Modelling and Resolving Software Dependencies

Daniel Burrows `<dburrows@debian.org>`

June 15, 2005

Abstract

Many Linux distributions and other modern operating systems feature the explicit declaration of (often complex) dependency relationships between the pieces of software that may be installed on a system. Resolving incompatibilities between different pieces of software is an NP-complete problem, and existing solutions require the user to manually resolve many "simple" dependency problems.

I present a simplified, abstract model of dependency relationships, and a restartable technique based on best-first-search to calculate resolutions.

Note. This is a work in progress; it sometimes lags behind or jumps ahead of the current state of the software it documents, and some of the details may be incomplete or unattended to. However, I hope that it provides some more insight into the direction in which `aptitude`'s problem resolver is headed – and in which I believe that other installation frontends should also consider heading.

1 Introduction

It is common nowadays for hundreds or thousands of software packages to be installed on a single computer system, and for many of these software packages to interact with one another. Because some combinations of software packages will not function properly – for instance, an application program might require a particular version of a graphics library – installing software manually while avoiding unexpected breakage is an increasingly unpleasant chore.

To address this problem, programs known as *package systems* were developed. A package system typically manages *packages* that consist of the files of a program or library, along with metadata such as the name and version of the package, a brief description of what it contains, and (most importantly for our purposes) a list of which other packages it requires or is incompatible with. The package installation software warns the user upon any attempt to install or remove software that would violate these constraints.

Unfortunately, the early versions of these tools replaced the chore of manual software installation with the chore of dependency resolution: for instance, installing a package of the popular game `wesnoth` might produce an error indicating that the user should find, download, and install a new version of the `SDL` graphics library. A new version of the `kmail` mail client might require the user to upgrade his or her entire operating system, indicating this fact by a slew of cascading error messages.

As a result of this so-called "dependency hell", new and more automated tools, such as `apt` and `up2date`, were developed. These tools maintain a database of software installed on the user's system, along with software available from any number of remote sites.

To install or remove a piece of software, the user issues a request to, for instance, "install wesnoth" or "upgrade kmail". The installation tool will proceed to find a set of package installations or removals which leads to a consistent result. Typically, it then presents this list of actions to the user and prompts for confirmation; the user can either accept the proposed solution, or reject it and proceed to fix the problem in a fully manual way. Once the user is satisfied with the proposed changes, the tool will download any new software packages and install them.

This approach has two major drawbacks:

1. The user interface for resolving dependencies is a "take it or leave it" proposition: there is no way for the user to ask the algorithm to find another solution. This means that if the algorithm makes a poor or undesired choice (which, as I will argue below, will inevitably occur from time to time) the user is forced to fall back to fully manual operation.

2. In at least some cases (particularly `apt`), the algorithm used in resolving dependency conflicts deals poorly – which is a euphemism for "not at all" – when there are more than two versions of a package to choose from[1]. For instance, if versions 1, 2, and 3 of package `A` are available, with 2 being the default version of the package, and if package B requires version 3 of package A, when the user tries to install package B, he or she will receive an error message indicating that the dependency on A cannot be fulfilled.

Another general difficulty in solving dependencies in these systems is that the package systems contain many features which, although they are arguably "syntactic sugar", tend to cause algorithms that operate on packages to become strewn with complex iteration constructs and unpleasant corner cases. Although some attempts have been made to find general models of package dependencies (for instance, the internal structures of `apt` can represent either Debian or Red Hat packages), the models with which I am familiar work by taking a "greatest upper bound" of the systems that they cover, leading to a generic framework that is, if anything, even more convoluted than the individual package systems that it covers.

Note. I have not yet performed an extensive survey of package systems, and it may be that there already exist systems that fix one, two, or all of the drawbacks listed above.

[1]More precisely, if more than one version other than the currently installed version (if any) exists.

2 Example: the Debian Package System

The Debian package system is implemented by a low-level tool known as `dpkg`. Debian packages are files with the extension `.deb`; `dpkg` can install a `.deb` file that has already been retrieved, or remove a package that is currently installed on the system. If dependency constraints are violated, `dpkg` will print errors messages and abort the installation after unpacking the packages.

The usual user interface to the package system is through one of the programs in the `apt` suite. `apt` is a high-level library which allows C++ programs to examine the set of installed packages, determine what actions are to be performed, and execute these actions (by, for instance, downloading package files and calling `dpkg` to install them). `apt`-based installation tools typically refuse to even begin any actions that will result in an inconsistent system state, and all of them provide a basic algorithm that resolves inconsistencies by adjusting package states until all dependencies are fixed.

