School of Engineering and Science

Master's Thesis

Solving the Bootstrap Problem for Free and Open Source Binary Distributions

Johannes Schauer

May 2013

Supervisor: Prof. Dr. Andreas Nüchter
Second supervisor: Dr. Pietro Abate

Abstract

To bootstrap a free and open source binary distribution means to recompile all its software packages for an architecture the operating system does not exist for yet. This task presents a major challenge because of the presence of millions of dependency cycles between software packages. Breaking those cycles is still done using a yearlong manual trial and error approach. Since the amount of interdependencies between software components grows over time, the task of manually bootstrapping a binary software distribution is expected to only become harder in the future.

In this thesis we present a solution to the *bootstrap problem*. We present tools, algorithms and techniques which turn bootstrapping one of the largest open source binary distribution into an automated, deterministic and fast process. Our tool chain allows to generate a dependency graph and assists in its analysis by offering several heuristics until in the end a build order can be devised. One of the heuristics we use for dependency graph analysis is a new cycle based approximate solution to the Feedback Arc Set Problem which outperforms existing heuristic solutions for this problem domain.

Contents

1 Introduction **1**
- 1.1 Binary Based Distributions . . . 1
- 1.2 The Bootstrap Problem . . . 2
- 1.3 Contribution of this Work . . . 4
- 1.4 Choice of Debian GNU/Linux . . . 5
- 1.5 Structure of this Thesis . . . 6

2 Basic Concepts **7**
- 2.1 Package Relationships . . . 7
- 2.2 Implicit Dependencies . . . 8
- 2.3 Architecture . . . 8
- 2.4 Identifying Packages . . . 9
- 2.5 Multiarch . . . 9
- 2.6 Multiarch Cross . . . 10
- 2.7 Control File Format, Packages and Sources Files . . . 10
- 2.8 Dependency Closure and Installation Sets . . . 11
- 2.9 Strong Dependencies . . . 12
- 2.10 Repositories and Metadata Repositories . . . 13
- 2.11 Availability . . . 14
- 2.12 Installability . . . 14
- 2.13 Build Dependency Cycles . . . 15
- 2.14 Build Profiles . . . 15

3 Dependency Graphs **17**
- 3.1 Graph Manipulation Pseudo Code . . . 17
- 3.2 Structure of the Build Graph . . . 18
- 3.3 Calculating and Partitioning Installation Sets . . . 19
- 3.4 Generating the Build Graph . . . 21
- 3.5 Structure of the Source Graph . . . 22
- 3.6 Generating the Source Graph . . . 23
- 3.7 Strong Source Graph . . . 23
- 3.8 Optimal Dependency Graph . . . 24
- 3.9 Rationale Behind Choice of Graph Types . . . 24

4 Toolset **27**
4.1 Distcheck 27
4.2 Buildcheck 28
4.3 Coinstall 28
4.4 Package Filter 28
4.5 Package Union, Intersection, Difference 29
4.6 Binary to Source and Source to Binary 29
4.7 Calculate Fixpoint 29
4.8 Calculate Build Closure 30

5 Processing Pipeline **33**
5.1 Preferring Native Compilation over Cross Compilation 33
5.2 Self-Contained Metadata Repository 34
5.3 Cross Phase 36
5.4 Native Phase 36

6 Enumerating all Cycles of a Directed Graph **41**
6.1 Implementation 41
6.2 Evaluation 43

7 A New Heuristic for the Feedback Arc Set Problem **45**
7.1 State of the Art 45
7.2 Preprocessing and Postprocessing Steps 46
7.3 A New Cycle Based Heuristic 47
7.4 Sorting Heuristics 48
7.5 Evaluation 49
7.6 Application to a Build Graph 52

8 Dependency Graph Analysis **55**
8.1 Need of a Human Developer 55
8.2 Finding Build-Depends Edges to Remove 56
8.2.1 Degree Based Heuristics 56
8.2.2 Enumerating Cycles 57
8.2.3 Strong Bridges and Strong Articulation Points 58
8.3 Removing Build-Depends Edges 58

9 Creating a Build Order **61**
9.1 Feedback Vertex Set Algorithm 61
9.2 Collapsing Strongly Connected Components in a Source Graph 63
9.3 Computing a Partial Order 64

10 Experimental Results **67**
10.1 Gentoo 67
10.2 Cross Phase 68
10.3 Setup 68

10.4 Benchmarks 69

11 Conclusion **71**
11.1 Related Work 71
11.2 Future Developments 72
11.3 Acknowledgments 72
11.4 Conclusion 73

Bibliography **75**

List of Listings **79**

List of Algorithms **81**

List of Figures **83**

List of Tables **85**

Chapter 1

Introduction

The term “bootstrapping” supposedly originated from the phrase of “pulling oneself over a fence by one’s bootstraps”, indicating an impossible action. This thesis describes methods to bootstrap one of the biggest free and open source distributions from zero to tens of thousands of packages. The supposed impossibility of this endeavor lies within millions of dependency cycles between software packages which have to be broken before a linear build order can be derived. Using an approximate solution to the Feedback Arc Set Problem, it will be described how a dependency graph can be made acyclic by only breaking a close to minimal amount of dependencies.

Most, if not all of today’s free and open source operating systems are package based. This division of the operating system into packages or *components* [46] is a natural consequence of the heterogeneous nature of those components. In a usual free and open source distribution, each of its components is being developed by a different group of people, in different programming languages, with different licenses, installation instructions, release cycles and terminologies. From the perspective of the developers of such a distribution, the developers of those components are called *upstream*. The person creating, upgrading, fixing bugs and otherwise maintaining a package within a distribution is called *maintainer*. The task of the developers or maintainers of a software distribution is, to make all its component work well together for the end user in a coherent manner. In the context of free and open source distributions we call those components *packages*.

1.1 Binary Based Distributions

Binary based distributions distinguish between *source packages* and *binary packages*. Source packages carry the source code of an upstream software project together with distribution specific metadata information and build instructions. Binary packages carry compiled executables and data files and are deployed on the system of the end user. Since source packages and binary packages often share the same name, we prefix source package names by `src:`. A source package with the name `foo` would here be written as `src:foo`.

Binary packages and source package are related to each other. Binary packages can declare *binary dependencies* on other binary packages. This kind of dependency indicates which other

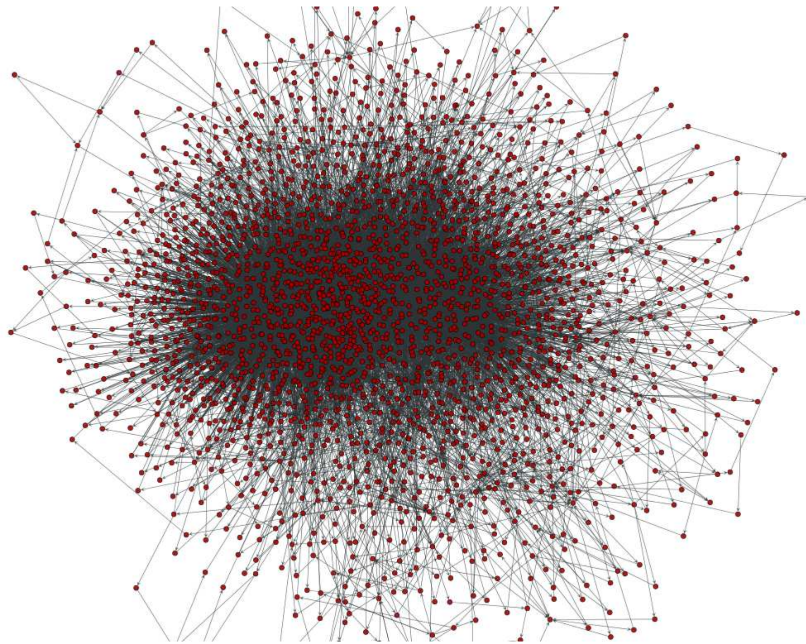


Figure 1.1: A strongly connected component of 977 vertices and 9073 edges. Due to its appearance and intractability, this representation of the graph is also called a "hairball" [31].

binary packages must be installed for a given binary package to be installable.

Source packages can declare *build dependencies* on binary packages. This kind of dependency indicates which binary packages must be installed for a given source package to be compilable. Source packages produce binary packages when they are compiled. Therefore binary packages relate to source packages by the source package they *build from*.

Usually, upstream software can be compiled with different sets of features enabled or disabled. The more features are enabled, the more binary packages are needed to be installed for the upstream software to compile successfully. For quality assurance, simplicity and reproducibility, binary based distributions compile their upstream sources with all possible features enabled. This avoids having to ship different versions of binary packages which each carry a different feature set. It simplifies dependencies between binary packages and makes it easier to reproduce software bugs. Therefore, the produced set of binary packages always carries the maximum amount of available features the upstream project offers. This means that most source packages need the maximum amount of build dependencies that the associated upstream project can require. This practice does not create any problems in the usual life cycle of a free and open source distribution as all binary packages are always available for installation. Therefore, even long lists of build dependencies can be satisfied without particular effort.

1.2 The Bootstrap Problem

The only time during the development of a binary based distribution where above practice creates problems is when the distribution is to be bootstrapped for a new architecture. A source

package `src:A` might need a binary package `B` to satisfy its build dependencies and thus become compilable. But the binary package `B` builds from the source package `src:B` which in turn needs binary packages created by `src:A` to be compilable. Since `src:A` cannot be compiled, `src:B` cannot be compiled either and vice versa. *Build dependency cycles* are created. The cycle in this example is between `src:A` and `src:B`. In reality, cycles contain up to a thousand elements where each element is involved in multiple cycles of similar length. In a dependency graph, those cycles form strongly connected components. In a typical dependency graph multiple strongly connected components of up to a thousand vertices exist. No source package that is part of a strongly connected component can be compiled until all the others are compiled. Thus, they all block each other from being compiled. Finding a way to make the dependency graph acyclic to allow the creation of a linear build order is called the *bootstrap problem*.

Every time the distribution is bootstrapped for a new architecture, the developer doing the bootstrap has to manually identify dependency cycles and break them by relaxing build dependencies of source packages. In above example, the cycle could be broken by `src:A` not build depending on `B` anymore. Then `src:A` could be built first, making the binary package `A` available. With this binary package, `src:B` can be built and thus make the binary package `B` available. This process usually takes months up to a year of human labor. The process is this lengthy because of the following reasons.

- There exist no tools for creating a dependency graph. A developer has to draw parts of the graph by manual depth first search of build and binary dependencies.
- The actual dependency graph contains nontrivial strongly connected components of up to 1000 vertices and 9000 edges. Figure 1.1 shows such a component. The solution a human can find to make such a component acyclic requires to modify much more source packages than are actually necessary.
- Meta data about which build dependencies can be dropped from source packages during a bootstrap is not available, so the developer has to find potential candidates by hand.
- Package relationships change throughout the life cycle of a distribution so above steps have to be re-evaluated manually every time a bootstrap is done.
- Cross compilation is not supported by source packages.

A typical work flow of a developer attempting a bootstrap would start with a *minimal build system* on the new architecture. This minimal build system has to be either cross compiled or a different distribution with better cross compilation capabilities has to be used as a basis. Starting from there, the developer would use his inside knowledge and intuition to select candidate source packages he wants to compile on the new platform. Since some of the build dependencies will not be available he will either modify that source package so that it can be compiled, or recurse this process deeper into the dependency tree. This process has to be repeated until it is possible to compile all source packages of the distribution which usually entail over ten thousand packages.

Furthermore, as can be seen in Figure 1.2, the size of the largest strongly connected in the dependency graph grows over time. So while in the past, dependency situations as complex as seen in Figure 1.1 were solved by hand, it is likely that the bootstrap problem and the work

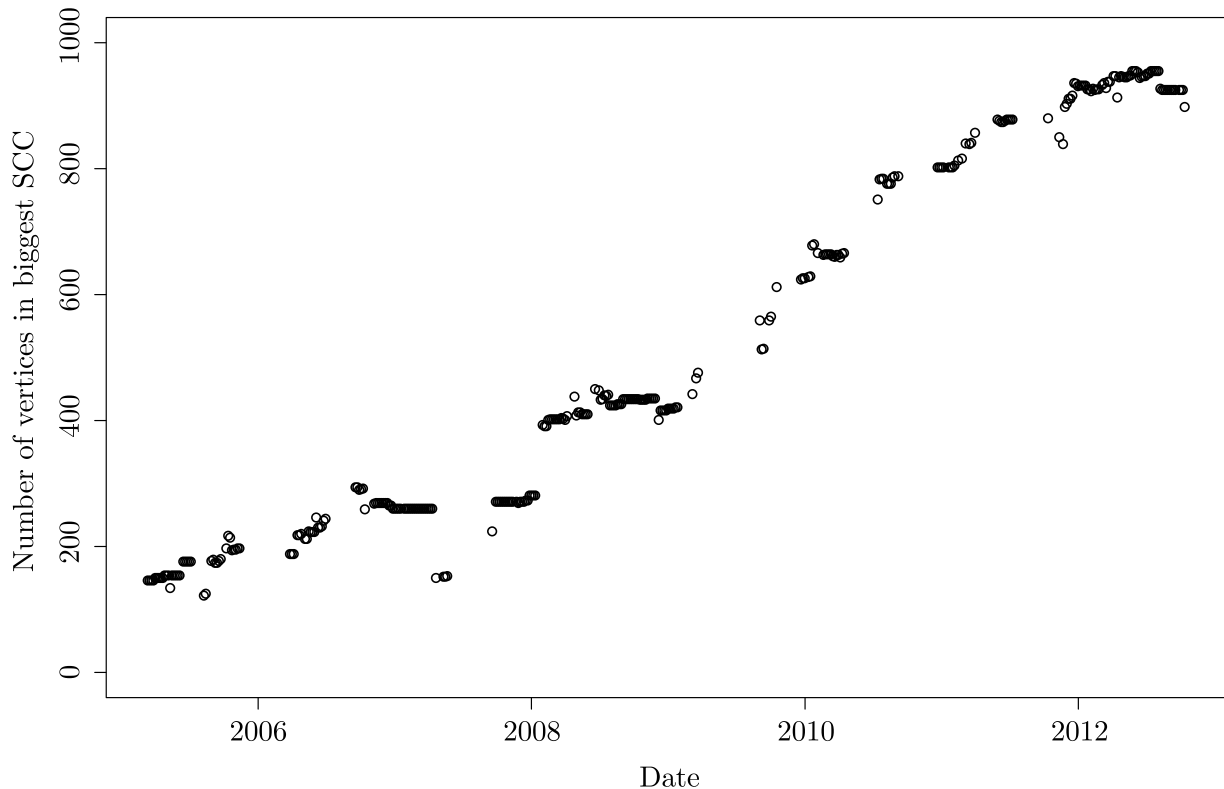


Figure 1.2: Amount of vertices in the biggest strongly connected component in the build graph of Debian Sid from 2005 to 2012

involved to complete it will continue to grow. This trend can be explained by the growing complexity of the source packages a distribution contains. As individual upstream software projects offer more functionality they also increase their amount required of build dependencies.

1.3 Contribution of this Work

In this thesis we show how the bootstrap problem was solved by making it automatic, deterministic and fast. With these properties it will not only be much easier to bootstrap a distribution but it will also be possible to continuously check a distribution for its current state of bootstrappability as a quality assurance measure.

We introduce a collection of tools and libraries called `botch`. `Botch` stands for Bootstrap/Build Order Tool Chain and is written in the OCaml programming language with some helper programs written in Python and Shell. It heavily depends on the `dose3` [5] library and the `ocamlgraph` [15] library. It is released under the terms of the LGPL and can be retrieved from gitorious [3]. More specifically, `botch` allows:

- To create customizable dependency graphs which can be annotated with metadata information and are output in the standard GraphML [12] format for easy processing by third party tools.

- To make big strongly connected components acyclic by using a new heuristic solution to the Feedback Arc Set Problem [37].
- To further analyze components by using an implementation of Johnson's algorithm [28] as well as degree based heuristics and algorithms to find strong bridges and strong articulation points [26].
- To calculate a build order with modifications to only a minimal amount of source packages through usage of a Feedback Vertex Set algorithm.
- To be used as a continuous integration tool for quality assurance, to check whether bootstrapping is still possible throughout the lifetime of a distribution.

The work done by `botch` is of purely theoretical nature. At no times are source packages actually compiled or are binary packages installed on a system. Instead, the analysis is done only on their metadata. A source package is assumed to be compilable if its build dependencies are met. A binary package is assumed to be installable if its binary dependencies are met. As package metadata is supposed to sufficiently describe the compilability of source packages as well as the installability of binary packages, the calculations done by `botch` should produce a valid result.

Further improvements were made to `dose3` to support multiarch cross compilation and build profiles. Build profile support was also added to the Advanced Package Tool (apt). Another contribution was the development of a test suite for various cycle enumeration algorithms as currently no well tested library includes such an algorithm. Lastly, another subproject was the development of a system mapping packages and dependencies between different distributions for a heuristic to find droppable build dependencies.

1.4 Choice of Debian GNU/Linux

This thesis will use the binary based distribution Debian GNU/Linux to demonstrate the functionality of `botch`. Our software is also fully compatible with all Debian derivatives like Ubuntu or Linux Mint. The choice of Debian for this work was made for the following reasons.

- **Popularity** Debian and its derivatives make 48% of unique daily visitors of distrowatch.com over the past 12 months (status January 2013).
- **Existing Research** Current research focuses on Debian and its derivatives [1,2,9,19,38, 49]. This makes it possible to draw from a large pool of existing expertise and toolsets like `dose3`.
- **Multiarch** No other distribution offers a general and formalized way to install binary packages of different architectures on the same system. This greatly simplifies cross build dependency resolution and even allows to break native dependency cycles.
- **Architecture Support** Debian is one of the only binary based distributions which offers support for more than a handful of architectures. Debian is being bootstrapped for a new

architecture about once per year. This makes the Debian project especially interested in a solution to the bootstrap problem.

- **Industry Backing** Companies like Linaro are regularly bootstrapping Debian based distributions like Ubuntu to the latest ARM based platforms. Currently the work of this thesis is being used for bootstrapping Ubuntu to the upcoming 64-bit ARMv8 instruction set architecture.
- **Size** As of January 2013, Debian based distributions contain the most number of packages (Ubuntu: 47600, Debian 38000). They are therefore best to show the feasibility of the developed algorithms on even the biggest existing problem sets.
- **Available History** Besides being able to retrieve package metadata back to the first release in 1995, all Debian repositories have been saved in six hour intervals since 2005 and therefore provide data to analyze the historical development of the bootstrap problem (see Figure 1.2).

1.5 Structure of this Thesis

After this introduction, chapter 2 will introduce the basic concepts needed to understand the chapters following afterwards. Chapter 3 will handle the different types of dependency graphs and how they are generated. In chapter 4 the tools and filters used for package selection and processing are explained. The following chapter 5 will then connect these tools to form processing pipelines. The output of them will be used as input for the algorithms generating dependency graphs. Chapter 6 describes Johnson's Algorithm for enumerating elementary cycles and our modifications to it. Chapter 7 introduces a new approximate solution to the Feedback Arc Set Problem based on cycle enumeration. This solution to the Feedback Arc Set Problem is one of the heuristics for dependency graph analysis as they are presented in chapter 8. The following chapter 9 then handles how a build order is generated from an acyclic graph. In chapter 10 we present how `botch` performs on real input data before drawing conclusions in chapter 11.

Chapter 2

Basic Concepts

This chapter will explain the fundamental concepts that are necessary to understand the contributions of this thesis. The chapter begins with items that are already well understood by developers of Debian based distributions like package relationships, multiarch and the control file format. All those concepts are documented in the Debian Policy Manual [27]. It follows terms like dependency closures, installation sets and strong dependencies which have their roots in academic publications. In the end we explain the basic concepts behind bootstrapping like availability, build dependency cycles and build profiles.

2.1 Package Relationships

Source as well as binary packages in Debian based distributions can declare explicit and implicit relationships to other binary packages. The only relationship that can be made toward source packages is the *builds-from* relationship. It can only be made by binary packages and indicates from which source package a binary package builds. In algorithms throughout later chapters, we use the function $s = \textsc{Source}(b)$ to retrieve the source package s from which a binary package b builds.

All other relationships are toward binary packages. One such relationship is the inverse of the builds-from relationship. It indicates for every source package, which binary packages it produces once it is compiled. In algorithms we call this function $B = \textsc{Binary}(s)$ to retrieve the set of binary packages B a source package s builds.

