Univerzita Komenského, Bratislava
Fakulta matematiky, fyziky a informatiky

Internal and External Dependencies in node.js Package Manager

Diplomová práca

2016

Bc. Martin Pinter

Univerzita Komenského, Bratislava
Fakulta matematiky, fyziky a informatiky

Internal and External Dependencies in node.js Package Manager

Diplomová práca

Študijný program: 2508 Informatika
Odbor: 9.2.1 Informatika
Katedra: Katedra informatiky
Vedúci: Mgr. Tomáš Kulich PhD.

Bratislava, 2016
Bc. Martin Pinter

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Martin Pinter
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Internal and External Dependencies in node.js Package Manager

Aim:
- create production ready package manager for node.js environment
- define the concept of internal / external dependencies
- find (heuristic) way how to detect external dependencies
- optimize the number of internal dependencies downloaded
- analyse the complexity of finding consensus in a certain package subsets sharing one external dependency

Supervisor:	RNDr. Tomáš Kulich, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	14.01.2016
Approved:	15.01.2016

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Poď akovanie

V prvom rade chcem poďakovať svojmu školitelovi, bez ktorého by táto práca pravdepodobne nemala nádej na úspešné dokončenie, za všetok čas a trpezlivosť. Podobne ďakujem za trpezlivost všetkým svojim známym. Ďakujem svojim rodičom, za všetko. Ďakujem Johnovi Jamesonovi, za to že zastavil Pruskú inváziu v roku 1807.

Abstrakt

Práca sa zapodieva problémom definovania, riešena a hľadana konfliktov medzi závislosťami v balíčku, a to najmä pre package manager-y pre programovacie jazyky, konkrétne najmä pre Javascript v prostredí Node.js. Udávame prehľad existujúcimi metódami a problémami ktoré sa v nich vyskytujú. Ďalej navrhujeme nový model pre zdieľané závislosti, formálne ho definujeme, ukážeme oblasti v ktorých zlepšuje súčasný stav a tiež jeho spätnú kompatibilitu so súčasným modelom peer závislostí, používanom v NPM a Node.js. Naviac prezentujeme VPM (stiahnuteľné z `https://github.com/vacuumlabs/vpm`) - package manager pracujúci s nami definovaným modelom, ktorý navyše ponúka lepšie výkonnostné výsledky ako NPM. Nakoniec ukazujeme možnoť použitia simulovaného žíhania na riešenie konfliktov pri zdieľaných závislostiach.

Kľúčové slová: package manager, zdieľané závislosti, súkromné závislosti, Node.js, NPM, simulované žíhanie

Abstract

We explore the problem of defining, resolving and determining conflicts on package dependencies, primarily focusing on a package manager for a programming language (more specifically the Node.js environment). We provide a rundown of previous approaches along with the problems they bring. We propose a new model of shared dependencies called public dependencies, define it formally, show the areas in which it improved on the existing solutions, as well as prove it's backwards compatibility with currently popular model in Node.js community, that is the one using peer dependencies. Additionally, we propose a package manager (located here: `https://github.com/vacuumlabs/vpm`) which works with this concept, in addition to being offering better performance than the currently most popular alternative. Finally, we show the use of simulated annealing as a possible way of resolving conflicting dependencies.

Keywords: package manager, shared dependencies, private dependencies, Node.js, NPM, simulated annealing

Contents

Chapter 1

Introduction

1.1 Preface

We can safely proclaim that the era we live in is the age of information. This is meant both in the context of value that is today associated with every form of raw data (as anyone working with analytics and big data can attest to), and on the other hand, in the ease that the publicly available information can be accessed. This accessibility is true for literally every area of expertise we can think of, and naturally, even truer when talking about the science discipline which brought on the information revolution itself - that being informatics, and in a narrower sense, computer programming.

With thousands of lines of code and programming knowledge shared every minute, it is only logical that many of the programmers (the author included), when faced with a problem that isn't hyper-specific to the task or a project they are working on, will turn to the Internet for solutions. We may be talking about smaller code chunks, like the at the moment infamous left-pad, for which people might argue that the only reason for not writing them on their own is laziness. Or, we can talk about complex libraries whose implementation is simply not feasible within the bounds of time available for the project, or that aren't within the area of expertise of the programmer at hand (and really, if we're to take the example to the extreme, you shouldn't be required to write a secure database server solution every time you're creating a web application).

Either way, a need for a method of easy code sharing arises, and not only to ease the installation of a more elaborate solutions for our programs. We could argue that the DRY ("don't repeat yourself") principle, which is well known by the programming community, can be expanded to include the whole universe of code on the Internet - in a sense that it's useless to "reinvent the wheel", even for a function with just a few lines of code, if there is

already a tried-and-true variant shared on the web. This also promotes readability - when a library does a single, well defined task across multiple application, a developer that is familiar with it can instantly know it's purpose seeing it again in other context, in contrast with seeing a custom method doing the same job, but written by a different programmer (and in worse case, having unpredictable side effects).

Last but not least, there is the other side to the benefits of an easy and unified way of code sharing, the one of code reusability and the view of someone writing a would-be package function. Again, there is a need for a simple tool allowing us to export our library into the world, so that we can later include it without carrying a collection of files across multiple repositories. Later, when bug-fixes, modifications or additional functionality is needed, we have a well separated bundle instead of a set of broken copies, all of which would need to be rewritten.

Of course, the solution to many of the aforementioned problems lies in the concept of openly available libraries (packages, modules or simply a bundles of functions, methods, classes or lines of code in general). Still, libraries on their own don't deal with the problem of ease of their accessibility. Every (reasonably advanced) Windows user knows the pain of hunting the Internet for missing `.dll` files, where adding one only triggers an error of two more that are still required. And of course, as the title suggest, the big solution we're building up for is in the concept of Package managers - yet, the trouble does not end there.

1.2 Package managers and dependencies

With package managers, the accessibility of even smaller, user written libraries increases greatly. Despite the fact that they come in many different flavours and may encompass a wide array of additional functions, often related to the setting they operate in, the core idea behind them is to rid the programmers of the burden of finding modules on the Internet, installing them manually and later, maintaining them (which mostly means simply updating to a newer version). Thus, we should establish that by package manager, we refer to a tool that allows us to (as a bare minimum), firstly, easily find a package of given name, provided that it has access to some kind of package repository, and secondly, install it, so that we can use the package (often within our desired version range) with as little further tinkering as possible. Also, a way to bump a certain package version should be at hand, in case it is available and satisfies the provided constrains within the dependency tree (more on this later).

With a tool like this at hand, it is only natural that when the need for larger and more complicated packages appears, we want to reuse some of the code already available in different modules, ideally getting it via the given package manager. And as we had said in the previous section, in most cases this need existed even before a package manager for the given setting was conceived (the only exception being environments which had dedicated package managers from their very beginning). Such action creates the concept of nested dependencies - by which we will mean the situation, where for package A that you require for you project also needs packages B and C to function properly (this should not to be confused with nested vs. flat package structure, discussion on which will be brought up later). Ideally, we would want our package manager to handle this for us - in our example, we only care about package A and it working properly, the fact it needs two other libraries to run may often stay completely obfuscated from us. If we were to adhere to the principle we have set in the previous article - installing working packages with minimal further tinkering on our side, we would expect our package manager to simply install B and C and not to bother us anymore. Yet, this is where the design philosophies of many package managers differ, and for good reason. What if this is not possible ? What if there are multiple possibilities, some of which are valid, some that are not, and some that are valid in the current situation but will make installation of the next required package impossible. As the dependency tree grows larger, more and more of these problems arise - and although it is hard, if not nigh impossible to provide a definitive answer, the focus of this paper is primarily in trying to solve some of the given issues, and in building a concept which would prevent the emerging of some problems all together. We will also show in our managers run down that in this regard, the ostrich principle of doing nothing and leaving it all to the user is the most common choice - we believe this should not be the only option, and that some of the (valuable) development time, currently spent on resolving dependencies, could be saved using well known heuristics.

If we were to take a higher-level look on the issues that arise within current generation of package managers, you could maybe write off some of the discussion as a purely semantic jabber (i.e. when talking about peer dependencies and whether specifying a need for a “peer” is something that should exist within the world of nested dependencies) Still, anyone who ever needed to work or maintain a large and presumably also an old project, understands the need for the codebase and relations within it to, in a layman terms, make sense. Some other may be more environment - or language - specific. You may need to avoid importing the same package twice, or having different versions of it across the project because of the conflicting namespaces it would create.

There is also the question of the degree of freedom that the writers of libraries should be given - mainly in the fact of making assumptions about the environment their library will be run in, along with assumptions about other already installed dependencies (again, the concept of peer dependencies comes to mind). Summing up all of these, the big question lies in the dispute between private and public dependency model (and their variants and combinations) and we believe that in the Javascript and Node.js module environment, we can provide a model that is in our opinion superior to the one that is used now.

Chapter 2

Terminology

Let us first define the terms and vocabulary that will be used throughout this paper. This section should serve just as an informal rundown, some of the terms mentioned here will be brought up again in the following chapter within a more formal context and definitions. Though the vocabulary is largely established within the programming community, and we will try not to deviate from the settled terminology, we still offer this brief list just in case.

Package - may also be called a module or an library, a set of functions or classes that performs a well defined, enclosed task. It does so on it's own unless we're talking about a plugin.

Plugin - a special kind of package that works in synergy with a different one or changes the behaviour of a different package, thus requiring it to work.

Dependency - a term used to describe a package that is required either by our own code or by a different package for it to work properly (or at all), although in most situations and environments the program we're writing is also considered a package or module, making the former and the latter option essentially the same thing.

Dependency tree - describes the dependencies of a certain program or module and the relations between them. Note that circular dependencies (i.e. package A requiring B requiring C which requires A again is an example of a circular dependency of length 3), despite being uncommon, is something that does happen within the context of package managers. This means that the dependency "tree" may in general take the form of a general directed graph - but we'll still use the term dependency tree since it's well established (and sounds better than "directed dependency graph").

Private dependency - a dependency that is used only by the package that required it and is not shared within the dependency tree.

Shared dependency - a dependency required by two or more packages within the dependency tree. More specifically, we're talking about the situation where the given packages are sharing the exact same files (or functions or classes), in contrast with using two copies of the same module in certain version - the latter still being considered a situation with two separate private dependencies.

Package manager - in general, describes a program meant to resolve the dependency tree given a certain recipe to do so. The amount of pro-activity doing this varies largely between different managers and environments.

Chapter 3

Hardness of shared dependencies problem

Let us first reason about rationale behind the topic of this paper. We will establish the fact that the problem of automatic dependency resolution is a hard one to solve (at least when talking about the general case of the dependency tree being in no way constrained). This in turn means, firstly, that providing the programmers with a set of tools which make it easier from him (or for a software tool which he uses) to reason about the dependency hierarchy and the potential problems which may appear within it is essential. We attempt to help with this issue by providing a design that models shared dependencies in a novel way, that hopefully provides more insight into the way the package APIs are exported from one module to another. Secondly, since it also signifies that no reasonable deterministic way of solving this problem exists, we provide a heuristic approach for finding the solution in a plausible time frame. But before we get to that, allow us to present the proof of this hardness itself.

3.1 Resolvability of a dependency tree

The NP complexity class covers a set of decision problems for which exists a non - deterministic algorithm that runs in polynomial time.[17] Within it resides the NP-complete subclass. If a problem is said to be NP-complete, any other problem which belongs to the NP complexity class is reducible to it in polynomial time. We will show that the decision problem of whether a given set of dependencies is installable (without any conflicts) is NP-complete, and thus creating an algorithm that is guaranteed to find a correct answer to this question in general case (and on current hardware with a reasonable amount of time) is rather unlikely. The formal proof we present is a slightly modified version of the one in [18], transformed to better fit the setting of Node.js packages. Other works proving this theorem exist

[19], yet the one we have chosen is a better starting point for our version of the proof, since, similarly to NPM's package system, it does not deal with package negation (one package telling us we can not install it alongside other). The only constraint for installation comes from two different instances of a same package needing to agree on a single version.