In the Debian package system, each package may have one or more versions, but at most one version of each package may be installed at any given time. The basic relationships between packages are *dependencies* and *conflicts*. For instance, version 6.14.00-1 of the `tcsh` command shell depends on version 2.3.2.ds-4 or greater of the `libc6` package and version 5.4-1 or greater of the `libncurses5` package: it may not be installed unless an appropriate version of each of these package is installed. On the other hand, the same package conflicts with all versions of the `tcsh-kanji` and `tcsh-i18n` packages: `tcsh` may not be installed at the same time as either of these packages.

A single dependency may name several packages, combined with and OR operator (indicated by a vertical bar). For instance, version `1.4.48` of the `debconf` package depends upon `debconf-i18n | debconf-english`; in order to install debconf, you must also install one of these two packages.

Last but not least, dpkg supports what are known as “virtual” packages. Each version of each package may *provide* one or more package names; each named package will become a virtual package. Virtual packages can have the same name as a normal package, in which case they are known as *mixed* virtual, or they can exist only through being provided by a normal package, in which case they are known as *pure* virtual. An unversioned dependency on a virtual package will be satisfied if any provider of the name is installed, while an unversioned conflicts will require that all providers of the name are removed – but as a special case, direct or implicit self-conflicts are ignored. Versioned dependencies and conflicts do not, as of this writing, follow provided package names.

For instance, the virtual package `mail-transport-agent` is used to identify packages containing mail transport agents. Every such package both provides and conflicts with the virtual package, and packages that require a mail transport agent depend on it.[2]

[2]Due to a quirk in how `apt` resolves dependencies, dependencies on a virtual package are required to include a real package as an alternative: for instance, `bugzilla` depends on `sendmail|mail-transport-agent`.

3 An Abstract Model of Dependency Relationships

As can be seen in the previous section, real dependency systems are complex. This tends to complicate the business of reasoning about how to find solutions to dependency problems, and to cause algorithms that manipulate dependencies to become horribly messy. In situations like this, it is often a good idea to find a simpler, more mathematical model of the problem being analyzed. Of course, it is well-known that package dependencies can be reduced to the satisfaction of Boolean equations, but such a reduction is arguably *too* extreme: it certainly results in a mathematical model, but the model it produces hides the structure of the original problem. The following section describes an alternate model which is sufficient to capture any dependency problem (at least in Debian) and retains the structure of a package system.

3.1 Basic Concepts

In this simplified model, the only objects in the world are packages, versions of packages, and dependencies. Packages will typically be denoted by $p_1, \dots$; versions will typically be denoted by $v_1, \dots$; and dependencies are of the form $v \to \{v'_1, \dots\}$, indicating that the version v requires the versions $\{v'_1, \dots\}$. The package associated with a version v is denoted by $PkgOf(v)$.

To represent the state of the entire system, the following sets are defined:

- $\mathcal{P}$ is the set of all packages.
- $\mathcal{V}$ is the set of all package versions.
- $\mathcal{D}$ is the set of all dependencies.[3]

Throughout this paper, I will assume that $\mathcal{P}$ and $\mathcal{V}$ (and hence $\mathcal{D}$) are finite.

3.2 Reduction of Debian Dependencies to the Model

As claimed above, it is possible to reduce a Debian dependency system to this abstracted model. The reduction proceeds in approximately the following way:

- $\mathcal{P}$ is the set of Debian packages.
- $\mathcal{V}$ is the set of versions of those packages, plus one additional version for each package. This version represents the state of the package when it is not installed. Versions corresponding to versions of the Debian package are indicated by $p\!:\!n$ where n is the version number, while the "uninstalled" version is indicated by p_u.

[3]Not the set of all *potential* dependencies, but the set of all dependencies asserted in the current package system.

- For each dependency of the version v of a package on $\texttt{A}_1$ | ..., accumulate a set S containing all the matching versions of each named package, combined with every package version that provides a named package (if the dependency is unversioned).

  For instance, if v declares a dependency on `A (>= 3.2) | B`, versions 3.1, 3.2, and 3.3 of package `A` are available, versions 1, 2, and 3 of package `B` are available, and package `C` version 3.14 provides `B`, then $S = \{\texttt{A}{:}\,3.2, \texttt{A}{:}\,3.3, \texttt{B}{:}\,1, \texttt{B}{:}\,2, B{:}\,3, \texttt{C}{:}\,3.14\}$.

  $\mathcal{D}$ contains $v \rightarrow S$ for every such dependency.