Other relationships govern which binary packages must be installed or not be installed to install other binary packages or to compile source packages. These relationships can either be positive or negative and thereby create a dependency or conflict relationship, respectively.

Binary packages specify them to indicate what other binary packages they require to be installed or not installed on the same system. Any given binary package can only be installed if all dependencies are met and none of its conflicts apply. These types of relationships between binary packages are called *binary dependencies* or *binary conflicts*, respectively.

Source packages specify relationships to binary packages to indicate what binary packages they require to be installed or not installed for successful compilation of the source package. A source package can only be compiled once all its dependencies are met and none of its conflicts

apply. These types of relationships of source packages toward binary packages are called *build dependencies* or *build conflicts*, respectively. If the source package is being cross compiled, the relationship is called *cross build dependency* or *cross build conflict*, respectively.

Binary dependencies of binary packages as well as build dependencies of source packages are expressed in conjunctive normal form. This allows different sets of binary packages to satisfy a binary or source package's dependencies. In algorithms we use the function $D = \textsc{Deps}(p)$ to retrieve the set of disjunctions D specified in the dependency relationship of a binary or source package p. Binary conflicts as well as build conflicts are expressed as a conjunction.

2.2 Implicit Dependencies

Some binary packages are marked as `Essential:yes`. Every binary package implicitly depends on all binary packages that are marked as being essential. This means that all essential packages (and their binary dependencies) must always be present on every Debian based system.

Source packages always implicitly build depend on the package `build-essential`. This package explicitly depends on the C library development headers, the GCC and G++ compilers, GNU make and Debian specific development tools. This means that source packages do not need to explicitly build depend on essential build tools themselves. On the other hand, even source packages that do not need a C compiler to be built implicitly depend on the `build-essential` package.

A not yet officially decided upon implicit build dependency was suggested for cross compilation. When cross compiling a source package for architecture `X`, then the package `crossbuild--essential-X` is an implicit build dependency. The package `crossbuild-essential-X` would then itself explicitly depend upon a C cross compiler for architecture `X`. A patch was written that implements this additional implicit dependency for `dose3`.

2.3 Architecture

Binary and source packages specify their architecture. Binary packages specify the architecture for which the binaries they contain were compiled for. If their content is architecture independent, then they are assigned to the special architecture `all` and are thus called `Architecture:all` binary packages. Source packages specify the architecture for which they can be natively compiled. Source packages can therefore specify a list of architectures. It is also possible to assign the special architecture `any` to them. It means that this source package can be compiled for any architecture.

During cross compilation, a source package is compiled on one architecture for another architecture. The GNU terminology for these architectures is *build architecture* for the architecture the source package is built **on** and *host architecture* for the architecture the source package is built **for**. The new architecture for which the bootstrap is carried out is called the *target architecture*. During cross compilation, the host architecture is the target architecture. During native compilation on the target architecture, build and host architecture equal to the target architecture.

2.4 Identifying Packages

To successfully establish inter-package relationships, a unique way to refer to packages is necessary. In Debian, binary packages are unique if their name, version and architecture matches. Source packages are unique if their name and version matches. Binary and build dependency relationships toward binary packages have to mention the name of the binary package and can optionally specify a version restriction. Debian package managers already allow to depend on a binary package of a specific architecture as well but this is not yet supported by the Debian archive tools. Internally, `botch` and `dose3` represent binary packages and source packages as CUDF package instances. CUDF stands for Common upgradeability description format [50] and is able to encode inter-package relationships not only for Debian packages but also for RPM [20] and Eclipse OSGi packages [22].

2.5 Multiarch

The default case in binary distributions is that only binary packages of one architecture can be installed on a system. Since there was a growing need of users of the `amd64` architecture to have libraries of the `i386` architecture available, the package `ia32-libs` was introduced. `ia32-libs` was a binary package containing hundreds of `i386` shared libraries but was marked as `amd64` so that it could be installed on `amd64` systems. It allowed executing proprietary `i386` applications which are not available for `amd64`. Multiarch [32] was introduced to solve that problem in a clean way and unified way for all architecture combinations. It also led to other advantages in the area of cross build dependency resolution and breaking of dependency cycles which is of specific importance for this work.

Multiarch is a feature which is only found in Debian based distributions. It allows to simultaneously install binary packages of different architectures next to each other. It also allows the binary package of one architecture to satisfy the dependencies of another binary package of a different architecture. Since the tools presented in this thesis make use of it, multiarch support of the `dose3` library was extended.

Multiarch is a property of binary packages. The multiarch field can have three possible values. A summary of these rules can be seen in Table 2.1 if the last column is ignored as that column only applies for multiarch cross dependency resolution.

- `Multi-Arch:same` A package with the same name and version but different architecture can be installed alongside this package. If a package of architecture `X` depends on a `Multi-Arch:same` package `A`, then `A` must be available in architecture `X` as well.
- `Multi-Arch:foreign` This package can not be installed together with a package of the same name but different architecture. Instead, this package is able to satisfy the dependencies of a binary package of a different architecture than itself. If a package of architecture `X` depends on a `Multi-Arch:foreign` package `A`, then `A` of any architecture can satisfy this dependency.
- `Multi-Arch:allowed` Not this package but the package depending on it decides which architecture of it satisfies its dependencies. If a package depends on a `Multi-Arch:allowed`

	foo	foo:any	foo:native
no Multi-Arch field	host architecture	-	build architecture
Multi-Arch: same	host architecture	-	build architecture
Multi-Arch: foreign	any architecture	-	-
Multi-Arch: allowed	host architecture	any architecture	build architecture

Table 2.1: Multiarch cross build dependency resolution for different build dependencies on `foo` for different multiarch properties of `foo`

package `A`, then it can declare which architecture of `A` it requires.

2.6 Multiarch Cross

An extension to the multiarch specification, describes how to use multiarch for cross compilation [43]. Support for multiarch cross dependency resolution was added to `dose3`. Table 2.1 visualizes the established dependency resolution rules.

Each of the columns represents one of the three ways a source package can express a cross build dependency on a package `foo`. The rows represent different packages `foo` with different values for their `Multi-Arch` field. The table cells show which architecture of `foo` is picked for each combination or whether a combination is invalid.

Without any qualifier, build dependency resolution works like normal multiarch binary dependency resolution: the build dependency must be of the host architecture except if the build dependency is `Multi-Arch:foreign`. Using the `any` qualifier, `Multi-Arch:allowed` packages of any architecture will satisfy that build dependency. Using the `native` qualifier, all except `Multi-Arch:foreign` packages satisfy that build dependency in their build architecture.

Since during native compilation the build architecture is equal to the host architecture, these cross build dependency resolution rules have no effect during native compilation. In particular, if in above table "build architecture" was replaced by "host architecture", then the resulting rules would be the same as explained in the last section.

2.7 Control File Format, Packages and Sources Files

The aforementioned metadata like package name, version, architecture and dependencies is stored in Packages or Sources files. They store metadata for a set of binary and source packages respectively and are usually distributed by distribution archive mirrors. All these files share the same format, called *control file format* which is based on the RFC822 [16] format, commonly known for its use in email.

Listing 2.1 shows the metadata for a package called `libgpm-dev` (line 1). It has version `1.20.4-6` (line 2) and is specific to the architecture `amd64` (line 3). Since it is `Multi-Arch:same` (line 4) it can be installed together with versions of this package that were built for other architectures. Since it is not essential (line 5) it is not implicitly depended upon by every other package. The source package it builds from is called `gpm` (line 6). It provides another package

called `libgpmg1-dev` (line 7). Therefore any package depending on `libgpmg1-dev` will have this dependency satisfied by `libgpm-dev` as well. For it to be installed, the package `libgpm2` in exactly version `1.20.4-6` must be installed (line 8). Additionally, either `libc6-dev` or `libc-dev` must be installed. This line expresses the aforementioned conjunctive normal form by expressing logical conjunctions with a comma and logical disjunctions with a pipe symbol. `libgpm-dev` can never be installed together with the package `libgpmg1-dev` as it conflicts with it (line 9).

```
1 Package: libgpm-dev
2 Version: 1.20.4-6
3 Architecture: amd64
4 Multi-Arch: same
5 Essential: false
6 Source: gpm
7 Provides: libgpmg1-dev Depends: libgpm2 (= 1.20.4-6), libc6-dev | libc-dev
8 Depends: libgpm2 (= 1.20.4-6), libc6-dev | libc-dev
9 Conflicts: libgpmg1-dev
```

Listing 2.1: The metadata for the binary package `libgpm-dev`

Listing 2.2 shows the metadata for the source package `gpm` (line 1) from which the binary package `libgpm-dev` of listing 2.1 builds. It has the same version number `1.20.4-6` as the binary package that was built from it (line 2). The binary packages a source package builds have the same version as the respective source package. It can be built on any architecture (line 3). If it is to be built, the binary packages `autoconf`, `autotools-dev`, `quilt`, `bison`, `texlive-base`, `texinfo` and `texi2html` must be installed first. Additionally the package `debhelper` must be available in a version greater or equal than `6.0.7`. Either the package `mawk` or `awk` must be installed too.

```
1 Package: gpm
2 Version: 1.20.4-6
3 Architecture: any
4 Build-Depends: autoconf, autotools-dev, quilt, debhelper (>= 6.0.7),
5  mawk | awk, bison, texlive-base, texinfo, texi2html
```

Listing 2.2: The metadata for the source package `gpm`

2.8 Dependency Closure and Installation Sets

A dependency closure is the set of binary packages closed under their dependency relationship. This is, if all dependency relationships from a starting binary or source package were followed (including all disjunctions) recursively, then the resulting set of visited binary packages would

form the dependency closure of the starting binary or source package. The dependency closure of a binary or source package therefore contains all binary packages that this binary or source package directly or indirectly could possibly require to satisfy its runtime or build dependencies respectively.

Since disjunctive dependencies are also followed, a dependency closure often contains more binary packages that are actually necessary to fulfill a given binary or source package's dependencies. It is also possible for two packages in a dependency closure to conflict with each other. To retrieve the dependency closure DC of a binary or source package p, we use the function $DC = \textsc{DependencyClosure}(p)$ in algorithms throughout later chapters.

An installation set is a subset of a dependency closure. An installation set must fulfill the following properties:

- Every package in the installation set has its dependencies satisfied
- No two packages in the installation set conflict with each other

Usually there exist different choices of installation sets and choosing one is the task of a solver and its given metrics. We use the function $IS = \textsc{Compute}\text{IS}(p)$ to return one valid installation set IS for a given binary or source package p.

The set of packages which satisfies the dependencies for installing or building more than one binary or source package is called a co-installation set. The same conditions as for normal installation sets hold. Some sets of binary or source packages might not have a co-installation set as they themselves or their respective dependencies conflict with each other.

Dependency closures, installation sets and co-installation sets can be formally defined. Their definitions can be found in [38].

2.9 Strong Dependencies

The set of *strong dependencies* of a binary or source package is the intersection of all its possible installation sets. In other words: a binary package is a strong dependency of another binary (or source) package, if the latter cannot be installed (or compiled) without the former. The strong dependency set therefore creates a lower bound and at the same time a strict minimum requirement for installation or build dependency satisfaction of any binary or source package. The concept of strong dependencies was first introduced in [1].

Non-disjunctive dependencies of a package are automatically also a strong dependency but it is wrong to assume that disjunctive dependencies are automatically not strong. Consider the dependency situation between the packages `a`, `b` and `c` as shown in Listing 2.3. In this scenario there is an immediately spottable strong dependency of `c` on `b`. But `b` is also a strong dependency of `a` even though it appears in a disjunction. It is so because it is not possible to install `a` without `b` because even if `c` was chosen from the disjunction in `a`, `b` would be required by `c`.

```
1  Package: a
2  Version: 0.1
3  Architecture: i386
4  Depends: b | c
5
6  Package: b
7  Architecture: i386
8  Version: 1.0
9
10 Package: c
11 Version: 0.3
12 Architecture: all
13 Depends: b
```

Listing 2.3: An example package setup to explain strong dependencies

2.10 Repositories and Metadata Repositories

A *repository* is a set of binary packages and source packages including their metadata information and the actual binaries and source code they contain, respectively. Package managers have access to repositories to facilitate the retrieval and installation of new or upgraded packages on the system of the end user. Repositories are usually either accessed over HTTP or FTP or can be stored on a local filesystem.

The dependency analysis of this thesis is carried out on the metadata stored inside a repository. We call a repository that only consists of metadata like package names, versions and their relationships but without the binary content or source code a *metadata repository*. The metadata information for binary and source packages is stored in Packages and Sources files respectively. Those two files conform to the control file format explained in an earlier section. To start a bootstrap analysis with `botch` the only needed input are a pair of those Packages and Sources files for the desired distribution and target architecture.

Most repositories offered by binary distributions are self-contained. A self-contained repository is a set of binary packages B and a set of source packages S for which the following holds:

- All binary packages B (except `Architecture:all` binary packages) must be compilable from the source packages in S.
- All source packages in S must be compilable from the binary packages in B.

The exception for `Architecture:all` packages is made because in a bootstrap situation, those binary packages do not need to be compiled anymore. It is therefore not necessary to make source packages that build `Architecture:all` binary packages part of the problem.

`Essential:yes` binary packages are an implicit dependency of every binary package and the `build-essential` package is an implicit build dependency of all source packages. Therefore all

`Essential:yes` binary packages and the `build-essential` binary package plus all binary dependencies of both are always part of a self-contained repository. Same holds for the source packages that build those binary packages except for those binary packages that are `Architecture:all` for reasons explained earlier.

2.11 Availability

Throughout the bootstrap process we associate the boolean property of availability to packages. Availability means that the package including all its data files exists for a given architecture. This property is particularly interesting for binary packages as source packages are always available for all architectures. Additionally, all `Architecture:all` binary packages are always available for any architecture as they are architecture independent. To make an architecture dependent binary package available it must be compiled natively or cross compiled. It is the goal of the bootstrap process to make all architecture dependent binary packages of the target architecture available.

This property is not to be confused with the availability of only the metadata of a given binary package for an architecture. Meta data information and full metadata repositories can always be generated for all binary packages for all architectures. The set of available binary packages refers to an actual repository of binary packages with their binary and data content available in addition to their metadata.

Having full metadata information available in form of metadata repositories before the actual binary packages associated with it are available is also a necessary property to analyze the dependency relationships between binary and source packages for the target architecture. For example, calculating an installation set for a given binary or source package depends on this metadata information. In all cases where the function ComputeIS is called during this thesis, it is called with the knowledge of a full metadata repository for the target architecture. This means that ComputeIS ignores the availability of binary packages for its computations.

The goal of the bootstrap process is to make more and more binary packages available through cross or native compilation until all architecture dependent binary packages are available for the target architecture and all source packages can be compiled on the target architecture without running into dependency cycles. The bootstrapping process starts with no binary packages of the target architecture being available. Some source packages have to be cross compiled so that enough binary packages for a minimal build system become available. The rest of the binary packages is then made available through native compilation.

2.12 Installability

Installability means that a package can be installed and more precisely, that there exists an installation set of this binary package for which all its members are available for a given architecture. Installability is therefore stronger than availability and installability of a package also implies its availability.

Throughout this thesis we sometimes use the term installability for source packages. For a source package to be installable it means that all binary packages of one of its installation sets

are available. Installability for source packages is therefore equal to its compilability.

It must also be noted that whenever the terms “install” or “compile” are used throughout this thesis, no binary package is actually installed and no source package is actually compiled on a system. Instead, here those terms only mean that there exists an installation set for which all binary packages are available. This definition hides the fact that in practice, the compilation of a source package or the installation of a binary package can still fail even though their dependencies are satisfied. Though, should this happen, then it is a bug in the respective source or binary package which has to be fixed. By Debian policy, the dependency relationships must be enough to warrant successful compilability and installability of source or binary packages, respectively.

In a healthy distribution all binary packages are available and installable and all source packages are compilable. During bootstrapping only a subset of all binary packages are available and therefore even fewer are installable.

In algorithms we use the function $I = \text{FindInstallable}(A, P)$ to retrieve the set $I \subseteq P$ of packages that is installable with a given set of available binary packages A. Note that in contrast to the ComputeIS function, FindInstallable does not consider a full metadata repository but only considers the given available binary packages for testing if dependencies can be satisfied. If FindInstallable is applied to a set of source packages, then it will return the set of compilable source packages.

2.13 Build Dependency Cycles

During the bootstrapping of a distribution, only a small subset of all binary packages is initially made available through cross compilation. To make more binary packages available their respective source packages have to be compiled. But those source packages might build depend on binary packages that are not yet available themselves and whose source packages can't be built for the same reason. *Build dependency cycles* are created.

Build dependency cycles do not only exist during native compilation but also during cross compilation of the minimal build system. In that scenario, build dependency cycles are created because not all of the cross build dependencies of source packages can be satisfied by packages of the build architecture but require not-yet-built binary packages of the host architecture. But since in most source packages at least some cross build dependencies can be satisfied by packages of the build architecture, there exist much fewer build dependency cycles during cross compilation than during native compilation. It has been shown that in practice, a minimal build system can be cross compiled by only modifying a dozen source packages [51].

2.14 Build Profiles

Build dependency cycles are broken by building some source packages with fewer build dependencies. The build dependencies that were removed are said to be *dropped* and the source package is said to be built in *reduced* form. It is not only called reduced because of the fewer amount of its needed build dependencies but also because of the reduction in its activated feature set. To drop build dependencies, the source package must be modified to not build a certain feature, to not run its test cases or to not generate its documentation.

To satisfy the goal of the bootstrap to be deterministic, those changes are recorded in the metadata and build instructions of the source package. We call these changes to the build dependencies and build instructions of source packages a *build profile*. Building a source package with one or more profiles activated might not only result in a different set of required build dependencies but also in fewer binary packages being generated.

The changes made to a source package to implement a build profile must not affect it's compilation when no build profile is activated. Only when a source package is requested to be compiled with a certain build profile enabled, then it is to require less build dependencies and in turn provide less binary packages or functionality.

The syntax to indicate build profiles for source packages is not finalized yet. Therefore, the preliminary syntax that botch understands is likely to change in the future. Nevertheless, support for the preliminary syntax was added to `dose3`.

Chapter 3

Dependency Graphs

This chapter will explain the structure of the two types of dependency graphs used in this thesis. Generating dependency graphs is necessary to identify build dependency cycles, break them by only modifying a close to minimal amount of source packages and generate a build order from an acyclic graph. All tools support outputting graphs in the GraphML [12] format for consumption by third-party tools. In particular, the Python graph-tool module which is based on the Boost Graph Library was used to generate representations of dependency graphs as can be seen in Figure 1.1. Dependency graphs are always directed and contain no multi-edges. We differentiate between two different kinds of dependency graphs.

At first, we introduce the syntax we use for graph manipulation in the pseudo code. Then, the structure of the so called *build graph* will be described and afterward how it is generated. After that, the structure of the *source graph* will be explained and how it is generated from the build graph. The following sections will discuss the generation of the strong source graph as well as an optimal dependency graph. The last section will explain the rationale for providing two types of graphs.

3.1 Graph Manipulation Pseudo Code

Throughout this as well as other chapters we present pseudo code involving graph manipulation. This section is to give an overview of the imperative syntax that we use to illustrate our algorithms.

$G.add_edge(src, dst)$ adds an edge to the graph G from vertex src to vertex dst. If either of src or dst do not exist yet in G, they are created.

$G.remove_edges(E)$ removes the given set of edges E from G. The vertices they formerly connected are left in the graph.

$G.remove_vertex(v)$ remove vertex v from G. All edges that connected to or from v are removed as well.

$G.nontrivival_scc$ returns the set of nontrivial strongly connected components in G. Each element in this set represents a strongly connected component as a set of vertices the

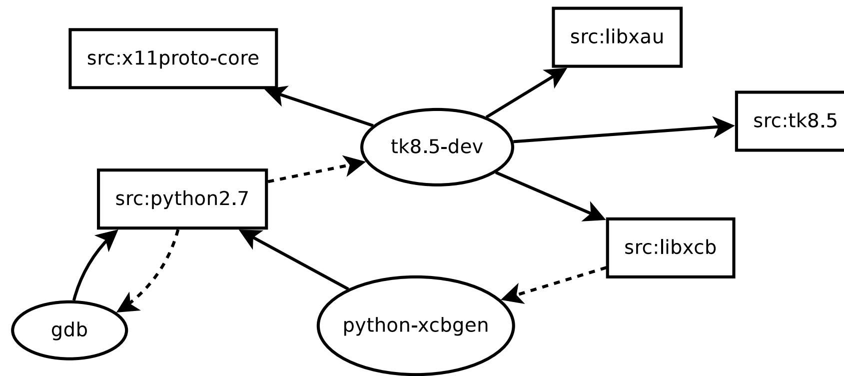


Figure 3.1: An example of a build graph. The displayed dependency relationships exist but many are left out for clarity.

respective component consists of.

