Comparison of PBO solvers in a dependency solving domain

Paulo Trezentos
ISCTE / Caixa Mágica
paulo.trezentos@iscte.pt

Linux package managers have to deal with dependencies and conflicts of packages required to be installed by the user. As an NP-complete problem, this is a hard task to solve. In this context, several approaches have been pursued. *Apt-pbo* is a package manager based on the apt project that encodes the dependency solving problem as a pseudo-Boolean optimization (PBO) problem. This paper compares different PBO solvers and their effectiveness on solving the dependency solving problem.

1 Introduction

Software installation is the process of installing programs assuring that specifically required software is pre-installed and that defined actions are taken before or after the copy of the files into the file-system [22, 24]. Although this is a common problem among Microsoft and Open Source Operating Systems (GNU/Linux, BSD,...) [25] we will focus on the later ones, since a progress in this field would be applicable to all environments, including applications like Eclipse or Firefox [15].

The installation process comprises retrieving the package, solving the software dependency tree, retrieving and installing the software dependencies and finally installing the package and executing the associated install scripts [8].

The dependency graph represents the software dependencies and sub-dependencies needed for a package to work properly after installation [5]. The restrictions imposed by the graph may have no solution (for instance, due to broken dependencies), only one solution, or several solutions. Criteria such as the minimum number of packages or freshness can be defined to rank the solutions in terms of their quality. Finding a solution consists in defining the sub-set of packages that meets the dependency requirements. This process is called dependency solving. One approach to dependency solving is to encode the problem as a pseudo-Boolean optimization (PBO) problem using existing solvers for finding the optimal solutions. This approach is applied in *apt-pbo*, a meta-installer tool based on *apt* that will be described in this paper.

This paper is organized as follows. In section 2 we provide background information about PBO. Section 3 depicts the *apt-pbo* tool and its architecture. Section 4 presents empirical results of experiments conducted with the several solvers. Finally, in section 6 are presented the concluding remarks.

2 Background

Pseudo-Boolean Optimization (PBO) is a special case of Integer Linear Programming (ILP) where variables are Boolean. For this reason, it is often called 0-1 ILP. This is the case of our package selection problem, where a package being present in the final solution can be easily encoded as a Boolean variable being assigned value 0 or 1.

I. Lynce and R. Treinen (eds.): Workshop on
Logics for Component Configuration (LoCoCo 2010)
EPTCS 29, 2010, pp. 23–31, doi:10.4204/EPTCS.29.3

Pseudo-Boolean functions are a generalization of Boolean functions with a mapping $\mathscr{B}^n = \{0,1\} \mapsto \mathbb{R}$ [2, 7]. Pseudo-Boolean functions in polynomial form are widely used in optimization models in different areas like statistics, computer science, VLSI design and operations research.

A PBO problem can be formally defined as follows [2]:

$$\begin{aligned} &\text{minimize} \sum_{j \in \mathbb{N}} c_j \cdot x_j \\ &\text{subject to} \sum_{j \in \mathbb{N}} a_{ij} l_j \geq b_i \\ &\qquad x_j \in \{0,1\}, a_{ij}, b_i, c_j \in \mathbb{N}_0^+, i \in M \\ &\qquad M = 1, ..., m \end{aligned} \tag{1}$$

where each c_j is a non-negative integer cost associated with variable x_j, $j \in \mathbb{N}$ and a_{ij} denotes the coefficients of the literals l_j in the set of m linear constraints, being a literal a Boolean variable or its negation.

Recent algorithms for solving the PBO problem integrate features from recent advances in Boolean satisfiability (SAT) and classical branch and bound algorithms.

3 System Overview

3.1 Architecture

Apt is a meta-installer widely used in Linux distributions. However, apt solves dependencies in a very straightforward way and in a large number of occurrences fails to deliver a solution.

The *Apt-pbo* application [23] belongs to a new generation of meta-installers that not only are capable of finding a solution but are flexible to allow the user to customize which solution fits best the needs.

The architecture of *apt-pbo* has different hooks to integrate modules. This architecture allows exchange of modules. For example, changing the PBO solver being used is an extremely easy task.