- For each conflict declared by a version v on the package p, accumulate a set S containing all the *non*-matching versions of p, including the "uninstalled" version, and insert $v \rightarrow S$ into $\mathcal{D}$. Furthermore, if the conflict is not versioned, then for each package p' and version v' of p' such that v' provides p, let $S = \{v'' \mid PkgOf(v'') = p' \wedge v''\ \text{does not provide}\ p\}$ and insert $v \rightarrow S$ into $\mathcal{D}$.

  For instance, if v conflicts with `A`, of which versions 3.2 and 3.3 are available, versions 2 and 3 of `B` provide `A`, and no other versions of `B` are available, then $S = \{\texttt{A}{:}\,3.2, \texttt{A}{:}\,3.3, \texttt{A}{:}\,\texttt{UNINST}, \texttt{B}{:}\,2, \texttt{B}{:}\,\texttt{UNINST}\}$.

  Note. In reality, extra care must be taken to screen out self-conflicts in this process, but the description above is complicated enough as it stands!

Remark. Although the above reduction is complicated to describe, its major steps must be performed whenever any program is analyzing dependencies: for instance, when listing all the versions that can fulfill a dependency, it is necessary to iterate over all members of each OR and to search their providing packages as necessary. Thus, an "on-the-fly" reduction in an algorithm written for the generic model is conceivably almost as efficient as an algorithm that works with the Debian package structure directly.

3.3 Installations

An *installation* represents a choice about which package versions to install; it is a function that maps each package to a version of that package.

Definition 1. An installation I installs a version v, written $I \rhd v$, if $I(PkgOf(v)) = v$.

Definition 2. An installation I satisfies a dependency $d = v \rightarrow S$ if either $I \not\rhd v$ or $I \rhd v'$ for some $v' \in S$.

Definition 3. An installation I is *consistent* if $I \vdash d$ for all $d \in \mathcal{D}$.

Definition 4. If I is an installation, then $I; p \mapsto v$ is a new installation which installs the version v of the package p and leaves all other packages unchanged:

$$(I; p \mapsto v)(p') = \begin{cases} I(p'), & p' \neq p \\ v, & p' = p \end{cases} \tag{1}$$

As a shorthand, the following notation indicates that a particular version of a package is to be installed:

$$I; v = I; PkgOf(()v) \mapsto v \tag{2}$$

4 The Dependency Resolution Problem

4.1 Problem Definition

Let I_0 be an inconsistent installation. We would like to find a consistent installation that is "similar to" I_0. This is the dependency resolution problem. In a real package manager, it corresponds to a situation in which the user's requests have resulted in an invalid state (with unsatisfied dependencies or conflicts); the goal of the package manager in this situation is to find a "small" and "good" set of changes that result in a valid state.

Note. This problem is poorly defined: "small" and "good" are not precise terms. The goal, from a UI point of view, is to not change too many packages, but to make reasonable decisions: for instance, if the user has requested that some packages be installed and these installations cause dependency clashes, "solving" the problem by cancelling the installations is probably not the desired result. However, while it might have obviously wrong solutions, *this problem has no principled correct solution*, because it is possible that if several different users view a single dependency problem, each prefers a different solution from the others. In other words, some of the information necessary to find the "best" solution is inside the user's head.

Thus, the best we can do is to define some criteria for "goodness" (to prioritize solutions that are more likely to interest the user) and allow the user to see alternatives to an undesired solution.

4.2 Dependency Resolution is NP-complete

In order to find a "good" solution, we must first find *any* solution to the existing set of dependencies. Unfortunately, as shown below, this is an NP-complete problem.

Theorem 5. *Dependency resolution is NP-complete.*

Proof. Proof is by reduction from CNF-SAT to the problem "does a consistent installation I exist?"

Create one package for each variable and for each clause in the SAT problem. For each variable x, let the versions of the corresponding package be $x\colon 0$ and $x\colon 1$; for each clause, create exactly one version. For each SAT clause let v_c

be the package corresponding to the clause, and insert $v_c \rightarrow S$ into $\mathcal{D}$, where for each literal of a variable x appearing in the clause, S contains $x\colon 0$ if x is a negative literal and $x\colon 1$ if x is a positive literal. This reduction is clearly polynomial-time; I claim that a solution to this set of dependencies exists if and only if a solution to the corresponding SAT problem exists.