$G.scc_with_vertex(v)$ retrieves the strongly connected component in G which contains the vertex v.

$G.has_cycle$ returns true if G is cyclic and false otherwise.

3.2 Structure of the Build Graph

The build graph contains two types of vertices and two types of edges. Vertices represent either source packages or installation sets of binary packages. They are called *source vertices* and *installation set vertices*, respectively. Edges can be either *build-depends* or *builds-from* edges. Build-depends edges can only connect source vertices to installation set vertices. Each build-depends edge represents one build dependency of the source package. Thus it connects a source package with the installation set vertex of one of its build dependencies. This installation set vertex holds an installation set of that build dependency. Builds-from edges connect installation set vertices to source vertices. Each builds-from edge represents the builds-from relationship of one or more binary packages within an installation set to the source package they build from.

Figure 3.1 shows an example for a build graph. Throughout this thesis, when displaying dependency graphs we will use the following conventions: source vertices are represented by rectangles, installation set vertices are represented by ellipses, build-depends edges are dashed arrows and builds-from edges are solid arrows. One can see from this build graph, that the source package `src:python2.7` build depends on the binary packages `tk8.5-dev` as well as on the binary package `gdb`.

It is important to remember that even though installation set vertices are labeled with a single binary package name in Figure 3.1, they still an installation set of multiple binary packages. The label merely represents the binary package for which the installation set was generated. This explains why the installation set vertex of `tk8.5-dev` in Figure 3.1 has multiple outgoing builds-from edges. Only one of those edges points to the source package that `tk8.5-dev` itself builds

from, namely `src:tk8.5`. The other builds-from edges point to the source packages from which the binary packages in the installation set of `tk8.5-dev` build.

We can also observe another property of the build graph that is implied by the definition above. Source vertices and installation set vertices can only have installation set vertices and source vertices as their successors, respectively. Because of that, a build graph cannot contain self-cycles and all cycles have an even number of edges with alternating build-depends and builds-from edges. In Figure 3.1, we can identify two cycles: one is the two-cycle between `src:python2.7` and `gdb` and one is the four-cycle between `src:python2.7`, `tk8.5-dev`, `src:libxcb` and `python-xcbgen`.

A build graph can only contain one source vertex for each source package as source vertices are unique in the same way source packages are unique: by their name and version. Since the same binary package can have different installation sets, there can be different installation set vertices belonging to the same binary package but associated with different installation sets of that binary package. Installation set vertices are unique by the binary package they are associated to and the installation set that was calculated for them.

3.3 Calculating and Partitioning Installation Sets

The reason why it must be possible for a build graph to contain two or more installation set vertices belonging to the same binary package but with different installation sets associated to them lies in the existence of the conflict relationship between binary packages. Picking installation sets for each build dependency of a source package individually might result in the union of those sets not being a valid installation set. Some pairs of binary packages from different individual installation sets might conflict with each other. It is therefore important to calculate the installation set for each build dependency of a source package in a way such that they do not conflict with the installation sets chosen for the other build dependencies of the same source package.

```
1  Package: b
2  Depends: c | d
3
4  Package: c
5  Conflicts: f
6
7  Package: d
8
9  Package: e
10 Depends: f
11
12 Package: f
```

```
1 Package: a
2 Build-Depends: b, e
3
4 Package: g
5 Build-Depends: b
6 Build-Conflicts: d
```

Listing 3.4: An example setup of binary packages (left) and source packages (right) to explain the reason behind dependency partitioning. Package versions and architectures are omitted for clarity.

Consider the dependency situation as shown in Listing 3.4. Lets suppose that installation sets for binary packages were chosen individually. Calculating an installation set for `b`, a solver might choose `c` from its disjunction. This installation set for `b` would not be applicable for the source package `src:a` because `src:a` also depends on `e` which depends on `f`. Since `c` was chosen for the installation set of `b`, `f` and `c` conflict with each other. To resolve the conflict, a different installation set must be chosen for `b`. It must be chosen such that it does not conflict with the installation set chosen for `e`. A valid choice would be to choose `d` instead of `c` from the disjunction of `b`.

Another aforementioned property was, that it must be possible to have installation set vertices in the graph that belong to the same binary package but contain different installation sets. Suppose the installation set of `b` contained `d` in accordance with what was required for `src:a`. The source package `src:g` also depends on `b` but at the same time it conflicts with `d`. Therefore, to satisfy the dependencies for `src:g`, the installation set of `b` must contain `c` and not `d`. So in the end there must be two installation set vertices associated to `b` in the dependency graph. One would contain the package `d` and `src:a` would connect to it. The other would contain the package `c` and `src:g` would connect to it.

The function PartitionDeps shown in algorithm 3.1 was developed by Pietro Abate and calculates installation sets for all build dependencies of a source package such that they are not in conflict with each other. It does so by intersecting an installation of the source package itself with the dependency closure of each build dependency. The result of this function is a set of tuples with the first entry being a direct build dependency of the source package and the second entry being its chosen installation set. This tuple can then later be used to create the appropriate installation set vertex.

Algorithm 3.1 Partitioning algorithm

1: **procedure** $\text{PartitionDeps}(S)$
2: $\quad IS \leftarrow \text{ComputeIS}(S)$
3: $\quad DEPS \leftarrow \{\text{Choose}(disj \cap IS) \mid disj \in \text{Deps}(S)\}$
4: $\quad$ **return** $\{(p, \text{DependencyClosure}(p) \cap IS) \mid p \in DEPS\}$

An installation set of S is computed in line 2. As stated earlier, this installation set is generated from a full metadata repository and might therefore contain binary packages which are not yet available.

Line 3 of the algorithm needs additional explanation. The function Deps returns the set of disjunctions of the source package. Each of those disjunctions is intersected with the installation set calculated for the source package itself. Since, for the dependencies of a package to be satisfied at least one element from each disjunction must be installed, the intersection of a disjunction with an installation set can never be empty. If the result of intersecting the disjunction with the installation set is a set with more than one member, then one arbitrary element is chosen by the function Choose. Making this choice arbitrary is a valid operation as only one element of each disjunction has to be considered and choosing these elements from those that are part of an installation set avoids conflicts.

In line 4, the intersection between the dependency closure of p and an installation set IS of the source package is calculated. The result of this operation must be a valid installation set

of p because p is known have a valid installation set in IS itself and at the same time can not require more packages in its installation set than in its dependency closure.

As an example we apply PartitionDeps on `src:a` as shown in 3.4. The installation set computed for `src:a` might consist of `b`, `d`, `e` and `f`. In this case it is indeed the only valid installation set for `src:a`. Applying the function Deps on `src:a` would return two sets, each with cardinality one. One set would contain `b` and the other would contain `e`. The intersection of each of these sets with the chosen installation set, would result in the same set of sets as already calculated by Deps and therefore, the function Choose just picks that one element of each of the two sets. The variable $Deps$ now has the value $\{b, e\}$ At last, the dependency closure for those two packages is calculated. The dependency closure of `b` contains `c` and `d`. This set is intersected with the installation set chosen for `src:a` and the result would be a set just containing `d`. The dependency closure of `e` contains `f` and intersecting it with the installation set of `src:a` keeps it. We can observe how the installation set chosen for `b` (just containing `d`) and the installation set chosen for `e` (only `f`) are the valid installation sets we already obtained in the example above.

3.4 Generating the Build Graph

Algorithm 3.2 Generating a build graph

```
 1: procedure BuildGraph(S, A)
 2:     for all s ∈ S do
 3:         P ← PartitonDeps(S)                        ▷ see Algorithm 3.1
 4:         for all (p, is) ∈ P do
 5:             if is ⊈ A then
 6:                 G.add_edge(s, (p, is))             ▷ add a build-depends edge
 7:                 for all b ∈ is do
 8:                     if b ∉ A then
 9:                         src ← Source(b)
10:                         G.add_edge((p, is), src)   ▷ add a builds-from edge
```

The dependency graph can be created by traversing all source packages iteratively as can be seen in 3.2. The function BuildGraph receives the set of all source packages and the set of already available binary packages. It then iterates through all source packages, creating source vertices for them, connecting those to installation set vertices and them again to other source vertices. An alternative implementation would start from a subset of all source packages and then recurse on the source packages to which a connection was made during the build graph creation. This recursive implementation was implemented as well with the ability to choose the maximum depth.

For each source package, the function partition calculates installation sets for each of its build dependencies (line 3). The set of tuples returned by the function partition is traversed (line 4). For each build dependency for which the calculated installation set is not a subset of the available packages (line 5), a new build-depends edge is added (line 6). If an installation set is a

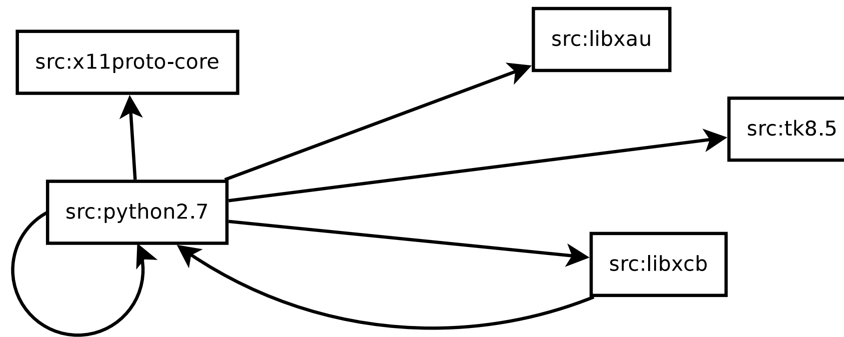


Figure 3.2: An example of a source graph. The displayed dependency relationships exist but many are left out for clarity.

subset of the available package, then it can already be installed. In that case, no binary package of that installation set needs to be compiled anymore. It is therefore not necessary to add that installation set to the build graph. This also means that throughout the bootstrap process, as more and more binary packages are made available, the calculated build graph will become smaller and smaller until it only consists of source vertices without connections between each other. In that state, the bootstrap process is finished and all source packages can be compiled in any order.

All binary packages in the remaining installation sets are traversed as well (line 7) and for each binary package which is not an element of the available packages, a builds-from edge is added to the build graph (line 10). No connection is made for binary packages that are already available because them being available implies that the source packages they would build from does not have to be compiled. This can be because the source package was natively compiled earlier, the source package was cross compiled or the binary package is `Architecture:all`.

3.5 Structure of the Source Graph

When calculating a build order from an acyclic build graph, one is only interested in an order of the source vertices and not the installation set vertices. Furthermore, it is not possible to calculate a strong subgraph from edge annotations in the build graph. We therefore introduce a second type of graph that we call a *source graph*.

In contrast to a build graph, a source graph only contains source vertices. It has only one edge type between source vertices and can contain self-edges. A source vertex representing a source package `src:a` is connected to another source vertex representing a source package `src:b` if one or more of the binary packages in the installation set chosen for `src:a` builds from `src:b`.

Furthermore, a source graph can easily be created from a build graph while preserving the relationships between source packages in the build graph. This is essential for generating a partial order consistent with the original build graph.

Figure 3.2 shows an example for a source graph that was generated from the build graph shown in Figure 3.1. As one can see, if a source vertex was connected to another source vertex via an installation set vertex in the build graph, then those source packages are now di-

rectly connected in the source graph. Additionally, the two-cycle in the build graph was turned into a self-cycle of `src:python` with itself and the four-cycle turned into a two-cycle between `src:python` and `src:libxcb`.

3.6 Generating the Source Graph

A source graph can be generated in two different ways. It can either be generated from scratch or a build graph can be turned into a source graph. Algorithm 3.3 does this conversion using all vertices V of the original graph as input.

Algorithm 3.3 Converting a build graph into a source graph through path contraction.

```
1: for all v ∈ V do
2:     if isInstSet(v) then
3:         for all src1 ∈ Pred(v) do
4:             for all src2 ∈ Succ(v) do
5:                 G.add_edge(src1, src2)
6:         G.remove_vertex(v)
```

The algorithm is a special case of path contraction. It contracts all paths between two source package vertices which are connected by a single installation set vertex. For this purpose, the algorithm iterates over all vertices of the build graph (line 1) and if they are an installation set vertex (line 2) it adds an edge between every pair of its predecessors and successors (line 5).

Alternatively, a source graph can be generated from scratch in a similar but simpler way than the build graph was generated. The algorithm can be seen in Algorithm 3.4.

Algorithm 3.4 Generating a source graph

```
1: procedure SrcGraph(S, A)
2:     for all src1 ∈ S do
3:         IS ← ComputeIS(src1)
4:         for all p ∈ IS \ A do
5:             src2 ← Source(p)
6:             G.add_edge(src1, src2)
```

Since no installation set vertices have to be added, the partitioning of the installation set of the source package is not necessary. Instead, the algorithm simply connects a source vertex to all other source vertices which build the binary packages in the installation set that are not available. If it could be guaranteed that for each source package the same installation set was generated, then the output of both algorithms presented in this section would be the same.

3.7 Strong Source Graph

The build graph and source graph are not unique. Depending on the choice of installation set for each source package, different graphs can be generated. The choice of installation sets is

not crucial if the source vertex is not part of a dependency cycle. But if it is, then choosing a different installation set might break this dependency cycle. Therefore, the information about which dependencies are strong is especially useful when analyzing small cycles.

To determine which source packages of a nontrivial strongly connected component are optional and which are not, one can calculate the strong subgraphs of a source graph. This subgraph is made of all source vertices connected by edges that are marked as strong. Edges are marked as strong if the binary package they represent is a strong dependency of the respective source package. The result of this operation is called a *strong source graph* and it gives a lower bound on the number of source packages that have to be involved in a strongly connected component.

3.8 Optimal Dependency Graph

Since the generated dependency graph depends on the choice of installation sets, `botch` allows the user to influence this choice and thereby allow a fine-grained control over the generation of the dependency graph. It does so by allowing the user to specify not-strong build dependencies that are to be excluded from the generated installation set on a per-source-package basis. It provides the user with the facility to experiment with different choices of installation sets for example in case a cycle with a not-strong edge exists. We did not find a satisfactory solution to automate any such customizations.

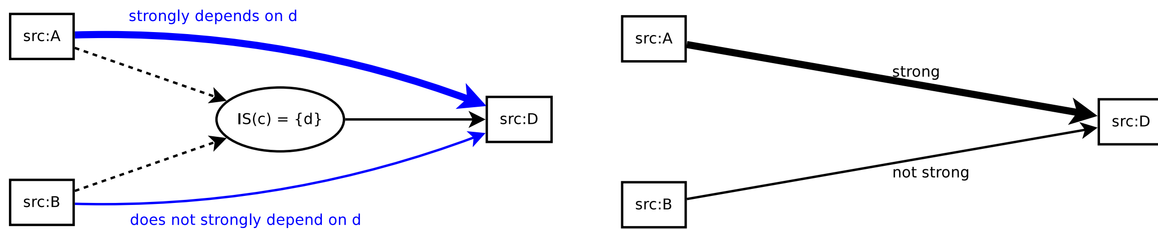
For example, using constraint solvers, a dependency graph could be generated which conforms to a property like minimum number of source packages in strongly connected components. To test such a possibility `aspcud` [21] was used and instructed to find a self-contained repository of minimum cardinality. The result was only 1.7% smaller than the non-optimized repository calculated using the `dose3` solver.

Furthermore, the actual goal for which should be optimized is "easy bootstrappability" which cannot be quantified or calculated. A smaller dependency graph might as well be harder to solve than a bigger one due to the different selection of packages. An algorithm cannot decide which build dependencies are "easier" to break. This topic will be covered in more detail in chapter 8. For these reasons, the computation of an optimal solution with respect to dependency graph generation was not further pursued. Instead, as stated initially, the user is able to manually customize individually generated installation sets if he sees it fit.

3.9 Rationale Behind Choice of Graph Types

Both graph types are necessary because of the certain operations that can be done on one but not on the other. A build graph is the natural choice over the source graph when looking for build dependencies to remove from source packages. The source graph hides the information about which build dependency leads to a connection to which source package. In a build graph, on the other hand, the removal of one build-depends edge will automatically remove all connections to the source packages associated by that installation set vertex.

A source graph is not only the natural choice for deriving a build order but is also the only way to generate a strong subgraph. Suppose we would want to calculate the strong subgraph



(a) The curved, blue arrows indicate a strong (or not-strong) dependency on a binary package built by the source package at the destination of the arrow. They do not appear in the build graph.

(b) The bold upper arrow indicates a strong dependency and the thin lower arrow indicates a not-strong dependency on a binary package built by the source package at the destination of the arrows.

Figure 3.3: Example explaining why there are no strong build graphs. The source graph on the right corresponds to the build graph on the left.

of a build graph. We would then have to annotate build-depends and builds-from edges as being strong or not. An algorithm would then follow the edges marked as strong to extract the strong subgraph in the same manner as it is done with the source graph. The problem is, that builds-from edges cannot be marked as strong in a meaningful manner.

Suppose the dependency situation as depicted in Figure 3.3. Figure 3.3a shows a build graph and 3.3b its corresponding source graph. While the builds-from edge from the installation set vertex of `c` to the source vertex of `src:D` could be marked as strong or not strong depending on whether `d` is a strong dependency of `c`, this would not be a useful information. It would not because it would not reflect the actual strong or not-strong dependency between `src:A` or `src:B` and the binary package `d` built by the source package `src:D`. The blue arrows indicate that `src:A` strongly depends on `d` but `src:B` does not.

Furthermore, suppose that the dependency of `src:A` on `c` was not strong. Then `src:A` could still strongly depend on `d` but as the build-depends edge from `src:A` to `c` would not be marked as strong, the strong subgraph would be calculated incorrectly. Therefore, the only way to reliably calculate a strong subgraph from a build graph would be if the following two conditions were met. Firstly, build-depends edges would not be marked as strong or not strong and during the generation of a strong subgraph they would therefore always be followed. Secondly, builds-from edges would carry the information whether the source source package they point to builds a strong dependency of any source package that is the predecessor of the installation set vertex they come from.

Instead, we use a source graph to annotate strong dependencies. As one can see in Figure 3.3b, the blue arrows from the build graph in Figure 3.3a can directly be represented by the edges in a source graph. There is therefore no need to introduce more complexity.

Chapter 4

Toolset

The needed input for build or source graph creation is three-fold: a metadata repository for ComputeIS, a list of source packages to compile and a list of available binary packages. Furthermore, when analyzing the cross compilation phase, it would be beneficial to only work on the relevant subset of the full repository to save time. This argument also holds during native compilation where the process of dependency graph analysis can be speeded up by first concentrating on building a meaningful subset of the full repository. A meaningful subset would be a smaller self-contained repository. We found out that all these tasks can be accomplished by the same set of tools but combined together in different ways. In this chapter we describe those tools individually.

All tools are designed by the UNIX philosophy. Each tool executes exactly one algorithm, the exchange format between the tools is ASCII plain text and every tool is a filter. Different tasks can be accomplished by building different pipelines out of these tools.

The plain text exchange format are the aforementioned RFC822 based control files which contain package metadata. Apart from commandline switches, these files are the only input to all tools. The result of their calculations is also again in the same control file format. Specifically, the metadata repository is given as a pair of Packages and Sources files. The set of source packages to compile as well as the set of available binary packages are also given in the control file format.

Not all tools in this section are part of `botch` itself. Where possible we use existing tools that the user might already be familiar with.

4.1 Distcheck

`Distcheck` is part of `dose3` and a Debian quality assurance tool to check installability of binary packages by testing if their dependencies can be satisfied. It is able to output the calculated installation set as well as an error report in case the dependencies of a binary package can not be satisfied. It is available for installation as part of the Debian binary package `dose-distcheck` and the executable is named `dose-debcheck` because `distcheck` also allows to analyze dependencies between rpm and Eclipse packages. Here, we are only interested in checking Debian metadata.