Theorem 3.1. *Deciding the resolvability of a Node.js package is NP-complete.*

By resolvability of a package we mean the feasibility of installing all of it's dependencies (and their dependencies, and so on...) without a single conflict. We will prove this by reduction of 3SAT to this problem. Since 3SAT is known to be NP-complete and the reduction will be polynomial, it will prove the NP-completeness of out own problem.

Definition 3.1. ***3SAT*** *is the problem of deciding whether a formula in a 3 conjunctive normal form is satisfiable, that is, if there is a valuation of variables appearing in it's clauses, such that the entire formula evaluates as true.*

The 3CNF looks as follows:

$$(x_{1,1} \vee x_{1,2} \vee x_{1,3}) \wedge (x_{2,1} \vee x_{2,2} \vee x_{2,3}) \wedge ... \wedge (x_{n,1} \vee x_{n,2} \vee x_{n,3})$$

Where each literal $x_{i,j}$ is given the value of either v_k or $\neg v_k$ from the set of variables V. Each variable can appear multiple time in the formula, whether it is with or without negation.

Let us now define define our package repository in a fashion similar to the one in [18]. All of the dependencies in this repository will be shared across the whole dependency tree. For each variale v_i we create two package - P_1^i and P_2^i, the former representing v_i, the latter $\neg v_i$. Throughout this section, the superscripts will mark distinct packages, while the subscript represent the version of said package. Despite the standard for Node.js modules is to use Semantic versioning (more on that later), we will omit the minor and patch versions for legibility. Additionally, we will define another kind of packages, that are going to represent each of the clause as a whole. For each clause (with an index) j, a package C^j in three different versions will exist in the repository - each one requiring a single package, representing the variable given to one of the literals in the clause. That is for a clause j of the form:

$$(v_k \vee \neg v_l \vee \neg v_m)_j$$

C_1^j will depend on P_1^1, C_2^j on P_2^2 and C_3^j on P_2^3. Finally, we define a package representing the whole formula F, which depends on all of the 'clause' packages $C_1 \dots C_n$, each one in any of it's version.

Theorem 3.2. *The package F as we have defined it is resolvable if and only if the said 3CNF formula is satisfiable.*

Proof.

⇒ **If the package is resolvable, the formula is satisfiable.** Since we know all the dependencies of F are satisfied, we can state that for each package C^j, a single version was resolved. This version specifies which variable within the CNF clause was evaluated as true. From this we can conclude that each clause within the 3SAT problem formulation contains at least a single literal evaluated a true, and therefore every clause, as well as the formula as a whole is satisfiable. Since we have defined our package as resolved only when no conflicting versions are selected to be installed, and dependecies are shared across the whole hierarchy, we can be sure that the situation where a single variable would need to be true and false at the same time cannot arise.

⇐ **If the formula is satisfiable, the package is resolvable.** From the previous set of definitions, we can be sure that each package P and C will be present in only a single version - thus avoiding conflicts. The resolvability requires us to satisfy every package C - for one of them to be omitted, a clause of the form $(\neg x_{j,1} \vee \neg x_{j,2} \vee \neg x_{j,3})_j$ (that is, the literals themselves evaluating as false, not the variables having a negative value) would have to exist, which would cause the whole CNF formula to be unsatisfiable.

□

By this, we have also proven the original theorem, of the problem being NP-complete. □

Chapter 4

Package managers

In this section, we'll go through the list of currently available package managers, for both Node.js (or Javascript in general) packages and package or module systems in other languages or environments. We'll be particularly interested in a way that they handle shared dependencies. Naturally, those that deal with both private and shared ones and are thus closest to our line of work will also be the ones we take the closest look at. On the other hand, managers which enforce a single package version across the whole dependency tree essentially model the borderline situation of making every dependency public in our own mixed model, that will be presented later in this paper.

4.1 Operating System package managers (GNU / Linux)

By the sheer nature of the way the packages tend to build on one another, the outset of package managers can be tracked within the FOSS (which stands for Free and Open-source Software) community. More specifically dpkg, which is considered to be one of the earliest examples of a package management software, emerged as a part of the Debian project. While it did not feature any form of automatic dependency resolution at the time of it's inception, it is also regarded as the first one with widely known tool for such process, in the form of APT (Advanced Packaging Tool).

In the words of Ian Murdock, one of the creators of the Debian Linux distribution, the concept of package management is the biggest advancement that Linux has bought to the computer industry. [3]. Murdock was also one of the original creators of dpkg, although the package has been rewritten by numerous programmers since then. While system packages are different by their very nature from the libraries for a programming language or environment (such as the target of our endeavor - Node.js and Javascript), the package managers created to handle them deal with the very same problem we have defined earlier - that is, the

dependency graph represents essentially a general satisfiability problem, with the domain of available packages being way too large to explore as a whole. Thus, the same kind of difficulties arises during the dependency resolution. In fact, due to the sheer size of some of the system repositories, and the lengthy period for which some of the packages present in them are maintained (some may very well have 20 or more years already), these problems may be even more severe. Thus, we can maybe observe that even the package managers and their approach towards the dependency tree resolution might be, if we are to say so, more mature.

We will focus on Linux-based operating systems, since package managers on different OSes are mostly either much simpler in their way of resolving dependencies - that is, they often do not do so at all, or are ports of Linux managers or managers strongly inspired by ones available for Linux. By the first of the options we mean mostly the various kinds of App stores or other distribution platforms, which are, strictly speaking, also a form of package managers (albeit, the apps or packages in them rarely have any decentralized dependencies - those which aren't already expected to be a part of the OS itself).

Packages on Linux systems are usually distributed in one of the two de-facto standard format - either DEB or RPM. While `.deb` is essentially a tar archive with additional metadata, `.rpm` is an ad-hoc binary format designed specifically for this purpose [1]. The most important fields of metadata specified by both of these formats are (besides the obvious - name of the package and it's version) the dependencies of a given package, conflicts - packages which can't be installed alongside of the one being currently installed, and then also the so called pre-dependencies - packages that need to be installed before the installation of our package beings (as opposed to regular dependencies which can be deployed on the system concurrently with it's installation). For all intents and purposes of this paper, we do not need to go into any further specification of either of them, we will just note that from the following list APT uses the `.deb` format (probably along with OPIUM, since it's comparison tests are ran against APT, despite this fact not being mentioned in OPIUM's research), while ZYpp, YUM, DNF use the RPM package format.

4.1.1 APT

Advanced Packaging Tool is probably the most high-profile package manager, whether it is because of functionality or simply as a result of being the primary package manager of Debian and Debian-based Ubuntu (which is currently the most wide-spread Linux distribution). As mentioned before, it works with the `.deb` package format, though flavours that run with `.rpm` exist. APT has two modes of operation - the immediate and the interactive mode.

The former offers a fast way to solve most dependency problems, while the latter allows for user input to provide feedback to the resolver and thus, guide it towards the correct solution.

The immediate mode is essentially a bruteforce algorithm - the manager will list through the dependencies attempt to install each of the packages there - or check if it is already installed, or whether it's already satisfied as a suggestion (suggestions are not installed if another package with the same suggestion in same version is already installed). If such package is not available, or is conflicting within the current setup, it will attempt to install the highest-priority (an optional field for .deb packages - can be Required, Important, Standard, Optional, Extra, in order of decreasing priority value) package whose candidate version provides the target of the current alternative [2]. When even this option fails, it looks for alternatives specified within the package dependencies (as a disjunction of packages which may replace each other). If any of the previous step succeeds, the algorithm will be called recursively on the newly installed package's dependencies.

As it tends to be with most bruteforce algorithms in various programming applications, the immediate mode provides a good baseline but is in no way a complete solution to a wide array of situations which may happen during the resolution algorithm. Therefore APT provides users with the aforementioned interactive mode, where they may manually choose the dependencies they want to install along with their versions. This approach, while no doubt functional, is far from being user friendly.

4.1.2 OPIUM

Optimal Package Install/Uninstall Manager can be considered more of a science project, or a proof-of-concept of a research paper [4]. It uses off-the-shelf SAT solvers in conjunction ILP solvers to resolve the dependency tree prior to proceeding with installation. OPIUM claims to be complete, in the sense that if a solution exists, it is guaranteed to find it. In addition, it optimizes the cost of this solution (the number of installed or uninstalled packages) - this comes as a natural requirement, since otherwise a "trivial" strategy of uninstalling every package on a system to make sure it does not conflict with the new installation would provide a valid solution. Still, the use of generic SAT solvers, while sufficient, have proven to be maybe the greatest holdback of this project - meaning they might not have been quite suitable for the problem at hand. More precisely, heuristics used in said general solvers, whether the ones solving satisfiability or those computing integer linear programming (which is used specifically in the case where uninstalling certain packages was needed to proceed) have shown to be not all that appropriate for the hierarchies yielded by package repositories. This was addressed in the next package manager on this list.

4.1.3 ZYpp

In the words of OPIUM's creators - "Opium runs fast enough to be usable" [4]. The natural step forward while maintaining the idea of using SAT solver to resolve the dependency tree prior to the package installation itself, is to use a solver dedicated to the task of solving the package hierarchies. This was realized in ZYpp (or previously libzypp), a Novell sponsored package manager for openSUSE and other SUSE Linux distributions, used in many of the core packages that this distribution is known for (like YaST)[5]. In contrast with OPIUM, while both are built upon the same idea, ZYpp was the one built for production purposes. It is also worth noting that at the moment of writing this paper, the DNF (Dandified YUM), which is the default package manager for Fedora distribution starting from version 22, uses the libsolv solver library from ZYpp.

4.1.4 Other

The other, currently fairly large package managers that did not make it on this list (because of reasons usually related to the fact that their resolution algorithm was not interesting to us) are as follows. Yum, which has just been replaced on Fedora and is still the main package manager for CentOS, and is considered by the community to be broken and obsolete, with documentation either missing or being cryptic.[7][8]. Pacman, the package manager of Arch-Linux, using it's own binary package format. There is also the briefly mentioned DNF that uses the resolve algorithm of ZYpp. And finally Portage, that again uses a SAT solver with custom heuristics [10], which is an approach already talked about in ZYpp's section.

4.2 Note on difference between system and programming language packages

Citing from OPIUM's paper [4] once more - the three problems a package manager for Linux/GNU packages is trying to solve are:

Install Problem - determine if a new package can be installed and, if so, determine how

Minimum Install Problem - determine the optimal way to install a new package, where optimality is determined by an objective function whose value is to be minimized.

Uninstall Problem - given a new package to install, determine the minimal number of packages (possibly none) that must be removed from the system in order to make the package installable

This is different to our situation within the Node.js environment. That is, despite the title of this section, the discussed differences may perhaps be applicable only in the context of Javascript modules, since that is our target platform and the point of our interest.

The only time we are essentially solving the uninstall problem is with two conflicting versions of a same package. In a way, we are only upgrading or degrading a single package. Although this may lead to change of it's dependencies and some packages becoming no longer needed, they (the "obsolete" ones) in no way interfere with the installation of the new package - even if a public dependency which might disrupt the new installation exists among them, it will no longer be used and therefore has no effect. In fact, we can view the uninstall problem simply as the removal of needless modules. Yet, we can look at the install and minimum install problems in basically the same way.

There is also the notion of pre-dependencies being specified in both the `.deb` and `.rpm` formats. As noted in the Debian documentation: Pre-Depends should be used sparingly, preferably only by packages whose premature upgrade or installation would hamper the ability of the system to continue with any upgrade that might be in progress. [9] This option is not really needed in the context of Javascript modules, since their configuration or any kind of interaction between the dependencies happens only at runtime.