Suppose that there is an assignment that solves the SAT problem. Define an installation I as follows: if p corresponds to a clause, $I(p)$ is the single version of p; if p corresponds to a variable x, $I(p) = x\colon 0$ if x is FALSE in the SAT solution and $I(p) = x\colon 1$ if x is TRUE in the SAT solution. Now, consider any dependency $d = v \rightarrow S$. From the construction above, S and v correspond to a clause of the SAT instance. At least one literal in this clause must be assigned the value TRUE (otherwise the clause is not satisfied); let x be the corresponding variable. If the literal is positive, then (by construction) S contains $x\colon 1$; since x must be assigned the value TRUE. $I \rhd x\colon 1$. Hence, $I \vdash d$. On the other hand, if the literal is negative, then S contains $x\colon 0$ and $I \rhd x\colon 0$, so $I \vdash d$. Thus, I is a consistent installation.

On the other hand, suppose that there is a consistent installation I. For all variables x, let p be the corresponding package; if $I(p) = x\colon 0$, assign FALSE to x, and if $I(p) = x\colon 1$, assign TRUE to x. Now consider any clause in the SAT problem: from the construction above, $\mathcal{D}$ contains a dependency $v_c \rightarrow S$ where v_c is the single version of the package corresponding to the clause. Since we must have $I \rhd v_c$ and since I is consistent, there must be a version $v' \in S$ such that $I \rhd v'$. But from the construction, there is some x such that v corresponds to either $x\colon 1$, where x appears as a positive literal in the clause or $x\colon 0$, where x appears as a negative literal in the clause. Thus, the clause is satisfied, and so the assignment described above satisfies all clauses.

Therefore, dependency resolution is NP-complete. $\square$

4.3 Don't Panic

Although the problem at hand is NP-complete in general, there is good reason to think that the instances that arise in practice are tractable. It is well-known that many NP-complete problems have "easy" and "hard" instances: some instances of the problem can be solved quickly by relatively naive algorithms, while others are intractable even using the most sophisticated techniques available.

In the particular case of package dependencies, the traditions that have grown up around package tools seem to encourage the creation of easy instances of the dependency problem; furthermore, the user's desired installation is typically consistent or "almost consistent" (meaning that few dependencies are violated). It is usually straightforward, when solving problems in an ad hoc way, to isolate a small part of the dependency graph in which the problem occurs; for instance, by informally applying a constraint such as "don't solve dependencies by removing core library packages". Once this is done, the problem can be declared either solvable or unsolvable on the basis of a quick analysis of that region of the graph.

In fact, when even relatively basic search techniques are applied to many typical dependency problems, the difficulties that arise are related not to a paucity of solutions, but rather to an excess of them. That is, finding *a* solution is easy, but finding the *right* solution is more problematic. Indeed, in the Debian framework there is always at least one solution: removing every package on the system will satisfy all the dependencies. However, for obvious reasons, this is not a solution that we want to produce!

4.4 Solving Dependencies Through Best-First Search

This problem statement suggests the use of a relatively simple algorithm – best-first search – to resolve dependencies. To briefly review, best-first search works by keeping a priority queue, known as the "open" queue, of potential (or partial) solutions. The priority queue is sorted according to some heuristic function that quantifies the "goodness" of each node (often in terms of nearness to a full solution). In each iteration of the algorithm, the "best" partial solution is removed from the queue. If it is a full solution, the algorithm terminates; otherwise, each "successor" node is generated and placed in the queue.

There are two main issues to resolve:

- How should successors be generated?
- What heuristic should be used?

To generate successors, we could simply enqueue all possible changes to a single package. However, this would result in a gigantic branching factor (over 1500 branches at each step in the current Debian archive), and it would cause the algorithm to consider adjusting packages that were utterly irrelevant to the problem at hand, as well as changing a package multiple times (which can lead to choices being made for reasons that are obscure to the user). A more focussed approach is needed.

Similarly, we could simply use the number of currently unsatisfied dependencies as our heuristic, but this does not provide any guidance as to how dependencies should be resolved. If `A` depends on `B`, `A` is installed, and `B` is not installed, it is usually better to install `B` than to remove `A`; however, a straight count of broken dependencies would consider both solutions to be equally "good".

4.4.1 Generating Successors to a Partial Solution

An obvious way of generating the successors of a given solution is to do it on the basis of unsatisfied dependencies. If the installation I does not satisfy the dependency $v \rightarrow S$, we know that v is installed but no member of S is. To resolve this dependency, we can either install a different version of $PkgOf(v)$ or install any element of S. Applying this rule to each "broken" dependency in turn will produce a set of successors that each solve at least one dependency (although they may break others in the process).

However, this approach still has the potential to "run in circles" by installing one version of a package, encountering broken dependencies, and then moving to a different version (possibly after resolving some dependencies of the intermediate version). The problem resolver of `apt`, for instance, sometimes confuses users by exhibiting this behavior. To fix this, I enforce a simple rule in generating solutions: a solution should never modify a package twice.

