser [1] from the EDOS project [3]. Once we have the dependencies and conflicts encoded as a CNF instance, we have to add the basic installation to it, i.e. the unit clauses that represent the installed packages in a basic installation of the Debian distribution. After applying the unit clause rule [6], also referred to as *Boolean constraint propagation* when applied iteratively, we get a CNF instance that represents the constraints of a Debian distribution taking into account that it has the basic packages installed. This is the Debian *basic installation* instance that we used to generate the instances for the installability problem described in the following sections. This instance has around 17,000 variables and 25,000 clauses.

4.3 Experimentation with the installability problem

The installation of new packages is one of the actions performed more often by the users. The aim of this experimentation is to show the likelihood that a user tries to get an invalid installation. The instances used for the experimentation with the installability problem extend the basic installation as follows:

1. Pick a subset of packages I to install from the distribution, using the Debian Popularity Contest[3] statistics, to take into account the most installed packages in Debian distributions. This way, a package

[2] http://snapshot.debian.net/archive/2008/05/01/debian/dists/stable/main/binary-i386/Packages.gz

[3] http://popcon.debian.org/

installed 4,000 times has two times more possibilities of being picked than another package that has been installed 2,000 times.

2. For each package $p_i \in I$, add a unit clause, with a positive literal x_i, to the basic installation instance. These unit clauses encode the packages that we want to install in the system.

If the resulting instance is satisfiable, then the set of packages I is installable on a system with the basic installation. For the experimental investigation, we generated 7,000 instances. These instances refer to installations ranging from 10 to 700 additional packages, with an interval of 10, where for each specific number of packages were generated 100 instances.

The experimentation with the installability problem[4] is shown in Figure 2. We used Minisat [7] version 2 to solve the instances. The left plot shows the mean CPU time needed by Minisat to solve all the instances of each set, and the right plot shows the ratio of satisfiable instances over the total number of instances. Both plots report values with respect to the number of packages to install.

We can appreciate that the time needed to solve the instances is always very small, and also decreases when the number of packages increases. A more detailed analysis was performed distinguishing the time to solve satisfiable and unsatisfiable instances. We observed that satisfiable instances take around 0.04 seconds and unsatisfiable instances take around 0.015 seconds, regardless the number of packages to be installed. Therefore, despite all the instances being easy to solve, satisfiable instances are slightly harder than unsatisfiable instances. This behaviour can be explained with the experiments described in the following section.

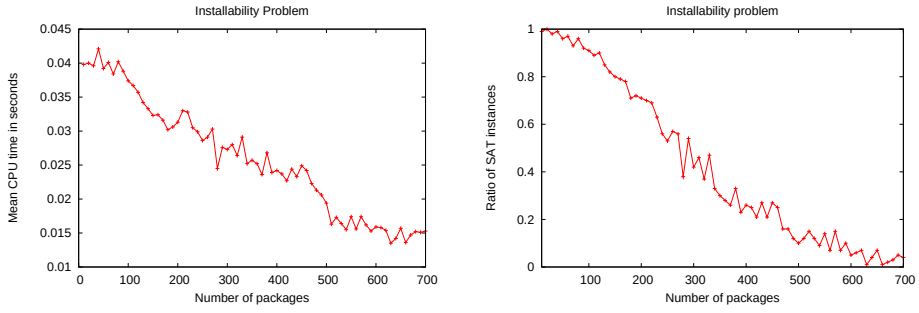

Figure 2: Installability problem. Mean CPU time to solve the instances (left plot). Ratio of satisfiable instances over the total number of instances (right plot).

[4]Experimentation performed on a Intel Core 2 Duo CPU P8400 2.26GHz with 4GB of RAM.

4.4 Experimentation with unsatisfiable cores

Given the significant number of unsatisfiable instances, it should be interesting to investigate what makes these instances unsatisfiable. One measure commonly used to characterize unsatisfiable formulas is the analysis of *unsatisfiable subformulas*. Given an unsatisfiable formula φ, a formula $\varphi' \subseteq \varphi$ that is still unsatisfiable is said to be an unsatisfiable subformula. Unsatisfiable subformulas are also named *unsatisfiable cores* or simply *cores*.

More interesting is the analysis of *minimal* unsatisfiable subformulas (MUS). An unsatisfiable subformula φ' is said to be minimal if and only if $\forall_{c \in \varphi'}\ \varphi' - \{c\}$ is *satisfiable*. Hence, the removal of any clause from a minimal unsatisfiable subformula makes the formula to become satisfiable. The shortest explanation for unsatisfiability is the *smallest minimal unsatisfiable core*.

The knowledge of the number and the size of the minimal unsatisfiable cores will give us a hint about the number of explanations of infeasibility and the number of clauses that are involved in it.

In what follows, whenever we refer to unsatisfiable cores we will assume that those unsatisfiable cores are minimal.