In our tests, the overhead of the external calls is not significant since the number of iterations is extremely low.

Figure 1 depicts a typical installation flow of apt-pbo.

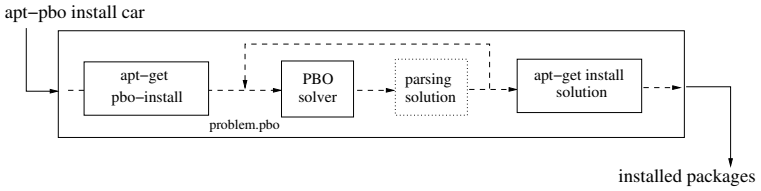

Figure 1: High level processing flow of *apt-pbo*

The *apt-pbo* application is called with the operation *install* and the desired package as arguments, which map the usage of *apt-get*.

The components of the figure have the following role:

- **apt-get pbo-install**: we have a modified version of *apt-get* installation software. Apt is one of the most used meta-installers and is adopted by Linux distributions like Debian, Ubuntu and Caixa Mágica. The modifications introduced by the author created a new method called *pbo-install* which, given a specific package, calculates the dependency tree and writes the PBO encoding. The PBO encoding is composed of a set of PB-constraints and an objective function.
- **PBO solver**: the *problem.pbo* formula is solved by the PBO solver. We have used and tested different solvers as will be detailed in section 4.
- **parsing solution**: apt-pbo has a module that parses the solver solution and, if necessary, establishes a new iteration with *apt-get pbo-install*.
- **apt-get install solution**: when the final package set solution is reached, the user is asked for permission and the removal and installation of packages are performed using apt and dpkg / rpm.

3.2 PBO encoding

As presented in the previous section, apt-pbo pbo-install encodes the the problem as a Pseudo-Boolen Optimization.

This encoding has two parts: constraint and objective function definition-

Constraints definition

In a pseudo-Boolean formula, variables have Boolean domains and constraints are linear inequalities with integer coefficients.

Encoding relations of the dependency tree as constraints is a straightforward task. The following translations will be used:

- *Installation*: p_1[1] is the package that we want to install: $p_1 \geq 1$.
- *Dependency*: p_1 depends on x_1 should be represented as $x_1 - p_1 \geq 0$. This means that installing p_1 implies installing x_1 as well, although x_1 may be installed without p_1. If p_1 also depends of y_1, we should add $x_1 - p_1 \geq 0$.
- *Multiple versions*: if a package p_1 requires the installation of a package x having different versions, for example x_1 and x_2, then we should encode the requirement that installing package p_1 requires installing either package x_1 or package x_2. Hence, such requirement may be encoded with constraint $x_1 + x_2 - p_1 \geq 0$.
- Conflicts: if a package has an explicit conflict with other package, for instance if y_3 conflicts with x_1, then this conflict is encoded as $x_1 + y_3 \leq 1$. Remember that there is a conflict for each pair of different packages corresponding to the same unit.

Objective Function Definition

In the Objective Function we define what we plan to minimize.

Two approaches might be followed: we minimize a single criterion (e.g. the number of packages) ou multiple-criteria.

We will start by presenting single criterion.

Minimizing Package Removal

To minimize the number of removed packages, even if newer packages exist, one should use the following *objective function*, where $PI'_1..PI'_N$ is the set of packages already installed:

[1] For simplicity, the tuple representation of a package as $(p,2)$ will be now represented as p_2

$$f_1(P) = \min(1 - PI'_1) + ... + (1 - PI'_N)$$

In order to minimize the objective function, the solver will try to set variable PI_i to 1 which will imply not removing installed applications.