Definition 6. If the original installation was I_0, then for any I and any $d \in \mathcal{D}$ such that $I \not\models d$, the installation $I' = I; v$ is a successor of I for d if $v \neq I_0(PkgOf(v))$ and $I(PkgOf(v)) = I_0(PkgOf(v))$.

One might wonder whether this approach risks overlooking solutions: for instance, maybe it really is necessary to "go in circles" in order to find a particular solution. However, as shown below, if a solution cannot be generated through the application of the successor rule defined above, then there is a "simpler" version of that solution (one which modifies the states of fewer packages) that can be generated. To prove this, I first will introduce some definitions and notation.

Definition 7. Let I_1, I_2 be installations. The following notation is used to denote the "distance" from I_1 to I_2 (defined as the number of packages whose mappings differ between I_1 and I_2).

$$\langle I_1, I_2 \rangle = |\{p \mid I_1(p) \neq I_2(p)\}| \tag{3}$$

Definition 8. Let I_1, I_2 be installations. An installation I' is a *hybrid* of I_1 and I_2 if for all p, either $I'(p) = I_1(p)$ or $I'(p) = I_2(p)$.

Note. An alternative phrasing is that if I' is a hybrid of I_1 and I_2, then for all v such that $I' \rhd v$, either $I_1 \rhd v$ or $I_2 \rhd v$.

Definition 9. If I' is a successor of I with respect to I_0 for the dependency d, then $I \overset{I_0}{\underset{d}{\Rightarrow}} I'$. If there exist $I_1, \ldots, I_n$ and $d_1, \ldots, d_n$ such that $I_1 \overset{I_0}{\underset{d_1}{\Rightarrow}} I_2 \overset{I_0}{\underset{d_2}{\Rightarrow}} \ldots \overset{I_0}{\underset{d_{n-1}}{\Rightarrow}} I_n$, then $I_1 \overset{I_0}{\underset{*}{\Rightarrow}} I_n$.

Lemma 10. *Let I_c be any consistent installation (if one exists) and I_0 be any installation. For all hybrids I of I_0 and I_c and all dependencies $d \in \mathcal{D}$ such that $I \not\models d$, there exists an I' such that $I \overset{I_0}{\underset{d}{\Rightarrow}} I'$, I' is a hybrid of I_0 and I_c, and $\langle I', I_c \rangle < \langle I, I_c \rangle$.*

Proof. Consider any hybrid I of I_0 and I_c such that I is not a solution and any $d = v \rightarrow S \in \mathcal{D}$ such that $I \not\models d$.

Suppose that $I_c \not\rhd v$. Since I is a hybrid of I_0 and I_c, $I_0 \rhd v$. Thus, $I \overset{I_0}{\underset{d}{\Rightarrow}} I'$, where

$$I' = I; I_c(PkgOf(v)) \tag{4}$$

On the other hand, if $I_c \rhd v'$ for some $v' \in S$, then $I_0 \not\rhd v'$. Therefore, $I \overset{I_0}{\underset{d}{\Rightarrow}} I'$, where

$$I' = I; v' \tag{5}$$

In either case, clearly I' is a hybrid of I_0 and I_c and $\langle I', I_c \rangle < \langle I, I_c \rangle$, proving the lemma. □

Theorem 11. *For any consistent installation I_c and any inconsistent installation I_0, there exists a consistent installation I'_c such that I'_c is a hybrid of I_0 and I_c, and $I_0 \underset{*}{\overset{I_0}{\Rightarrow}} I'_c$.*

Proof. Proof is by repeated application of the previous lemma. Consider any inconsistent hybrid I of I_0 and I_c. Let I^+ be the I' shown to exist in the previous lemma for an arbitrary d such that $I \not\models d$, and define a sequence $I_1, \ldots$ as follows:

$$I_k = \begin{cases} I_{k-1} & \text{if } I_{k-1} \text{ is consistent} \\ I_k^+ & \text{otherwise} \end{cases} \tag{6}$$

I claim that this sequence converges; i.e., that for some finite n and all $m > n$, $I_n = I_m$. Proof: let $D_k = \langle I_k, I_c \rangle$ and $n = \langle I_0, I_c \rangle$. By the previous lemma, $D_k \geq D_{k+1}$ for all k, and $D_k = D_{k+1}$ if and only if I_k is a solution. Thus, if I_k is not a solution, we have $D_k \leq n - k$. But by definition, $D_k \geq 0$ for all k, so clearly I_{n+1} is a solution (else we have $0 \leq D_{n+1} \leq -1$).