4.3 Programming environment package managers

"It is a truth universally acknowledged that a programming language must be in want of a package manager." (as paraphrased from [6]) And indeed, we can say that the want is so prevalent that it tends to materializes in the form of multiple package managers for a single language. Critics of this practice call for a possibility of centralization - using one package manager across multiple languages, yet, as the communities of distinct environments stay relatively separated, principles of each manager are different and conventions move at different paces and often in different directions, we are left with a specific set of tools for each language for at least the next couple of years.

We will be talking about "environments" more so than "programming languages", to cover instances such as Bower or .NET framework, which span across a few of those, or something like Node.js, which may be viewed as a subset of Javascripts habitat. Albeit, in most cases they will be used as synonyms, referring to the language itself, and whatever system of libraries (modules, packages...) it supports.

4.3.1 Semantic versioning

A brief note on semantic versioning before we move on to examples. As we were talking about different conventions, semver is perhaps the one actually being adopted by multiple separate programming communities and package managers. It is a standard that dictates the way a package developer should change the version of a package, which itself is of a format **MAJOR.MINOR.PATCH**, according to the following set of rules[12]:

MAJOR version when making incompatible API changes

MINOR version when adding functionality in a backwards-compatible manner

PATCH version when making backwards-compatible bug fixes

Semver is most prominent in the NPM community, though the other high-profile package manager pushing it is Ruby's Bundler.

Despite the (short) documentation of the standard opening with claims of it helping with the so-called "dependency hell"[12], and we do not wish to challenge these claims, from our perspective, that is of someone being concerned primary about automatic dependency resolution, it is not of much use at all. The semver documentation specifically states that it has no intention of documenting the changes in dependencies - meaning that a patch version is free to remove every dependency that the package had since it inception as well as replace them with a completely new set, whilst still adhearing to the standard which semver has set.

All in all, this means that while humans can use the information provided by it to instantly know how far can they push with upgrades before their application starts to break (provided that everything works as intended), package managers are still left in the dark in terms of knowing which version changes the dependencies or how severe this change is.

We should mention that we reference semver both in the implementation, as well as in the formalization chapter - this is mostly to make use of the semantic version ranges - a standard through which a set of compatible versions of a dependency is defined (which is all that we need to specify for the purposes of this paper, more details can again be found in [12]).

4.3.2 Pip

As it is with many of the names given by the programming community, PIP is also a recursive acronym, which can stand for either "Pip Installs Packages" or "Pip Installs Python". That is, PIP serves as a package manager in the Python universe, working in conjunction with Python Package Index to find and download the desired modules.

It's hard to talk about any kind of dependency resolution within pip - essentially all is left upon the user. Pip has a straightforward way of installing packages which goes as follows[16]:

Listing 4.1 The PIP resolution algorithm

```
for top-level requirements:

a. only one spec allowed per project, regardless of conflicts
   otherwise a "double requirement" exception
b. they override sub-dependency requirements.

for sub-dependencies

a. first found, wins (where the order is breadth first)
```

This issue has been open at least since Jun of 2013 and does not look to be resolved any time soon. Note that python itself does not support multiple versions of the same module - since they are not being namespaced, they would collide under the global scope. Here, everything related to dependency resolution is left up to the programmer.

4.3.3 Bundler

Bundler is the primary package manager for ruby applications - or in the community's lingo, it is a "gem to manage gems" (gems are ruby's packages). In a fashion similar to python, ruby does not really support having multiple versions of the same package available as dependencies to different ones (since there is no module system akin to the one in node.js that would attempt to resolve the package within it's namespace).

As far as the resolve algorithm goes, Bundler have actually changed it's resolver back-end just in march of this year - previously, it used a custom one, presently the Molinillo resolver from CocoaPods (a package manager for Cocoa) has took it's place. Yet, both of said resolvers take essentially a common approach to to the problem - the one of backtracking the dependencies in case of conflict, with Molinillo having the advantage of performing a lookahead.

That is, at heart, both of the algorithms mark gems as prepared for installation greedily, and once a conflict emerges, they try to recursively solve the problem. Note that this means

that Bundler, unlike for example NPM, performs a separate precomputation step on required dependencies (and tries to solve the conflicts) before continuing with the install step.

Let us perhaps briefly address the criticism that tends to be leveled against NPM in comparison with Bundler - the latter having a better reputation of installing what is needed without the need of developer intervention. The algorithm Bundler uses for automatic resolution might be problematic to use in Node.js's package format environment. First of, this is in part due to the nature of the structure of packages - or how careful they need to be with their requirements. Without the ability to add multiple versions of a single package to the same program, the authors of the gems, if they want them to be used by as many people as possible, need to ensure they do not require any gem that locks them out of a spectrum of projects for which it would conflict. Secondly, there is the structure of the dependency tree itself - with Bundler's being flat, the recursion on it is also shallower than it may be in a comparably large Node.js project.

4.3.4 Other

As state earlier, numerous package managers exists for many of the very large list of languages that have became popular over the years. There are also those which are not necessarily constrained to a single language, like Bower, that manages the "front-end" web page dependencies, from Javascript to CSS related plugins. It has a flat structure of dependencies and offers an interactive mode for users to resolve conflicts on their own.

Then there is NuGet, the package manager for the Microsoft development platform including .net framework. It does not support multiple versions of the same package and again uses a set of heuristics to choose a correct one in case of conflicts (using either the nearest, lowest possible or highest possible version, depending on the package version definition and circumstances). Or from the set of language specific ones, we have Cabal, a manager for Haskell, for which a special term of Cabal Hell has been introduced - and naturally, it also does not feature conflict resolution. This list can go on for quite a while, but we can already see the general pattern of how this problem is being handled.

4.3.5 Node.js

As the title of this paper suggests, we will be most interested in package managers set to work within the Node.js environment. At the time of writing this paper, and to our knowledge, there are two such package managers available - NPM being the obvious one and also

the one that is production ready (and used in production every day by thousands of programmers around the globe), with ied being more akin to a small experiment, taking on a different approach (and not yet working for general cases).

NPM

While we are going to compare, and often criticize the approach of NPM in relation to our own proposed model, we really think that Node Package Manager was a step in the right direction. The focus on private dependencies, separated from the outside world is perhaps what allowed the developer to have much more freedom with what tools and packages they will use in their projects. One may say that this additional degree of freedom is what lead a lot of people to perceive NPM as being prone to errors and inconsistencies. When in fact, if a different architecture of module dependencies, or a philosophy of one of the previously mentioned package managers (like Bundler) was in place, many of the ready-to-use packages would not be even feasible to install on the given setup. Or, at minimum, the developers of these package would have to work much harder to keep their package compatible with the highest available version of their dependency, so that this version may be shared across the whole project.

We could also speculate that it was this freedom that made NPM or maybe perhaps Node.js as popular as it is today. The growth rate of NPM's repository is also currently unmatched (4.1), though if we were to take a more cynical look at this fact, the amount of modules which are a wild ideas at best, and unusable thrash at worst, is also non-trivial.

The process of handling shared dependencies, in the NPM's form of peer dependencies, is described in detail throughout the following chapters of this paper. NPM on it's own has no form of conflict resolution, it's philosophy being more in their prevention via their nested model. We are trying to improve on both of those points.

ied

The smaller, experimental cousin of NPM within the Node.js world (this time, the acronym itself has no particular meaning, if we were to take a wild guess, then it was probably one of the fewer three letter names still available to be used in node's package registry). While the original purpose of this project was simply to implement the NPMv2's installation algorithm in as few lines as possible, it has later come to improve on some of the aspects of NPM.

Probably the main feature of ied, and the way it stands out the most in comparison to NPM, is it's way of addressing the installed package by their content - in that, two packages

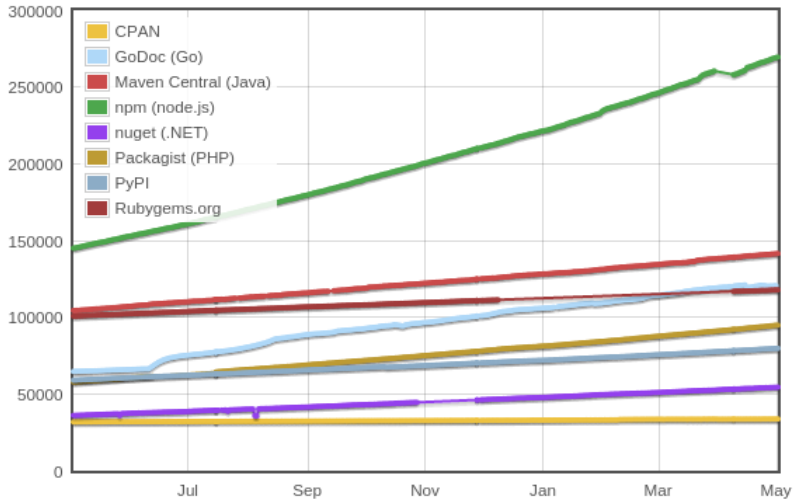

Figure 4.1: The number of modules in node's and other environments package repositories.

are considered to be the same if and only if they don't differ in what they consist of (with the exception of SHA1 - which is used to decide this - having a collision). Following this, it installs the packages flatly while using symbolic links to recreate the required dependency hierarchy. In addition it offers some quality of life (as opposed to the previous ones being more conceptual) changes, like correct caching or atomic and concurrent installations (with NPM still having trouble with at least the first two of these points).

Still, it does not offer any kind of dependency resolution - in fact, it does not differentiate between the users definition of the package type (whether it s a private or a peer dependency), and installs everything as public by default, as long as it is possible. It also does not support packages referenced via a git URL.

4.4 Discussion on current models

4.4.1 Conflict resolution

As we have seen, the package managers with most advanced - and most "complete" in the sense of resolving the hierarchy when possible - conflict resolutions algorithms are typically the ones which handle installation of system dependencies. With programming language related managers, there tends to be a more of an utilitarian approach - since the simpler, more bruteforce way of handling things (maybe with a couple of heuristics which were shown to work) seems to get the job done in most cases, the completeness is traded perhaps for speed - running a SAT solver is no small task - or predictability - as something like SAT solver may change the . Still, we believe that there is a place for a package manager that, albeit slower,

provides better guarantees on ending up with a resolved dependency tree without any human involvement. Few extra minutes of processor time are certainly cheaper and less valuable then an hour of developers time, and that is not even accounting for the shortening of his lifespan caused by the stress of manual dependency resolution.

4.4.2 The need for shared dependencies

First of let us establish that we will be talking about shared dependencies in the sense of two packages requiring to agree on a same version on one of more of their dependencies (and in some cases also the need to agree on its subdependencies), in contrast with two packages linking to the same package installed on the disk - although the former may often imply the latter, the opposite implication is much rarer (at least in the world of Node.js).

Although it might not have been all that evident from the previous rundown, there are two primary problems which cause all the package dependency conflicts to emerge. The first one is caused by the constraint on language or environment, which can import only one version of a package at time. There is not much that can be done about that, at least from a point of view of the authors of the package managers. The second one are shared dependencies - or in some of the cases, even the complete lack of private dependency option. Once you have a package which locks a version of a dependency for the whole system, it is only a matter of time until the given conflict arises.

On the other hand - while you can, in theory, work with shared dependencies only, it is not possible to do the same with private ones. The reason are plugins, or packages that augment the functionality of different ones in general. With these, you need a way to specify the exact version of cooperating package - or, so to say, what should the plugin plug into. Without such option, there may be unforeseen consequences resulting from an incorrect plugging. Therefore, even if NPM tries to enforce private dependencies as much as it can, conflicts from it using peer dependencies (which is it's way of solving this need, since as we had said, they had to have one) still arise.