Minimizing the Number of Installed Packages

In this case, the total number of packages installed in the system is to be minimized. Having $P1..PN$ as the new packages targeted to be installed - either existent or new - the objective function will be:

$$f_2(P) = min\ P1 + ... + PN$$

Maximizing the Freshness of Packages

Consider $P1_1..P1_{k1}$ to be different software versions or releases of package $P1$. Also, consider $v(P1_1)$ to be the normalized distance (a constant, for the purposes of the PBO problem) between the package $P1_1$ and the newest version present in repository R. Then the optimization function is:

$$\begin{aligned} f_3(P) = & min\ (P1_1 * v(P1_1) + ... + P1_{K1} * v(P1_{K1})) + \\ & (PZ_1 * v(PZ_1) + ... + PZ_{KN} * v(PZ_{KN}))... \end{aligned}$$

The value of $v(Pi_{Ki})$ is zero if the package is the newest in the repository.

Multicriteria optimization

However, in the real world installing a package follows multiple criteria and even if one is more important than the others that can lead to non-desired solutions.

Trying to satisfy different criteria when finding the set of packages for a software installation falls in the multicriteria decision making (MCDM) set of problems [11].

Apt-pbo integrates the different objective functions of the previous section as a multiobjective problem (MOP):

$$min\ (f_1(P), f_2(P), f_3(P))$$

with P as the available packages and f_1, f_2 and f_3 as the existent objective functions.

The multiobjective problem is solved transforming it into a single objective problem through *weighted sum scalarization*.

Apt-pbo uses the following coefficients, λ, representing the overall utility for the user: Removal Cost - W_r (weight given to the cost of a removal of a package), Presence Cost - W_p (weight given to the presence of a new or an already installed package) and Version Cost - W_v (weight representing the cost of having an older version in the solution when a newer exists).

The objective function is then defined as:

$$min\ (W_r \cdot f_1(P) + W_p \cdot f_2(P) + W_v \cdot f_3(P))$$

4 Experimental Results

We performed experiments on a large set of different repositories, packages and systems hosted at O2H Lab cluster of 164 Xeon CPU cores[2] with Linux installed in Xen virtual system machines and inside a *chroot* environment. In what follows we report the results of this evaluation.

The goal of the experiments performed was to simulate the installation of software in a Linux environment and test the different PBO solvers against the same criteria.

A comparison of SAT and PBO solvers has been performed extensively through international competitions and benchmarks [3, 18, 9]. Since the solving algorithm can benefit greatly from the structure of the problem, it was considered important to evaluate different PBO solvers on solving this problem. As mentioned in section 3, *apt-pbo* is structured in a modular form, thus allowing the replacement of one PBO solver by another compatible solver.

For testing purposes, four solvers were considered:

- *minisat+* [10]: from the same authors of *minisat*, a well known SAT solver, and actually based on *minisat*, *minisat+* encodes PB-constraints into SAT.
- *bsolo* [12]: bsolo is a PBO solver, which was first designed to solve instances of the Unate and Binate Covering Problems (UCP/BCP) and later updated with pseudo-Boolean constraints support.
- *wbo* [17]: from some of the same authors of bsolo, participated in the PB'09 competition.
- *opbdp* [2]: an implementation in C++ of an implicit enumeration algorithm for solving PBO.

Besides the solvers mentioned above, Pueblo [21] was also considered but not included since the only available version is dynamically linked and the libraries needed are old and not available in the testing infra-structure. Nevertheless, an old Linux system was installed (Debian Etch) and some ad-hoc tests were performed with Pueblo. These tests revealed that Pueblo has a poor performance for this specific type of problems and no further efforts to port Pueblo were made.

The tests consisted of 1,000 installation of packages over a Debian Lenny Linux system. Two different scenarios were tested: "conservative" and "aggressive".

The weights in the objective function (section 3.2) are the same in both scenarios adopting a balanced configuration between updates and removals.

The difference are trhe active repositories. In the "conservative" scenario only Lenny repositories were active (main and updates). In the "aggressive" the *Sid* (development version) and *Backports* repositories were also present. Table 1 summarizes the differences between scenarios. In fact, 12,000 more packages were present in the "aggressive" scenario and more than the double of the total space accounted by *apt-pbo* for mapping packages, dependencies and conflicts.

4.1 Aggressive scenario

Table 2 summarizes the results of the evaluation performed in the context of the aggressive scenario.