Therefore, the theorem holds with $I'_c = D_{n+1}$. □

4.4.2 Scoring

The second key ingredient of a best-first search is a scheme for ordering search nodes, typically by assigning a numerical value to each prospective solution. In doing so, we must balance two priorities: the desire to find a solution quickly, and the desire to find a *good* solution.

The most obvious way to guide the search towards a solution is to "reward" avenues of inquiry that decrease the number of unsatisfied dependencies. This is not, of course, guaranteed to produce a solution quickly; however, in practice, it seems to be a sufficient hint for the algorithm to reach a goal node in a reasonable number of steps[4]. Finding "good" solutions is somewhat more difficult, not least because of the fact that "good" is an ill-defined property. The experimental implementation of this algorithm in `aptitude` uses the following general criteria to assign scores to nodes:

- Each version of each package is assigned a separate score. By default, removing any package is heavily penalized, altering packages which were automatically installed recieves a smaller penalty, maintaining the state of an automatic package makes no contribution to the score, and maintaining the state of a manually installed package receives a bonus.[5]

[4]Most searches seem to converge in under 5000 steps.

[5]In actuality, all that is calculated is the difference between the initial total version score and the final total version score.

- A penalty is applied to each search node based on its distance from the root of the search. This works to favor "simpler" solutions and penalize more complex ones.
- Nodes that resolve all dependencies are given an additional bonus – usually a relatively minor one. Goal nodes are moved through the priority queue in the normal manner, rather than being floated directly to its head, in order to ensure that solutions that are particularly "bad" are not produced unless it is absolutely necessary to do so.

Thus, letting $B(I)$ be the set of dependencies that are "broken" (not satisfied) by I and letting $h(v)$ be the score of the version v, the total score of an installation is

$$h(I) = \alpha_B|B(I)| + \alpha_L\langle I, I_0\rangle + \alpha_G\delta(0, |B(I)|) + \sum_{p\in\mathcal{P}} h(I(p)) \tag{7}$$

where α_B, α_L, and α_G are weighting factors and δ is the Kronacker delta function (i.e., $\delta(i, j)$ is 1 if $i = j$ and 0 otherwise). In the current implementation, $\alpha_B = -100$, $\alpha_L = -10$, and $\alpha_G = 50$.

5 Reducing the Branching Factor

5.1 One Dependency at a Time

The algorithm laid out above is sufficient to solve many of the dependency problems that are encountered in practice. However, some problems still cause the search to take an unacceptable amount of time to reach a solution. The problems observed fall into two main categories:

1. Too many reverse dependencies.

   In order to calculate the score of a successor of an installation (and of course to analyze that solution later on) it is necessary to generate the set of dependencies which are not satisfied by that successor. However, there are some one hundred thousand dependencies in the Debian archive; so that it completes in a reasonable amount of time, the current implementation uses the obvious optimization of only testing those dependencies which either were previously broken, which impinge on the package version being removed, or which impinge on the package version being installed.[6]

   Unfortunately, some packages have very many reverse dependencies. For instance, if I removes the system C library, over a thousand dependencies will be unsatisfied – and simply generating the successors of this node will require time at least *quadratic* in the number of reverse dependencies of `libc`. This can impose a significant performance penalty on the process of successor generation.

[6]Recall that a successor to I will install version v of p, removing $I(p)$ in the process.

2. Removing the bottom of a dependency chain.

   When an important library such as GTK is removed, it is necessary to propagate the removal "up the dependency tree". However, the search technique outlined above will search exponentially many installations before settling on this solution. Aside from the goal node of "keep the library on the system", the first step of the search will enqueue one node for each package depending on GTK; each node will remove its corresponding package. As these nodes are processed, pairs of packages will be removed from the system; then triples, and so on, until the full power set of all packages depending (directly or indirectly) on GTK is generated. Worse, at each step, solutions that suggest installing GTK (and removing many packages) will be generated.

There is a simple solution to both of these problems. Instead of generating successors for every dependency, it is sufficient to generate successors for a single, arbitrary dependency (as shown in Theorem 11). In theory, this could lead to somewhat less optimal ordering of generated solutions, but this doesn't seem to be a major problem in practice and the decrease in the problem's branching factor is well worth it.

5.2 Exclude Supersets of Solutions

One simple way to trim the search tree is to drop any search node I that is a "superset" of a full solution I_c – meaning that I_c is a hybrid of I and I_0. This has the additional beneficial effect of preventing solutions from being offered to the user which are just a previously-displayed solution with some extra, redundant actions added to it.

5.3 Forbidden versions

If we have a choice between removing `p` and installing `q`, and we choose to remove `p`, why should we ever install `q`? This question leads to yet another way of reducing the problem's branching factor.

