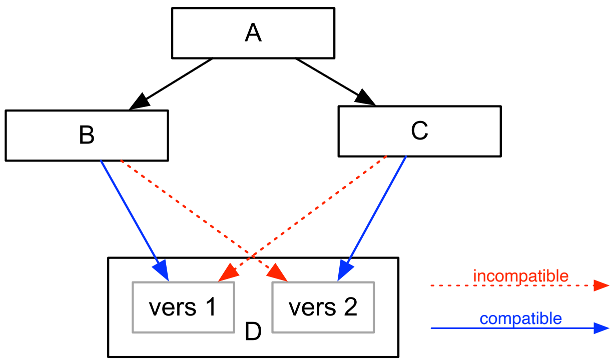
Version SAT

Posted on Tuesday, December 13, 2016.

Dependency hell is NP-complete. But maybe we can climb out.

The package version selection problem is to find a set of dependencies that can be used to build a top-level package P that is complete (all dependencies satisfied) and compatible (no two incompatible packages are selected). There may be no such set, because of the diamond dependency problem: perhaps A needs B and C; B needs D version 1, not 2; and C needs D version 2, not 1. In this case, assuming it's not possible to choose both versions of D, there is no way to build A.



A package manager needs an algorithm to select package versions: when you run `apt-get install perl`, it may assume you mean the latest version of Perl, but then it has to find a way to satisfy Perl's transitive dependencies, or else to print an understandable explanation of why Perl can't be installed. You might reasonably wonder: how expensive is it to solve this problem, in the worst case? You probably don't want your package manager to take hours or days or years to decide whether it can install Perl.

Unfortunately, the version selection problem is NP-complete, which means that we're exceedingly unlikely to find an algorithm guaranteed to run quickly on every input. This post gives a proof of NP-completeness for version selection, looks at how existing package managers cope, and briefly discusses possible approaches to avoid an NP-complete task.

Proof of NP-Completeness

To consider NP-completeness, we need to shift from our modern world of algorithms with rich outputs to the limited world of complexity theory, where algorithms have one boolean output: yes or no. In this world of complexity theory, we'll define the VERSION problem (they're always all caps) to ask whether there is a valid version selection. This boolean VERSION problem is only half of our original problem, and we can prove that it's NP-complete. To do so, we need to prove two separate facts: that VERSION is in NP and that VERSION is NP-hard.

A problem is in NP if every “yes” answer has an easily-checked polynomial-size explanation.

VERSION is in NP, because any “yes” answer can be explained by listing the selected package versions. This list is no bigger than the input and can be checked for correctness in time no worse than quadratic in the input (probably linear, depending on details of the computing model).

A problem is NP-hard if an efficient solution for that problem can be adapted into an efficient solution to *every* other problem in NP. That's a pretty tall order, but it is enough for us to show how to adapt an efficient solution for VERSION into an efficient solution for one other NP-hard problem (call it HARD) and then rely on the fact that someone else has proven that an efficient solution for HARD can be adapted into an efficient solution for every other problem in NP.

A useful example of an NP-complete (in NP and NP-hard) problem is 3-SAT. In 3-SAT, the input is a boolean formula over some number of boolean variables, constrained to be a conjunction (an AND) of some number of disjunctions (ORs) of three literals each, where a literal is a variable or its negation. For example, here is an input for 3-SAT ($\wedge$ means AND, $\vee$ means OR, and $\neg$ means NOT):

$$(\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor \neg x_4) \land (\neg x_2 \lor \neg x_2 \lor x_3) \land (x_2 \lor x_2 \lor x_2)$$

It is satisfiable by exactly one assignment to the variables—$x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$—so the answer is yes.

If we extend it to add one more clause,

$$(\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_2 \lor \neg x_3 \lor \neg x_4) \land (\neg x_2 \lor \neg x_2 \lor x_3) \land (x_2 \lor x_2 \lor x_2) \land (x_1 \lor \neg x_2 \lor x_4)$$

then it is unsatisfiable by any assignment to the variables, so the answer is no.

The general form of a 3-SAT instance is a formula F that is the conjunction of clauses C_1 through C_n over variables V_1 through V_m, where each C_i is a disjunction of three literals, each of the form x_j or $\neg x_j$ for some variable x_j . Duplicate literals in a clause are allowed, as in $(\neg x_2 \lor \neg x_2 \lor x_3)$ and $(x_2 \lor x_2 \lor x_2)$ above.

We can convert any 3-SAT instance to a VERSION instance with the same answer. About the package manager we will assume only that:

1. A package can list zero or more packages or specific package versions as dependencies.
2. To install a package, all its dependencies must be installed.
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

We'll abbreviate package `P` version `V` as `P:V` (now using fixed-width font for packages to distinguish from the standard math italics for formulas). A dependency on `P:V` must be satisfied by version `V` exactly, not `V`−1 and not `V`+1.