4.2 Buildcheck

`Buildcheck` works similar to `distcheck` except that it checks buildability of source packages instead of installability of binary packages. Just as `distcheck` it can output the calculated installation set and reports errors with different verbosity levels. For this work, `buildcheck` was extended to also allow to check multiarch cross compilation besides native compilation. It can be installed through the `dose-builddebcheck` Debian package and contains an executable of the same name.

4.3 Coinstall

The `Coinstall` tool is part of `dose3` as well and calculates a co-installation set of a set of binary packages. It throws an error if no co-installation set exists. The installation set is output in the control file format to be consumed by other tools.

4.4 Package Filter

The tool `grep-dctrl` is part of the `dctrl-tools` package in Debian. `Grep-dctrl` can perform absolute filtering (but no relative filtering) on all package properties. More specifically it allows to filter package properties by using regular expressions or by inequality comparisons of version numbers. Different filters can be combined using logical expressions. Output and input to `grep-dctrl` are files in the Debian control file format.

```
1 $ grep-dctrl -X \( -FEssential "yes" --or \
2 >        -FPackage "build-essential" \) < "./packages-amd64" > "./minimal"
3 $ deb-coinstall --bg="./packages-amd64" --fg="./minimal" \
4 >        --deb-native-arch=amd64 > "./minimal-amd64"
```

Listing 4.5: Selecting packages for the minimal build system

We use `grep-dctrl` to allow to select the packages that should be present in the minimal build system. The minimal build system should at least contain all `Essential:yes` packages and the `build-essential` package. Therefore the `grep-dctrl` in Listing 4.5 filters the input binary package list `packages-amd64` by the value of packages' `Essential` and `Package` fields. Since the resulting list misses the binary dependencies of the selected packages, a co-installation set is calculated by a call to the `deb-coinstall` tool. Listing 4.5 demonstrates how the tools are connected together as a pipeline. The input to `grep-dctrl` is the list of all binary packages of architecture `amd64` and the output of `grep-dctrl` is the file `minimal` which serves as an input to the next step in the pipeline when `deb-coinstall` is called.

4.5 Package Union, Intersection, Difference

Using the python module `apt_pkg`, simple scripts were written which perform the set operations union, intersection and difference on Debian control files. Conforming to what was explained in earlier chapters, they treat packages as unique if their name, version and architecture match. Therefore, even if there were duplicate packages in the input, the output is always unique. The output is ordered first by package name, then by version, then by architecture.

4.6 Binary to Source and Source to Binary

The tool `bin2src` turns a list of binary packages into a list of their corresponding source packages from which they build according to the builds-form relationship. Conversely, the tool `src2bin` turns a list of source packages into a list of binary packages that those source packages build. Those two tools are usable implementations of the functions SOURCE and BINARY in `bin2src` and `src2bin`, respectively.

The tool `bin2src` per default does not return the source packages for binary packages that are `Architecture:all` because in a bootstrapping situation, those source packages are not of interest. For example, if the tool `bin2src` is applied to the result of `deb-coinstall` as shown in Listing 4.5, then the result is the set of source packages that needs to be cross compiled to produce that list of binary packages.

4.7 Calculate Fixpoint

This tool calculates all those source packages which can be compiled on a minimal build system or any other selection of available packages without having to break build dependency cycles. It allows to make the maximum number of binary packages available before the dependency situation has to be analyzed by calculating dependency graphs. This is useful because the dependency graph is the smaller the more binary packages are available. The algorithm was first introduced in a paper co-authored by the author of this thesis [4].

The tool is given a set of source and a set of binary packages and will try to compile all source packages with only the limited amount of binary packages available to satisfy build dependencies. Should there be any source packages that can be compiled, it will retrieve the binary packages they produce and add them to the next iteration. Should no more source packages be able to be compiled, the algorithm stops.

A recursive implementation of this algorithm can be found in Algorithm 4.1. It takes as input the set of available binary packages A and the set of all source packages S. It finds the subset of all those source packages that can already be compiled in A (line 2). Should this set be empty, it returns a tuple of the now newly available binary packages and compilable source packages. Otherwise it adds the binary packages generated by the compilable source packages to the set of available binary packages (line 7). It updates the set of compilable source packages and the set of source packages to compile and re-runs the algorithm.

The fix point algorithm also computes the first steps of a build order. Therefore, in line 6, the set of source packages that was found to be compilable at that point can be output. A more

Algorithm 4.1 Compilation Fix Point

```
1: procedure F(C_i, A_i, S_i)
2:     NS ← FindInstallable(A_i, S_i)          ▷ Sources compilable in A_i
3:     if NS = ∅ then
4:         return (A_i, C_i)                   ▷ available binaries, compiled sources
5:     else
6:         output(NS)                          ▷ print the source packages compilable in this step
7:         B_{i+1} ← Binary(NS) ∪ A_i          ▷ A_i plus all binaries from NS
8:         C_{i+1} ← C_i ∪ NS                  ▷ Overall set of compilable sources
9:         S_{i+1} ← S_i \ NS                  ▷ Sources left to compile
10:        return F(C_{i+1}, B_{i+1}, S_{i+1})
11: Fixpoint ← F(∅, A, S)
```

trivial implementation of the fix point algorithm would work by using `buildcheck`, `src2bin` and the python tools for calculating union and difference and iteratively call them on all source packages to find out more source packages that already have their build dependencies satisfied. The advantage of having this algorithm in one tool is, that otherwise package lists would have to be parsed and written out multiple times which costs time.

4.8 Calculate Build Closure

The algorithm executed by this tool allows to find a set of source packages and a set of binary packages that together make a self-contained repository, starting from an initial set of source packages. This tool therefore calculates the contents of self-contained metadata repositories which are a subset of the repository for the full distribution. A repository of a full distribution contains metadata for tens of thousands of binary and source packages. By cutting this amount to only a few hundred, the execution time of algorithms generating and analyzing the dependency graphs is cut by an order of magnitude. Just as the fixpoint algorithm, the build closure algorithm was already presented in [4].

The result of calculating such a subset is still meaningful. For example when calculating the amount of packages to cross compile, it is not needed to calculate installation sets to cross compile tens of thousands of packages and investigate a dependency graph of thousands of vertices. Instead it is enough to focus on the few hundred packages that are actually needed for a minimal build system. The same holds during native compilation. The self-contained repository that is created from the minimal build system includes the biggest strongly connected component of the distribution. One can therefore focus on solving this component without the clutter of another twenty thousand packages that are not part of the problem.

The tool is given an initial set of source packages. It computes the union of installation sets for all those source packages and then retrieves the source packages that build those binary packages. This step is repeated until no new source packages are needed to compile the required binary packages.

A recursive implementation of this algorithm can be found in Figure 4.2. It gets as input

Algorithm 4.2 Build Closure

```
1: procedure F(B_i, C_i, S)
2:     NB ← ⋃_{s∈S} ComputeIS(s)
3:     if NB = ∅ then
4:         return (B_i, C_i)                  ▷ available binaries, compiled sources
5:     else
6:         B_{i+1} ← B_i ∪ NB                 ▷ B_i plus additionally needed binary
7:         C_{i+1} ← C_i ∪ S                  ▷ Overall set of compiled sources
8:         NS ← Source(NB) \ C_{i+1}          ▷ Sources for binaries minus compiled
9:         return F(B_{i+1}, C_{i+1}, NS)
10: Closure ← F(∅, ∅, S)
```

the set of initial source packages and calculates the union of their installation sets (line 2). If the resulting set of binary packages is empty, then the algorithm finishes as no more source packages have to be compiled. Otherwise, it retrieves the source packages needed to compile those binary packages minus those source packages that already have been compiled. The list of needed binary packages and the list of already compilable sources are updated as well. The result is used as the input for the next iteration. The calculated self-contained repository is not unique and depends on the chosen installation sets.

Chapter 5

Processing Pipeline

This chapter will connect all the tools of the previous chapter together to produce meaningful output. The diagrams in this chapter are to be read as follows:

- **Solid arrows** represent a flow of binary packages.
- **Dotted arrows** represent a flow of source packages.
- **Dashed arrows** represent textual user input.
- **Rectangular boxes** represent filters. There is only one input to the filter, which is the arrow connected to the top of the box. Outgoing arrows from the bottom represent the filtered input. Ingoing arrows to either side are arguments to the filter and control how the filter behaves depending on the algorithm.
- **Ovals** represent a set of packages.
- **Boxes with rounded corners** represent set operations like union, intersection and difference between two or more input package lists.

5.1 Preferring Native Compilation over Cross Compilation

The bootstrap process is divided into two phases: the cross and the native phase. During the cross phase an initial set of source packages is cross compiled to produce enough binary packages for a minimal build system. The minimal build system must at least contain binary packages that are marked as `Essential:yes` as well as the package `build-essential`. This is because both are implicit dependencies for installing any binary package and compiling any source package, respectively. Using this minimal build system, native compilation is started to compile all of the rest.

Cross compilation should be kept to the bare minimum and native compilation should be preferred. The reasons are:

- Upstream projects do often not support cross compilation and are neither interested in accepting patches implementing it

- Cross compilation adds much complexity to the build process and patches must be maintained and updated as the upstream projects updates
- `autotools` and `cmake` support cross compilation but especially big projects use their own build systems without any cross compilation support
- During native compilation, the compiled binaries are run to be tested for errors; this cannot be done during cross compilation and therefore introduces errors
- Cross compilation will only be done during bootstrapping and will therefore be little tested and prone to stop working after package upgrades
- Some generators of binary data cannot yet handle cross compilation. An example is the generation of `*.gir` files needed for GObject Introspection. Other examples are programming languages which produce platform specific bytecode but do not support cross compilation.

On the other hand one reason to introduce better support for cross compilation and to cross compile more packages is, that there exist fewer build dependency cycles during cross compilation than during native compilation. The amount of dependency cycles is less because build dependencies on multiarch binary packages can be satisfied by existing binary packages of the build architecture instead of having to rely in not-yet-existing binary packages of the host architecture (see Table 2.1). But as stated above, implementing cross compilation support is often hard and experience shows that it is in most cases harder to make a source package cross compile than it is to make it compile with dropped build dependencies. But since `botch` allows the developer to make any arbitrary selection for what source packages he sees fit to be cross compiled, `botch` does not make restrictions for either case. Therefore, if cross compilation becomes more common in the feature, then the only input that has to be adapted is the amount of cross compiled packages as selected by the user via `grep-dctrl`.

5.2 Self-Contained Metadata Repository

A self-contained metadata repository can be created for the cross as well as for the native case. In neither case, creating a self-contained repository is strictly required and the full repository of the distribution can be used just as well. But in both cases it will speed up any tool working on the data and will still allow a self containing system to be bootstrapped.

Figure 5.1 shows how the tools can be connected together to create a self-contained metadata repository. The user first chooses the binary packages for the minimal build system using `grep-dctrl`. As stated earlier, `Essential:yes` packages and the package `build-essential` are required.

The developer can add any additional packages to the list that he wants to have available in the minimal build system. It would for example make sense to also have the package `debhelper` available since 79% of source packages build depend on it. It would therefore be hard to bootstrap it without introducing many build profiles that allow to not build depend on debhelper. Such a change would also be very intrusive with respect to the build scripts. So instead, some more

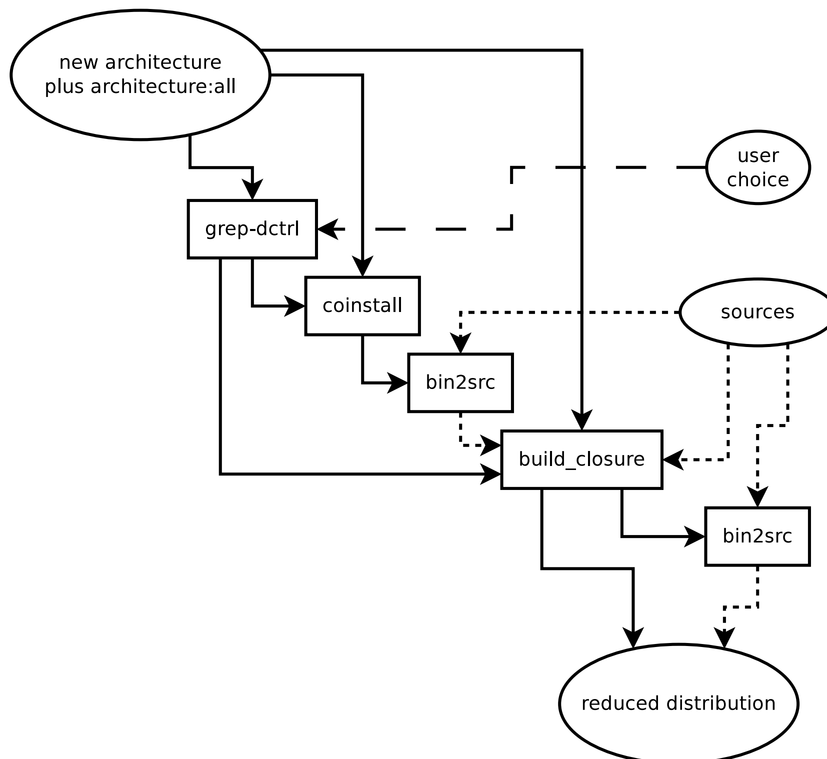


Figure 5.1: Processing pipeline for creating a self-contained repository

source packages are picked to be cross compiled as an easier and cleaner option. Examples like this show that it is necessary to allow the user full control over which binaries are chosen for cross compilation as the decision whether or not cross compilation or a build profile is preferred is a decision only a human can make. This topic will be discussed in more detail in chapter 8.

To guarantee that the binaries chosen in the first step can be installed together, the tool `coinstall` is used to calculate a co-installation set for the selection of binary packages. It will throw an error if no co-installation set exists. These last two steps equal the example invocation seen in Listing 4.5.

The tool `bin2src` is executed on the result of the `coinstall` tool. It associates all binary packages that are not `Architecture:all` with their respective source packages. Those source packages are then used as the initial input to `build_closure`. This tool will then execute its algorithm to calculate the set of binary packages which are part of a self-contained repository containing the selected minimal build system.

To retrieve the set of corresponding source packages, `bin2src` is executed again on those binary packages. The resulting selection of binary packages and source packages can be checked whether they form a valid self-contained repository using the `distcheck` and `buildcheck` tools. They will verify if all binary packages can be installed and if all source packages can be compiled, respectively. The requirement that in a self-contained distributions all binary packages must have been produced by its source packages is already implied by the list of source packages being generated by the `bin2src` tool.

5.3 Cross Phase

The cross phase as depicted in Figure 5.2 reuses the pipeline to produce a self-contained repository which was introduced in the last section but it will execute the `build_closure` tool for cross compilation instead of native compilation. The initial selection by `grep-dctrl` should be as small as possible because native compilation is preferred over cross compilation.

The set binary packages calculated by `build_closure` are of the target architecture and finding the source packages that build them using `bin2src` yields the source packages that have to be cross compiled. This selection forms a self-contained repository with respect to cross compilation instead of native compilation.

The `build_fixpoint` tool is used to check whether some of those source packages can already be cross compiled without having to break build dependencies. The output of `build_fixpoint` is the list of source packages that are cross compilable without the need of any changes to source packages. The difference between the input of `build_fixpoint` and its output is the final list of source packages that must be cross compiled to have the initially selected binary packages available in the new system.

The list of cross compilable source packages calculated by `build_fixpoint` is also converted to their corresponding binary packages by `src2bin`. Its output is the list of binary packages that are already available through cross compilation. That output together with the binary packages of the old architecture make the overall list of available binary packages.

The output of this pipeline, namely the set of available binary packages and the set of source packages to compile can then be used as the input to the algorithm creating a build graph for cross build analysis. The calculation can be speeded up by using the self-contained repository with respect to cross compilation that was calculated by the `build_closure` tool. Experiments in practice show, that only a dozen source packages have to be modified to break enough dependency cycles so that a minimal build system can be cross compiled. In theory, we cannot analyze this case yet as missing multiarch information in binary packages prevents cross build dependency resolution for a number of source packages to succeed. Chapter 10 will go into more detail about this topic.

5.4 Native Phase

The native phase as depicted in Figure 5.3 is similar to the cross phase. There are two major differences:

- The `build_closure` tool is executed for native compilation and not for cross compilation.
- The available packages are those which have been cross compiled during the cross phase. They are therefore part of the input for `build_fixpoint` which is executed for native compilation as well.

The reason for `build_fixpoint` still receiving packages from an existing architecture is because multiarch might help resolving some build dependencies. If a platform is able to work with binaries from multiple architectures, then packages which are marked as `Multi-Arch:foreign`

can be used to satisfy dependencies of binary packages of the new architectures. This makes `Multi-Arch:foreign` binary packages immediately available for dependency resolution on the new platform.

The reason why `build_closure` is not assisted by binary packages of an existing architecture is, because this part of the pipeline is exactly the calculation of a self-contained repository. As noted earlier, calculating a smaller, self-contained repository is not a strict requirement, so the whole algorithm can also be executed without these steps. On the other hand, the execution is orders of magnitude faster when calculating a smaller self-contained repository first. The processing pipeline without calculation of a self-contained repository first is shown in Figure 5.4. In the source code of `botch`, both native pipelines (with and without calculating a self-contained repository) are realized by executing the associated commands one after another in the form of two shell scripts.