To each solution node I, attach a set F of *forbidden* versions; the successors of I are restricted to those which do not install any version in F. For all successors I' of I, let $F' \subseteq F$; furthermore, if a successor I' of I is generated by removing the source version of a dependency, then all of the targets of that dependency are members of I'_F. This new successor relationship is formally defined in Figure 2 on page 15.

This has the effect of forcing the algorithm to "stick with" a decision to forgo installing the targets of a dependency in favor of shifting the source.

Note. This technique could just as well be applied by expanding the forbidden set when generating successors for the *targets* of a dependency: that is, forbidding a different version of the source of a dependency to be installed. The decision regarding which exclusion principle to use was made on the basis of a

$$\frac{I \not\models d \qquad d = v \to S \qquad I(v) = I_0(v) \qquad PkgOf(v') = PkgOf(v) \qquad v' \notin F}{(I, F) \overset{I_0}{\underset{d}{\Rightarrow}} (I; v', F \cup S)}$$

$$\frac{I \not\models d \qquad d = v \to S \qquad v' \in S \qquad I(v') = I_0(v') \qquad v' \notin F}{(I, F) \overset{I_0}{\underset{d}{\Rightarrow}} (I; v', F)}$$

Figure 1: Successor generation with forbidden versions

conjecture that we are more likely to encounter a “hard” dependency problem when moving “up” a dependency chain than when moving “down” it.

Of course, it is important to verify that cutting off wide swathes of the search space in this manner does not impede our ability to generate solutions:

Theorem 12. *Let I_c be any consistent installation (if one exists) and I_0 be any installation. There exists an I'_c such that I'_c is a hybrid of I_0 and $I_0 \overset{I_0}{\underset{*}{\Rightarrow}} I'_C$.*

Proof. Let $F_0 = \emptyset$. I claim that there exists a sequence $(I_1, F_1), \ldots$ such that for all $k \geq 0$,

- For all $v \in F_k$, $I_c \ntriangleright v$.
- $I_0 \overset{I_0}{\underset{*}{\Rightarrow}} I_k$
- Either $k = 0$, I_{k-1} is consistent and $I_k = I_{k-1}$, or $\langle I_k, I_c \rangle < \langle I_{k-1}, I_c \rangle$.

Proof is by induction on k. Suppose that a sequence $(I_1, F_1), \ldots, (I_k, F_k)$ exists satisfying the above condition. If I_k is consistent, then let $I_{k+1} = I_k$ and $F_{k+1} = F_k$; the inductive hypothesis is satisfied immediately.

Otherwise, consider any $d = v \to S \in \mathcal{D}$ such that $I_k \not\models d$ (since I_k is inconsistent, at least one such d exists). If there is a $v' \in S$ such that $I_c \triangleright v'$, then let $I_{k+1} = I_k; v'$ and $F_{k+1} = F_k$. Clearly $I_c \ntriangleright v''$ for all $v'' \in F_{k+1}$ and $\langle I_{k+1}, I_c \rangle < \langle I_k, I_c \rangle$; since we additionally have $(I_k, F_k) \overset{I_0}{\underset{d}{\Rightarrow}} (I_{k+1}, F_k)$, the inductive hypothesis holds.

If instead $I_c \ntriangleright v'$ for all $v' \in S$, then since I_c is consistent, $I_c(PkgOf(v)) \neq v$. Let $I_{k+1} = I_k; I_c(PkgOf(v))$ and $F_{k+1} = F_k \cup S$. $I_c \ntriangleright v''$ for all $v'' \in S$ by definition and clearly $\langle I_{k+1}, I_c \rangle < \langle I_k, I_c \rangle$. In addition, $I_k \overset{I_0}{\underset{d}{\Rightarrow}} I_{k+1}$ by Figure 2; therefore, the inductive hypothesis holds.

Thus, the claim is established: such a sequence exists. Following the logic of Theorem 11, we can see that for $n = \langle I_0, I_c \rangle$, I_n is a consistent installation. Furthermore, from the construction above, I_n is a hybrid of I_0 and I_c. Thus, the theorem is established with $I'_c = I_n$. □

5.4 Use Logical Necessity

In combination with the tracking of forbidden versions, it is also possible to detect *forced installations* and *essential conflicts*. A forced installation is one which is logically necessary given I and F: for instance, if we have $d = v \rightarrow \{v'_1, v'_2\}$, I has touched v (i.e., $I(v) \neq I_0$), $v'_1 \in F$, and $v'_2 \notin I_F$, then the only permissible successor given d is $(I; v'_1)$. An essential conflict is a dependency for which *no* successors can be generated: for instance, if in the previous example we instead had $v'_2 \in F$, then d would be an essential conflict.

