APT 3.0 dependency solver

An orthodox approach to dependency solving, leading to a SAT solver comparable to DPLL.

Julian Andres Klode

APT Developer
Ubuntu Foundations @ Canonical Ltd

Contents

Introduction 1

Algorithm 5

Evaluation 6

Conclusion 6

Introduction

APT is the high-level package manager used in Ubuntu and Debian. It's job is to install software packaged in the .deb format, currently about 80 000 packages.

Historically, since its inception about 30 years ago, APT has been using an ad-hoc two solver approach to solve dependencies:

The first solver follows the dependency graph recursively, marking dependencies for install and conflicts for removal. There is no separation between the input to the solver and the state of the solver itself.

The second solver iteratively resolves any conflicts that remain between the choices following from the first solver using heuristics.

Having grown over 30 years in an ad-hoc fashion, the solver is not trivial to maintain, seemingly simple changes can cause problems (years) later. Particular issues are:

1. If a package depends on `foo=1` and we install `foo=2` instead, the package is being removed. Even if it is manually installed and foo is automatically installed.
2. Unsatisfiability is not always nicely explained

We intend to replace that ad-hoc dual solver approach with a methodical approach which, starting from the set of manually installed packages minus any packages that should be removed, calculates a final set of packages that shall be installed, using backtracking to handle conflicts.

Expected behavior from APT solvers

APT's solver is reasonably predictable in most cases as it is a greedy algorithm, a new solver should behave reasonably close to the existing solver that it does not work in ways that are unexpected.

The Mancoosi project (TBD: bib) and others employed general purpose optimization problem solvers to the space of dependency solving, given certain optimization goals such as "minimize the number of packages removed" and "minimize the number of changes made". While these resulted in "optimal" solutions, such solutions were not always the expected ones.

Case 1: The order of dependencies matters

For example, a certain desktop environment had a search engine with multiple storage backends. APT's greedy solver picked the first backend in the list, resulting in the prefered backend being installed, whereas alternative APT solvers from the Mancoosi project picked the less performant sqlite backend because it caused fewer packages to be installed.

This points to one of the most interesting differences in Debian packaging: If a package depends on `A | B`, the options are not equal. We should always install the first installable option unless another one has already been marked for install.

Overview of the new approach

Ignoring optional dependencies for now, the solver maintains two sets: 'needs' and 'rejects'. The 'needs' set determines the versions to be installed, the 'rejects' set determines the versions to not be installed. Versions not in either set are undecided.

In addition to those sets, the solver maintains a work queue of dependencies that need to be resolved, these dependencies are essentially represented as two fields: the version having the dependency, and all versions that can satisfy the dependency.

The working queue is a priority queue: We process items with less choices before items with more choices. This effectively yields us unit propagation for more or less free, any dependencies with exactly one solution will be solved first.

Consider the packages A depending on X or Y, and B depending on Y. Installing both of them has two valid solutions: ABXY and ABX. To start the solver we enqueue the items [A] and [B], the solver now processes the following steps:

1. Add A to the needs set, enqueue `A depends X or Y`
2. Add B to the needs set, enqueue `B depends Y` (at the front, because it only has 1 choice)
3. Process `B depends Y` (it has 1 choice) by marking Y and enqueuing it's dependencies
4. … Potentially process further dependency chains from Y with less than two solutions …
5. Finally get to the `A depends X or Y`, see that `Y` is already in the needs set, and finish.

If we instead processed the items in the order they were added for example, we would have had to make a choice between X and Y and then either install all packages or backtrack if we ended up with a conflict.

Conflicts

When we process an item and decide to install a given choice, we iterate over all its Conflicts and add them to the rejects set. For example, given an install of A conflicts X, we first add `A` to the needs set, then add `X` to the conflicts set, and finally enqueue any dependencies of A.

Conflicts hence do not remove manually installed packages, but only remove automatically installed packages and prevent the automatic installation of new packages.

An exception can be made at the start of the solver for the purpose of dist-upgrades: If there is a strong chain from a package to be installed that Conflicts and Replaces a package that was previously manually installed, that package may be considered automatically installed again and those not to be removed.