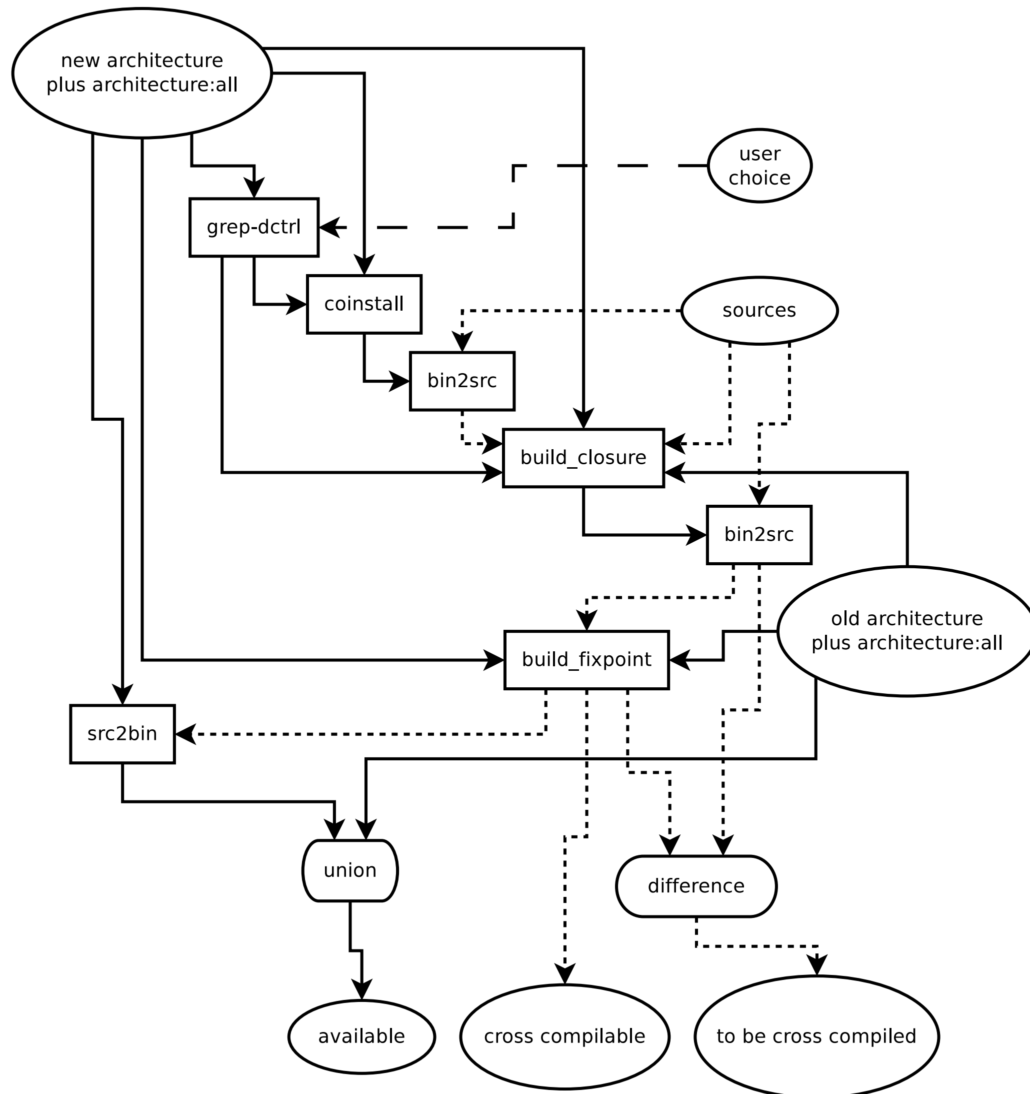


Figure 5.2: Processing pipeline for cross compilation

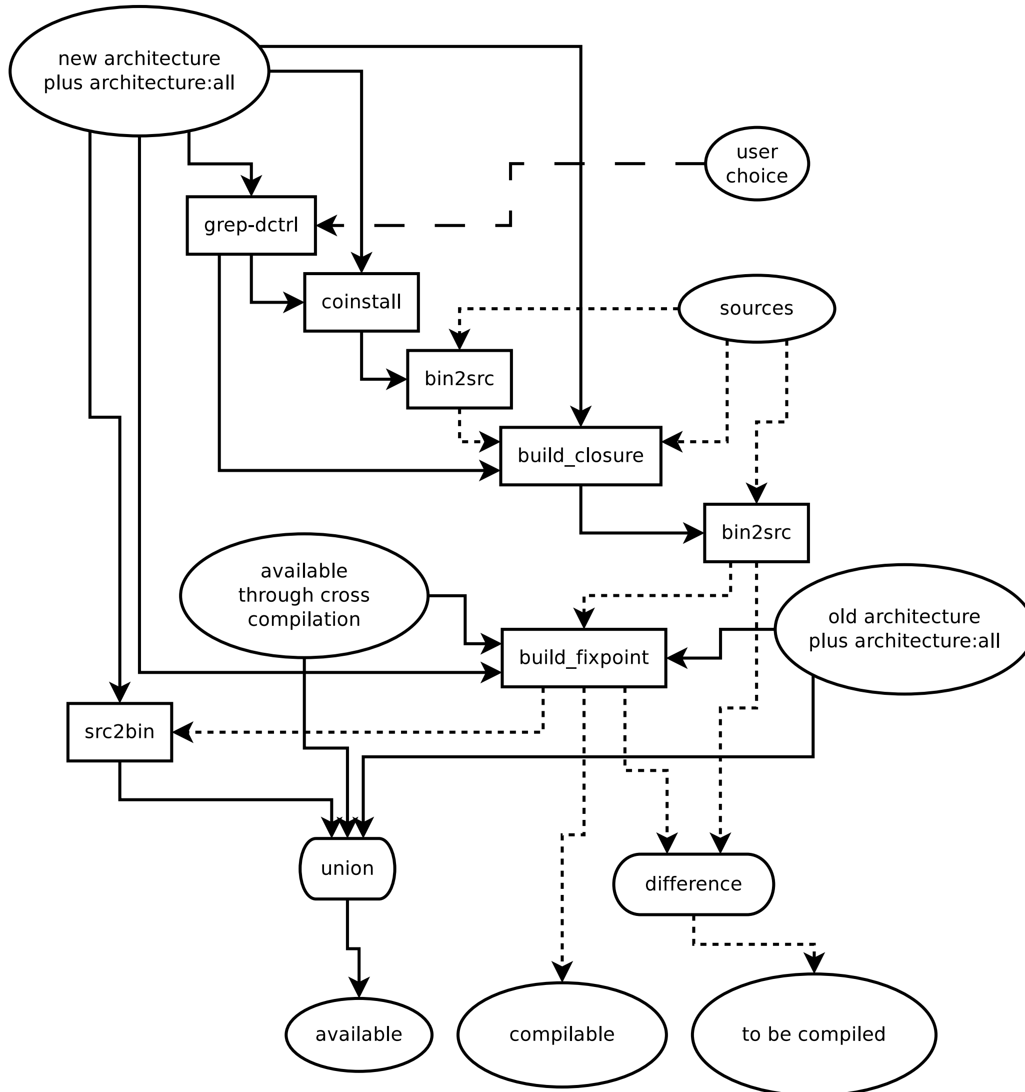


Figure 5.3: Processing pipeline for native compilation using a self-contained repository

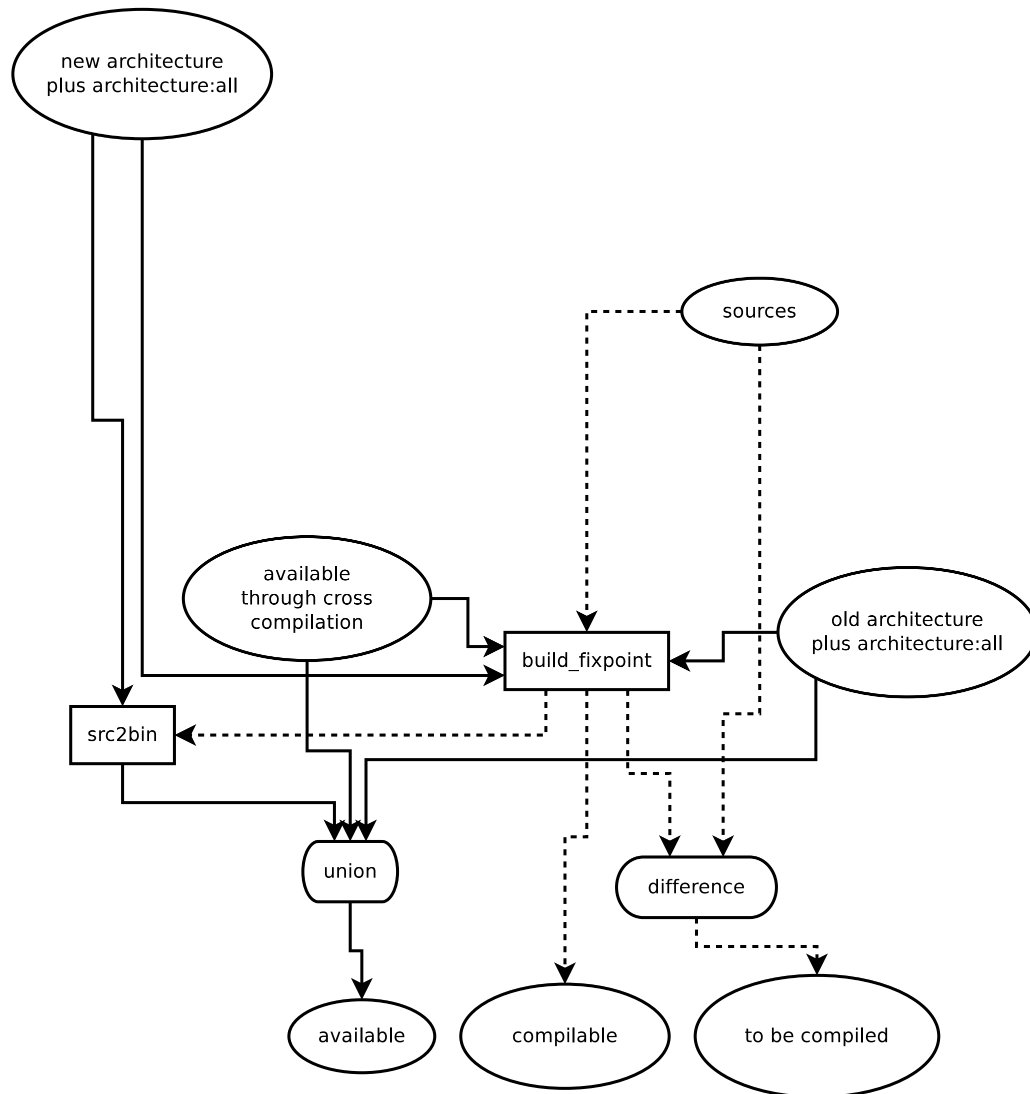


Figure 5.4: Processing pipeline for native compilation without a self-contained repository

Chapter 6

Enumerating all Cycles of a Directed Graph

This chapter covers the algorithm we use for finding cycles in a directed graph. This algorithm will later be used for our new heuristic for the Feedback Arc Set Problem. It will also allow the developer to find small cycles or edges with lots of cycles through them in the dependency graph.

Let n be the number of vertices, e be the number of edges and c be the number of cycles in the graph. According to the authors of [39], the algorithm by Donald B. Johnson [28] is complexity wise the asymptotically fastest existing algorithm to enumerate cycles with an upper bound of $O((n+e)c)$.

This work uses a slightly modified version of Johnson's algorithm which allows to limit the result by the maximum length of the returned cycles and by the maximum amount of cycles. There existed two implementations of Johnson's algorithm in OCaml by Pietro Abate. One used an imperative and the other a recursive algorithm but both suffered from bugs which were fixed in the context of this project. The imperative implementation is used per default by `botch` as it is slightly faster than the recursive one. It was also submitted for inclusion into the `ocamlgraph` library.

Cycle enumeration algorithms are not implemented in popular graph libraries like `ocamlgraph` or the Boost Graph Library, so a testbed was written to ensure the correctness of the developed algorithm. This chapter first covers Johnson's algorithm and then its evaluation in comparison with other implementations.

6.1 Implementation

Pseudo code on how Johnson's algorithm was implemented can be seen in algorithm 6.1. The original paper used a Pascal-like pseudo code. The more modern pseudo code presented here should be much more readable.

The algorithm proceeds in a depth first search and backtracking fashion just as the original. The function Circuit recursively searches the unblocked successors of vertices for a new cycle and blocks traversed vertices to avoid double counting and double visiting. The function Un-

Algorithm 6.1 Enumerating all cycles by D.B. Johnson

```
1: procedure JOHNSON(G, maxlength, maxamount)
2:     path ← empty list of vertices
3:     blocked ← empty map from vertex to boolean
4:     b ← empty map from vertex to vertex set
5:     result ← empty list of vertex lists
6:     procedure UNBLOCK(n)
7:         if blocked[n] then
8:             blocked[n] ← false
9:             for i ∈ b[n] do UNBLOCK(i)
10:            b[n] ← emptylist
11:    procedure CIRCUIT(thisvert, startvert, comp)
12:        closed ← false
13:        if (path.length ≤ maxlength) ∧ (result.length ≤ maxamount) then
14:            path.push(thisvert)
15:            blocked[thisvert] ← true
16:            for nextvert ∈ thisvert.successors do
17:                if nextvert = startvert then
18:                    result ← result + path
19:                    closed ← true
20:                else if not(blocked[nextvert]) ∧ CIRCUIT(nextvert, startvert, comp) then
21:                    closed ← true
22:            if not(closed) then
23:                UNBLOCK(thisvert)
24:            else
25:                for nextvert ∈ thisvert.successors do
26:                    b[nextvert].add(thisvert)
27:            path.pop()
28:        return closed
29:    for scc ∈ G.nontrivival_scc do
30:        for vertex ∈ scc do
31:            if result.length ≤ maxamount then
32:                component ← G.scc_with_vertex(vertex)
33:                for v ∈ component do
34:                    blocked[v] ← false
35:                    b[v] ← empty vertex set
36:                CIRCUIT(vertex, vertex, component)
37:                G.remove_vertex(vertex)
```

BLOCK recursively unblocks vertices in case the current execution of CIRCUIT didnt lead to a new cycle and the algorithm has to backtrack. The variable *blocked* tracks the vertices which are part of the currently evaluated path. It avoids to search vertices which are part of the current path already. The variable b maps vertices to sets of vertices. It stores for every vertex the set of vertices which are its predecessor and have already been found to be part of at least one cycle. This information is crucial so that when the function CIRCUIT did not find a cycle when exploring a vertex and has to backtrack, all relevant vertices are unblocked.

The only functional change that was made to the algorithm was the addition of lines 13 and 31. They limit the amount of cycles the algorithm outputs as well as the maximum cycle length. The maximum cycle length is enforced by limiting the maximum path length and thereby reducing the search area around each start vertex.

6.2 Evaluation

Since there existed no well tested implementation of cycle enumerating algorithms, a test suite was written which ran seven different cycle enumerating implementations on 380 randomly generated input graphs and compared their output. Since the algorithms are written by different authors in different programming languages and use different enumeration algorithms, their correctness is likely if they all agree on the same solution for a given graph. The test suite is not part of the `botch` project but is released under the GPL on `https://github.com/josch/cycle_test`. It is currently also used to test a submission of a C++ based cycle enumeration function for inclusion into the Boost Graph Library.

The first two implementations are based on the code of Pietro Abate and are the functional and imperative implementations of Johnson's algorithm in OCaml. The third implementation is an implementation of the cycle enumeration algorithm by Tarjan [47] in Python. Tarjan's algorithm is more simple but has a higher computational complexity with $O(n \cdot e \cdot c)$. The fourth test code is based on an implementation of Johnson's algorithm in Java by Frank Meyer. The fifth implementation is based on an extension of Johnson's algorithm by Hawick and James [25]. They propose a new algorithm which is also able to handle directed graphs with self-cycles and multi-edges. In their publication they present source code snippets in the D programming language which were combined into a running implementation. The sixth codebase is based on a partial implementation of Johnson's algorithm by Aric Hagberg in Python using the `networkx` graph library. It was fixed to properly calculate strongly connected components as well as to enumerate cycles in a deterministic order. The last implementation by Louis Dionne is using the Boost Graph Library and implements the algorithm by Hawick and James in C++.

All algorithms enumerated the same cycles on the given input graphs. Table 6.1 shows an overview of the different algorithms used. As one can see, the C++ and D implementations of the algorithm of Hawick and James perform fastest. The slowest implementation is the one by Frank Meyer done in Java, possibly due to how it uses an adjacency matrix to store the graph. The implementation of Tarjan's cycle enumeration algorithm performed well with respect to the fact that the algorithm has a worse complexity than Johnson's. On the other hand, the maximum amount of vertices in the tested graphs was probably low enough for the algorithm to still be fast. The imperative implementation of Johnson's Algorithm in Ocaml was found to be

enumeration algorithm by	**implementation language**	**main author(s)**	**average runtime (mm:ss)**
Johnson	OCaml (functional)	Abate, Schauer	01:51
Johnson	OCaml (imperative)	Abate, Schauer	01:50
Tarjan	Python	Schauer	02:02
Johnson	Java	Meyer	04:40
Hawick, James	D	Hawick, James	01:17
Johnson	Python	Hagberg	02:53
Hawick, James	C++	Dionne	01:18

Table 6.1: Benchmark of cycle enumeration algorithm on 380 random graphs with averages of five runs

slightly faster. Since it is also a bit more readable than the functional implementation, it was chosen for inclusion in `botch`.

Chapter 7

A New Heuristic for the Feedback Arc Set Problem

The core of the bootstrap problem is to find enough build dependencies to drop so that the dependency graph becomes acyclic. As it is desirable to modify a close to minimal amount of packages, the problem is equivalent with finding a minimum Feedback Arc Set.

We call a Feedback Arc Set a set of edges (or arcs) which, if removed from a graph, make that graph acyclic. The Feedback Arc Set Problem is the problem of finding such a set with minimum cardinality, called minimum Feedback Arc Set. The problem is NP-complete and one of Karp's 21 NP-complete problems [29]. An alternative formulation of the problem is called the maximum acyclic subgraph problem. It is the problem of finding the maximum number of edges of a graph such that the resulting subgraph still remains acyclic. Both problems are equivalent with respect to their optimal solutions but error bounds for polynomial approximations are only known for the maximum acyclic subgraph problem [23].

This chapter will start with an overview of some existing approximation algorithms calculating small Feedback Arc Sets. The next section will then give preprocessing steps and postprocessing steps to speed up Feedback Arc Set algorithms or even let them produce better results without significantly increasing their runtime. The following section will then introduce a new algorithm we call CycleFAS because it makes use of enumerating cycles. The next sections will introduce the other heuristics we tested CycleFAS against and then present the results of these benchmarks. The last section will outline how these general solutions for the Feedback Arc Set Problem can be adopted to the problem of finding build dependencies to break in a build graph.

7.1 State of the Art

The Feedback Arc Set Problem can trivially be approximated with linear complexity and an error bound of 0.5 by using an algorithm presented in [30]. Given a cyclic graph, choose an arbitrary ordering of its vertices. Then partition the edges into two sets. One set with the edges that point forward in the chosen order and the other set pointing backward. Lastly, remove the set with lower cardinality between those two from the graph. The resulting graph will be acyclic.

The set of removed edges will make the Feedback Arc Set. Since the maximum cardinality of both edge partitions cannot be greater than half the total number of edges, this is also the upper bound for the size of the Feedback Arc Set this method can find.

There exist only few solutions that give upper bounds for the size of their result. Let n be the number of vertices, e be the number of edges and d_{max} the maximum vertex degree of a graph. The best known solution [8] finds a Feedback Arc Set in $O(v \cdot e)$ with at most $\frac{e}{2} - \Omega(\frac{1}{\sqrt{d_{max}}})e$ edges. Due to the complexity of the algorithm, most implementations [34, 41, 45] instead choose a simpler version [18] which we will call EadesFAS in this thesis. In the worst case, it returns a bigger slightly Feedback Arc Set of at most $\frac{e}{2} - \frac{n}{6}$ edges but it does so in only $O(e)$ time.

There are different approaches to the Feedback Arc Set Problem. Some use combinatorial methods [17], simulated annealing [6] or a divide and conquer approach [44]. Plenty of research [11] has been done on heuristics that fall into the category of sorting. Sorting based heuristics modify the order of vertices. Edges pointing backward with respect to an order are made part of the Feedback Arc Set. Sorting based heuristics try to find an order such that the amount of backward edges is minimized.

7.2 Preprocessing and Postprocessing Steps

Before applying any Feedback Arc Set algorithms to a graph there are certain operations that can be carried out on the graph to reduce the amount of computation needed later. One such operation is to calculate the nontrivial strongly connected components of the graph. All vertices and edges that are not part of a nontrivial strongly connected component are also not part of a cycle. It does therefore not make sense to consider those edges to be part of the Feedback Arc Set. Instead, only the subgraphs of nontrivial strongly connected components should be the input to any Feedback Arc Set algorithm. The components are then solved individually and the Feedback Arc Set for the full graph is the union of the Feedback Arc Sets found for each strongly connected component. The solutions found for individual strongly connected components do not influence each other. Therefore, an optimal solution for each strongly connected component will also result in an optimal solution for the whole graph.

Another preprocessing step is the removal of all self-cycles. Any edge that forms a self-cycle is automatically part of the Feedback Arc Set. Those edges can therefore be removed and added to the Feedback Arc Set immediately.

A preprocessing step for sorting heuristics specifically is to consider two-cycles. Two-cycles have to be broken but as they contain exactly two edges, this measure makes sure that only exactly one edge is added to the Feedback Arc Set per two-cycle. Firstly, all edges involved in two-cycles are removed and stored before running a Feedback Arc Set algorithm. After it finishes processing the graph, each pair of edges that was formerly removed and stored is investigated. From each such pair, one edge will be a forward edge and the other will be a backward edge with respect to the order calculated by the sorting heuristic. The forward edge is inserted back into the graph while the backward edge is added to the Feedback Arc Set.

A postprocessing step can be applied to all heuristics that do not determine the Feedback Arc Set by the backward edges of a vertex order, that is, to all heuristics that are not based on sorting. After such an algorithm finished executing and determined a Feedback Arc Set, the

vertex order induced by this Feedback Arc Set is calculated. Each edge in the found Feedback Arc Set is then investigated and removed from the Feedback Arc Set and reinserted into the graph if it is found to be a forward pointing edge with respect to the calculated vertex order. This postprocessing step can be further refined by calculating a partial vertex order instead of a total order. A method for such an order is shown in chapter 9. It makes use of the fact that after removing the calculated Feedback Arc Set from the graph, some vertex pairs loose their relationship with each other. If it is found that an edge in the Feedback Arc Set connects two vertices that were found not to relate to each other in terms of a partial vertex order, then this edge can be removed from the Feedback Arc Set and reinserted to the graph. This reinsertion gives an ordering to the involved vertices.

7.3 A New Cycle Based Heuristic

Every valid Feedback Arc Set must contain at least one edge from every cycle in the graph. This consideration let to the development of a new heuristic to find a small Feedback Arc Set. The algorithm works by enumerating cycles in the graph and then greedily removing those edges with most cycles through them until all of the enumerated cycles are broken. Since enumeration of all cycles in the graph would be costly, only cycles up to a certain maximum length are enumerated. This maximum length is incremented and the algorithm re-run until the graph is cycle free.