If any essential conflicts exist in an installation I, it is discarded immediately (rather than, for instance, generating successors for all the solvable dependencies). If any forced installations exist, they are accumulated and a successor formed by adding these installations to I is placed into the open queue.

6 Non-mandatory Dependencies

In addition to the standard Depends metadata, Debian also has a class of dependencies known as "recommendations". In the words of section 7.2 of Debian's technical policy:

> Recommends: This declares a strong, but not absolute dependency.
>
> The Recommends field should list packages that would be found together with [the recommending package] in all but unusual installations.

Package management frontends adopt a variety of strategies to deal with recommendations, ranging from completely ignoring them to treating them nearly as strictly as dependencies. The current best practice seems to be the rule "install recommendations when a package is first installed; ignore them otherwise".

In this section, I will propose one way in which the above theory and algorithm can be extended to accomodate these non-mandatory relationships.

6.1 "Hard" and "Soft"

The information content of a recommendation is equivalent to that of a dependency, and so it makes sense to represent a recommendation in our formal model as a special type of dependency. I will divide dependencies into two classes: "hard" dependencies and "soft" dependencies. "soft" dependencies, of course, represent recommendations[7].

Now, although "soft" dependencies need not be satisfied in an eventual solution, we would like the algorithm to at least *try* to satisfy them, and in fact it should to make a reasonably significant effort to satisfy them. In order to ensure that this is done, I suggest the following techniques:

[7]Or, to be more precise, recommendations of packages that are not presently installed, in accordance with the abovementioned rule.

$$\frac{I \not\models d \qquad d = v \to S \qquad \overset{d \notin C}{I(v) = I_0(v)} \qquad PkgOf(v') = PkgOf(v) \qquad v' \notin F}{(I, F, C) \overset{I_0}{\underset{d}{\Rightarrow}} (I; v', F \cup S, C)}$$

$$\frac{d \notin C \qquad I \not\models d \qquad d = v \to S \qquad v' \in S \qquad I(v') = I_0(v') \qquad v' \notin F}{(I, F, C) \overset{I_0}{\underset{d}{\Rightarrow}} (I; v', F, C)}$$

$$\frac{I \not\models d \qquad d \text{ is soft}}{(I, F, C) \overset{I_0}{\underset{d}{\Rightarrow}} (I, F, C \cup \{d\})}$$

Figure 2: Successor generation with soft dependencies

- Extend the state of search nodes with an additional set C, representing the dependencies that have been "closed" by being examined at least once. As shown in Figure 2, extend successor generation to permit the algorithm to "give up" on any open soft dependency: in addition to generating successors for the various way of solving that dependency, it will also generate a successor in which no package states are changed, but the dependency is closed anyway.
- Penalize broken soft dependencies, to reward solutions that fulfill soft dependencies.

This has not yet been tested, but will likely require some rebalancing of the various weighting factors previously discussed in order to produce reasonable results.

7 Implementation

A prototype implementation of this resolver algorithm exists in the experimental branch of `aptitude`. The implementation is composed of two pieces, which are assembled via C++ templates: a search algorithm for a generic dependency problem, and a runtime translation of APT dependencies to the generic form outlined above. It does not implement "soft" dependencies, although their future inclusion is planned.

The current implementation seems to perform reasonably well: in the cases that I have tested, solutions are generated quickly enough for interactive use. However, the order in which solutions are offered is sometimes surprising: for instance, if the installation of a package causes problems, it is common for the first generated solution to be "cancel this installation". While, as noted above, there is no perfect solution even in principle and any static weighting is likely to occasionally produce odd results, I expect that some of these problems can be fixed through adjustments of the score function.

8 Future Work

As noted above, the score function needs to be adjusted and soft dependencies need to be supported. In addition, some consideration of the following questions seems worthwhile to me:

1. Is it ever possible to "divide and conquer" a dependency problem?

   Rationale: as noted towards the beginning of this paper, informal analyses of dependency problems often seem to adopt a "divide and conquer" approach. Moreover, such an approach would have several important user interface benefits: for instance, it would avoid the tendency of the algorithm to produce the Cartesian product of all the different ways to solve each isolated group of dependency problems.

   I do not, however, see an obvious simple way of performing such a division.

2. Can and should "overly similar" solutions be detected and dropped?

   When a solution implicates a large number of packages, the current algorithm tends to produce many solutions which differ only slightly from one another. From a user-interface perspective, it might be desirable to drop some of these solutions. What metric, if any, should be used to perform this dropping?