Overall this is the strongest difference from apt's previous approach to solving: The classic APT solvers would happily remove a manually installed package with a `Depends: foo (= 1)` if you replace `foo=1` with `foo=2`.

Optional dependencies

We extend the solver design with two additional sets ‘wants’ and ‘likes’ which collect the ‘Recommends’ and ‘Suggests’ of installed packages.

When choices need to be made, we will first try the choices that are in the ‘wants’ set, then we try the choices in the ‘suggests’ set.

Automatically installed packages

The solver does not try to keep automatically installed packages on the system, they will remain by the nature of dependencies. To aid in dependency solving, the solver will, similarly to optional dependencies first consult the set of automatically installed packages before it tries other choices.

Backtracking

When solving dependencies we could have picked choices that end up conflicting with dependencies in the package. For example, if `A depends X | Y` and `X depends Z`, and `Z conflicts A`, if we chose to install `X` we end up with a conflict.

Such a conflict can be resolved by means of backtracking. For backtracking we introduce a depth counter and record for each decision the depth at which it was made. Then when we reach a conflict we go mark the conflict, and go back up one genernation.

In the example above, we would

0. Mark
1. Increase the depth counter to 1
2. Mark X for install by A at depth 1
3. Install Z by X at depth 1
4. Find out that Z has a conflicts with A
5. Decrease the depth counter to 0
6. Set Z as rejected at depth 0
7. Set X as rejected at depth 0 (following the recorded reason for installing Z)
8. Remove all other decisions of depth = 1
9. See A depends X | Y again, note that X is rejected and try Y.

Algorithm

Facts

Let

- $\mathcal{V}$ be the set of versions in the apt cache (literals)
- $\mathcal{I} \subset \mathcal{V}$ be the set of installed versions
- $\mathcal{M} \subset \mathcal{I}$ be the set of manually installed versions
- $\mathcal{A} = \mathcal{I} \setminus \mathcal{M}$ be the set of automatically installed versions

Let

- $\mathcal{D}_V \subset \{D1 \vee \ldots \vee D_n \mid D_1, \ldots, D_n \in \mathcal{V}\}$
- $\mathcal{C}_V \subset \{C \mid C \in \mathcal{V}\}$

denote the dependencies and conflicts of $V \in \mathcal{V}$. These correspond to the formulas: $V \rightarrow D1 \vee \ldots \vee D_n$ and $V \rightarrow \neg C$.

(TODO: Optional dependencies)

Let $|D_1 \vee \ldots \vee D_n| = n$ represent the number of choices in a given "or group".

Solver state

For depth $i \in \mathbb{N}$, and step $j \in \mathbb{N}$:

Let

- $needs_{ij} \subset \mathcal{V}$ denote the set of versions that shall be installed
- $rejects_{ij} \subset \mathcal{V}$ denote the set of versions that shall not be installed
- $wants_{i}j \subset \mathcal{V}$ denote the set of versions that we want installed later (optional dependencies)
- $likes_{ij} \subset \mathcal{V}$ denote the set of versions that are also suggested by packages (more optional)

Let $allversions(V)$ denote the ordered set of all (allowed for install) versions of the package that V is a version of.

Let $work_{ij} \subset \{V \rightarrow D \mid D \in \mathcal{D}V\}$ denote the work queue of unsatisfied dependencies.

Let $needs_{00} = rejects_{00} = wants_{00} = likes_{00} = \emptyset$.

Iteration

Let the symbol $\perp$ determine termination of the solver (mostly fatal), and $\top$ denote termination of that level.

$$needs_{i,j+1} = \begin{cases} \perp & \text{if } \forall d \in w : d \in rejects_{ij} \\ needs_{ij} & \text{if } \exists d \in w : d \in needs_{ij} \text{ (already installed)} \\ needs_{ij} \cup d & \text{if } \exists d \in w : d = \{w\} \text{ (single choice)} \\ \top & \text{ otherwise} \end{cases}$$

$$rejects_{i,j+1} = \begin{cases} \perp & \text{if } needs_{i,j+1} = \perp \\ \top & \text{if } needs_{i,j+1} = \top \\ rejects_{ij} \cup \{d \in \mathcal{C}_d\} & \text{otherwise} \end{cases}$$

Evaluation

Conclusion