Algorithm 7.1 Calculating a Feedback Arc Set

1: **procedure** PartialFAS(C_i, FAS_i)
2:     **if** $C_i = \emptyset$ **then**
3:         **return** FAS_i
4:     **else**
5:         $e \leftarrow$ EdgeWithMostCycles(C)
6:         $D \leftarrow$ CyclesThroughEdge(e)
7:         $C_{i+1} \leftarrow C_i \setminus D$
8:         $FAS_{i+1} \leftarrow FAS_i \cup \{e\}$
9:         **return** PartialFAS(C_{i+1}, FAS_{i+1})
10: **procedure** CycleFAS($G, maxlength$)
11:     **procedure** RecCycle(FAS_i, N)
12:         **if** $G.has_cycle$ **then**
13:             $C \leftarrow$ FindCycles(G, N)
14:             $P \leftarrow$ PartialFAS($C, \emptyset$)
15:             $G.remove_edges(P)$
16:             $FAS_{i+1} \leftarrow FAS_i \cup P$
17:             **return** RecCycle($FAS_{i+1}, N + 1$)
18:         **else**
19:             **return** FAS_i
20:     **return** RecCycle($\emptyset, maxlength$)

Algorithm 7.1 shows how the CycleFAS function operates. It receives a graph and an

initial maximum cycle length. The input graph should already be cleaned up according to the aforementioned preprocessing steps. The function RecCycle is called recursively with an initially empty Feedback Arc Set. It checks whether G is cyclic (line 13) and if not, return the current Feedback Arc Set. If G is still cyclic it finds all cycles up to length N in G (line 13). Those cycles are given to the PartialFAS function which will return a set of edges which break all the found cycles (line 14). The edges are removed from the graph (line 15) and added to the Feedback Arc Set (line 16). The function is then executed again with N incremented.

The PartialFAS function is called like this because it does not calculate a Feedback Arc Set but only a subset of it as it only has knowledge of a limited number of cycles in the graph. If the set of cycles is empty, it returns the current solution as-is (line 3). Otherwise, it retrieves the edge with most cycles through it (line 5) and then identifies all the cycles that cross this edge (line 6). Those cycles are then removed from the given set of cycles (line 7) and the edge is added to the solution (line 8). Breaking a cycle this way means that the association other edges might have had with that cycle is removed. The function is recursively called with a growing Feedback Arc Set and less and less cycles.

7.4 Sorting Heuristics

To evaluate this new approximate solution to the Feedback Arc Set Problem, we compare it to several sorting based heuristics. This includes the aforementioned widely used EadesFAS [18] algorithm. The heuristics we benchmark against have been implemented according to a comparative study by Brandenburg and Hanauer [11]. More specifically, we benchmark CycleFAS against the following algorithms.

The EadesFAS algorithm was introduced in [18]. It keeps two lists of vertices: one to store sink and the other to store source vertices. Sinks are vertices with an out-degree of zero. Sources are vertices with an in-degree of zero. The algorithm proceeds by identifying all sink and source vertices in the graph, adding them to their respective lists and removing them from the graph. Once all such vertices have been processed, the vertex with the highest out-degree minus its in-degree is treated as a source, added to the respective list and removed from the graph. The algorithm proceeds until there are no vertices left in the graph. The two lists are then concatenated and form the vertex order from which the Feedback Arc Set is determined by identifying the backward edges according to this order. In [14] an improved version is introduced which we call EadesImprovedFAS here. It differs from the original algorithm by finding the maximum *absolute* difference between in-degree and out-degree of all remaining vertices. Then if the out-degree of the found vertex is bigger than its in-degree, it is treated as a source, otherwise as a sink.

The InsertFAS heuristic was described in [13] as an application of classic insertion sort to the Feedback Arc Set Problem. For each vertex in the graph, the algorithm inserts it into the best position of the already sorted list of vertices. The best position is determined by the least number of backward edges which will later make the Feedback Arc Set. A conceptual weakness of InsertFAS is that determining the best position of a vertex only takes the already sorted vertices into account and ignores the still unsorted list of vertices. On the other hand it has the unique property that when being applied to a reversed vertex order, it will always calculate a

solution that is at least as good as the initial order as the authors of [13] show. It is therefore possible to use a vertex order reversal and one application of InsertFAS as a postprocessing step to any other Feedback Arc Set algorithm. The result will always be at least as small as the initially calculated Feedback Arc Set but might also even improve it.

The SiftFAS algorithm visits every vertex exactly once as given by the initial ordering and inserts the vertex into its best position in the full vertex list. Therefore, for each insertion of a vertex, its connections to all other vertices are taken into account to determine the position with the least amount of backward edges. This algorithm has previously been used to minimize edge crossings in graph drawing [40]. Even though SiftFAS is similar to InsertFAS it does not guarantee to produce a better result on a reversed vertex order as InsertFAS does.

The MoveFAS heuristic was introduced in [14] as an improvement to the InsertFAS heuristic. For each position in the vertex ordering it moves the vertex at that position to a better position in the ordering. This means that vertices that were moved forward in the vertex order will be considered again at a later point. It can also mean that a vertex is not considered at all in case the currently investigated vertex is moved only one position forward in the ordering.

The algorithms InsertFAS, SiftFAS and MoveFAS all depend on an initial ordering of vertices to determine in which order they are traversed. Each of the algorithms can be executed multiple times on the output of the previous iteration. Since all three algorithms are monotone [11], applying them multiple times will not produce worse results. The iteration stops once convergence is reached. We call those new algorithms InsertMultiFAS, SiftMultiFAS and MoveMultiFAS according to the used base algorithm, respectively.

Another variation of those three algorithms makes use of the property that applying InsertFAS on a reversed ordering, gives a result that is at least as good as the initial ordering. In this variation, one of InsertMultiFAS, SiftMultiFAS or MoveMultiFAS is applied and the result is then reversed. InsertFAS is then applied on this reversed result and that output is used as the input for the next iteration until convergence is reached. We call those algorithms InsertRevFAS, SiftRevFAS and MoveRevFAS depending on the algorithm they execute.

7.5 Evaluation

The speed of CycleFAS is greatly determined by how long the cycle enumeration takes. By adjusting the initial *maxlength* value, the execution time can be adjusted. Different choices of *maxlength* not only lead to different execution times but also different results.

Table 7.1 shows how CycleFAS performs on a build graph of 977 vertices. The smallest initial maximum cycle length is four because all two-cycles have been removed in a preprocessing step. For the chosen build graph there existed 36 two-cycles. This is expressed in the columns "sum of found cycles" and "Feedback Arc Set size" as consequently both cannot be smaller than 36. The maximum cycle lengths are a multiple of two because cycles in a build graph always have a length of a multiple of two. The largest initial maximum cycle length was chosen to be 16 as the runtime started to become infeasible. The column "amount of iterations" shows how often the function RecCycle was executed. There is no visible connection between the initial cycle length and the size of the Feedback Arc Set. As one can see, even choosing small values for the initial cycle length and having a small runtime results in Feedback Arc Sets with small

initial cycle length	runtime (hh:mm:ss)	resident memory (MiB)	sum of found cycles	amount of iterations	Feedback Arc Set size
4	00:00:07	48	36+387	6	36+57
6	00:00:07	48	36+1523	9	36+59
8	00:00:07	48	36+5189	7	36+59
10	00:00:09	53	36+25944	3	36+58
12	00:02:56	106	36+148162	5	36+68
14	01:09:11	497	36+797313	2	36+55
16	43:11:51	2979	36+5314457	3	36+57

Table 7.1: Performance of the Feedback Arc Set algorithm on a 977 vertices strongly connected component

sizes.

We now compare CycleFAS to other algorithms that were introduced earlier. We compare CycleFAS to EadesFAS, its improved version EadesImprovedFAS as well as to the pure InsertFAS, SiftFAS and MoveFAS algorithms and its variations. Additionally we use CycleFAS and EadesFAS as an input to InsertFAS, SiftFAS and MoveFAS and their variation to create hybrid algorithms.

As input graph we use 12 source graphs of different sizes and from different distributions. The graphs have been extracted from the past three Debian and Ubuntu releases as well as from six snapshots of Debian Sid from 2008 to 2013. Source graphs were used instead of build graphs because EadesFAS and EadesImprovedFAS cannot handle build graphs. Additionally, runtime requirements for algorithms based on InsertFAS, SiftFAS and MoveFAS were too large for graphs with around 1000 vertices. The size of the input source graphs ranged from 137 to 389 vertices and from 1655 to 9499 edges. It is entirely possible that opposite results would be achieved on graphs with a different structure. For this work though, only the performance for dependency graphs is relevant.

CycleFAS requires the initial maximum cycle length as an argument. We therefore chose to benchmark CycleFAS with three different choices of the initial maximum cycle length: three, four and five. Larger values only resulted in longer runtimes without producing better results. In contrast to a build graph, cycles in a source graph are not of a multiple of two.

The results of our benchmarks can be seen in Figure 7.1 and Figure 7.2. They show box plots of the achieved Feedback Arc Set size as well as the required runtime for running the respective algorithm on each of the 12 input graphs. To make the results achieved by each algorithm comparable even though they were achieved on different input graphs, they were normalized with respect to the smallest Feedback Arc Set found and the average runtime, respectively for each Figure.

The values depicted in Figure 7.1 are the result of calculating $\frac{v-min}{v}$ where v is the length of the Feedback Arc Set as calculated by an individual algorithm and min the minimum result achieved by any algorithm operating on the same graph. The closer the values are to zero, the

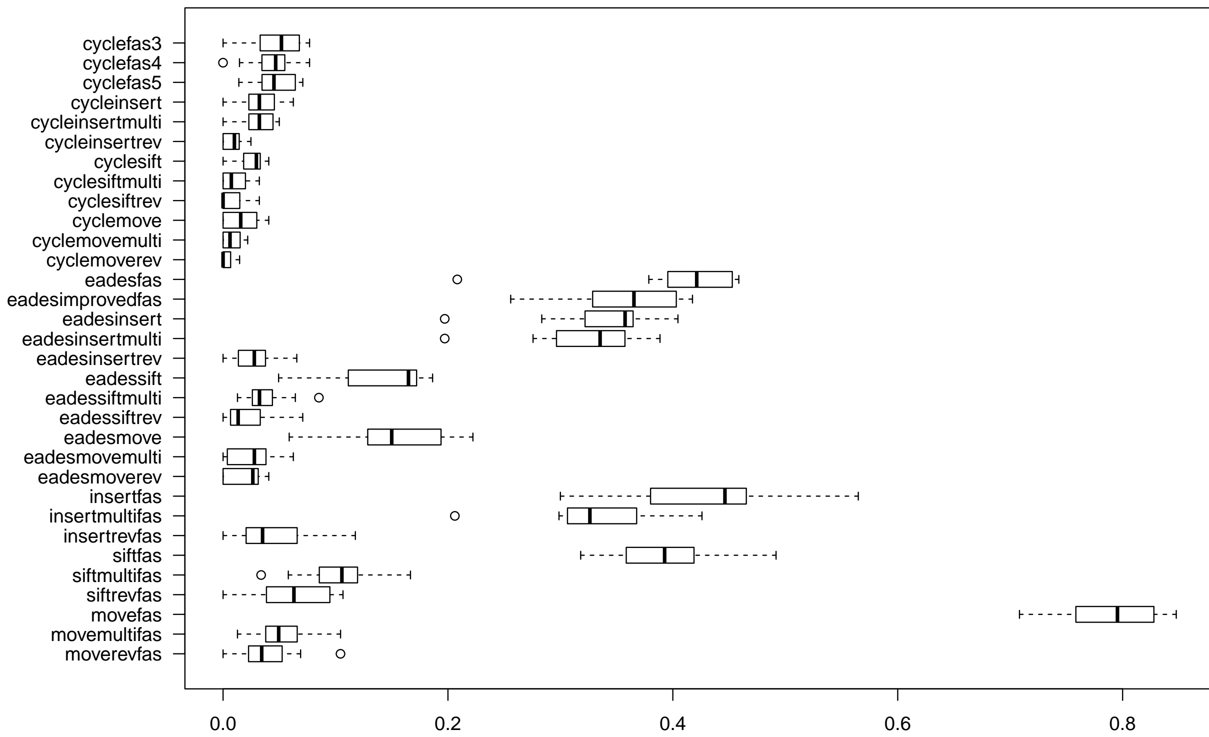


Figure 7.1: Solution quality (smaller is better)

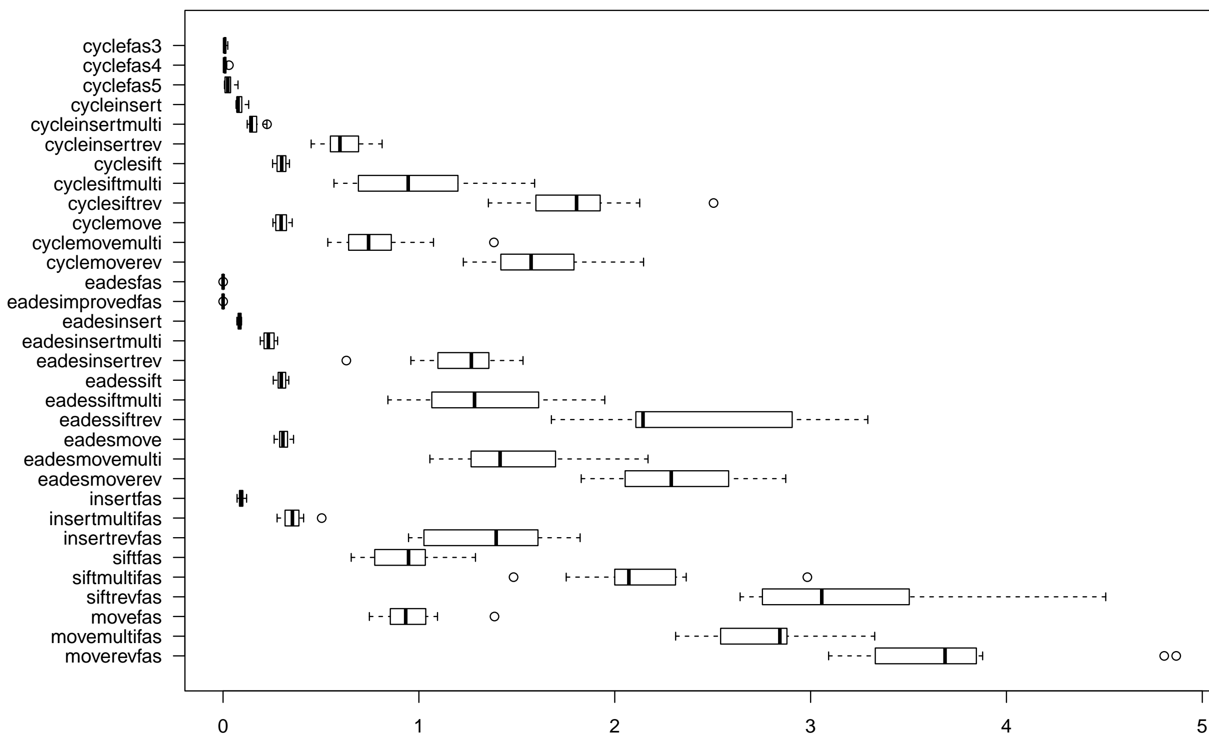


Figure 7.2: Required runtime (smaller is better)

better the result.

In a similar manner, the values depicted in Figure 7.2 are the result of calculating $\frac{mean}{v}$ where v is the time it took to execute the algorithm and $mean$ is the average time it took for any algorithm operating on the same graph. Values around one therefore stand for a comparatively average runtime while smaller values indicate smaller runtimes. Values between zero and one could've been made better visible by using a logarithmic scale but this has been avoided to keep linearity for readability.

It can be observed that the non-hybrid CycleFAS and EadesFAS algorithm are the fastest. While EadesFAS executes in fractions of a second, CycleFAS with the given initial cycle length often takes between two or five seconds. On the other hand, non-hybrid EadesFAS also ranks between the algorithms with the worst output. From the other non-hybrid algorithms, the only ones that produce a Feedback Arc Set as small as CycleFAS are the algorithms using the reversing technique (*RevFAS algorithms) but they are also the slowest.

Looking at the hybrid algorithms it can be seen that unsurprisingly all of them provide a better result than CycleFAS alone. This is expected as all of them refine an existing result which was given by CycleFAS. Their quality always ranges within 5% of the optimal solution and their main difference is their respective runtimes.

In summary it can be said that while EadesFAS is faster than any other algorithm it also produces much worse results and cannot be applied to build graphs. CycleFAS on the other hand produces Feedback Arc Sets in a matter of seconds which are as small as solutions calculated by other algorithms which take hours to complete.

Additionally, for an even better result, CycleFAS can be used as the input for a hybrid algorithm. Executing InsertFAS on the result of a previous execution of CycleFAS produces a result that is better than any non-hybrid algorithm while at the same time only needing a fraction of their runtime.

7.6 Application to a Build Graph

In the end we are interested to apply Feedback Arc Set finding algorithms to a build graph so that the developer can get the set of build dependencies that would make sense if they were droppable. On the other hand a build graph also contains builds-from edges which are not breakable. The algorithms must therefore be adapted to handle unbreakable edges.

The CycleFAS heuristic can be adapted by changing the function EdgeWithMostCycles to only return build-depends edges. The EadesFAS and EadesImprovedFAS algorithms cannot be adapted as they only allow two choices for each vertex to be placed which in some cases results in builds-from edges being a backward edge. The algorithms based on InsertFAS, SiftFAS and MoveFAS have to be adapted such that during a search for the new position of a vertex, no positions that would make a builds-from edge a backward edge are taken into account.

A build graph cannot contain self-cycles but it contains two-cycles. Two-cycles in a build graph are always composed of one build-depends and one builds-from edge. As only build-depends edges can be removed, one preprocessing step for build graphs is the removal of all two-cycles by adding the respective build-depends edges to the Feedback Arc Set.

Another modification that has to be done to the CycleFAS algorithm is, that the maximum cycle length in each iteration of RecCycle can be increased by two instead of being incremented by one. This is because in a build graph, all cycles have a length that is a multiple of two.

Chapter 8

Dependency Graph Analysis

Current binary distributions do neither implement droppable build dependencies nor do they annotate them in the metadata of source packages. Until enough source packages are modified with droppable build dependencies (or build profiles) the process of bootstrapping a distribution cannot be fully automated but requires human interaction. Furthermore, as a distribution changes over time and dependency relationships change, it might not be possible anymore to make the dependency graph acyclic with the existing build profiles. A human is again required to add additional build profiles or change existing ones.

After explaining the requirement for a human developer, this chapter handles the part of `botch` which assists a developer in analyzing a dependency graph and finding a good selection of candidate source packages to modify. In the last section it will outline the techniques that are available to the developer to modify source packages accordingly.

The measures explained in this chapter are only required if the distribution does not yet contain enough source packages with build profiles to the dependency graph acyclic. This is the case for all binary distributions as of now.

8.1 Need of a Human Developer

Approximate solutions to the Feedback Arc Set Problem and other heuristics can only give suggestions to the developer on which build dependencies to check for being droppable from their respective source packages or not. Solutions given by any heuristic can never guarantee that their suggestions are possible to be implemented in practice.

The reason for this is the nature of the task that has to be carried out. When being suggested to drop build dependencies to make a dependency graph acyclic, source packages have to be checked for whether or not this suggestion can be implemented. This checking involves to understand the build system and the software itself, to read the source code and documentation. If the build dependency can be dropped, the developer also has to make a decision on whether it is safe enough to drop that build dependency or whether an important feature would be disabled. The developer also has to choose between several options when modifying a package to require less build dependencies. He can implement a build profile but it might be easier to modify the package to cross compile or split the source package. Any of this means not only to

adapt the metadata of the source package, but also to extend the build scripts, prepare patches, write documentation for the added changes and communicate with the maintainers of the source package to make sure they agree with those changes. Neither the decisions that have to be taken nor any of the listed actions that have to be performed can be carried out by a machine.

Therefore, until enough source packages are modified with build profiles that allow build dependencies to be dropped, there will always be some amount of human labor needed. This limitation when analyzing inter dependency graphs between software components has been observed by others as well [33, 36]. A bootstrap can only ever be fully automatic, once enough source packages were modified with build profiles as suggested by the heuristics listed in this chapter.

