# Search heuristics and optimisations to solve package installability problems by constraint programming

Sébastien Mouthuy[*], Luis Quesada[†], Grégoire Dooms[‡]

August 2, 2006

**Abstract**

The aim of this work is to present a constraint programming formulation of the package installability problem. The package installability problem deals about determining whether a package of a repository could be installed or not. This problem is known to be NP-complete.Efficient search heuristics will be presented, leading to few failures. Our solver is able to determine whether all packages of the Debian distribution can be installed in less than 6 minutes. At the end, implementation of optimisation criteria are also presented.

## 1 Introduction

The problem we want to solve is to check whether all packages of a repository can be installed. Such repositories are common in Open-Source software distributions like Debian, Fedora,... These distributions support a set of software ( one piece of software is called a package). This set is called a *repository*. The question is to check whether a repository is *trimmed* as defined in [1], this means all packages of a repository can be installed. This property is very important in the maintenance of distribution like Debian. People used to softwares like apt-get could think that such problems are very easy to solve. But it has been shown [1] that the package installability problem is NP-complete, by reducing it to 3SAT. Tools like apt-get are thus incomplete. As we will see later, real repositories can be checked efficiently by using smart search heuristics.

We will use the same notation as [1]. A $package$ is a couple $(unit, version)$, e.g. $gnome - 2.6$. A unit is the name of the package like $gnome$, $apache$, ... The set of all packages in a repository will be denoted by $P$.

[*]sebastien.mouthuy@student.uclouvain.be
[†]luque@info.ucl.ac.be
[‡]dooms@info.ucl.ac.be

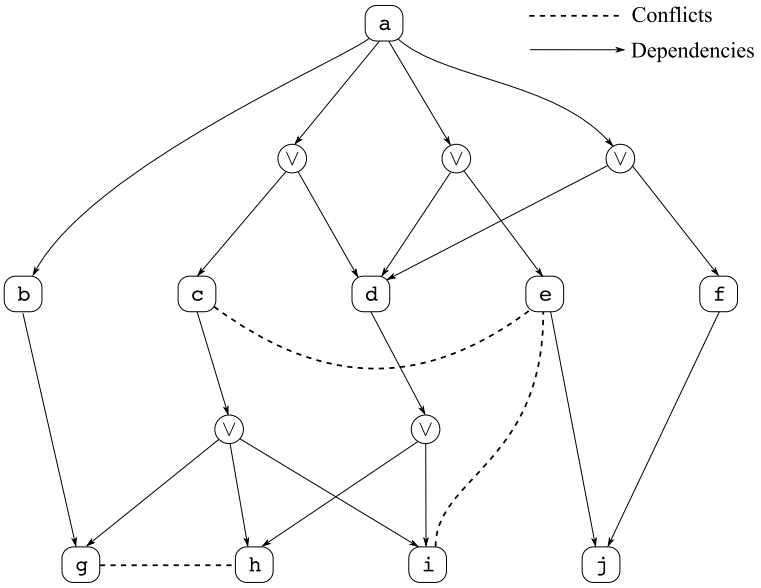


Figure 1: Dependencies between packages of a repository

Dependencies in a repository will be specified by a function $D : P \rightarrow \wp(\wp(P))$ .The dependencies of a package $a$ will be defined by a set of sets of packages: $D(a) = \{d_1, \ldots, d_n\}$ with $d_i \in \wp(P), \forall i : 1 \leq i \leq n$. This means all $d_i$ should be satisfied in order to install $a$. $d_i$'s are called disjunctive dependency as only one package $p \in d_i$ should be installed in order for $d_i$ to be satisfied.

Conflicts are defined as couples of packages $(P_1, P_2)$ that cannot be installed together. The set of conflicts in a repository is denoted by $C \subseteq P \times P$.

A repository is defined as a tuple $R = (P, D, C)$. $\pi$ will usually denote the package we want to install. For convenience the set of units in a repository will also be defined: $units(P) = \{u \mid \exists v : (u, v) \in P\}$.

