Debian dependency resolution in polynomial time

Niels Thykier

Debian Developer
Release Manager

21. August 2015
DebConf15 2015
Heidelberg, Germany

Outline

Outline

Introduction

What do I want to achieve with this talk?

- I hope to debunk some myths about what makes dependency resolution "really hard" and "tractable".
- I will be sharing some of our findings from optimizing Britney. Not all will apply to you(r problem).
- For some, I will be stating "the obvious" and "nothing new".
- I do *not* have the "one-size-fits-all" solution. Sorry.

Defining the problem the problem

- When you run “apt install eclipse”, two distinct problems are solved before the package is installed.
  1. Apt figures out what is needed to install the package.
  2. Apt/dpkg computes a series of “unpack”, “configure” etc. actions.
- The first problem is what I will be talking about.
- The second problem is basically an “install plan” or an “ordering problem”.

Why skip over the "install plan"

- Despite being an important part of installing packages, it is *not* dependency resolution.
- Assuming we had no cycles[1], this is a trivial partial-ordering problem:
  - Define all actions (e.g. "unpack pkg/version/arch").
  - Compute partial-ordering constrains ("configure pkgA/..." before "configure pkgB/...")
  - Sort items such that all their constrains are satisfied.
  - Given no cycles, we always can construct a directed-acyclic graph (DAG).
  - Said DAG is your install plan - "just" feed it to dpkg and you are done.

Run-time of all of this?

[1] Known invalid assumption

Run-time of the "install plan"

- Something like graph search $O(|V| + |E|)$ ($\leq O(n^2)$) plus sorting $O(n \cdot log(n))$.
- Even with cycle detection, complexity remains the same (Tarjan's algorithm).
- Cycle breaking can be non-trivial, but ...
  - The fewer (postinst) scripts, the fewer "unbreakable cycles".
  - Remove all dependency cycles and this problem is indisputably "trivial".
  - Yes, that is easier said than done, but that is at most what it takes.
- Of course, if it is "too simple" for you, then you can always add more features on top to make it harder.

So, moving on ...

The players in “hard” problem-game

Notable tools affected by the “hard” problem:

- APT, aptitude, cupt, etc.
- Britney
- DOSE, edos, etc.

Notable tools **not** affected by the “hard” problem:

- dpkg - it only verifies a given solution, which is polynomial.
- DAK’s rm command - it does a “cheap local” check.[2].

[2] It is primarily “slow” for other reasons

Outline

What makes the problem "hard"?

- The "options" (alternatives, virtual packages).
  - This also includes "normal" $pkg(>= 1.0)$ versions.
  - And especially unversioned dependencies with multiple versions of said packages.

With options removed, everything else becomes piece of cake.

What makes the problem "hard"? Necessary evil

But what happens if we remove "options"[3]:

- The problem is trivially reduced graph search $O(|V| \cdot |E|)$.
- Either we got no versioned dependencies OR we would have "strictly equal" versioned dependencies only.
- Without versioned dependencies, upgrades would be "fun". Lots of "fun".
- With "strictly equal" versioned dependencies, *every single* upload would be "fun". Lots of "fun".

So the complexity is a necessary evil.

[3] You can also make it trivial in a different way. However, it involves removing negative dependencies (which be "fun" in its own way)

An example of the problem

A simplified problem:

```
Package: coreutils
Version: 8.23-4
[...]
Depends: libc6 (>= 2.17)
```

Quiz: Given only that **libc6/2.19 is known to be installable**, can we immediately conclude that **coreutils/8.23-4 is also installable?**

- With negative dependencies?
- Without negatives dependencies?

An example of the problem - with answers

A simplified problem:

```
Package: coreutils
Version: 8.23-4
[...]
Depends: libc6 (>= 2.17)
```

Quiz: Given only that **libc6/2.19 is known to be installable**, can we immediately conclude that **coreutils/8.23-4 is also installable?**

- With negative dependencies: No, libc6 (et al) could have a Breaks/Conflicts on coreutils.
- Without negatives dependencies: Still no, libc6 (et al) could depend on `coreutils (<< 8.23)` (or `coreutils (>= 8.24)`).

Outline

The big picture of the problem

In the big picture, we tend to optimize for (co-)installable packages.

- If there is Breaks/Conflicts, it will generally be versioned with an upper-bound (e.g. `Breaks: coreutils (<< 8.23)`).
- If there is a (circular) dependency, it will generally unversioned or versioned with a lower-bound (e.g. `Pre-Depends: coreutils (>= 8.23)`).
- Alternatives/virtual packages are limited in numbers.