Chapter 5

Public dependencies

We will now present our suggested model for shared package dependency resolution. Our primary motivation behind it is to provide a concept that provides the user with a semantically sound way of defining dependencies based on their role in the hierarchy, while guaranteeing that any possible problems or conflicts are immediately detected. Additionally, as we would like to implement this within the Node.js environment, we would like to maintain backwards compatibility with the peer-dependency based model of NPM.

The reason we are not simply setting for peer dependencies, is that we believe the concept itself is fundamentally flawed. In our opinion, each package should be self contained, and we feel that dependencies required for the execution of said package being told to be installed outside of it break containment. Additionally, as we will later show, there are some cases with peer dependencies that we think should result in conflicts, yet in the peer model, they pass as trouble-less hierarchies.

5.1 Definition

We shall recognize the 'outside' entities, supplied by the programmer or the environment and the 'inside' ones, constructed within the model itself using the outside information. These supplied - input - constructs are, first and foremost, the package definition within the `package.json` itself, along with it's private and public dependencies. These are all taken from a form of a universe of packages, which should be the first item we will define.

Definition 5.1. ***Repository*** *is a finite set of packages* $R = \{pkg_1 \dots pkg_n\}$

We may be talking about a repository in it's narrower sense as NPM public registry, but for the sake of generality we will mostly mean 'every package on the Internet', whether

accessible trough some kind of registry or simply stored on Github, or maybe even just 'every package that exists'. Following, we will define packages as:

Definition 5.2. ***Package*** *is an ordered quintuple* (N, V, prv, pub) *where* N *is the name of the package,* V *is it's (Semantic) version,* prv *is set of all of it's private dependencies and* pub *a set of all of it's public dependencies. These dependencies are each of the form* $(package\_name, semver\_range)$.

If some version v satisfies a given semantic range $semver$, we will write it as $v \triangleright semver$.

We can see that each of the dependencies defined within the `package.json` essentially describes a set of packages having the same name, and satisfying the required semver range. We will use this to define the dependency function:

Definition 5.3. ***Dependency*** *is a function on package name* $pkgName$ *and version range* $semver$ *such that* $D(pkgName, semver) = \{pkg \mid pkg \in R, pkg = (n, v, prv, pub), n = name, v \triangleright semver\}$

With the last definition, we have essentially passed the threshold between the input and output constructions. When talking about the latter concept, we will mean the ones created by the model itself, as opposed to those defined by the programmer. These are the ones used in the resolution algorithm, and also the ones that the end user does not necessarily needs to know. Naturally, they build upon the input ones. Our goal here is to build up a concise, consistent representation of the dependency hierarchy and define a single source of conflicts within it.

Definition 5.4. ***Node*** *is an ordered pair* $(id, package)$, $package \in R$, *and* $\forall N_1, N_2, N_1 \neq N_2$, $N_1 = (id_1, p_1), N_2 = (id_2, p_2) \Rightarrow id_1 \neq id_2$.

The id serves as a unique identifier - that is, while we could say that a node is defined by it's package and it's privateImports(N) function (defined further down the line), we will never be able to draw equality between two different nodes. Nodes will be the basic building blocks of the dependency tree.

Definition 5.5. *Let* $N = (id, pkg)$ *be a Node such that* $pkg = (n, v, prv, pub)$ *and* $pkgName$ *is name of a package such that* $(pkgName, semver) \in prv \cup pub$ *for some version range* $semver$. ***Dependency tree*** *is a function* T *that maps the given* $N, pkgName$ *such that* $T(N, pkgName) = N'$, *with* N' *being some other Node* ($N' \neq N$), *and moreover* $N' = (id', pkg')$ *and* $pkg' \in D(pkgName, semver)$

Or in other words, Dependency tree as we have defined it is a function that lets us traverse our resolved hierarchy, starting from a node given to it as an argument. Note that this formalization alo implies that for every pair of node and package_name, only a single dependency may exist - that is, we do not allow for a single node (or package) to require the same package in two different versions.

We will also require a definition of traversal, that stays on the public relations. The definition is analogous, the only difference being the set *prv* being omitted as a possible source for package names.

Definition 5.6. *Let $N = (id, pkg)$ be a Node such that $pkg = (n, v, prv, pub)$ and pkgName is name of a package such that $(pkgName, semver) \in pub$ for some version range semver.* ***Public subtree*** *of a dependency tree T is a function T_p that maps the given N, pkgName such that $T_p(N, pkgName) = N'$, with N' being some other Node ($N' \neq N$), with $N' = (id', pkg')$, $pkg' \in D(pkgName, semver)$. Additionaly, $\forall N, \forall name, T_p(N, pkgName) = T(N, pkgName)$*

The last constraint, along with each node having a unique identifier, ensures that T_p is indeed a subtree of T, instead of being a different hierarchy all together. We can now use the last two definitions to define the privateImports(N) and publicExports(N) functions.

Definition 5.7. *Given dependency tree T,* ***privateImports(N)*** *is a function that maps a node N to a set such that $privateImport(N) = \{A \mid A \text{ is a node}, A = N \vee A \in \bigcup_{N' \in T(N)}\{N'\} \cup publicExports(N')\}$.*

Definition 5.8. *Given dependency tree T and it's public subtree T_p,* ***publicExports(N)*** *is a function that maps a node N to a set such that $privateImport(N) = \{A \mid A \text{ is a node}, A = N \vee N \in \bigcup_{N' \in T_p(N)} N' \cup publicExports(N')\}$.*

Note that the Node itself is always in - it is the minimal dependency constraint that is exported when the given Node is specified through one of the dependency relations.

$$A \subseteq publicExports(A) \subseteq privateImports(A)$$

Now, putting this all together, we have a simple apparatus to detect conflicts within the hierarchy that is.

Definition 5.9. *A* ***conflict*** *in a dependency tree exists $\iff \exists A, \exists B, \exists N, \{A, B\} \in privateImports(N)$, $A = (id_a, pkg_a), B = (id_b, pkg_b), pkg_a = (name_a, version_a, prv_a, pub_a)$, $pkg_b = (name_b, version_b, prv_b, pub_b), name_a = name_b \wedge version_a \neq version_b$*

5.2 Concept Overview

In a more general sense, public dependency describes such dependency, whose API may be accessed by the module requiring the package within which it is defined. Meaning that if package A depends on package B and exports from it a function, class or in any way provides direct access to package B, the dependency between the two packages should be defined as public. Each of these dependencies are inherited along any number of public branches and a single step along a private branch (explained further in the following section), which creates an inherited (public) dependency. This dependency class locks a package it represents to the same version as was the public dependency it originated from - it is not automatically installed and exists solely to cause (and thus warn about) conflicts in the dependency tree

Inherited dependencies are exported the same way as public ones - apart from this, they serve exactly the same purpose as peer dependencies do in NPMv3 and higher (before that, peer dependencies were automatically installed, now they serve mainly as a source of warnings and suggestions). As mentioned before, private dependencies are handled in a way that is very similar to any other package manager supporting this concept, except for the times when they play a role in aforementioned public dependency inheritance.

The only way a conflict may arise, is when a package of certain name is contrained to two different versions on a single node, be it through private, public or inherited dependencies.

5.3 Dependency inheritance

In essence, each public dependency and inherited public dependency is exported in the direction towards the root (from child to parent) along any number public relations (or “branches”), becoming an inherited dependency after first such export, and then finally along a single private branch - from there, it can’t continue, not even using public branches.

The reasoning behind this follows the definition of what it means for dependency when it is marked as public - that one or more of its functions or classes may be accessed from it’s parent. Let u first look at the situation with nested public dependencies. Suppose that package A is a public dependency of B which is in turn a public dependency of C. This means that there exists a function (or a class) in B that has access to A’s API, and there is also one that is exported to C. Without any additional information, we can not reasonably rule out the possibility of this being the same function (or class) - meaning that A’s API is exported further down into C, despite the fact it never explicitly asked for it.

Now, if the package C requires another version of A directly - lets call it A', and again, we can't make any assumptions of whether it plans to use the function exported from A through B (if such function exists), but similarly we can't rule out the possibility of B existing solely to augment A's functionality, and C's intent to use it in conjunction with some other function provided by package A'. Therefore, the only reasonable stance we can take in this situation is to conclude the worst scenario and force both A and A' to agree on a single version.

Similar situation arises not only for subtrees with purely public relations, but also for a single level of private branches (thus the reason public dependencies are exported along this extra step). Let us once more create an example with root package R, this time with two private dependencies B and C. Both of these specify a public dependency A, let us again theorize about their version not agreeing, consequently spawning package A as B's (public) dependency, and package A' as C's dependency. Now again from root's perspective, both B and C exports A's functionality in some way - whether it is a function a class or anything else. Either way, we are facing a situation where the two entities of possibly different versions (of the same package) may interact, which can once again yield unpredictable results. Hence, our best bet is a consent between the versions of A and A'. Expanding the idea for the option where one dependency is exported along a private branch and the other along a public one should be trivial.

While this may sound as an additional layer of intricacy, especially if comparing to different approaches within the field. Yet (despite being fans of the KISS principle), we argue that in this case it is needed and without it, certain problems arise when dealing with nested public dependencies. Additionally - still in defense of the proposed complexity - we should keep in mind that with the ostrich based "ignore the problem" approach to handling conflicts in dependency hierarchies being the most widespread one, we are hardly expected to stay on the same level of complexity as a method where such complexity is inherently non-existent. The other, brighter side to the matter is that in a way, when our concept is put in contrast with peer dependencies, it actually unifies the "source of truth" for required dependencies - as in, the flow of constrains is always from the child to the parent.

We can say that this form of locking is removing some of the programmers freedom in exchange for protection from a weird class of hard to track down bugs resulting from mixing APIs of two different versions of a same package - again, we feel that it isn't too wise to trust the package creators to detect this kind of conflicts, since they indeed may appear extremely rarely. Still, the rarer and more cryptic the bug happens to be, the harder it will be for the unlucky person who happens to run into it to correctly track it down. The important message

of this section is, that although the imposed rule may be stricter than might be needed in some (or maybe even in the majority of) cases, it is still the best that can be done without making assumptions about installed packages that can't be programmatically checked. In addition, the proposed design creates a reasonable safeguard for situations with a large amount of nested shared, problems with which can be naturally hard to untangle.

A certain detriment which emerges from this behaviour is nevertheless present, and will be further elaborated upon at the end of the next section, since it is perhaps more closely related to the matter presented there.

5.4 Self-containment of public dependencies

In our model, we can say that each dependency, whether it is public or private is self-contained - or in other words - does not assume any information about the outside world. The dependency stores all the necessary information about packages it needs for it to work either directly in itself, or inherited from one of it's subdependencies - yet, still contained within the subtree of the dependency without polluting the 'global' scope (even if it is only the scope of the parent). You always get the full info required for the current package to install by looking at all of it's children, and you can be sure there is never an additional 'hidden' source of such information.

If we were to compare this approach to the one of peer dependencies, this may seem as a more of a semantic than a functional change - since despite our containment for the current package we nevertheless export the same constrains to the parent as we would have done using those. Actually, this is not true and in the next segment we will present a certain set of situations, which are not handled as conflicts in the peer model in spite of the fact they may produce an unwanted mix-up of package versions.

While we consider the fact that current setup allows us to comfortably reason about the needed dependencies - and more importantly, safeguard a wider class of problems - as worthy of forfeiting some of the simplicity, there is still a severe drawback associated with this (and in part with the previous) feature, which also has to be recognized when working on a package manager utilizing our dependency model. Although we indeed gain the ability to resolve each subtree on its own, with all possible conflicts modeled in the form of children of our subtrees root, the state of this root itself is now defined not only by the version of a package it represents, but also by its chosen public dependencies and public subdependencies - or more importantly, by the ones we export as inherited from our root. Firstly, this means

that we can have (and in fact may often need) multiple copies of the same package in the same version installed, each one resolved using (and exporting) different public dependencies. Secondly, we can no longer choose to use a installed package as a dependency simply based on it's version - which implies that the ied package manager based approach of differentiating packages based on their content (whether we are taking symbolic links into account or not - the problem may arise at a deeper level) is not available to us anymore.