8.2 Finding Build-Depends Edges to Remove

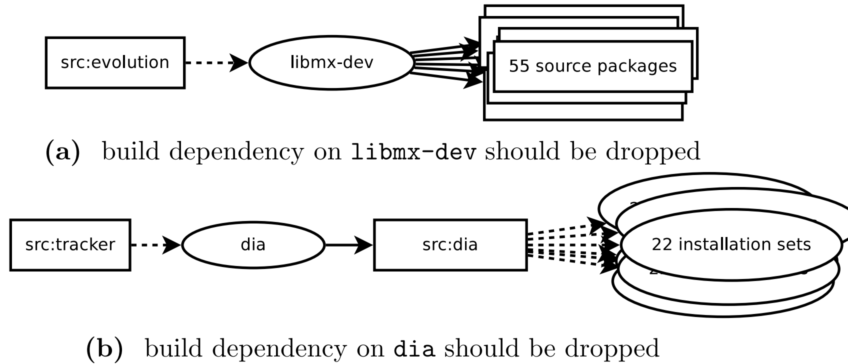
The dependency graph only rarely contains individual cycles. It is common that strongly connected components up to hundreds of vertices have to be turned into an acyclic graph by removing build-depends edges. The most obvious solution for making a nontrivial strongly connected component acyclic is to find a small Feedback Arc Set. How to generate such a set has been explained in the last chapter. But since the build-depends edges a Feedback Arc Set algorithm finds might not be removable from their respective source packages in practice, `botch` offers a couple of other heuristics as well.

8.2.1 Degree Based Heuristics

The most simple heuristics are those also employed by human bootstrappers in the past. One is to show those vertices in the dependency graph with the least number of outgoing edges. Those are the source packages which have only a small number of build dependencies missing. Allowing to drop these build dependencies will make the associated source packages compilable and break other dependency cycles by making more binary packages available.

The other one is to maintain a list of binary packages which are known to be droppable as build dependencies from most source packages. We call binary packages that qualify for this list weak build dependencies. This heuristic would list all those source packages which are only missing weak build dependencies. The list of weak dependencies has no further impact on the dependency analysis other than serving as an input for this heuristic. The content of this list is entirely up to the user of `botch` and usually contains binary packages responsible for documentation generation.

Figure 8.1 visualizes examples for two other heuristics. Figure 8.1a shows an example for a heuristic that finds source packages which build depend on a binary package whose installation set draws in lots of other source packages through builds-from edges. In the example shown in Figure 8.1a, it would be beneficial if `src:evolution` could be built without `libmx-dev` because then its connection to 55 other source packages would be broken. Similarly in Figure 8.1b, it would simplify the dependency graph a lot if `src:tracker` would not build-depend on `dia` because the source packages the installation set of `dia` builds from draw in 22 more unavailable binary packages.



(a) build dependency on `libmx-dev` should be dropped

(b) build dependency on `dia` should be dropped

Figure 8.1: Example of two in-/out-degree ratio based heuristics

Lastly in this category, `botch` offers a variety of general heuristics that depend on vertex degrees. It shows for source and installation set vertices individually the vertices with the highest and lowest in-degree or out-degree. It shows for both kinds of vertices those with the highest and lowest ratios between in-degree and out-degree. Finally it allows to display the highest or least connected vertices in the graph.

8.2.2 Enumerating Cycles

To make a graph acyclic, all its cycles must be broken. The smallest cycle in a build graph is of length two and contains exactly one build-depends and one builds-from edge. Since only build-depends edges can be broken, enumerating all two-cycles shows those build dependencies which must be removable from their respective source packages as there is no other way to break the associated two-cycle.

One has to make sure that the choice of installation set does not play a role in the existence of a two-cycle. Therefore, instead of enumerating two-cycles on the build graph, self-cycles are enumerated on the strong subgraph of the source graph.

One can identify two classes of two-cycles or self-cycles. Firstly there are those that are created by build dependencies of a source package on a binary package the source package builds. Such source packages mostly belong to compilers where the compiler or parts of it are written in the same language that it compiles. The build system for those source packages has to be adopted such that the source package can bootstrap itself without needing itself. Examples for source packages with such cycles are those for Free Pascal, Lisp, Haskell, Scheme, SML and Vala.

The other type are cycles because of build dependencies on any other binary package but where this binary package strongly depends on a binary package the source package builds. This kind of cycle is most unintuitive and can only be found through dependency graph analysis. An example is the unexpected self-cycle of the implementation for the Multicast DNS Discovery service avahi with itself through its build dependency on the graphical toolkit library GTK. This kind of cycle can quickly vanish or new ones appear once the interdependencies between binary packages change as the distribution is developed.

Beyond two-cycles it can also be beneficial to enumerate four-cycles or six-cycles because they

only give two and three different options, respectively, for breaking them. A skilled developer might immediately see that from the two build-depends edges that break a four cycle, one is very hard but the other one easy to break.

Furthermore, `botch` allows to search for cycles up to a given length in the build graph and then display an ordered list of build-depends edges with most cycles through them. This heuristic is the same as the concept of "shared dependencies" which was introduced in [36]. The authors of that publication propose a visual representation for shared edges. Their findings are hard to apply here because of up to 9000 edges involved in a single strongly connected component. `Botch` therefore only prints a list of the top most build-depends edges which, if removed, would break most cycles.

8.2.3 Strong Bridges and Strong Articulation Points

An edge is a strong bridge if its removal from the graph increases its amount of strongly connected components. It is therefore an edge which, if removed, will split a strongly connected component into two or more smaller components. In the same fashion, a vertex is a strong articulation point if its removal splits a strongly connected component. A fast method for finding strong bridges and strong articulation points using vertex and edge dominators has been introduced in [26]. `Botch` allows to enumerate both and sorts them by the amount of strongly connected components they split the input component into.

Calculating strong bridges that are build-depends edges have the immediate implication that modifying the associated source package accordingly will break the strongly connected component down into smaller ones. Strong bridges that are builds-from edges are only useful if the installation set vertex they connect from has a low in-degree. As builds-from edges can not be removed themselves, the build-depends edges that are incident into the associated installation set vertex would have to be removed instead.

Calculating strong articulation points for source vertices allows to quickly find those source packages which the developer might want to focus on earlier. Again, calculating strong articulation points for installation set vertices is only useful if they themselves have a low in-degree as only build-depends edges can be broken.

8.3 Removing Build-Depends Edges

Without any or with too few source packages with build profile information, build dependency cycles have to be broken during the cross as well as the native phase by relaxing their build dependencies. The following techniques exist to remove build-depends edges from the build graph and make it acyclic:

- Introduce build profiles
- Move dependencies from `Build-Depends` to `Build-Depends-Indep`
- Use `Multi-Arch:foreign` packages to satisfy dependencies
- Choose different installation sets for not-strong dependencies

- Split the source package in question
- Make binary packages available through cross compilation

Introducing a build profile or extending an existing one is the most obvious way to break build dependency cycles. It involves modifying a source package such that if it were compiled with a build profile activated, it would require less build dependencies at the expense of providing less functionality. Such changes include to build without a certain feature, to not generate documentation or to not run test cases.

Besides the `Build-Depends` field which indicates what build dependencies a source package has, there also exists the `Build-Depends-Indep` field [27]. This field is to specify those build dependencies of a source package which are only needed to build `Architecture:all` binary packages. Initially this field was introduced to lower the work of build servers because `Architecture:all` binary packages do not need to be rebuild on every architecture as they are architecture independent. The same can be said during bootstrapping: there is no reason for source packages to generate `Architecture:all` binary packages as they are already available. Therefore, during dependency graph generation, build dependencies listed in the `Build-Depends-Indep` field are ignored. One way to break build dependency cycles is therefore to find out that a build dependency is only used to generate `Architecture:all` binary packages and then move it from the `Build-Depends` field to the `Build-Depends-Indep` field.

Depending on the capabilities of the CPU and kernel or possibly even assisted by qemu user mode emulation [7], it might be possible to run existing Debian binary packages on the new hardware architecture. In this case, these foreign binary packages can be used to break build dependency cycles if they are `Multi-Arch:foreign` and can therefore satisfy dependencies of packages with a different architecture than their own. The only manual intervention required from the user is to supply the information about the foreign architecture. The dependency resolution algorithms will automatically take `Multi-Arch:foreign` packages into account.

If not all edges of a build dependency cycle are marked as strong, then choosing a different installation set for one vertex in the cycle will break the cycle. One has to be careful though, since choosing a different installation set might introduce other cycles which are even harder to break. The benefits and merits of calculating a minimal graph were discussed in chapter 3. Since other cycles might be introduced by the choice of a different installation set, this might not always be favorable. Using a solver it would be possible to generate a solution that picks installation sets in a way so that the size of strongly connected components is kept minimal.

Splitting a source package can help if compiling part of the original source package can be done effortlessly and would help providing binary packages for satisfying build dependencies of others. This method heavily interferes with the organization of packages in the distribution and therefore probably not preferable in most situations.

If none of those techniques is applicable for a build dependency cycle, then the last resort for breaking it, is to cross compile enough source packages, so that one build dependency in the cycle is installable. This option is always the last resort as from a purely technical standpoint it might be hard to make certain source packages cross compile but it should always theoretically be possible to do so. To break a dependency cycle, there are two ways to find out which source packages to cross compile. Either a source package which is part of the cycle can be cross

compiled. This will make the binary packages generated by this source package available and therefore break the cycle. Otherwise, an installation set along the cycle can be picked and all source packages to which the installation set vertex connects to using a builds-from edge can be cross compiled to make the binary packages of the installation set available. It must be noted that when choosing a source package to be cross compiled, one might still run into cross build dependency cycles. But as there are less cycles in the cross phase as there are in the native phase, those cycles will be easier to solve.

Chapter 9

Creating a Build Order

A build order is the final output of `botch` once enough source packages were modified with build profile information so that the graph can be made acyclic. This chapter will first introduce the method that is used to select the source packages in the build graph which are to be profile built. The next section will handle a method which allows a build order to be generated even for graphs that cannot yet be made acyclic. The last section covers the generation of a partial vertex order of an acyclic graph.

9.1 Feedback Vertex Set Algorithm

It can easily be tested whether a nontrivial strongly connected component can be made acyclic with the given set of build profile information. One first identifies all source packages which implement a build profile. One then drops all build dependencies that those build profiles mark as being droppable from those source packages. If the resulting dependency graph is acyclic, then enough source packages contain build profile information. An ordering algorithm can now deduce a build order from the acyclic graph.

As build profile mechanisms are introduced into distributions, it is expected that package maintainers add build profiles to the source packages they maintain without those changes being strictly needed. Additionally, as the dependencies within a distribution change over its lifetime and new build profiles are introduced, old and now possibly unneeded build profile information will still be present in source packages. It is therefore expected that in the future, the amount of source packages with a build profile will exceed the amount that is needed to make the dependency graph acyclic.

While the approach of just building all source packages that implement a build profile in reduced form will certainly work in theory, in practice it is desirable to profile build a close to minimal amount of source packages. This is because building source packages with a build profile always bears the risk that the produced binary packages behave differently than those built without a build profile. It might therefore come to unexpected build failures in source packages that build depend on those binary packages. To minimize this risk, an algorithm is needed which only picks a close to minimal amount of source packages to be built with a build profile.

Picking those source packages means to pick a close to minimal amount of source vertices in the build graph which are then profile built to make the graph acyclic. This problem is similar to the Feedback Vertex Set Problem. The Feedback Vertex Set Problem is the problem of finding the minimum number of vertices which, if removed from the graph, make the graph acyclic. In this case, we do not want to find vertices to remove but source vertices to replace with their respective build profile version.

We will use the same intuition that was used to develop the Feedback Arc Set algorithm. Firstly, cycles up to a specific length are enumerated. Then the source package which, if it were profile built, would break the most cycles is picked as being profile built and added to the Feedback Vertex Set. The process is being repeated until the graph is cycle free.

Algorithm 9.1 Calculating a Feedback Vertex Set with alternative functions for application to a build graph in comments on the right hand side.

```
 1: procedure PartialFVS(C_i, FVS_i)
 2:     if C_i = ∅ then
 3:         return FVS_i
 4:     else
 5:         v ← VertexWithMostCycles(C)        ▷ SourceWithMostRemovableCycles
 6:         D ← CyclesThroughVertex(v)          ▷ RemovableCyclesThroughSource
 7:         C_{i+1} ← C_i \ D
 8:         FVS_{i+1} ← FVS_i ∪ {v}
 9:         return PartialFVS(C_{i+1}, FVS_{i+1})
10: procedure CycleFVS(G, maxlength)
11:     procedure RecCycle(FVS_i, N)
12:         if G.has_cycle then
13:             C ← FindCycles(G, N)
14:             P ← PartialFVS(C, ∅)
15:             ∀v ∈ P : G.remove_vertex(v)      ▷ ∀s ∈ P : G.modify_source(s)
16:             FVS_{i+1} ← FVS_i ∪ P
17:             return RecCycle(FVS_{i+1}, N + 1)                    ▷ N + 2
18:         else
19:             return FVS_i
20:     return RecCycle(∅, maxlength)
```

The general form of the Feedback Vertex Set algorithm can be seen in Algorithm 9.1. One can see its similarity with the Feedback Arc Set algorithm in Algorithm 7.1. The only difference is that all occurrences of edges in the Feedback Arc Set algorithm is now replaced with vertices. The vertex with most cycles through it is found (line 5) and the cycles through this vertex are retrieved (line 6). Found vertices are removed from the graph (line 15) and a set of vertices is returned. Otherwise the program logic follows the same rules.

This general Feedback Vertex Set algorithm can easily be turned into one that selects only a close to minimal amount of source packages to be profile built by replacing the function VertexWithMostCycles with SourceWithMostRemovableCycles, CyclesThroughVer-

TEX with REMOVABLECYCLESTHROUGHSOURCE and $G.remove_vertex$ with $G.modify_source$. Instead of finding the vertex with most cycles through it, the source vertex with most cycles through it which would be broken if the source package was profile built must be found. Instead of finding all cycles through a vertex, the cycles which would be removed if the found source package was profile built are found. And instead of removing vertices from the graph, the selected vertices are modified in the graph according to their build profile information using $G.modify_source$.

Additionally, the maximum cycle length can be increased by two instead of being incremented by one because the algorithm works on a build graph. Nearly the same pre processing steps that apply for the Feedback Arc Set Problem can be applied here as well. For example source packages which are involved in two-cycles automatically have to be profile built.

Using this method, nontrivial strongly connected components can be made acyclic by only profile building a close to minimal amount of source packages. An algorithm would handle each nontrivial strongly connected component individually. It would first check whether enough source packages have build profiles for making the current component acyclic. If this is the case, then it would run the algorithm outlined in this section to find a smaller set of source packages to profile build.

After all nontrivial strongly components have been processed in this fashion, the build graph is converted into a source graph by using Algorithm 3.3. It does not make sense to run this algorithm on a source graph as a source graph hides which connections to other source packages are made because of which direct build dependency. It is therefore hard to change a source graph according to a requested build profile.

9.2 Collapsing Strongly Connected Components in a Source Graph

The graph resulting from the lastly explained algorithm might still be cyclic. While there might be enough source packages with build profile information to make some strongly connected components acyclic, this might not be the case for other strongly connected components. Therefore, the overall graph might still contain cycles.

It would be an inconvenient limitation if `botch` could only generate a build order once enough source packages contained build profile information to make the complete graph acyclic. Especially in a situation where no build profiles exist yet, this means that one would have to edit over a hundred source packages before it were possible to calculate a build order. Furthermore, a developer might decide that some nontrivial strongly connected components are to be left unsolved until a later point so that he can focus his attention on other tasks.

The situation is solved by introducing a new vertex type in source graphs. This new vertex type represents strongly connected components and vertices of this type are therefore called SCC vertices. By collapsing all vertices which are part of a nontrivial strongly connected component into a single vertex, any cyclic graph can be made acyclic. Every SCC vertex saves the source packages that are part of it.

Source vertices that are part of a nontrivial strongly connected component can be collapsed into a single vertex by using vertex contraction. Algorithm 9.2 shows how the contraction

Algorithm 9.2 Collapsing a source graph.

```
1: procedure Collapse(G)
2:     for scc ∈ G.nontrivial_scc do
3:         for vertex ∈ scc do
4:             for succ ∈ vertex.successors do
5:                 if (isSCC(succ) ∧ succ ≠ scc) ∨ (isSrc(succ) ∧ succ ∉ scc) then
6:                     G.add_edge(scc, succ)
7:             for pred ∈ vertex.predecessors do
8:                 if (isSCC(pred) ∧ pred ≠ scc) ∨ (isSrc(pred) ∧ pred ∉ scc) then
9:                     G.add_edge(pred, scc)
10:            G.remove_vertex(vertex)
11:    ReplaceSelfCyclesWithSCC(G)
```

algorithm operates. It iterates over all nontrivial strongly connected components that are left in a source graph (line 2). For each vertex in a component (line 3) it gets its successors and predecessor vertices. If the neighbor is an SCC vertex but not the SCC vertex the original vertex is part of, an edge is added. If the neighbor is a source vertex but not a source vertex that is part of this strongly connected component, an edge is added as well. Adding these edges connects the neighbors of the original vertex to a new SCC vertex representing the current strongly connected component. In the end, the visited vertex is removed (line 10).

The function ReplaceSelfCyclesWithSCC will iterate over all edges in the graph and make those source vertices which belong to a self-edge SCC vertices as well. Since at this point, all the nontrivial strongly connected components are already processed, the remaining self-cycles cannot be part of them.

9.3 Computing a Partial Order

Now that the graph is surely being made acyclic, a build order can be calculated. A build order is a partial order over the reachability relationship of source vertices in the source graph through the edges connecting them. It is a partial relationship because not for all possible pairs of source vertices, such a relationship is defined. In particular, pairs of source package vertices for which this relationship is not defined can be built in parallel. Since bootstrapping a distribution means to compile all its tens of thousand of source packages, the information about which packages can be built in parallel can potentially help to considerably speed up the bootstrapping process.

Algorithm 9.3 shows a trivial implementation of the ordering algorithm. An error is returned if the input graph should contain a cycle (line 3). This is important because the rest of the algorithm does not detect cycles and would therefore run into an infinite loop if the input graph would be cyclic. The algorithm starts with all vertices in the graph that are sink vertices and therefore have no successor (line 4). Source vertices are vertices with an in-degree of zero. Those vertices are the first elements in the order and as such already added to the result (line 5). They are also added to the set of processed vertices (line 6). Their predecessors are added to the set of vertices that will be checked next (line 7). A loop starts which executes until the set of vertices

Algorithm 9.3 Computing a partial order

```
 1: procedure PartialOrder(G)
 2:     if G.has_cycle then
 3:         return error                                  ▷ G must be acyclic
 4:     init ← G.get_sinks                               ▷ get vertices without successors
 5:     result ← init :: [ ]
 6:     processed ← {v | v ∈ init}
 7:     tocheck ← ⋃_{v∈init} v.predecessors
 8:     while tocheck.cardinality > 0 do
 9:         new ← {v | v ∈ tocheck ∧ v.successors ⊆ processed}
10:         tocheck ← tocheck ∪ (⋃_{v∈new} v.predecessors)
11:         tocheck ← tocheck \ new
12:         processed ← processed ∪ new
13:         result ← new :: result
14:     return result
```

to be checked is empty (line 8). In the loop body, first all vertices are found for which all its successors are a subset of the already processed vertices (line 9). In a source graph, this would select all those source packages which only depend on those source packages which already have been compiled. Next, the union of the predecessors of those new vertices is added to the set of vertices that is to be checked (line 10). The set of newly found vertices is removed from this set (line 11) and added to the set of processed vertices (12). The set of newly found vertices is then appended to the resulting build order.

The result is a list of sets. The source packages belonging to the source vertices within each such set can all be built in parallel as they do not depend upon each other. The calculated build order can be given to a build daemon together with the information of which source packages are to be built with enabled build profiles.

An important aspect that was omitted from Algorithm 9.3 for clarity is the rebuilding of profile built source packages in their full form once all their build dependencies can be satisfied. Additionally, suppose that a source package `src:A` was built with one of its build dependencies satisfied by a binary package `B` which itself was built by a profile built source package `src:B`. The binary packages produced by `src:A` were produced from potentially incomplete binary packages as `src:B` was profile built. To ensure that all produced binary packages are built with their full feature set, all source packages involved in a strongly connected component are rebuilt once the strongly connected component is solved. In the end of the build order, all source packages should be recompiled to further minimize any possible error.