These dependencies are illustrated in FIG-1[1]. The V nodes represents a disjunctive dependency. For example, in order to install package d, we should install either package h or package i. On this example, $D(a) = \{\{b\}, \{c, d\}, \{d, e\}, \{d, f\}\}$.

# 2 Preprocessing

An entire distribution contains many packages (more than 33'000), so it is worth preprocessing them. We can easily compute a set of packages that will not be of interest for a given problem. For example, if we want to install a unique package $\pi$ having not many dependencies, a lot of packages don't need to be considered because they would never satisfy a required dependency.

An easy solution to remove some not-relevant packages is to compute the set of packages $\Delta(\pi)$ that could be required in order to install a given package $\pi$. This

[1] This figure comes from [1]

can be achieved easily by traversing the dependency graph from $\pi$ along the dependency edges. All reached nodes are of interest, all others can be discarded without eliminating any solution of the problem. In FIG-1, we have $\Delta(c) = \{g, h, i\}$.

The *dependency closure* $\Delta(\pi)$ was introduced and defined formally in [1]. They show $\Delta(\pi)$ always exists and can be computed by traversing the dependency graph. In our implementation, the dependency graph like the one in FIG-1 is traversed by DFS and each node not reached at the end of the traversal is discarded.

# 3 Formulation of the package installability problem as a CSP

In this section we will formulate the package installability problem as a constraint satisfaction problem (CSP). We will use the same notations as in the introduction.

## 3.1 Inputs and solution

Here are the inputs of the CSP:

$R = (P, D, C)$**:** A repository

$(u^*, v^*) \in P$ **:** the unit and the version of the package for which we want to check whether it could be installed

The solution our solver should return is a function $install : U \rightarrow \mathcal{Z}$ indicating for each unit the version we should install in order to install $(u^*, v^*)$ while respecting all dependencies constraints.

## 3.2 Constraints of the CSP

The solution should satisfy different constraints.

First, we need to constraint the solution to respect all dependencies:

$$\forall (u, v) \in P, install(u) = v \Rightarrow \forall d \in deps((u, v)), \exists (u', v') \in d \mid install(u') = v'$$

.

Second, conflicts should be taken into account:

$$\forall \Big( (u_1, v_1), (u_2, v_2) \Big) \in C, \quad install(u_1) = v_1 \Rightarrow \ install(u_2) \neq v_2$$

Third the version of the unit we want to install should be imposed:

$$install(u^*) = v^*$$

# 4 Description of our current solution

From the problem formulation above, we can express the installability problem as a CSP using finite domain (FD) variables. In this formulation, we are using a FD variable $V_u$ per unit $u$ in the repository. This variable contain the version of this

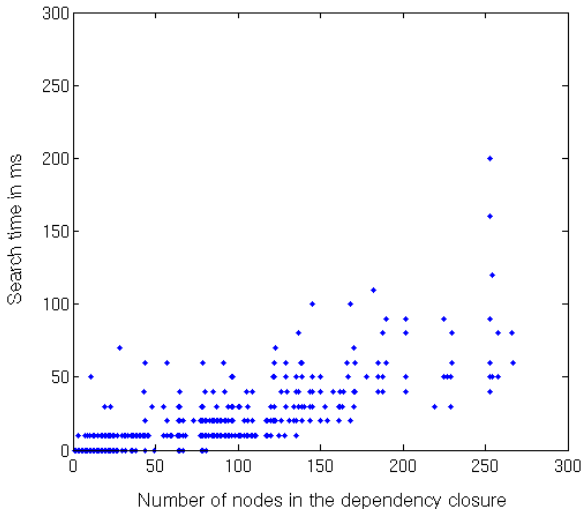


Figure 2: Time taken in function of the number of nodes after preprocessing

unit that should be installed[2]. $0$ means that no version of the package should be installed. These variables implement the $install$ function.