The major exceptions being: ...

The big picture of the problem - the exceptions

The major exceptions being: ...

- Packages from "unstable". Usually because packages are not built yet or "occasionally" due to transitions.
- "mutually exclusive" packages (e.g. providers of sendmail)
- Version ranges ("rare"): `foo (>= 1.0 ), foo (<< 2.0 )`
- Strictly equal versions: `foo (= ${binary:Version})`
  - almost exclusively used in binaries built from *the same source*.

Exponential run-time is all about choice!

What makes the problem hard? In a nut shell, given:

```
Package: starting-package
Depends: foo1 | foo2 | foo3 | ... | fooN

Package: foo{1..N}
Depends: bar1 | bar2 | bar3 | ... | barM | good

Package: bar{1..M}
Depends: bad

Package: bad
Conflicts: starting-package

Package: good
```

Solve for `starting-package`.

What makes the problem easy? We do!

Very few writes software like this.

What makes the problem easy? Except dictionaries

Very few writes software like this.

- There are a handful of exceptions including aspell-dictionary, ispell-dictionary, wordlist, etc.

What makes the problem easy? Except dictionaries and Multi-Arch

Very few writes software like this.

- There are a handful of exceptions including aspell-dictionary, ispell-dictionary, wordlist, etc.
- Technically, arch:any Multi-Arch foreign (or "allowed" with pkg:any dependencies) packages *can* also cause "unnecessary" extra "options" as well.

If you think about it

If you think about it, to trigger this case (examplified):

- We would need N distinct implementations of `awk`.
- They would have to depend on any one of M distinct (but equally valid) `libc` implementations.

The dictionaries exampled

The reason the dictionaries "blow up" is because they work with pure interchangeable "data".

- Data packages themselves often have no or very trivial dependencies.
- The blow up often is limited to "1 dependency level".

Certainly, if you have N of these "1 dependency level" blows up, you still have an issue.

Outline

Installing in the easy case

Installing becomes trivial some common cases. The conditions required:

- Single suite
- Single architecture

Concrete example: Lintian

Lintian is "mostly trivial":

- Lintian has 26 distinct dependency clauses
- With single suite and architecture, there is *at most one package* satisfying each of its clauses.
- Some of its dependencies have "non-trivial" dependencies such as: `debconf (>= 1.2.0) | debconf-2.0`

By the time, the solver must "guess", all the dependencies of `debconf` have already been resolved.

When do SS and SA happen?

Some particular common (mostly) polynomial cases:

- Installing build dependencies in "clean" unstable/stable Debian chroots (SAN)
- Installing packages in a pure Wheezy or Jessie system with Multi-Arch (SN)
- Installing packages in a pure Squeeze (or older) system (SAN)
- Installing packages in unstable+experimental OR stable + backports with "strict" preference for one of the suites (S)
- Testing migration (Britney) (SAN)

Legend:

- N: No special heuristics / code required. The data itself exhibits the traits.
- S: "single suite" restriction (partially or fully)
- A: "single architecture" restriction (partially or fully)

Reasons for polynomial run-time

A couple of reasons why this happens:

1. With single suite + architecture, there is at most one version of any "package/version/arch" to satisfy dependencies.
2. We forbid the use of alternatives implementations for libraries as it is fragile, which limits "options" in many packages.
3. We only want so many implementations of awk, dpkg, etc.
4. Breakage is usually quite "obvious" (despite a "exploded options list").
5. We discourage (and in many cases) forbid the use of permanent negative dependencies.

Can we somehow make it simple without the first item?

Outline

Dist-upgrading - the basics

What happens when we do a simple stable to stable upgrade?

- Replace old release with new release in sources.list
- Have apt update its caches
- Have apt upgrade all packages with a new versioning in the new release

When is the dist-upgrade complete?

Long story short. Upgrade is complete when all packages on the system:

- are also present in the new release,
  - That is, old packages are removed.
- have the same version as their counterpart in the release
  - That is, they have been upgraded if there was a new version
- include the essential packages of the new release.
  - This involves installing new packages, so we will ignore that.

Simplified upgrade scenario

Let us take a simple setup:

- Upgrade from Wheezy to Jessie (i.e. sed s/Wheezy/Jessie/g)
- Wheezy has ~30000 packages
- Jessie has ~35000 packages
- The system has 2000 packages installed from Wheezy
- Assume that all Wheezy packages got a new version in Jessie

Pop quiz on upgrades

What is the maximum problem size of this upgrade?