Example 2 Consider the following formula with five clauses:

$$\omega_1 = \neg x_1 \vee x_2$$
$$\omega_2 = x_1 \vee x_2$$
$$\omega_3 = \neg x_2$$
$$\omega_4 = x_3 \vee \neg x_2$$
$$\omega_5 = \neg x_3 \vee \neg x_2$$

This formula is unsatisfiable and this may be explained by two minimal unsatisfiable cores. The first one contains clauses ω_1, ω_2 and ω_3. The second one contains clauses $\omega_1, \omega_2, \omega_4$ and ω_5. The unsatisfiable core containing clauses ω_1, ω_2 and ω_3 is the smallest minimal unsatisfiable core.

In order to analyze the unsatisfiable cores contained in the unsatisfiable instances from the installability problem, we have run the CAMUS tool [10] on each of them. CAMUS is able to provide for a given formula all of its minimal unsatisfiable cores.

Figure 3 gives the median number of unsatisfiable cores for each set of 100 instances, as well as the median size of the smallest and largest unsatisfiable cores. The median number has been chosen instead of the mean number for being the most consistent value. It has been observed that huge variations on these values may occur. Most often, for instances with the same number of packages we observe that the number and sizes of the unsatisfiable cores are quite similar, with only a few exceptions. But given that these exceptions can differ as much as one order of magnitude, they would have a strong impact in the mean value.

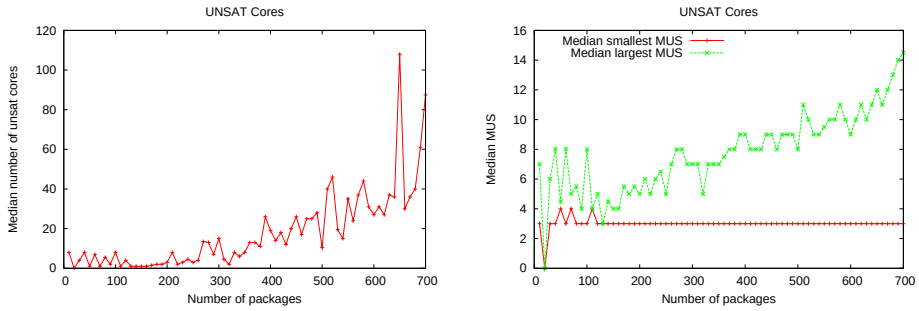

Figure 3: Unsatisfiable installability problem. Median number of unsatisfiable cores (left plot). Median smallest and largest unsatisfiable cores (right plot).

Not surprisingly, the plot on the left shows that the number of unsatisfiable cores tends to increase as the number of packages to be installed increases. In other words, the number of potential conflicts increases as the number of packages to install increases.

On the other hand, from the plot on the right we conclude that the size of the smallest unsatisfiable core does not depend on the number of packages to install. With only a very few exceptions, the smallest unsatisfiable core has size 3. An explanation for this number is straightforward: you have two unit clauses representing two different packages to be installed that clash with one binary clause that expresses that these two packages cannot be both installed. Nonetheless, much larger minimal unsatisfiable cores can be found, specially as the number of packages to be installed increases. Indeed, the size of the largest unsatisfiable core tends to increase as the number of packages to install increases. Having large unsatisfiable cores means that unsatisfiability is explained by a large number of clauses, which in practice represent more elaborated conflicts (when compared with those represented with only three clauses).

We can clarify with these experiments the behaviour pointed out in the last sentence of the previous section. Unsatisfiable instances are slightly easier to solve than satisfiable instances because as we increase the number of packages to install, we also increase the number of unsatisfiable cores. This makes that most of the instances with a higher number of packages to install can be shown to be unsatisfiable using only unit propagation.

Overall, these results indicate that in general there are many unsatisfiable cores of different sizes and therefore one may choose between a variety of explanations for infeasibility.

4.5 Experimentation with the maximum installability problem

The analysis of the unsatisfiable cores characterises the infeasibility of the problem but somehow not in a complete way. For example, the number of minimal unsatisfiable cores does not suffice to determine the number of clauses to be removed to achieve satisfiability. If there are no clauses belonging to more than one core, then the number of clauses to be removed corresponds to the number of cores. Otherwise the number of clauses to be removed may be smaller as removing a clause that belongs to more than one core will destroy the cores that clause belongs to. For example, the formula given in Example 2 has two minimal unsatisfiable cores but it suffices to remove one clause (either ω_1 or ω_2) to make the formula satisfiable.

The goal of this section is to show the maximum number of packages we can install for several installation profiles, and the mean CPU time needed for installing each set of packages. Obviously, if a problem instance is satisfiable then all packages may be installed. On the other hand, for an unsatisfiable problem instance at least one package cannot be installed. Hence, from this experimentation, we can get the minimum number of packages that we need to remove from the original installation profile, in order to make the installation satisfiable.