As we can observe, both *wbo* and *bsolo* are able to solve all the instances but *wbo* has a better performance (4.45 seconds on average per transaction). *Minisat+* comes in third place, not only with a lower number of instances solved, 355, but also with a poorer performance, taking on average more than two minutes to solve a problem. *wbo* has also a smaller standard deviation than *bsolo*. The *average time* consists in the time, in average, per installation transaction.

[2]The infra-structure is integrated in the ADETTI / ISCTE centre of RNG Grid.

Table 1: Characterization of packages - conservative and aggressive scenarios

Measures	Conservative	Agressive
Total package names	30014	42007
Total distinct versions	24100	51337
Total dependencies	147085	326891
Total Provides mappings	5146	10962
Total dependency version space	602k	1358k
Total space accounted for	7284k	14,9M

Table 2: PBO solvers benchmarking - Aggressive scenario

	bsolo	wbo	minisat+	opbdp
# Solved	1,000	1,000	355	47
# Timeouts	0	0	645	953
Average time	00:07.79	00:04.45	02:30.16	07:16.49
Standard deviation	00:02.83	00:01.19	01:29.33	35:13.02

Figure 2 compares *wbo* and *bsolo* varying the number of the installed packages per transaction. There is a smooth growth by *wbo* and a more unstable line of growth in a much more unpredictable fashion by *bsolo*. Since *minisat+* and *opbdp* had a significant number of timeouts, they were not included in the graph.

4.2 Conservative scenario

In the conservative scenario, development repositories are not active and therefore there is a much more steady environment for dependency solving.

In this case, the four solvers were able to find the solutions before the timeout of 150 seconds. In fact, on average they performed under 3 seconds with the exception of *minisat+*.

Table 3: PBO solvers benchmarking - Conservative scenario

	wbo	bsolo	minisat+	opbdp
# Solved	1000	1000	1000	1000
# Timeouts	0	0	0	0
Average time	00:02.6	00:02.62	00:06.22	00:02.55
Standard deviation	00:00.8	00:01.1	00:01.4	00:01.1

Figure 3 depicts the size of the problem *vs* time. Although on average opbdp performs better than minisat+, the figure shows that as the size of the problem grows opbdp is more sensible to peaks and outliers.