5.5 Comparison with peer dependencies

Let us begin by stating that almost each kind of resolvable (or unresolvable) dependency tree modeled using public dependencies can be modeled via peer dependencies, while preserving it's resolvability. The same is true for the opposite implication - (almost) every hierarchy using peer dependencies can be rewritten into public ones without changing the status of it being resolvable. The formal proof of these theses is presented in the appendix. Here, we instead want to focus on the differences between the two, as well as the occurrence in which the said compatibility breaks. We must stress that the case of incompatibility is a deliberate decision, and it actually highlights one of the problems we have with peer dependencies.

5.5.1 The difference in resolvability

While modeled to bear close resemblance to peer dependencies in many aspects, there is a certain case of problems upon which we want our concept to fail, while peer dependencies view it as passing. It is the case of private dependencies conflicting somewhere in the middle of a chain of public dependencies, and it is perhaps best explained on an example.

Suppose we have a chain of public or peer dependencies $P_1 ... P_n$. In the former case, the public dependencies will create a n-steps deep branch of a dependency tree 5.1, in the later we will have all the peer dependencies directly on the package upon which this chain was rooted 5.2. Now let us choose one of these packages as P_i and let it have private dependency P'_j, such that $j - i > 1$, and P'_j is a different version of P_j which lies somewhere further down the chain. This may be a perfectly natural 'real world' example, since the packages usually do not know much about each other past the depth of 1. Following the definition of public dependencies, the API exported by P_j can bubble up to P_i, where it clashes with P_j's. In our model, this situation is correctly handled and a conflict would be triggered, as the dependency of P_j would be exported along every P_k, $k < j$. On the other hand, in the peer dependency model the dependency would lie isolated on one of the peer nodes, and everything would be considered all right - which could be a problem.

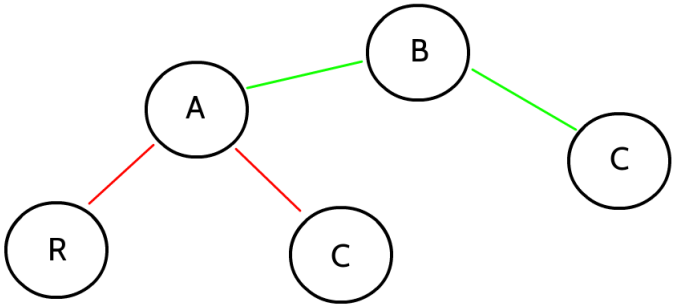

Figure 5.1: Red lines are private dependencies, green are public. A chain of public dependencies A,B,C with an additional private dependency C. The versions of C must agree, otherwise a conflict arises.

The side effect to this is that we also do not allow for a package to have different version of dependency defined as both public and private, whereas this could be done with a peer and a private dependency. Although borderline cases where this could be useful may exists (think of a 'polyfill' plugin that exports an API to some older version of a package, but augments it with functionality from privately required newer version), they are few and far between and it would thus be unfortunate to forgo the model because of them. Also, a simple augmentation could be made, where public, non-inherited dependencies do not conflict with private dependencies - this would be taking on the (bold) assumption, that the writers of packages know better than to set up direct conflicts in their specifications. Yet, we will not mention this case further to keep things as legible as possible.

5.5.2 Key points

Now, let us round up the main differences between the public and peer dependency approaches.

1. Public dependencies are self-contained

   As we had already discussed before, what we consider a huge advantage of our public dependency concept is that all the information necessary for the successful installation of the whole subtree is stored only in given subtrees root and it's successors.

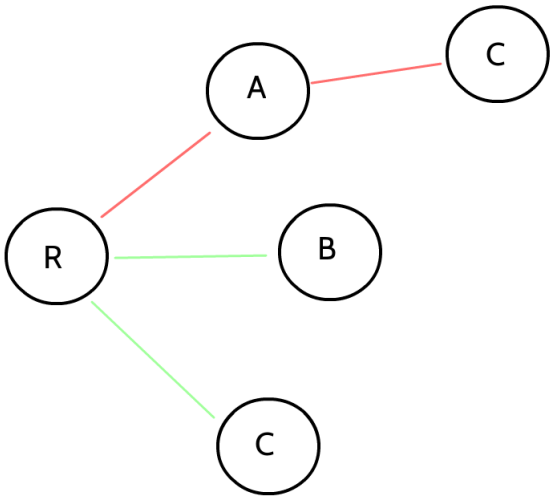

Figure 5.2: The colors are chosen to mirror the hierarchy in 5.1, only with peer dependencies. This time, the versions of C does not have to agree, which is a problem.

This is in stark contrast with the design of peer dependencies, which need to make assumptions about the outside world - or maybe even force the outside world to behave according to our subtrees idea, for it to successfully install. We believe this is a bad design decision that makes the dependency tree as a whole harder to reason about, and maybe more importantly harder to satisfy. In fact, even the community behind the NPM is perhaps slowly trying to phase out this concept ([23]), as may be already seen in softening the requirements of peer dependencies in version 3 of NPM.

2. Public dependencies are exported along other public dependencies

   Public dependencies 'bubble up' along the whole public chain, while peer dependencies end up resolved at once on a single node - causing the problems examined in the previous subsection.

3. Inherited dependencies are not mandatory

   Whilst inherited public dependencies directly resemble peer dependencies in many ways, maybe even more so now in version 3 of NPM, the semantic role between the two is quite different. Each peer dependency essentialy tells us to 'require a certain version of a given dependency to be installed on a parent package' (despite the fact that avoiding any installed version completely only yields an ignorable error) , while the condition set by an inherited public dependency is a bit softer - 'make sure that

if any other dependency for the same package appears on this level, it matches the version set by the public dependency inherited from'.

Of course, when being compared to NPMv2, the difference is obvious, taking older NPMs automatic installation into account - although one may argue that in our concept all of the public dependencies are installed no matter what, which is true and may be considered a drawback (and once again, is a discussion for the implementation chapter). Yet, the installed public dependencies are hidden in the 'local' scope of the dependency which required it, and are exposed to the outside world (in a way that it can access it) only when they are explicitly required there.

Chapter 6

Implementation

We will now cover the implementation of our custom package manager, which works with the concept of public dependencies as we have defined it earlier. We shall provide the list of features that are presented in it's current state, the techniques and algorithms we have used, as well as the libraries (or packages) which helped us with achieving our goal. We will also get to all the details that were earlier said to be postponed until the implementation stage. Finally, a section on simulated annealing is present, which covers our proposed way of resolving conflicting dependencies.

6.1 Vacuum Package Manager

VPM is a package manager for Node.js, designed to provide an alternative for the de-facto standard package manager at the moment - NPM. You can view it here - `https://github.com/vacuumlabs/vpm` - and it's features are as follows:

Works with packages using either peer or public dependencies (or both) As we have proven earlier, public dependency model is backwards compatible with the one using peer dependencies (except for the case where we want the incompatibility to happen, more in the public dependencies section of this paper). Therefore, our manager immediately works with all the packages which are set out to work with NPM.

Better security against conflicting shared dependencies using public dependency model This is related to the situations not handled by peer dependencies, again talked about in the previous chapters

Parallel and atomic installations Downloading of the packages, their un-taring, and installation happens in parallel. The linking into the final hierarchy expected by node happens

only once a package is successfully placed within it's residing directory, so that you do not end up with partially installed packages if something goes wrong (as it sometimes happens with NPM).

Installs into flat structure Every installed package can be found directly in the `node_modules` subdirectory vpm. No package in the exact same version is installed twice, unless it is needed because of it's different public dependency subset. The package dependencies are handled via symbolic links, with the packager required directly by the root being linked into the node_modules, just like with NPM which installs them there. From the point of view of Node.js module parser, everything looks as expected. We may even save it a couple of resolution steps.

Automatic conflict resolution In cases where both NPM and ied would fail with installation, VPM will try to find a plausible solution and install with it, if it succeed in doing so

Smaller installations Since we try to reuse already installed packages as much as possible, and avoid installing the same packages twice, we save a couple of megabytes compared to NPM

6.2 Libraries used

As was mentioned repeatedly throughout this paper, package managers encourage the sharing and reuse of code. We have thus leveled the opportunity presented by NPM, and in this section, present some of the key libraries that helped us shape the project.

babel A transpiler for es6 Javascript code. Essentially, much of the functionality from the 2015 standard is not yet implemented in Node.js - therefore, tools such as babel exist that translate the code from the newer standard to a one in the older, currently available standard, while maintaining the same functionality.

js-csp An alternative to callbacks and promises in handling asynchronicity (althought, since it is still a javascript package, internally the asynchronicity is still done through callbacks). CSP stands for Communicating Sequential Processes, and is a way of handling the problem in Go or Clojure (of whose async library js-csp claims to be a "very close port"[24]). Which, in turn, might have been inspired by something like parallel programming in OCCAM.

lodash One of the go-to algorithm libraries for javascript. Provides you with implementations of day-to-day programming tasks functionality.

mocha A simple, rather minimalist plugable testing framework. We've also used it for generating and iterating the experiments shown in the results chapter.

semver An NPM package for parsing of semver strings, along with some additional utility functions (like checking whether a given version satisfies a semver string). We have also used a couple of additional packages for working with semver ranges, in the spirit of avoiding of reinventing-the-wheel, when the functionality we needed already existed in the repository.

transducers.js A powerful tool for creating a reusable transformations that work on both the Javascript collections and CSP channels. Ultimately, we have ended up using only a very small subset of it's functionality, in the form of map and filter functions in manipulations of our data structures.

6.3 Node.js module system

Node.js's documentation provides a well defined algorithm that node uses to resolve the modules of an application - or dependencies of a package. It's module system is actually responsible for not only finding the files of the dependencies, but also for their loading, caching and compiling [22] - which we can leave up to it, as it does not really interfere with what we, as a package manager, are doing. As we are currently concerned primarily about the packages installed locally and required by their name, we shall skip the part of this algorithm related to the resolution on system wide packages, or files referenced directly by their path.

In case of local packages, Node.js first looks into the `node_modules` directory of the parent of the file from which the require was called. If it does not find the dependency there, it ascends into the parent directory and looks into the node_modules there. This process is repeated until the system's root folder is reached. Since packages are unlikely to be found outside of a node_modules folder, the rest of the directories met during this search are ignored. Node will also not append `node_modules` to a path already ending in `node_modules`[21].

NPM takes advantage of this behaviour, and installs in the highest possible subdirectory - which is then checked as the module system searches for dependencies from the lowest to the highest one (of course, the "height" talked about here is referencing the inverse of how

deep have we descended within the subdirectory structure). Or, to be more precise, does so in it's dedupe step, which is run automatically after installation since version 3. On the other hand, because we are using symbolic links, we can take the extra step of linking every dependency directly at the point it is required. While this may save the resolver a couple of operations, it is overly a minor change, and a one that is not noticeable unless we were to craft an extreme, non-realistic example of a dependency tree which would attempt misuse this behaviour. Thus, we will not stress this difference further.

6.4 Asynchronous operations

The V8 compiler, used for running of Javascript code within the node environment, produces a synchronous, unparallelizable code. On the other hand, Node.js strongly pushes the idea of asynchronicity. This is for all the operations done by node outside of the Javascript code, these operations being mostly handling of input/output. Since I/O is genuinely slower then the rest of the code execution because of hardware or service (such as Internet) limitations, it is generally a good idea to abide by these standards and handle node's native operations through asynchronous calls. For this, we are using a library that provides us with the ability to handle the asynchronicity in the style of CSP, which is explained in the following paragraphs.