- 30000 packages
- 35000 packages
- 37000 packages
- 65000 packages
- 67000 packages
- Trick-question, the right answer is not on this slide

More on the upgrade

Further remarks:

- For every upgraded package, our problem size decreases by 1 (*without* Wheezy)
- Once we are done upgrading, we end up in a "single suite" situation
  - This is generally true for pure stable to stable upgrades
- 65000 would be correct if we *kept* Wheezy in the sources.list

Finding upgrades

Upgrading ought to be as simple as:

- Mark new essential packages for install
- Upgrade all binaries to their new version (if they have one)
- Remove all binaries not in the new suite
- Resolve missing dependencies if any

This basically solves upgrading by installing packages. We can do better than this.

On polynomial upgrades

A couple of remarks about deterministic polynomial upgrades:

- It is intended as a problem size reducer, not a perfect solution.
- It relies on the fact that solution verification is $O(|V| + |E|)$
- It exploits common "patterns" in dependencies.

Theory behind polynomial upgrades

- We have a system that is **not** broken
  - Theory: "Given a set of installable mutually co-installable packages called I"
- Where we can add, remove or replace one or more packages to/from/in this system (with another package) in linear time.
  - Theory: "We can compute $I_{new} = (I \setminus R) \cup A$ in linear time"
- And then verify that the modified system is still **not** broken in polynomial time
  - Theory: "We verify that I_{new} is a valid solution in $O(|I| + |E|)$"

Issue of the day, how do we compute what to add/remove or replace (A and R)?

Finding packages in a jiffy

- Group binaries from the same source together
  - Due to "Depends: `foo (= ${binary:Version})`"
  - Side-effect: Also resolves intra-source Breaks/Replaces.
- Try upgrading each of these groups in (any) order
- If result is upgradable alone, commit
- Rinse and repeat until you reach a fix point (that is, nothing new can upgrade)

Results from simple upgrades

Depends on largely on what is installed:

- A number of packages could be upgraded
  - Example test: libc6, ant, man-db*, eclipse*, xterm*, etc.
  - Not tested in "all configurations"
- Basically, `foo (>= version-in-wheezy)` implies we can upgrade foo to the Jessie version.
  - Common for libraries without ABI breaks
  - Enables upgrading of "stable" packages before their reverse dependencies
- Algorithm is fairly limited:
  - Example issues: perl packages, python packages, etc.
- Run-time: $O(|I|^2 * (|I| + |E|))$ (ballpark estimate)

Improving the upgrades

Further ideas:

- Combine groups if their relations might need it
  - Example: dpkg needs new version of libc
  - Example: dpkg breaks old version of readahead-fedora
- Account for "renamed" packages (ABI breaks) and transitional packages
  - Example: bar/wheezy depends on libfoo1, bar/jessie depends on libfoo2.
  - Example: bar depends on foo. In Jessie, foo depends on one single foo-replacement.
  - In both cases, the replacement is built from the same source as the package being replaced.
- If they have "trivial" (no-guess) solutions, resolve those. Otherwise, give up and try another package.

Outline

Smarter installing

We can also improve installing a bit:

- Find packages that are "identical" (apart from their "name" and "version")
- Find packages that are clearly "superior" to another one

Eqv. packages - staring into the abys

If you stare long enough at the "dictionary problem" (aspell-* etc.):

```
$ wdiff aspell-{ru,uk}
Package: [-aspell-ru-] {+aspell-uk+}
Version: [-0.99g5-20-] {+1.7.1-1+}
Architecture: all
Depends: aspell, dictionaries-common (>= 1.23~)
Provides: aspell-dictionary
```

Eqv. packages - what is going on?

- Textually the packages are different
- Semantically, they differ only by name and version
- Simplified view: Reverse dependencies *could* in theory make them different

Find eqv. packages

- They have the same (effective) dependencies
- They have the same (effective) negative dependencies
- They satisfy the same clauses of their reverse dependencies

This can be done in polynomial time for all packages.

Substituable packages

Beyond eqv. packages, we can also find superior packages

- The pkgA have the same or fewer (effective) dependencies compared to pkgB
- The pkgA have the same or fewer (effective) negative dependencies compared to pkgB
- The pkgA satisfies at least all the same clauses of the reverse dependencies of pkgB

Work in progress, but many easy solution in sight.

Thanks!

Questions?

Niels Thykier
niels@thykier.net

about the slides:
available at https://anonscm.debian.org/cgit/users/nthykier/talks.git