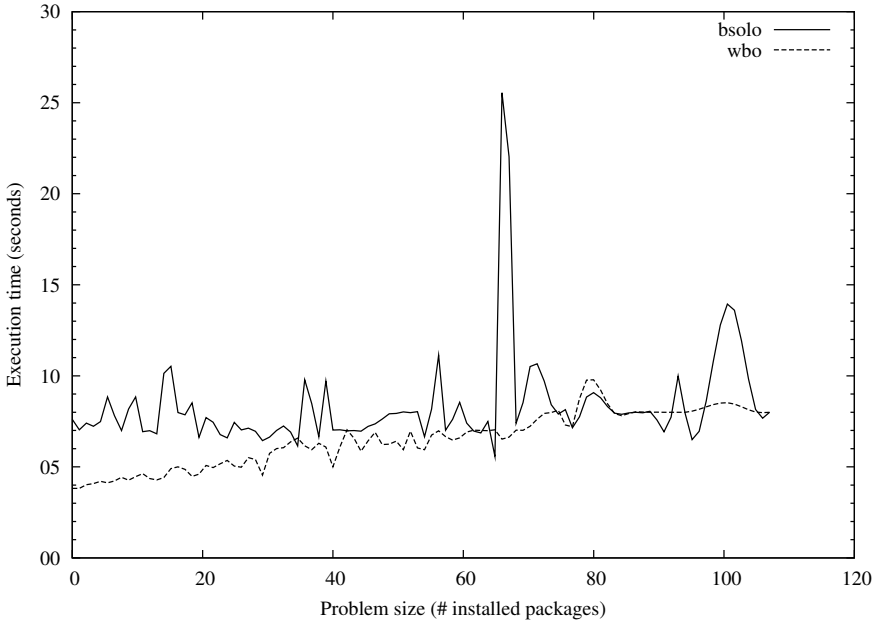

Figure 2: PBO solvers graph - Aggressive scenario

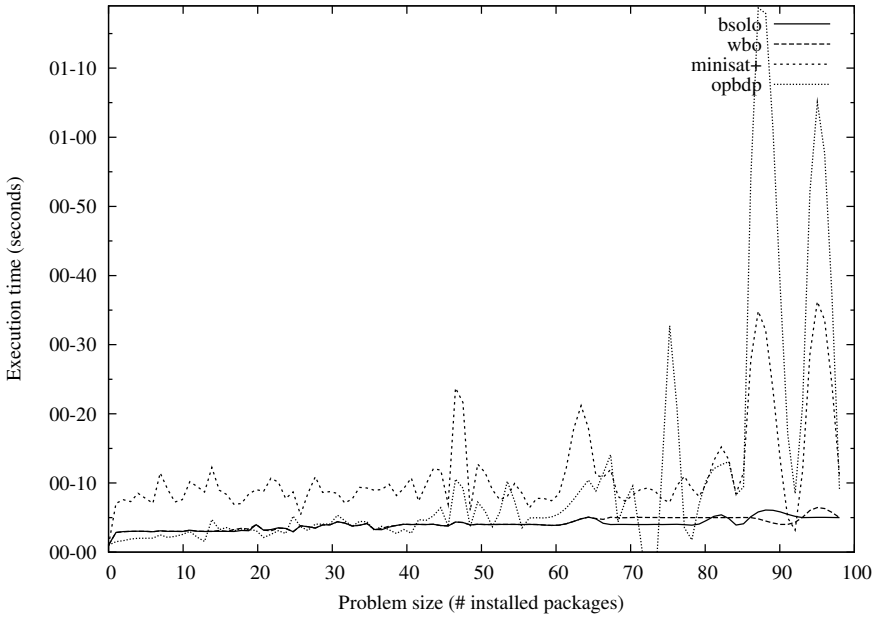

Figure 3: PBO solvers graph - Conservative scenario

5 Related Work

The use of Boolean Satisfiability (SAT) [4] for solving the dependency problem has first been proposed in the context of the EDOS FP6 project [16, 6] which had impact in other research efforts [13]. An alternative formulation using constraint programming techniques has been described in [20], including the use of different heuristics for improving the quality of the solution found.

6 Conclusions

The PBO solvers evaluated follow different theoretical approaches and therefore are expected to have different results. However, some results of the tests performed are interesting to recall: *wbo* is the solver that performed better in both scenarios and with a more stable behaviour. *bsolo* has also interesting results in both scenarios.

Although *wbo* is the solver with better time results, there are other aspects to take in account: *minisat+* is open source and can be enhanced to address more difficult problems as the presented ones in the aggressive scenario. Being open source is a critical point to a Linux distribution that might adopt such a tool.

Future work will consist in analysing, jointly with the authors of the PBO tools, possible enhancements of the tools as a result of this evaluation. Another direction for future work is to study the possibility of the solvers returning a non-optimal solution when the timeout is reached.

Finally, this article can be extended to study other solvers such as SCIP [1] and boolean optimization engines such as SAT4JPB [14] or MsUnCore [19].

6.1 Acknowledgments

Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898.

References

[1] Tobias Achterberg (2004): *SCIP - a framework to integrate Constraint and Mixed Integer Programming*. Technical Report 04-19, Zuse Institute Berlin. `http://www.zib.de/Publications/abstracts/ZR-04-19/`.

[2] P. Barth (1995): *A Davis-Putnam based enumeration algorithm for linear pseudo- boolean optimization*. Technical Report, Technical Report MPI-I-95-2-003, Max Planck Institute.

[3] Daniel Le Berre, Olivier Roussel & Laurent Simon (2009). *International SAT 2009 competition*. Http://www.satcompetition.org/2009/.

[4] Armin Biere, Marijn J. H. Heule, Hans van Maaren & Toby Walsh, editors (2009): *Handbook of Satisfiability*, *Frontiers in Artificial Intelligence and Applications* 185. IOS Press.