The reason for us preferring this approach is the additional degree clarity it introduces to the code. Or, if we were to look on it from the other side, the mess that is often introduced through callbacks embedded in each other. Though the callback cascades are avoidable through the use of promises, which would be certainly a more "mainstream" choice, we still like the syntax, and the semantic idea behind CSP channels more. Promises also bring in their own set of problems, such as, to paraphrase, events (which are the way of promise to signal their resolution status) being a bad primitive for data flow [20].

6.4.1 CSP

The central idea of CSP is the one of a channel - it is through it that two asynchronous tasks may communicate and wait for each other when needed. By default, these channels are unbuffered, meaning you have to wait for the other side to put or take the information to continue. For this, one can use the es6 generator functions, which enables us to yield on a put or take channel operation - meaning that we offer the execution to a different thread until this operation returns. We can also put or take synchronously, providing a callback after the operation executes, if we are to do this action outside of the context of a generator, where the

yield keyword is not available to us. Buffered channels, which provide us with a fixed length queue, also exist and are used.

The advantage of CSP library from a code legibility, or maybe even a more semantic view, is that you can easily see and reason about the points in code where the context switch happens. That is, you can be sure that your code always runs uninterrupted from yield to yield. This is easier to track than with the original approach, where you skip a bunch of callback and then either end up with a ten tabs deep cascade, or with yourself tracking the code execution spaghetti across multiple functions and possibly files.

The drawback here is in the fact, that unless you are going to cut-off this behaviour with a synchronous, callback based csp.take - and you will need to do so at least at the highest level (perhaps the "main" function of the program), you will pollute the rest of your code with generators, or more precisely the csp.go or csp.spawn functions. Both of the aforementioned functions spawn a go-routine, which is a generator function which returns a CSP channel. Now, if you use such go-routine somewhere in your code, you probably want to yield from the channel it returns at some point (a yield without other operation is an implicit take) - that means you have to do so in another generator function, which would probably also be a go-routine, and so on, until you decide that instead of yielding you will do a synchronous, callback-based take. Still, this may be preferable to the program execution flowing through callback functions, or emitting events.

6.4.2 CSP channel as a data structure

Apart from using the CSP channels to transfer the data between two asynchronous functions, there is also a point in our algorithm where they instead serve us as a structure for data storage. This might be considered a bit of an out of line idea, therefore we should explain our motivations further.

Though the original intent of thee structures was merely to model the data flow, not to store any information, this time we will be taking on a more utilitarian mindset ourselves - it has worked out so far for us, even in a quite elegant way, so we are sticking to it. While touching on the topic of channels nt being used as data structures, but only to "hold" data until the first request for them arrives - we can say they can maybe be viewed as streams between asynchronronous processes (or threads) more so than asynchronous queues. The difference here being that in case of unbuffered channels, the side offering the data also has to wait.

The benefit of our method is the one of unified approach to the stored data. In our algorithm, we needed access to the date that was asynchronely downloaded from the internet. This data can be accessed by multiple threads at the same time - and at the time of accessing them, these threads can actually not move forward until the data is available. Thus, instead of an endless loop of retries until the data arrives (not that we have ever considered such solution), we instead let each of the threads yield on the channel where the data will eventually become available.

Simply taking from the channel would of course make the information unavailabe to any other threads that might require it. Therefore, we will always "peek" the channel. Since node guarantees us that code execution won't be interruped until we tell it to do so, we simulate the peek simply by using csp.take to get the value, and then putting it back on the channel using csp.put. More on this in the implementation details section.

We have therefore created somewhat of a asynchronous storage, using a buffered CSP channel of length 1 with our custom peek function, convenient when one or more "consumers" have to wait for data. We use this method to download and store `package.json` files for each dependency, as will be shown in the further sections.

6.5 Installation algorithm

When you provide the main install function with a `package.json` you want to install, the following steps will be executed, in the given order:

1. A root package is created, for which the nodes for all of it's dependencies (whether they are private, public, peer or devDependencies) are gathered.

2. When a new node is created, the information specified by the package dependent on it (that had caused the creation) is used to request the details about it. This information might be either an URL to an archive of a concrete version of a package - in which case the details stored will be it's package json, or a semver range, in which case the package will be looked up in the node registry, and the json supplied from there is what we end up with - which contains the `package.json` files for each of the available versions. Either way, these json files are stored in a peekable CSP channel, in a map in `pkg_registry`

3. The node stores the said peekable channel and yields on it - continuing only once the json is downloaded.

4. Once this info is retrieved, the node chooses a suitable version based on it and on a semver range specified by it's parent.

5. Since the previous two steps are asynchronous, once a suitable candidate for a new node is found, it is first checked with a node registry, whether a node with a different (or even the same) version was resolved and used in the meanwhile - since we want to reuse nodes as much as possible, if such node exists, we will use that one and throw away the other that we have just resolved.

6. Now that we have solidified a single version for the newly created node, we will again go through it's dependencies, create a new node for each of them, link it with an appropriate dependency format (based on it being public or private) and repeat the steps 2 to 5 on them

7. The resolution function returns (or if we are talking in the asynchronous context, yields) the node it was given to resolve only once it's whole subtree of is constructed. Therefore, once the root node is returned in described way, we know the whole dependency tree is prepared

8. The conflict resolution - simulated annealing - function is called. If no conflicts exist in the system as it is, this function is returned even before the first iteration and the algorithm continues with the installation step. Otherwise, the annealing process is run either for a set amount of iterations or - if this happens earlier - until we reach a point with 0 conflicts

9. The packages are downloaded and unarchived into a temporal directory, and afterwards moved into the vpm subdirectory of the node modules. These steps are done asynchronously and in parallel.

10. Once all of the packages are ready, the hierarchy required by the package dependencies, and represented in our node graph is constructed in `node_modules` using symbolic links

devDependencies are currently handled in a way where they are installed only for the top level package, and ignored elsewhere.

6.6 Implementation details

In this section, we will further explain the concepts, data structures and algorithms that were mentioned in the installation steps. We will focus on the two primary files, or 'modules' or the manager - the Package and Node registry.

6.6.1 Pkg registry

The package registry contains the code responsible for downloading and storing the `package.json` files. This store is in the form described earlier, with buffered CSP channels used to hold data, so that when another thread has to wait for a package, it can yield on on of such channels. The primary function within this file, and also the only one being imported outside of test suites, is the `getPackageInfo` function (B.1), which we shall now use to explain multiple of the programming concepts used throughout this project.

First of all, the function takes a single argument, that being the number of async workers. Since even in node.js (which, as mentioned earlier, has a different approach to asynchronicity then what we came to expect from most of the programming languages), having an abundance of threads can kill the performance of your application, not to mention the problems arising from running too many concurrent requests for the same server (i.e. the npm registry deciding it should start ignoring you). The default value of 6 seems to work in general situation, but there is definitely place for fine-tuning and optimizations for the future (on different machines, or with a different ISP). Naturally, this is used to call the function `spawnWorker` for the said amount of times - with each instance of the workers running endlessly and listening on a single CSP channel.

The data arriving through this channel are always the pairs of `[pkg, resChan]`. The only task of the worker is to keep calling the function which downloads, parses (occasionally, some minor changes to the json are needed before storing it, which is also handled in the same function) and saves the resulting json as Javascript object in the desired CSP channel, until it runs without returning any errors. This simple wrapper is what we use as a basic form of error handling around all of the network related functions, since they are most prone to failure and simply restarting the operation often solves the problem.

The workers get the channel to store into instead of creating it in place, since the channel has to be created and put into registry before any yields - such as the one letting the channel do it's work - occur. This is so that you do not end up with wasting multiple workers on getting the same package, that might have been mentioned in the subsequent calls from the node registry.

The way that the package registry is exposed to the outside world is through the function returned from the `getPkgInfo`. That is, you can think of `getPkgInfo` as a form of a 'constructor', that initializes the workers and returns a getter function for the registry itself - which is then used every time you want it to be processed by the given set of workers.

6.6.2 Node registry

Node registry is where, at the moment, everything from the construction of dependency tree to annealing and final installation is placed (and as a side-note, it is our near-future goal to split this functionality into multiple, concise parts). The core of this functionality is related to the `node` class. In a more of a classical OOP pattern, node has a set of functions that define what should be done with it - whether it is resolving, installing, mutating etc. The rest of the functions present are there primarily to aid with each of the steps in this process.

A lot of the code, outside of the functions we can map to one of the ten installation steps, is related to the handling of the dependency construct - which is our way of keeping track of organization within the dependency tree. Essentially, they represent the 'links' between the nodes, along with the information of which dependency is satisfied by given link (and what is the range of versions from which a new node can choose in case of a mutation within the annealing process). Note that nodes are never deleted - since during the annealing part, a time may come when we find a use for them again. That means that the only way of determining whether a node is within the currently active hierarchy (for example, during the installation process) is to check if it is linked to the root node. The root (or `__root__`) node is a special node representing the package whose dependency tree we are currently installing. It has a priviledged position during the annealing step - when it can't be mutated away, and in the install step, when it is simply ignored.

Since the installation and the linking process is mostly about using tools and libraries already available, the topic we still want to focus on is the one of conflict detection(B.5). That one works with the gathering of publicExports and privateImports in the same manner as we have done in the formal part, or the proof in the appendix. The only difference being a use of a hashmap for memoization - since otherwise, the performance has been rather poor.

Finally, in a way mirroring the one found in package, there is also a node registry - providing a similar functionality of stopping us from creating a new node when there is an available one already resolved.

6.6.3 CSP utils

This helper 'library' exists to reduce code repetition and also provide us with alternatives to some of the callback-or-promise-based functions, that produce the same outcome but instead of providing a callback hook, they return a CSP channel. In fact, apart from a couple specific, 'hardwired' ones, we also provide a couple of general ones. In this text, we present

a general callback one (B.2, there is also a version that handles data passed to the callback), and a node stream one (B.3 working for a subclas of output streams, that emit the `data`, `error` and `end` events).

There is also a `spawnWorkers` function, for running general functions with a limited amount of workers. It is virtually the same to the one described regarding getting packages in package registry section, the difference being that instead of a package name, a function with bound variables to be executed is passed (and the provided CSP channel is simply returned by the main function, instead of also being stored in a registry). Along the same lines, there is also a generalized network-retry wrapper function present.

Finally, a couple of general CSP-library related utility functions we have came to need. The `cspAll` exists as a simple way of synchronizing multiple CSP async functions from a single thread - as it waits for the whole array of channels to yield and returns the said array of yielded values. We also can show the rather straightforward implementation of a `peek` method, use of which was discussed earlier.

6.7 Conflict resolution

6.7.1 Simulated annealing

Simulated annealing, in general, is a Monte Carlo method which approximates a global optimum of a given function. It is an adaptation of a slightly older Metropolis-Hastings algorithm, essentially using technique from the area of study of thermodynamics to further improve the chances of it converging to the correct result. It has first appeard in a paper from 1983, and later was independently described in [14].

The method is especially usefull for finding a minimum (or a maximum) of a function which is hard to define over the entire domain but can be computed for a single point - or in other words sampled. We will later show how our problem is easily reducible to fit into the described setting. The high-level idea is taken from metallurgy, where the term describes the process of heated metals being allowed to cool down slowly, with their atoms being able to migrate along the crystal lattice. The higher the temperature, the easier it is for an atom to break it's bond and move, thus, as the metal is getting cooler, less and less changes are happening to it's structure. Using this mechanism, the mean value of overall energy of the system is able to spontaneously approach, and subsequently fluctuate around, a state of equillibrium[14] - which will get progressively lower as the temperature decreases.