Given a 3-SAT formula, we can create a package `F` representing the whole formula, packages `C1`, `C2`, ..., `C`*n* representing each clause, and packages `X1`, `X2`, ..., `X`*m* representing each variable.

Each package `X`*j* has two versions `X`*j*`:0` and `X`*j*`:1`. As assumed above, `X`*j*`:0` and `X`*j*`:1` conflict and cannot both be installed. `X`*j*`:1` being installed corresponds to $x_j = 1$ in the original formula.

Package `C`*i* has three versions numbered 0, 1, 2, each of which depends on a literal from the corresponding clause. For example, if C_5 is $(x_1 \lor \neg x_2 \lor x_4)$, then `C5:0` depends on `X1:1`, `C5:1` depends on `X2:0`, and `C5:2` depends on `X4:1`. `C`*i*`:`*k* being installed corresponds to C_i's k'th literal being true (and therefore C_i being true) in the original formula.

Package `F` depends on `C1`, `C2`, ..., `C`*n*. `F` being installed implies that all the `C`*i* are installed, which corresponds to all the C_i being true and therefore to F being true.

If the package manager can find a way to install package `F`, then a satisfying assignment for the original formula can be read out from the install status of `X`*j*`:1` for each variable x_j. Similarly, if the formula is satisfiable, the satisfying assignment gives one way the package manager could successfully install `F`. Therefore, we've converted the 3-SAT instance into a corresponding VERSION instance with the same answer, which establishes that 3-SAT can be solved using VERSION, so VERSION is NP-hard.

Since VERSION is in NP and is NP-hard, VERSION is NP-complete.

Implementations

The assumptions above are quite minimal: packages have a list of dependencies, a package's dependencies can change with its own version to version, a package's dependencies can be restricted to specific versions of those dependencies, and it is possible for two versions of a package to conflict with each other. That may be the bare minimum for a package manager to be useful. Some package managers might not allow a dependency to list a specific version, instead requiring a range, but we can easily change the version requirements 0 and 1 to $\leqslant 0$ and $\geqslant 1$. Some package managers might not assume that different versions of a package conflict by default, but it must be at least possible to specify such a conflict: there can't be two `/bin/bash` on a Unix system, or two definitions of `printf` built into a C program.

The assumptions are true of every package manager I have looked at: Debian's APT, RedHat's RPM, Rust's Cargo, Node's npmjs, Java's Maven, Haskell's Cabal, and more. The implication is that these package managers faces an NP-complete task. Each must choose between possibly taking a very long time to decide on an installation strategy or possibly reporting an installable package as uninstallable. (Of course, a given implementation may inadvertently do both.)

Knuth writes in Volume 4, Fascicle 6:

> The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics. Thanks to elegant new data structures and other techniques, modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago.

In practice, it does seem that modern package managers are moving toward using SAT solvers:

0install started with heuristics but found it necessary to switch to a SAT solver.

Chef, a systems integration framework, uses the dep-selector Ruby bindings for the Gecode constraint solver.

Dart's pub includes a backtracking solver that often takes a long time.

Debian's apt-get uses heuristics by default but can invoke a SAT solver and can take user preferences into account. The Debian Quality Assurance team also runs a solver to identify uninstallable packages in their repos.

Eclipse uses the sat4j SAT solver to manage installation of its plugins.

Fedora's DNF ("Dandified yum") uses a SAT solver in an experimental mode.

FreeBSD's pkg, also used by DragonflyBSD, uses the picosat SAT solver.

OCaml's OPAM can invoke a SAT solver locally or remotely over a network. Like with Debian's apt-get, OPAM's solver can take user preferences into account, and the OPAM repos are scanned for uninstallable packages.

OpenSUSE's package manager uses libsolv, "a free package dependency solver using a satisfiability algorithm." There is also OpenSUSE's zypper, which uses its own libzypp SAT solver.

Python's Anaconda uses a SAT solver but can take a long time.

Rust's Cargo uses a basic backtracking solver. It also allows multiple versions of a crate to be linked into the final binary.

Solaris's pkg, also used by Illumos and sometimes known as IPS, uses the minisat SAT solver.

Swift's package manager uses a basic backtracking solver.

[I would like to add more package managers here. If you know details for one (or something here is wrong), please email me or send a tweet. Thanks.]

Alternatives?

How should we react to the fact that package version selection is NP-complete? One reaction is to embrace the complexity and be thankful that SAT solvers are as good as they are. Another reaction is to ask whether this can possibly be a good idea. Maybe we should not be building tools that require solving this problem. Maybe something has gone wrong in the way we develop software.