This CSP problem was implemented in Oz [2]:

$$V_{u^*} = v^*$$

$$\forall\, package\ (u,v) \in P, \quad \forall \left((u_k^1, v_k^1), \ldots, (u_k^r, v_k^r)\right) \in D((u,v)),$$
$$\{\text{FD.impl } \text{V}_\text{u} =: \text{v } \{\text{FD.disj}\{\ldots\{\text{FD.disj}\text{V}_{\text{u}_\text{k}^1} =: \text{v}_\text{k}^1 \text{V}_{\text{u}_\text{k}^2} =: \text{v}_\text{k}^2\}\ldots\}\ \text{V}_{\text{u}_\text{k}^\text{r}} =: \text{v}_\text{k}^\text{r}\}\ 1\}$$

$$\forall\, conflict\ \Big((u_1, v_1), (u_2, v_2)\Big) \in C,$$
$$\{\text{FD.impl } \text{V}_{\text{u}_1} =: \text{v}_1\ \text{V}_{\text{u}_2} \backslash =: \text{v}_2\}$$

This Oz implementation can find an installation solution to all packages of the entire Debian distribution (33200 different packages) in less than 20 minutes on a notepad[3]. FIG-2 shows the search time in millisecond taken to find a solution in function of the number of nodes in the dependency closure (ie after preprocessing).

It would be interesting to create artificial repository that contain packages very hard to install. Ratio $\frac{\#Solution}{Searchspace}$ of current distributions (Debian, Mandriva,Fedora,...) is not as low as it could be; that is the reason why tools like apt-get can install most of the packages while being polynomial in time. As shown in [1], the temperature[4] of the installation problem under its SAT formulation is well under $4.2$, meaning current repositories are not as hard as they could be. Evolution of the time taken by the algorithms on these fake repositories would allow us to measure the real limitations of our solvers.

Another improvement that enhances performances is to keep in memory all packages of all installability solutions found so far. If a given package $p$ is con-

[2]Using integer for the version number of packages has been justified in [1]

[3]with a Pentium Intel Centrino 1,5GHz and 512 MB memory

[4]ratio $m/n$ where $m$ is the number of clauses and $n$ the number of variables in a SAT problem

tained in the solution of the installability of another package $p'$, then we don't need to check whether $p$ can be installed or not, as we already know that at least one solution exists. This small improvement can decrease the time required to analyse the entire Debian distribution to less than 6 minutes[5].

# 5 Searching Heuristic

The searching strategy is very important in constraint programming and has a great impact on performances.

Our searching strategy is based on the fact that we should decide to install a package only if it is required by another package we must install. So we begin branching on packages that are close to the package $\pi$ we want to install. A BFS traversal of the dependency graph from $\pi$ is done and all packages are concateneted in a list $L_p$ in the order they are visited. During searching, the algorithm branch on the first undetermined variable it finds in $L_p$. Pseudo code of this algorithm is shown in Algo-1 to Algo-4. $Solution$ is a data structure containing all the FD variables of the problem. $Solution.U$ is the FD variable representing the version of unit $U$ to install (or $0$ if we should not install any version of unit $U$). The function $processDeps$ is introduced here as its definition will change below.

**Algorithm 1** $branch(U_{to\ install}, V_{to\ install})$

$L_p \leftarrow computeBFSTraversal([(U_{to\ install}, V_{to\ install})], [])$
$distribute(L_p)$

**Algorithm 2** $computeBFSTraversal(Ls, Acc)$

**case** $Ls$
**of** $nil$ **then**
  **return** $Acc$
**of** $H|T$ **then**
  **if** $isVisited(H)$ **then**
    $computeBFSTraversal(T, Acc)$
  **else**
    $markVisited(H)$
    $Acc2 \leftarrow Append(Acc, [H])$
    $Lt \leftarrow Append(T, processDeps(D(H)))$
    $computeBFSTraversal(Lt, Acc2)$
  **end if**
**end case**