[5] Li Bixin (2003): *Managing dependencies in component-based systems based on matrix model*. In: *NETObjectDays'03*.

[6] Jaap Boender, Roberto DiCosmo, Berke Durak, Xavier Leroy, Marc Lijour, Fabio Mancinelli, Tova Milo, Mario Morgado, David Pinheiro, Rafael Suarez, Ralf Treinen, Paulo Trezentos, Jérôme Vouillon & Tal Zur. *WP2 - Formal management of software dependencies - Deliverable 2.2*. Http://edos-project.org/xwiki/bin/download/Main/Deliverables/edos-wp2d2.pdf.

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

[8] Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio & Stefano Zacchiroli (2009): *Towards maintainer script modernization in FOSS distributions*. In: IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems, ACM, New York, NY, USA, pp. 11–20.

[9] Heidi Dixon & Peter Barth (2009). *PB'09 competition: available results*. Http://www.cril.univ-artois.fr/PB09/results/results.php?idev=28.

[10] N. Eén & N. Sörensson (2006): *Translating Pseudo-Boolean Constraints into SAT*. Journal on Satisfiability, Boolean Modeling and Computation 2, pp. 1–26.

[11] M. Ehrgott (2000): *Multicriteria optimization*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag.

[12] F. Heras, V. Manquinho & J. Marques-Silva (2008): *On Applying Unit Propagation-Based Lower Bounds in Pseudo-Boolean Optimization*. In: Proceedings of International FLAIRS Conference.

[13] D. Le Berre & A. Parrain. (2008): *On SAT technologies for dependency management and beyond*. ASPL .

[14] Daniel Le Berre. *Sat4j: a reasoning engine in Java based on the SATisfiability problem (SAT)*. `http://www.sat4j.org`.

[15] Daniel Le Berre & Pascal Rapicault (2009): *Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution*. In: IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems, ACM, New York, NY, USA, pp. 21–30.

[16] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy & Ralf Treinen (2006): *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*. In: ASE, IEEE Computer Society, pp. 199–208. Available at `http://doi.ieeecomputersociety.org/10.1109/ASE.2006.49`.

[17] Vasco Manquinho, João Marques-Silva & Jordi Planes (2009): *Algorithms for Weighted Boolean Optimization*. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing.

[18] Vasco Manquinho & Olivier Roussel (2009). *PBO Competition 2009*. Http://www.cril.univ-artois.fr/PB09/.

[19] Joao Marques-Silva (2009): *The MSUNCORE MAX SAT Solver*. Technical Report, CASL/CSI, University College Dublin. `http://www.csi.ucd.ie/staff/jpms/pubs/pubsite/msu-booklet09.pdf`.

[20] Sébastien Mouthuy, Luis Quesada & Grégoire Doom (2006): *Search heuristics and optimisations to solve package installability problems by constraint programming*. Technical Report, Department of CSE, UC Louvain. Available at `http://www.info.ucl.ac.be/$\sim$pvr/report_ingi2800_C.pdf`.

[21] Hossein M. Sheini & Karem A. Sakallah (2006): *Pueblo: A hybrid pseudo-boolean SAT solver*. Journal on Satisfiability, Boolean Modeling and Computation 2, p. 2006.

[22] Judith A. Stafford & Alexander L. Wolf (1998): *Architecture-level dependence analysis in support of software maintenance*. In: ISAW '98: Proceedings of the third international workshop on Software architecture, ACM, New York, NY, USA, pp. 129–132.

[23] Paulo Trezentos (2009). *Apt-pbo Homepage*. Http://aptpbo.caixamagica.pt/.

[24] Marlon Vieira & Debra Richardson (2002): *Analyzing Dependencies in Large Component-Based Systems*. In: ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering, IEEE Computer Society, Washington, DC, USA, p. 241.

[25] Il-Chul Yoon, Alan Sussman, Atif Memon & Adam Porter (2008): *Effective and scalable software compatibility testing*. In: ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis, ACM, New York, NY, USA, pp. 63–74.