If package version selection is NP-complete, that means the search space of possible package combinations is too large and intricate for efficient systematic analysis; what about efficient systematic testing? If a search finds a conflict-free combination, why should we believe the combination will work? The absence of a version conflict may indicate only that the combination is untested. If a search doesn't find a conflict-free combination, how can that failure be explained to a developer in a way that makes it clear what to do next? Software is hard enough to get right without admitting NP-complete problems into our software configuration decisions. Let's reexamine how we got here and how we might escape.

The proof above depends on these of assumptions, copied from above:

1. A package can list zero or more packages or specific package versions as dependencies.

2. To install a package, all its dependencies must be installed.
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

The conventional wisdom, as I suggested above, is that these are roughly the "the bare minimum for a package manager to be useful," but maybe we can find a way to reduce them after all.

One way to avoid NP-completeness is to attack assumption 1: what if, instead of allowing a dependency to list specific package versions, a dependency can only specify a minimum version? Then there is a trivial algorithm for finding the packages to use: start with the newest version of what you want to install, and then get the newest version of all its dependencies, recursively. In the original diamond dependency at the beginning of this article, A needs B and C, and B and C need different versions of D. If B needs D 1.5 and C needs D 1.6, the build can use D 1.6 for both. If B doesn't work with D 1.6, then either the version of B we're considering is buggy or D 1.6 is buggy. The buggy version should be removed from circulation entirely, and then a new released version should fix the problem. Adding a conflict to the dependency graph instead is like documenting a bug instead of fixing it.

Another way to avoid NP-completeness is to attack assumption 4: what if two different versions of a package could be installed simultaneously? Then almost any search algorithm will find a combination of packages to build the program; it just might not be the smallest possible combination (that's still NP-complete). If B needs D 1.5 and C needs D 2.2, the build can include both packages in the final binary, treating them as distinct packages. I mentioned above that there can't be two definitions of `printf` built into a C program, but languages with explicit module systems should have no problem including separate copies of D (under different fully-qualified names) into a program.

Another way to avoid NP-completeness is to combine the previous two. As the examples already hint at, if packages follow semantic versioning, a package manager might automatically use the newest version of a dependency within a major version but then treat different major versions as different packages.

One rationale for such restrictions is that developers are likely not thinking about the entire space of all possible package combinations when building or testing software. It would help for the developers and their tools to agree about how software is built. If any of these approaches can be made to work in practice, it could go a long way toward simplifying the operation and understandability of language package managers.

Related Work

Proofs that Debian and RedHat package installation are both NP-complete are given in "EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies" (2005), pages 49–50. The difficult step in the reduction of 3-SAT to package installation is how to construct a disjunction. The EDOS proofs encode the disjunction using the package manager's ability to specify a list of alternatives for a single dependency, either directly (in Debian) or using "provides" directives (in RedHat). For example, these systems allow a pseudo-package `text-editor` to be defined that is considered installed when any of the real packages `ed`, `vi`, or `acme` is installed.

The dependency specifications for a language package manager like Rust's Cargo are dramatically simpler than those for Debian and RedHat, and so the EDOS proofs do not apply. One might therefore hope that language package managers face an easier (not NP-complete) job. The new proof above dashes that hope. (One way to view the proof above is that it simulates the "provides" directive in the last example by defining a `text-editor` package with three versions, one of which depends on `ed`, one on `vi`, and one on `acme`.)

By encoding the disjunction in the changing dependencies of different versions of a package, the new proof works without modification for both Debian's and RedHat's package managers but also applies to essentially any foreseeable operating system or language package manager. I suspect that most language package manager authors assumed the problem they faced was NP-complete, but I've been unable to find prior written proofs of that fact.

A few dependency systems use constraint solvers instead of SAT solvers, but the underlying problem is still NP-complete.

In 2008, Daniel Burrows wrote a blog post about using dpkg to solve Sudoku problems.

Thanks to Sam Boyer for pointing me at the EDOS report and for his excellent overview of package management.

Roberto Di Cosmo has written a number of followups to the EDOS report, listed here, in particular, “Dependency solving: a separate concern in component evolution management,” which contains an updated proof. That line of research applies SAT solvers but also works to take user preferences into account.

Another related line of work is “OPIUM: Optimal Package Install/Uninstall Manager” by Tucker et al., ICSE 2007. OPIUM was the starting point for 0install's solver.

Jaroslav Tulach discovered the same proof as above in 2009. Thanks to HN reader edwintorok for the link.

The discussion of Tulach's proof on LtU mentions Daniel Burrows's 2005 paper “Modelling and Resolving Software Dependencies,” but that paper's proof is more like the EDOS proof than Tulach's proof / the proof above.

Many readers sent additional links to references and to package managers with SAT solvers. Thanks to all.