This searching strategy seems nearly optimal since we rarely observe failures when searching for a solution to an installation problem. only $3800$ packages made our search algorithm fail at least once, nevertheless our algorithm fails at most 8

[5] with a Pentium Intel Centrino 1,5GHz and 512 MB memory

**Algorithm 3** $processDeps(Ls)$

**return** $Ls$

**Algorithm 4** $distribute(L_p)$

**case** $L_p$
**of** $nil$ **then**
  **return**
**of** $(U, V)|T$ **then**
  **if** $isAssigned(U)$ **then**
    $distribute(T)$
  **else**
    **choice**
      $post(Solution.U = V)$
      $distribute(T)$
    **or**
      $post(Solution.U \neq V$
      $distribute(T)$
    **end choice**
  **end if**
**end case**

times for the worst package (the entire distribution was composed of 33220 different $(package, version)$ couples). Length of branches of the search tree leading to a failure is at most 2: when we make a wrong decision, it is detected after at most two other decisions. So this searching heuristic seems really efficient.

# 6 Optimisation

On the recommendations of Roberto Di Cosmo (head of the EDOS project), we decided to investigate further how we could optimize the solution our solver returns.

Several criteria over the solution returned could be of interest:

**Overall freshness:** measured by the number of packages to install that are the last version of their unit

**Size:** the number of packages that should be installed

**Difference from the current system:** the number of packages whose status should change (package installation, removal or upgrade)

**Minimum storage requirements:** finding the solution requiring the smallest amount of disk space

**...**

We focused on the two first optimisations: freshness and size of the solution returned. We wanted to find a trade-off between speed and optimality. Because of the huge number of solutions for a given problem, optimality could require too much

computation. The search heuristic was modified in order to find very good solutions while maintaining efficiency.

## 6.1 Freshness of the solution

In the search heuristic we presented in 5, we did a BFS traversal on the dependency tree. When analysing a disjunctive dependency, all the packages were added to the list $L_p$ without any preference. We modified this algorithm in order to first append to $L_p$ the packages that are up to date (their version is the latest available). For example, in order to install gnome, we should either install Xfree86 either Xorg. Imagine a repository containing two versions of each unit: Xfree86-3, Xfree86-4, Xorg-1 and Xorg-2. Then $Xfree86-4 \mid Xorg-2 \mid Xorg-1 \mid Xfree86-3$ would be appended to $L_p$. The pseudo code of the new searching heuristic is nearly the same as the one presented in section 5. Only the $processDeps(Ls)$ procedure must be redefined. Its specifications are shown in Algo-5.

**Algorithm 5** $processDeps(Ls)$

**Require:** $Ls$ is a list of packages
**Ensure:** $\forall i, j \mid i < j, isLatest(Ls[i]) \vee not\ isLatest(Ls[j])$

On the Debian distribution tested, there were 20'000 distinct unit. With this change in the searching heuristic, we tried to install the latest version of each unit and we looked for the ones that required to install some packages which were not the freshest. Only 390 packages required to install such packages.

## 6.2 Smallest number of packages to install

The search heuristic explained in section 5 could install more packages than needed. This is due to the fact that the branching algorithm try to install the first undetermined package it encounters in $L_p$. But in the case of a disjunctive dependency that can be satisfied by two different units, we would try to install one package of these units. Taking into consideration that the different units could have very different dependencies, this could lead to a significant number of useless packages installed. Another reason why we could install more packages than needed is that a package $P_d$ could be in $L_p$ because $P_d$ satisfies a dependency of another package $P_{d2}$. If we don't install $P_{d2}$, $P_d$ and all its dependencies would nevertheless be installed.