Chapter 10

Experimental Results

In this chapter we will present execution times, memory requirements and the solutions generated by `botch`. The next section presents a heuristic for identifying droppable build dependencies using Gentoo. In the following section we explain why the cross phase is omitted from the analysis. The last two sections handle the benchmark setup and the obtained results.

10.1 Gentoo

To test the developed algorithms, some build dependencies had to be selected and marked as droppable. This selection should not be arbitrary but at the same time we established in chapter 8 that this information can not be generated by a machine but needs human interaction. But since we are only interested in metadata information and since an approximate solution is sufficient, the problem of automatically finding droppable build dependencies can be solved by finding a Linux distribution which provides the following:

- Allows to build source packages with different feature sets enabled or disabled
- Stores the information about which build dependency is required for each feature in a machine readable format
- Provides a similar selection of software packages as Debian so that package names can be mapped

Above requirements are met for the Gentoo Linux distribution which, in contrast to Debian, is a source based distribution [48]. While there exist repositories to download readily compiled binaries, the usual modality is to download and compile source packages. To fully utilize the advantages of manually compiling all source packages every time upon installation, Gentoo source packages allow to be compiled with a selectable subset of their features. These features are enabled and disabled by setting so called USE flags. In addition, Gentoo source packages also specify which dependencies are specific to USE flags being set or not set.

It is now possible to create a tool which parses Gentoo dependency information, retrieves those build dependencies which are marked as optional, maps them to Debian package names

and outputs them. The tool is not part of the `botch` project but released under the GPL on `https://gitorious.org/debian-bootstrap/gen2deb`.

While this approach can only extract metadata and not the necessary changes to the source code, the extracted information is highly error prone and should only be seen as a rough heuristic. The following problems can occur.

Debian and Gentoo use different versions of upstream software. The version numbers often differ only by a minor revision but due to this, packages are mapped without taking the version number into account. There are also some packages which are only in one distribution but not the other. Additionally, source packages might have been split in a different manner. Furthermore, many build dependencies of Debian source packages are pulled in indirectly through binary dependencies of its build dependencies. Gentoo source packages depend upon others more directly. This means, that many build dependencies which can be dropped in Gentoo cannot be dropped in Debian because the associated Debian package only indirectly depends on it. Lastly, Gentoo does not have the concept of `build-essential`. The Gentoo metadata information might therefore suggest to drop build dependencies which are implicit for Debian source packages.

10.2 Cross Phase

The cross build phase was skipped during testing. While `botch` has no problem resolving multiarch cross build dependencies and generating the associated dependency graphs, not enough binary packages are marked with multiarch yet. This means that for a large number of source packages, multiarch conflicts will occur and cross build dependency resolution will fail. As a result, with the current status of source and binary package metadata, the cross phase can not be sufficiently analyzed.

For now, we work around this by assuming the minimal build system can be created without breaking any cross build dependency cycles. Cross build analysis in practice which was done by Wookey [51] showed that a minimal build system containing over 100 binary packages can indeed be cross compiled with only profile building twelve source packages. This result also suggests that as long as cross compilation is frowned upon by distribution architects, `botch` is not strictly needed for cross build dependency analysis.

Therefore, we assume, that a minimal build system containing all `Essential:yes` packages, `build-essential`, `debhelper`, `apt` and all its dependencies exists. Analyzing the cross case becomes more important once cross compilation becomes more common and a greater number of binary packages is selected to initially be cross compiled.

10.3 Setup

The whole process is purely virtual and no Debian source packages are modified in practice through it. Instead, we solely rely on and work with package metadata. This metadata is enough to execute all the tools and algorithms explained so far.

There is not yet a format to store build profiles in source package metadata. On the other hand, a way to store such metadata is necessary so that functionalities of `botch` which requires build profile information can be tested. Therefore, droppable build dependencies of source

packages are stored in external plain text files. The content of those text files was aggregated from different sources.

The source from which most droppable build dependencies were retrieved was Gentoo by using the method outlined in the last section. Other information was manually supplied from developers that did bootstraps of Debian in the past. Those contributions were made by Wookey, Daniel Schepler, Patrick McDermott and Thorsten Glaser. Since self-cycles must be breakable, we also added all build dependencies which are involved in strong self-cycles to the list of droppable build dependencies. Finally, we added a list of build dependencies which was agreed upon to be droppable by nearly all source packages that build depend on them. Those build dependencies are almost exclusively used to generate documentation for the respective source packages.

Information about droppable build dependencies gathered by these methods was found not to be enough to make the dependency graphs acyclic. We therefore used the developed Feedback Arc Set algorithm to determine a set of eight build-depends edges which, together with the existing information, allows to make the graph acyclic. While the information automatically retrieved from Gentoo and manually supplied by developers provides a good estimate about which build dependencies are droppable in practice, it cannot be guaranteed that those eight forcefully chosen build dependencies can actually be broken. We nevertheless accept this issue as a negligible systematic error.

All testing was done with package data from the Debian Sid distribution as of 2013-01-01. The benchmarks were run on a 2.5 GHz Intel Core i5 machine. Any data was read and written to a RAM disk to minimize I/O delays.

10.4 Benchmarks

Benchmarks are carried out in two scenarios. The first scenario runs the analysis on a self-contained subset of the full distribution. The second runs on a full Debian Sid distribution. In both scenarios, first, the selection of packages for the minimal build system is made, then the fixpoint algorithm is run to calculate the set of available binary packages as input for dependency graph creation. After the dependency graph was generated, a tool analyzes it and outputs the result of all the previously explained heuristics for dependency graph analysis. In the end a build order is generated.

Generating the self contained distribution from the full distribution takes about one minute but drastically cuts the execution time for the other algorithms. The generated self-contained distribution measures 619 source and 2078 binary packages. The full distribution is 18613 source and 38433 binary packages big.

After a self-contained distribution is generated, calculating the dependency graph, takes the most amount of time. The advantages of working with a self-contained distribution show when comparing the execution times of dependency graph generation. It takes three seconds for the self-contained distribution, while generating it for the full distribution takes eight minutes. Residential memory requirements are at 134 MiB and 788 MiB, respectively. The generated build graph measures 1614 vertices for the self-contained distribution and 26606 vertices for the full distribution. If information about strong dependencies is desired to be embedded into the

dependency graphs, calculating them takes an additional six minutes of execution time and 769 MiB of additional resident memory for both cases.

Once the dependency graph is generated, the heuristics presented in chapter 8 are applied to the graph. In neither case, doing this analysis including calculation of a Feedback Arc Set takes more than four seconds. In both cases, the build graph contains one strongly connected component of 866 vertices. In addition, the build graph for the full distribution contains nine additional nontrivial strongly connected components with a maximum number of five member vertices. From this one can deduce, that solving the strongly connected component for a self-contained distribution first, will also solve the dependency relationships for the full distribution except for those nine small strongly connected components. One can therefore take advantage of the speed increase that generating a smaller self-contained repository brings and at the same time solve the biggest strongly connected component of the full dependency graph.

Generating a build order is similarly quick. It takes two seconds for the self-contained distribution and eight seconds for the full distribution. With the available information about droppable build dependencies, the self-contained distribution could be bootstrapped with just modifying 76 source packages. The full distribution can be bootstrapped with modifying 85 source packages. Those additional source packages are needed to break the nine additional nontrivial strongly connected components the full graph possesses. In terms of the generated build order, 60 steps are needed for the self-contained distribution and 66 for the full distribution. All source packages within each step can be compiled in parallel.

Chapter 11

Conclusion

In this chapter we start with presenting related work in this field, then list possible future developments. In the last two sections we give acknowledgments where they are due and at last, draw a conclusion.

11.1 Related Work

The author of this thesis published a paper [4] about the same topics in the proceedings of CBSE 2013 together with Pietro Abate. That paper is of more theoretical nature but describes the same processes. Many algorithms presented in this thesis were already presented in that paper and are mentioned as such in the text.

In [36] an approach called CycleTable is presented which is very similar to how our cycle heuristic works. Interdependencies are found to be "shared dependencies" if they are part of many cycles. The result is then visually presented to the user. In another paper [35] the same authors present an approach they call "enriched Dependency Structure Matrix (eDSM)". Both approaches use colors to highlight and visualize cycles in a graph. They are detailed further in the PhD thesis of Jannik Laval [33].

Further analysis of cycles between software components has been done in the area of C++ and Java classes. PASTA [24] allows to interactively refactor and arrange Java classes into hierarchies. The used heuristics are similar to those used in this paper. The Eclipse plugin Jooj [41, 42] does the same using the EadesFAS Feedback Arc Set heuristic. In [6] simulated annealing is used to remove dependency cycles.

In summary, existing papers presenting methods to identify cyclic dependencies between software components are limited to a far smaller problem size. While `botch` easily handles a repository of several ten thousand packages, other approaches focus on visualizing the interdependencies of only a few hundred software components. This limitation often comes from a focus on visualization of the problem. In `botch`, visualization plays a minor role and can only be achieved through third-party tools which allow to parse and display GraphML [12] data.

Furthermore, existing approaches concentrate on analyzing dependency relations between Java or C++ classes which rarely come close to the amount of packages present in current binary distributions. Analysis of dependency graphs for package based systems of the order

of magnitude as presented in this paper is done by the Mancoosi research project [1, 19, 38] but ignores source packages and focuses on constraint solvers, upgradeability, rollbacks and installability of binary packages. More specifically, the dependency graphs that are generated do not contain source packages but only binary packages. A good overview of these kind of graphs is given in the PhD thesis of Jacob Pieter Boender [10].

11.2 Future Developments

This work has so far been theoretical. Only dependency metadata has been analyzed. It needs to be confirmed that the calculated build order actually works in practice. A Debian Google Summer of Code 2013 project is out to test `botch` in practice. The task is to pick an existing architecture and suppose that no binary packages for it would exist. `Botch` would then be used to calculate a build order to bootstrap this architecture from scratch. Since source packages are already adapted to compile on the "new" architecture, this project can focus on introducing build profiles and evaluating the correctness of `botch` in practice.

Currently, `botch` only supports dropping build dependencies. In reality, building a source package with a build profile might also mean that a binary package it would otherwise build is not produced. This creates a problem if another source package build depends on this binary not-built binary package. The solution is to leave the original source vertex in the dependency graph and then add the profile built version of the same source package while maintaining the correct dependency relationships to surrounding installation set vertices as required. Then, in the build order, the profile built source package would be built first, and then the source package would be rebuilt fully at a later point to fulfill the build dependencies of other source packages.

Building a source package with a build profile also does not always mean that build dependencies are removed. In some cases it can also mean that a build dependency is replaced or added. Algorithms must be aware of this and not assume that profile building only removes edges but that it can also add edges. Both issues are not addressed yet by `botch` as there neither exists a format to store this information yet, nor does there exist data about which packages might require these features, if at all.

This work can be applied without modifications to all Debian based distributions. In theory, RPM [20] based distributions contain enough metadata information to serve as input for `botch` as well. Additionally, `dose3` includes a parser for RPM package metadata. One future addition is therefore support for RPM based distributions. If `botch` would support Debian as well as RPM based distributions, it would cover most of the available binary based software distributions.

11.3 Acknowledgments

This project started as a Google Summer of Code project for the Debian operating system in 2012. My thanks go to Pietro Abate and Wookey who were my mentors not only during that summer but also at any point I needed their help afterward. While Wookey supplied me with knowledge about the practical aspects of the bootstrap problem, Pietro Abate taught me the academic side of things. Without Pietro's plentiful help with `dose3`, `ocamlgraph` and without his patches and hints improving the source code of `botch`, this project would be nowhere close

to where it is right now. This thesis would also not have been possible without Andreas Nüchter agreeing to be my supervisor for it even though the topic does not fall directly into his field of research. My thanks also go to the Debian community for very fruitful discussions about the topic.

11.4 Conclusion

Bootstrapping binary based free and open source software distributions has been a major problem in the past. Adaption of binary distributions to new architectures took up to a year of manual guesswork. In this thesis we presented `botch`, a collection of tools which implement algorithms and heuristics to analyze the interdependencies between packages, generate a dependency graph and present the user with good solutions on how to make this graph acyclic. In the end, a build order can be generated. Once enough build profiles have been added through the assistance of the provided heuristics, the process of bootstrapping a binary distribution from scratch can be fully automated. `Botch` can then therefore be run regularly on the current versions of a distribution archive to check whether the distribution is still bootstrappable as a quality assurance measure. Should a distribution be found not to be bootstrappable anymore, then the provided heuristics simplify finding appropriate source packages to modify and make the distribution bootstrappable again. The execution time of all these algorithms is in the order of minutes for a full repository and down to seconds for a smaller self-contained repository. We showed that assisted by `botch`, bootstrapping a free and open source binary distribution from scratch can become automatic, deterministic and fast.

Bibliography

[1] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM 2009: International Symposium on Empirical Software Engineering and Measurement*, pages 89–99. IEEE, 2009.

[2] Pietro Abate, Roberto di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Mpm: A modular package manager. In *CBSE 2011*. ACM, 21/06/2011 2011.

[3] Pietro Abate and Johannes Schauer. botch. `https://gitorious.org/debian-bootstrap/botch`.

[4] Pietro Abate and Johannes Schauer. Bootstrapping software distributions. In *Proceedings of CBSE 2013*. ACM, 2013.

[5] Pietro Abate, Ralf Treinen, Roberto Di Cosmo, Stefano Zacchiroli, Jaap Boender, and Jakub Zwolakowski. dose3. `http://mancoosi.org/software/#index2h1`.

[6] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 103–112. IEEE, 2009.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.

[8] Bonnie Berger and Peter W. Shor. Approximation alogorithms for the maximum acyclic subgraph problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 236–243. Society for Industrial and Applied Mathematics, 1990.

[9] Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, and Fabio Mancinelli. Improving the quality of GNU/Linux distributions. In *COMPSAC*, pages 1240–1246. IEEE, 2008.

[10] Jacob Pieter Boender. *A formal study of Free Software Distributions*. PhD thesis, Université Paris Diderot, Paris 7, doctorale de Sciences Mathématiques de Paris Centre, 2011.

[11] KHF Brandenburg and Kathrin Hanauer. Sorting heuristics for the feedback arc set problem. 2011.

[12] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M Scott Marshall. Graphml progress report structural layer proposal. In *Graph Drawing*, pages 501–512. Springer, 2002.

[13] Stefan Chanas and Przemysław Kobylański. A new heuristic algorithm solving the linear ordering problem. *Computational optimization and applications*, 6(2):191–205, 1996.

[14] Tom Coleman and Anthony Wirth. Ranking tournaments: Local search and a new algorithm. *Journal of Experimental Algorithmics (JEA)*, 14:6, 2009.

[15] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ml functors. *Trends in functional programming*, 8:124–140, 2008.

[16] D. Crocker. STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. RFC 822 (Standard), August 1982. Obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148.

[17] Camil Demetrescu and Irene Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, 2003.

[18] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[19] EDOS Project. Report on formal management of software dependencies. EDOS Project Deliverables D2.1 and D2.2, March 2006.

[20] Eric Foster-Johnson. Red hat rpm guide. 2003.

[21] Martin Gebser, Roland Kaminski, and Torsten Schaub. aspcud: A linux package configuration tool based on answer set programming. *arXiv preprint arXiv:1109.0113*, 2011.

[22] Olivier Gruber, BJ Hargrave, Jeff McAffer, Pascal Rapicault, and Thomas Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Systems Journal*, 44(2):289–299, 2005.

[23] Refael Hassin and Shlomi Rubinstein. Approximations for the maximum acyclic subgraph problem. *Information processing letters*, 51(3):133–140, 1994.

[24] Edwin Hautus. Improving java software through package structure analysis. In *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[25] Ken A. Hawick and Heath A. James. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *Proc. Int. Conference on Foundations of Computer Science FCS*, volume 8, pages 14–20, 2005.

[26] Giuseppe F Italiano, Luigi Laura, and Federico Santaroni. Finding strong bridges and strong articulation points in linear time. In *Combinatorial Optimization and Applications*, pages 157–169. Springer, 2010.

[27] Ian Jackson et al. Debian policy manual. 1996.

[28] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[29] Richard Manning Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010.

[30] Bernhard H. Korte. Approximative algorithms for discrete optimization problems. *Ann. Discrete Math*, 4:85–120, 1979.

[31] Martin Krzywinski, Inanc Birol, Steven JM Jones, and Marco A Marra. Hive plots—rational approach to visualizing networks. *Briefings in Bioinformatics*, 13(5):627–644, 2012.

[32] Steve Langasek. Multiarch specification. `https://wiki.ubuntu.com/MultiarchSpec`, 2009.

[33] Jannik Laval. *Package Dependencies Analysis and Remediation in Object-Oriented Systems.* PhD thesis, Université des Sciences et Technologies de Lille, 2011.

[34] Jannik Laval, Nicolas Anquetil, and Stéphane Ducasse. Ozone: Package layered structure identification in presence of cycles. In *Proceedings of the 9th edition of the Workshop BElgian-NEtherlands software eVOLution seminar, BENEVOL*, 2010.

[35] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 113–122. IEEE, 2009.

[36] Jannik Laval, Simon Denier, Stéphane Ducasse, et al. Cycles assessment with cycletable. 2011.

[37] Abraham Lempel and Israel Cederbaum. Minimum feedback arc and vertex sets of a directed graph. *Circuit Theory, IEEE Transactions on*, 13(4):399–403, 1966.

[38] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006: Automated Software Engineering*, pages 199–208. IEEE, 2006.

[39] Prabhaker Mateti and Narsingh Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976.

[40] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using sifting for k-layer straightline crossing minimization. In *Graph Drawing*, pages 217–224. Springer, 1999.

[41] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[42] Hayden Melton and Ewan Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In Gillian Dobbie, editor, *Thirtieth Australasian Computer Science Conference (ACSC2007)*, volume 62 of *CRPIT*, pages 87–95, Ballarat Australia, 2007. ACS.

[43] Simon Richter, Loïc Minier, and Wookey. Multiarchcross specification. `https://wiki.ubuntu.com/MultiarchCross`, 2009.

[44] Youssef Saab. A fast and effective algorithm for the feedback arc set problem. *Journal of Heuristics*, 7(3):235–250, 2001.

[45] Christoph Daniel Schulze. *Optimizing Automatic Layout for Data Flow Diagrams*. PhD thesis, Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2011.

[46] Clemens Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[47] Robert Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[48] George K Thiruvathukal. Gentoo linux: the next generation of linux. *Computing in science & engineering*, 6(5):66–74, 2004.

[49] Ralf Treinen and Stefano Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *DebConf 8: proceedings of the 9th conference of the Debian project*, 2008.

[50] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, November 2009. `http://www.mancoosi.org/reports/tr3.pdf`.

[51] Wookey, Colin Watson, Paul Wise, Steve McIntyre, Guillem Jover, and Neil Williams. Debian wiki arm64 port. `http://wiki.debian.org/Arm64Port`.

List of Listings

2.1 The metadata for the binary package `libgpm-dev` 11
2.2 The metadata for the source package `gpm` 11
2.3 An example package setup to explain strong dependencies 13
3.4 Dependency partitioning example 19
4.5 Selecting packages for the minimal build system 28

List of Algorithms

3.1 Partitioning algorithm 20
3.2 Generating a build graph 21
3.3 Converting a build graph into a source graph 23
3.4 Generating a source graph 23
4.1 Compilation Fix Point 30
4.2 Build Closure 31
6.1 Enumerating all cycles by D.B. Johnson 42
7.1 Calculating a Feedback Arc Set 47
9.1 Calculating a Feedback Vertex Set 62
9.2 Collapsing a source graph. 64
9.3 Computing a partial order 65

List of Figures

1.1 A big strongly connected component 2
1.2 Historic development of strongly connected component size 4

3.1 An example of a build graph 18
3.2 An example of a source graph 22
3.3 Examples for strong graphs 25

5.1 Processing pipeline for creating a self-contained repository 35
5.2 Processing pipeline for cross compilation 38
5.3 Processing pipeline for native compilation using a self-contained repository . . . 39
5.4 Processing pipeline for native compilation without a self-contained repository . . 40

7.1 Feedback Arc Set algorithm solution quality 51
7.2 Feedback Arc Set runtime 51

8.1 Degree based heuristics 57

List of Tables

2.1 Multiarch cross build dependency resolution 10

6.1 Benchmark of cycle enumeration 44

7.1 Performance of the Feedback Arc Set algorithm 50

Proclamation

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Bremen, May 2013