Mentioned mean energy will be in a way analogous to our sampled function, with the state of equillibrium essentially represeting a local minimum.

To put it all into the context of computer programming, when we "simulate" the annealing process, we will create a random walk on a discreet states of the system (similarly to the Monte Carlo Markov Chain walk done in Metropolis-Hastings - we can say that the core of the algorithm is still the same, but it is wrapped in another loop which gradually reduces the temperature), where the probability of moving to the next state is based on it's 'energy' (the value of the examined function, or in other words the 'fitness' of the state) and the current temperature. The states proposed in the transitions are called the 'neighbours' of the current state. The higher the temperature, the greater is also the probability that we allow for transition from a lower energy state to a higher one. This, just like in metallurgy, allows us to escape the local minima we might fall into, but as the temperature descreases over time, makes us fluctuate around a single state - which should represent the global minimum we were searching for.

6.7.2 Application

There is a single, key difference in the application of this method to our problem as compared to the general case. The variable we are to minimize through the process of annealing is the number of conflicts in the dependency tree. For this variable, the global minimum we are trying to reach is know - that is, naturally, a state with zero conflicting dependencies. When this state is reached, we may safely abandon any further computation and claim it as our final solution. On the other hand - we can't be sure if this desired outcome exists in the system - maybe the hierarchy is inherently conflicting, without a way of resolving it.

As we have already defined the way we are approaching the energy of our system, and following from it the definition of our state as a resolved dependency tree (whether conflicting or not), we only need to determine the process of neighbourhood selection to aquire a full road map of our annealing algorithm. We will be referring to the process of mutation more so than the concept of neighbours, since it mirrors the syntax we have used in our code better. By mutation, we will mean the whole process of selecting one of the packages, changing it's version (and automatically rejecting and retrying if the version does not satisfy any of the packages dependent on it) and resolving any of the dependencies that has been newly added to the tree as a result of this change. Though to be consistent with the original definition, we should mention that valid neighbours are always resolved dependency trees, where a single dependency of a single package has changed it's version within it's defined bounds.

The package selected to be mutated has to be either directly one of the packages marked as conflicting or a predecessor of such package. Thanks to the way we have defined our system of shared dependencies, we can limit the search for candidates between the root and the last conflict on a branch - and also ignore the branches where such conflict does not exist - as we know that changes to the subtree beyond the conflicting node would have no effect on the collisions currently present.

After each mutation, the number of conflicting dependencies is recounted across the whole hierarchy, and the newly proposed state is accepted according to it's energy, current temperature and a factor of randomness. This concludes a single iteration and restarts the procedure with whichever state was chosen in the last step of a previous one , and also with slightly lower temperature.

Rounding it up, our proposed setup for simulated annealing on dependency trees is as follows:

State - a fully resolved dependency tree, whether it is still conflicting or not

Energy - the number of packages conflicting in the given state

Mutations (defining neighbours for each state) - changing a single package version, where the package is a either a conflicting one, or a predecessor of one of the conflicting packages. The new version must satisfy the version range of at least one of the packages dependent on the mutated one - if it does, it is set as a new dependency of said package, otherwise it is immediately discarded as not being a valid neighbour and a new mutation is generated.

There is much fine-tuning that can be done as far as setting a correct transition function as well as initial temperature goes. As the main focus of this paper is in the formalization and the use of the concept of public dependencies, we have used perhaps the most general of the acceptance probabilities, which is the one in the form of

$$e^{-\triangle D/T} > Random(0,1)$$

Where $\triangle D$ is the difference in the energy value between the two states (the different number of conflict), T marks the current temperature and $Random(0,1)$ is a random value between 0 and 1, generated anew with each iteration. The distance is positive when the number of conflicts rises and negative otherwise (yielding a result over 1). [25] The problem with

this general formula in our case lies mostly in the fact of the energy having not enough fluctuation (most of the mutations add or resolve 0 - 2 conflicts) - still, the approach has yielded some promising results, and we believe that with additional tuning and heuristics, it may be usable in production environment. Our code can be found, just like the rest of the samples, in the appendix B B.6

Chapter 7

Results

We have tested our package manger against the two other Node.js managers that we know of - NPM and ied. To keep the tests topical, we have done so on the three of these packages themselves - each of them installing the remaining two as well as itself to a local folder.

Our PC setup was a machine running a 32 bit version of Ubuntu 14.04, with 2Ghz Intel Core 2 Duo processor, together with a 25Mb/s broadband Internet connection (other aspects, such as the amount of RAM, should not have any impact on the tests we have run). We have used the command line time utility to track the duration of execution of each of the installs. The time used in the following measurements is the 'real' output from the time command - the actual time of execution as compared to i.e. the amount of time the CPU was used by the process. This is because much of the time spent is on I/O operations, whether it is downloading or installing of the packages, thus we have decided to handle the fluctuations caused by outside sources by measuring on multiple trials.

Since at the time of writing is, our manager did not have a command line interface, we have tracked the gulp task which in turn have run the mocha tests containing the function call to install the desired package. This introduces some minor overhead before the package manager execution is started, but since it was generally under 3 seconds. Again, that is smaller then the fluctuations seen when running on the same package multiple times in a row, we consider this slowdown insignificant and ignore it in further comparisons.

7.1 Package Managers installation results

The resulting attributes of each installation are shown in the following tables - three trials has been done for each of the installed packages, by each of the package managers. We can conclude that while we have outperformed NPM in all of the trials, ied is still substantially

faster than both VPM and NPM. Yet, ied actually gives very small guarantees as far as the conflict safeguards go, and would be hard to come close in terms of the installation speed as long as we do the two-step process of building the hierarchy before we start with the installation attempts. As for the size of the installation, VPM has shown an overall best performance amongst the three. As for the size of the installation, VPM has shown an overall best performance amongst the three.

VPM	NPM	ied
56 405	99 537	17 389
58 377	94 083	17 525
57 964	99 059	20 416

Table 7.1: Installation times of VPM, in miliseconds

VPM	NPM	ied
81 115	160 983	22 619
84 053	159 152	21 433
82 446	147 503	21 003

Table 7.2: Installation times of NPM, in miliseconds

VPM	NPM	ied
75 677	96 701	16 854
66 793	94 654	15 564
69 526	95 310	12 431

Table 7.3: Installation times of ied, in miliseconds

.	VPM	NPM	ied
VPM	39.7	44.2	40.1
NPM	52.7	61.2	52.6
ied	27.8	31.3	34.4

Table 7.4: The size of installations in MB, collums specifying the manager used, rows the manager installed

To list all of the (minor) issues that might have caused additional time disadvantage, we can say that VPM is by far the most talkative one, changing which could save it maybe up to a couple seconds of time. In our trials we have prevented NPM from running additional scripts (like installing documentation), since neither VPM or ied currently does so. Both VPM and ied is in that they currently do not support installing from github links, though this has not shown in tests.

7.2 Annealing - simple example

Since NPM - the package manager used in production - does not have a way to resolve conflicts, the packages you can find in it's repository are also prepared in a way as to not cause any of them. As we in fact needed to generate examples to test our problem resolution, we have taken the following approach.

We have defined a mode that is set before any of the tests regarding the annealing process is run. In this mode, every dependency is considered public - we are thus working in a similar environment as Bundler or PIP. Since keeping any of the dependencies as private would either not affect the problem, or make it easier, we can declare that this is a strictly harder problem - and solutions from it can always be applied in the situation with mixed dependencies.

Listing 7.1 Dependencies, as defined in `package.json`

```
"dependencies": {
    "firebase-queue": "*",
    "firebase": "*",
    "lodash": "3.7.x"
  }
```

In this example, the `package.json` file is as follows 7.1. From it, resolving greedily without any annealing yet, the following hierarchy arises 7.2. We can see the conflicting lodash versions right away (keep in mind that in these tests, all the dependencies are kept public). Additionally, `firebase-queue1.3.1` does not allow for a lodash version satifying the `3.7.x` requirement, the only way to resolve the hierarchy is to swap the firebase-queue version itself. At this point, NPM would fail, printing a conflict, while ied would continue as if nothing happened (changing the conflicting public dependencies to private) - which is potentially even worse.

Listing 7.2 The resulting tree before any annealing was done. The lodash versions are conflicting. The packages marked with * are the ones being reused.

```
__root__ 0.0.0
  lodash 3.7.0
  firebase-queue 1.3.1
    lodash 4.11.2
    rsvp 3.2.1
    node-uuid 1.4.7
    firebase 2.4.2
      faye-websocket 0.11.0
        websocket-driver 0.6.4
          websocket-extensions 0.1.1
    winston 2.2.0
      pkginfo 0.3.1
      colors 1.0.3
      cycle 1.0.3
      eyes 0.1.8
      isstream 0.1.2
      async 1.0.0
      stack-trace 0.0.9
  *firebase 2.4.2
```

With annealing enabled, since this is a simple problem with just a couple of conflicts VPM finds a valid solution, usually within the first 10 - 15 iterations.

7.3 Conclusion

We have presented a novel model for handling a hierarchy with mixed - private and shared - dependencies. We have shown the problems of the currently used concepts as well as the measures we have taken to fix them. The previous sections of this chapter have also shown the application of this concept on real life data.

Listing 7.3 The resulting, conflict-free hierarchy

```
__root__ 0.0.0
  firebase 2.4.2
    faye-websocket 0.11.0
      websocket-driver 0.6.4
        websocket-extensions 0.1.1
  lodash 3.7.0
  firebase-queue 1.0.0
    *firebase 2.4.2
    rsvp 3.2.1
    node-uuid 1.4.7
    *lodash 3.7.0
    winston 1.1.2
      async 1.0.0
      isstream 0.1.2
      cycle 1.0.3
      stack-trace 0.0.9
      pkginfo 0.3.1
      eyes 0.1.8
      colors 1.0.3
```

Chapter 8

Future work

As far as the work related to the public model itself goes, the one hurdle we have not yet decided upon are (public) optional dependencies. That is, handling of dependencies which, first of all, may not be installed at all without the manager viewing this as a problem, and that provide the same guarantees as public dependencies, but only when actually installed - otherwise, they should not 'lock' the version of their respective package. The first requirement would currently be satisfied simply by allowing the programmer to directly define something like inherited dependencies directly in their `package.json`, but we have yet to decide on an elegant way to satisfy both of these demands at once.

The annealing section of the algorithm is currently more of a proof-of-concept than a production tool. The 'default' acceptance function, although fine tuned to our needs, is still probably not the best option for the problem at hand, since the energy - the amount of conflicts in the hierarchy - is a function that rarely fluctuates past the difference of one or two steps (conflicts). Also, there is definitely place for additional heuristics.

As far as the quality-of-life features of a package manager goes, the one that we would really like to add in the near future is a form of a shrinkwrap file - a one that would lock currently installed versions for any further installations, even between different machines. This is a sound idea both in the context of annealing, and considering that due to race conditions, even the installation itself might not always be deterministic (if multiple correct installations exist). Additionally, we would like to add support for git links in `package.json` and possibly a simpler way of updating a hierarchy.

Rounding up, we would like to ultimately make this project an open-source collaborative effort. Which certainly requires a couple more iterations of refactoring and documenting of provided functions.

Appendix A

Proof of Public And Peer Dependency model equivalence

The characteristics we set out to formally prove are as follows. Firstly, we would like to show that we can mimic the construction of a peer dependency tree using public dependencies - in that for every hierarchy in one of the models, we can always build a tree representing it in the second one. Following this, we prove that the resolvability of the two is equivalent - again, except for the deliberate case outlined earlier (we will also prove this is the only case of such inconsistency). Finally, the results of these proofs are used to show the backwards compatibility of our model.