We changed the search algorithm in order to assign a value to a variable only if no other variable is assigned to a value that satisfies the disjunctive dependency. This solved the first point of the preceding paragraph. The difficulty of solving the second point is that we don't know a priori which packages we will install and thus we cannot build a minimal $L - p$ before branching. A concurrent approach is used to solve this easily: a port (see [3]) $Prt$ is created and its stream is used instead of a list $L_p$. A thread per unit $u$ is waiting for the corresponding variable $V_u$ to be set. Once a version is assigned to it, the thread appends to the stream of $Prt$ all the dependencies required by $V_u$. The element sent to $Prt$ are thus disjunctive dependencies: lists of packages. The initial value for $Prt$ is $[P_{to\ install}] \mid \_$. Pseudo code of this new search heuristic is shown in Algo-6 to Algo-7. The goal of $alreadySatisfied$ is to receive a list of packages $Ls$ as argument and to return the sublist of $Ls$ with

packages that can still be installed (this depends on choices made before in the search tree). $satisfied$ is returned if $Ls$ contains one package already installed. This means the dependency $Ls$ is already satisfied. if $alreadySatisfied$ returns $nil$ we fail (Algo-7 on line 5), because it means the dependency cannot be satisfied. The propagators should deduce it before getting there.

**Algorithm 6** $alreadySatisfied(Ls, Solution, Acc)$

```
case Ls
of nil then
   return Acc
of (U,V)|T then
   if Solution.U = V then
      return satisfied
   end if
   if V ∈ dom(Solution.U) then
      return alreadySatisfied(T, Solution, Append(Acc, [(U,V)]))
   else
      return alreadySatisfied(T, Solution, Acc)
   end if
end case
```

This two improvements can decrease the number of packages to install by a factor of two. In average, solutions found with this improvements contains 33% less packages to install than the solutions returned with the original search heuristic. The time decreases by 8%.

# 7 Future work

Some more preprocessing could allow us to compute an upper bound of how many packages need to be installed in order to install a given package $\pi$. This information could help us in branching more efficiently in the case of a disjunctive dependency.

It would also be useful to have a second criteria to sort packages satisfying a given dependency. Currently the procedure $processDeps$ shown in Algo-5 just put all the freshest packages at the top of $L_p$. But these freshest packages could also be sorted among them in function of another criteria . We could try first packages that seem to require the minimum number of other packages in order to be installed.

We would like also to create complex artificial repository to analyse the limitations of our solver. Future repositories could not be as simple as they are.

# 8 Conclusion

In this paper we presented a CSP formulation of the package installability problem. An implementation in Oz showed that contraint programming can be very efficient for this problem. Every package can be solved in less than 210 ms. The entire

**Algorithm 7** $distributeImproved(Ls, Solution)$

```
1: case Ls
   of H|T then
2:   case alreadySatisfied(H, Solution, nil)
     of satisfied then
3:     distributeImproved(T)
4:   of nil then
5:     fail
6:   of (U,V)|Ht then
7:     choice
8:       post(Solution.U = V)
9:       distributeImproved(T)
10:    or
11:      post(Solution.U ≠ V)
12:      distributeImproved(Ht|T)
13:    end choice
14:  end case
15: end case
```

Debian distribution can be solved in less than 6 minutes taking into account all the optimisation criteria's presented in this paper.

As shown in the previous sections, simple propagators can be used to formulate this problem. With current instances, design of good search is the most important point. This is due to the fact that current tools are polynomial in time and were implemented using heuristics. CSP implementations of this problem would show their real power if the complexity of repository increased. Then global constraints as $DomReachability$ could be of a great help in pruning early bad solutions.

The main advantage of using constaint programming for this problem is the flexibity of such approach. Adding additional constraints or optimisation criteria to the initial problem can be done easily.

## Aknowledgements

We would like to thank professor Peter Van Roy for his help and feedback all along the duration of this work. We would like also to thank Roberto Di Cosmo who answered to our questions concerning their current work on Open-Source repositories.

## References

[1] Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli, and Jérôme Vouillon. Maintaining large software distributions: new challenges from the foss era. In *FRCSS 2006: 1st International EASST-EU Workshop on Future Research Challenges for Software and Services*, 2006.

[2] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2004. Available at *http://www.mozart-oz.org/*.

[3] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.