The installability problem becomes now a maximum satisfiability (Max-SAT) instead of simply a satisfiability problem. This new problem will be called maximum installability problem. The traditional Max-SAT problem consists in maximizing the number of clauses that can be satisfied. The partial Max-SAT problem has additional expressiveness by distinguished between hard clauses (that must be satisfied) and soft clauses (for which the traditional Max-SAT problem applies). The instances used for the experimentation with the maximum installability problem have the same clauses as the instances that were used for the installability problem. In addition, the clauses of the basic installation are said to be hard clauses, whereas the unit clauses encoding the packages to be installed are said to be soft clauses.

The experimentation with the maximum installability problem[5] is shown in Figure 4. The left plot shows the mean CPU time needed by MSUnCore 4.0 [12] to solve all the instances of each set. We also tried other solvers like MiniMaxSat [8] or W-MaxSatz [5] but the performance was not as good as with MSUnCore with this benchmark. This comes as no surprise as MSUnCore is known for being particularly competitive on solving structured real-world instances.

In Figure 4 (left) we may observe that the time needed by MSUnCore to solve the instances increases as we increase the number of packages to install, but even for the harder set of instances it does not exceed 1.2 seconds. The right plot shows the mean number of packages we have to remove from

[5]Experimentation performed on a Intel Xeon CPU 5160 3.00GHz with 4GB of RAM.

the selected packages to install, in order to make the installation satisfiable. When we want to install only a few packages, usually we do not need to remove any package from the installation, as the instances are satisfiable. When the number of packages to install is increased, the number of packages to remove in order to make the installation satisfiable also increases. An interesting point is that, in general, by only removing 3 packages in installations of up to 700 additional packages, we can make the installations to become satisfiable.

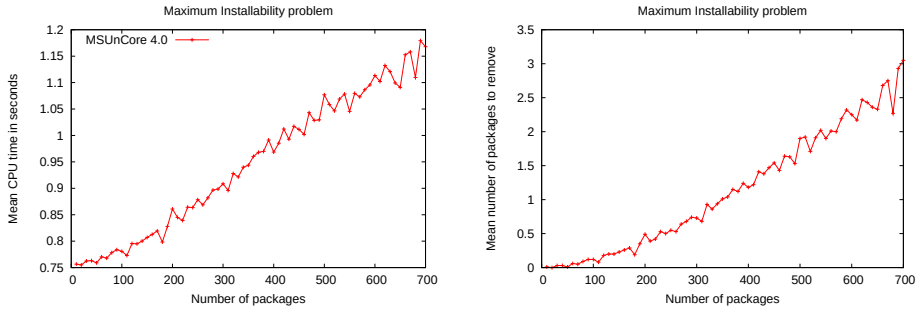

Figure 4: Maximum installability problem. Mean CPU time to solve the instances (left plot). Mean number of packages to remove to make the installation satisfiable (right plot).

5 Conclusions and future research directions

The management of a complex package-based software system implies an efficient handling of dependencies between packages. FOSS distributions represent an additional challenge with respect to this issue, due to the development being decentralised and being made by different individuals. As a consequence, the installation of new packages by the user is not always feasible.

In this work we have investigated instances from the software package installation problem. A complete characterization of these instances in terms of their hardness may provide additional light to solve inconsistencies. Experimental results show that the installability problem for FOSS distributions, even though being a NP-complete problem, is in general easy to solve. Hence, this problem should not have a negative impact in hypothetical users. Note, however, that the whole process associated with the installability problem may consume some time, but this is due to other parts of the process, such as reading the dependencies information from the repository.

As future work we plan to incorporate to our encoding some user preferences on which packages are preferred to be uninstalled in case not all the

packages can be installed. Additional requirements should include information about packages that the user wants to delete or update. This work will extend previous work in the area [13].

Finally, we should point out that the results of this investigation may also inspire the development of other SAT-based tools used in similar contexts [9].

References

[1] Ceve parser. http://www.edos-project.org/xwiki/bin/view/Main/Ceve.

[2] Debian GNU/Linux. http://www.debian.org.

[3] EDOS project. http://www.edos-project.org.

[4] MANCOOSI project. http://www.mancoosi.org.

[5] J. Argelich and F. Manyà. An improved exact solver for partial Max-SAT. In *Proceedings of International Conference on Nonconvex Programming: Local & Global Approaches, NCP-2007, Rouen, France*, pages 230–231, 2007.

[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[7] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT-2003, Santa Margherita Ligure, Italy*, pages 502–518. Springer LNCS 2919, 2003.

[8] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, 172(2–3):204–233, 2008.

[9] D. Le Berre and A. Parrain. On SAT technologies for dependency management and beyond. In *First Workshop on Analyses of Software Product Lines*, September 2008.

[10] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[11] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE'06*, pages 199–208, Washington, DC, USA, 2006. IEEE Computer Society.

[12] J. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design, Automation and Test in Europe (DATE)*, pages 408–413. IEEE, 2008.

[13] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th international conference on Software Engineering, ICSE'07*, pages 178–188, Washington, DC, USA, 2007. IEEE Computer Society.