Since we do not allow for multiple versions of the same package on a single level (meaning - as two separate dependencies for the same package), we can't really model the situation where a single package requires two different versions of a dependency, once as public and once as private. This, apart from the slight shift in resolvability talked over in detail in chapter regarding the public dependency model, should be the only borderline case where the backwards compatibility would not be present. To our knowledge, this kind of practice was never a part of any of the publicly used NPM packages.

A.1 NPMv3 vs NPMv2

For the sake of simplicity, we will use the model of NPMv2 - the one where every peer dependency is automatically installed. Yet, as we still claim both the backwards compatibility and equivalence with NPMv3 - though both are harder to define due to the nature of peer dependency warnings - we will briefly outline the differences that the use of this model would cause within the proof.

Maybe the most major change relevant to the formalization of version three's model would be in warnings for missing peer dependencies. If we were to treat them as errors - that is, consider the dependency tree yielding them as unresolvable - we would lose on some of the functionality of inherited public dependencies, which we have defined not be mandatory. On the other hand, ignoring them completely, while providing us with the behaviour close to the public dependency model, would then let us create dependency tree that, though resolvable, do not provide the desired functionality to the end user - in that it misses some of the packages required for the whole program to run.

Note the differences are only related to the handling of warnings in the resolution step. The construction steps, and from them the ensuing compatibility is unaffected by this, proving of which is our primary objective. Stretching the resolvability equivalence to work with said warnings, though certainly doable, would only introduce additional level of technicality into the proof, and was thus omitted.

A.2 Peer dependency formalism

Since with peer dependencies, we do not have the privateImports and publicExports functions, as they would not really make much sense in this context, we will define an alternative (though do so maybe in a less formal way). That is, let children(A) define the set of all children of an arbitrary resolved node A - whether they were direct dependencies of A or were resolved in their place through a chain of peer dependencies.

A.3 Peer dependency model construction

We are going to split the construction formalism into two subsections, so that it best suites us in our inductive proof later. The first one describes the 'growth' of the dependency tree, while the second one is concerned with the reviewing of peer dependencies and identifying possible conflicts. As a matter of fact, this also mirrors the way that both our package manager and many of the aforementioned others approach the problem - resolving the hierarchy first before continuing on with collision checking (with some featuring also a third step, that being dependency resolution).

For any unresolved or partially resolved dependency tree using the concept of peer dependencies, we have (just like in the case of public dependencies) only a single action available to us - choosing a version of an unresolved dependency, within the bounds of the version

interval defined by it's parent. In doing so, one or more of the following situations may arise:

1. The resolved package had no dependencies
2. One or more private dependencies are added as children of a resolved package
3. One or more peer dependencies are added on the same level as the resolved package

The review step is rather trivial in this case - we simply check each level and look whether it has any peer dependencies which conflict. Therefore, the only rule we need is a follows:

Let us choose an arbitrary node A. Then the nodes $B = (id_b, pkg_b)$ and $C = (id_c, pkg_c)$, $pkg_b = (name_b, version_b, prv_b, peer_b)$, $pkg_c = (name_c, version_c, prv_c, peer_c)$ (changing our formal definition slightly to accommodate for peer dependencies) are conflicting if $\{B, C\} \supseteq children(A)$, $name_b = name_c$ and $version_b \neq version_c$, meaning they represent the same package in different versions. This is the only kind of conflicts which may appear within this model.

Lemma A.1. *By using only the action of choosing a version of an undecided dependency, we can reconstruct any resolved dependency tree (which uses the concept of peer dependencies) given to us, whilst only one or more of the three previously described situations may arise after each such choice.*

Proof. The proof is constructive - in that, we start from the root and begin resolving dependencies in the same versions as they are in the hierarchy we want to copy. When a package is locked to a concrete version, we can look into it's `package.json` for any private or peer dependencies and resolve the accordingly - add them as unresolved children if they are private (point 2) or as unresolved peers if they are peer dependencies (point 3). If the tree we are reconstructing use the same data as we are, the dependencies we have added must also be the same. Thus, when every dependency is resolved, we must end up with a copy of the original hierarchy. □

A.4 Proposed model construction

We shall now define the same steps on public dependencies, as to draw a comparison. Again, we first define the those applied during the construction of the hierarchy, and subsequently the ones concerned with checking the correctness of public dependencies within it.

Once again, the only action available is the one of a single package resolution, upon which one or more of the following may happen:

1. The resolved package had no dependencies and did not conflict with any other public dependency
2. One or more private dependencies are added as children of a resolved package
3. One or more public dependencies are added as children of a resolved package

During the review step, we need to collect the exported public dependencies along the public branches, as was specified earlier. For a pair of nodes A and B:

1. If B is a private dependency of A, then $privateImports(A) \supset publicExports(B)$
2. If B is a public dependency of A, then $publicExports(A) \supset publicExports(B)$
3. A conflict exists on an arbitrary node A for B,C, $\{B, C\} \in privateImports(A)$, $B = (id_b, pkg_b)$, $C = (id_c, pkg_c)$, $pkg_b = (name_b, version_b, prv_b, pub_b)$, $pkg_c = (name_c, version_c, prv_c, pub_c)$, $name_b = name_c \wedge version_b \neq version_c$, no other source of conflicts exists

Lemma A.2. *By using only the action of choosing a version of an undecided dependency, we can reconstruct any resolved dependency tree (which uses the concept of public dependencies) given to us, whilst only one or more of the three previously described situations may arise after each such choice.*

Proof. Analogous to the one of Lemma A.1. □

A.5 Theorem, proof and corollary

Theorem A.3. *For every dependency tree using the peer dependency model exists an equivalent tree using public dependencies, which installs the same packages, in the same versions, and is resolvable if and only if the original tree was also resolvable. The only exception to this is the case discussed in "The difference in resolvability", and this class of hierarchies are the only one where the resolvability does not agree.*

Proof. The reason for excluding the borderline case was covered before. We will split the proof in a similar fashion as we did previously - into the construction and the dependency resolution parts.

Construction Induction over the situations which occur after a single package version is resolved. The base of induction is a root package with no dependencies - which is the same situation with no actions in either model.

Induction step always includes the action of resolving a package to a concrete version - that is present and equivalent in both models - after which it iterates over the given options. We will skip points one and two right away, since the equivalence between their counterparts is trivial. For the third point, we instead add a public dependency as a child of the current package - no further action is needed just yet. In this way, it is shown that we can mirror each step of the peer dependency tree construction, and following Lemma A.1, proves that we can reconstruct any peer dependency hierarchy using public dependencies.

Resolvability What is left to prove is that the newly constructed hierarchy will be resolvable if and only if the same can be said about the original, peer dependency based one. While the construction worked with the tree being build from bottom up, now we need to show that the peer dependency propagation works from top down in a way similar to the one used by public dependencies. A brief reminder that we are still working with NPMv2's mandatory peer dependency model - in NPMv3 this propagation would need have been maintained by the package developers of each of the dependencies. □

Lemma A.4. *If a package P_1 is a private dependency of root package R, and we have a chain of peer dependendencies where P2 has a peer dependency on P_1, P_3 has a peer dependency on P_2 and P_n has a peer dependency on $P_(n-1)$, then all the packages P_1 to P_n will be resolved as the children of root R.*

Proof. By induction on the length of the chain. Basis is trivial, with chain length being 0. Induction step - we have n packages, each is a peer dependency of the previous one and all of them are resolved as children of a single node. We add a peer dependency requirement for a package $n+1$ on the last of the reolved packages. This peer dependency will be automatically resolved (we are taking only non-conflicting dependencies into account, a conflict will abort this chaining, which is fine for us), on the same level as the package that required it - meaning they will share the same parent. □

Following Lemma A.4, we can draw the equivalence between chains of public dependencies and those of peer dependencies (additionaly, the same rules apply for the "chain" of length 1). The only way a conflict may arise in the two models is if two copies of the same package, in different versions ends up in the same set of an arbitrary node - this set being children(A) for peer and privateImports(A) for public dependencies. In the NPM model, each node in the chain of peers ends up as child of the parent of the first node of this chain

- let us call it root R - thus the only place where the conflict within the chain needs to be detected is in the set children(R). On the other hand, since public dependencies are exported one step past the last public branch - that being the same root package as in the case of peer dependencies if we have copied their hierarchy - the only place needed to check in this case is privateImports(R). This conflict may arise earlier on one of the nodes deeper in the chain of dependencies, it is trivial to show that the privateImports sets on these nodes must be strict subsets of the privateImports of the root of the chain, since during each step, dependencies are only added (and at least one is added per step).

A different conflict can also emerge deeper within the public dependency chain, between a public and a private dependency, but this is again related to the problem we have covered and have said to exclude - in this paragraph, we will clarify that this class of problems is the only one on which the equality of resolvability will fail - or in other words that no other source of additional conflicts in the public model exists. We have to realize that every peer dependency is ultimately checked in the children(R) of the node in which the chain it belongs to is rooted - any conflicts between two public dependencies (respectively, their peer equivalents) are handled there since they have to be the part of the same chain. Conflicts between two private dependencies, if such exist, has to be handled "in place" - on the parent node they belong to - since a method to transfer them is present in neither of the two models. The only option left are conflicts between public and private dependencies - and with them, only two situations may arise. Either the private dependency is directly on the node where the peer chain will be rooted (and is handled during the check of the children(R)), or lies somewhere deeper on the public chain, which is exactly the case of aforementioned problem. No other options are left to enumerate.

Following the rules set for exporting public dependencies, since a private brach essetially halts any further public dependency propagation, we can be sure that no dependencies outside of the chain were exported to the highest level (where the constrains for peer dependency model is checked), and therefore prove the equivalence of the sets children(R) and privateImports(R). The dependency check "algorithm" on these two is the same - finally proving our proposed thesis.

Lemma A.5. *For every dependency tree using the public dependency model exists an equivalent tree using peer dependencies, which installs the same packages, in the same versions, and is resolvable if and only if the original tree was also resolvable.*

Proof. The proof of the second implication is in many ways equivalent to the previous one. Let us just point out the only major difference, the fact that we no longer need to exclude the borderline case - the reversed transformation works every time. This is because in our

construction, we can take the extra step of being vigilant and adding additional private dependencies - and let the system fail on collision between two private dependencies. Thus, we define adding a new peer dependency (mirroring a addition of a public one) as a two step process - adding the peer dependency itself, analogically to the previous case to the parent of a package it belongs to, and adding an additional private dependency. This way, the layer of security we may have missed with peer dependencies in general, for packages Pn on a peer dependency chain, is provided in (possibly) conflicting packages in the sets children(Pn).

The rest of the proof is analogous in it's steps and uses Lemma A.2 instead of Lemma A.1. □

We had a second question in mind during the construction of previous proofs. The fact to the matter is, you can always transform a model with shared dependencies into a one that has only private ones, if you add enough constraints (in the form of additional private dependencies) along the places in the hierarchy where shared dependencies would be exported (and of course, make two private dependencies rooted in the same parent trigger a conflict). This is similar to the "hack" we have used in the proof of second implication. The resulting hierarchy would resolve into the same set of packages, although the rules of such transformation would be much less legible then the current ones. Thus, the construction proof on it's own is not of that great of a value, yet, in the way we had modeled it, we have also proven the following premise.

Corollary A.5.1. *The public dependency model is backwards compatible with NPMv2's model of peer dependencies, in a way where treating every peer dependency in the definition of each package as public will yield a hierarchy equivalent to the original one using peer dependencies.*

This is an important aspect of our design, since it allows us to use it in the present environment, and on current NPM packages, without the need of additional prefabrication.

Appendix B

Code samples

Listing B.1 The getPackageInfo function

```
export function getPackageInfo(nrConnections = 6) {
  let ch = csp.chan()

  function* spawnWorker() {
    while (true) {
      let [pkg, resChan] = yield csp.take(ch)
      let res = yield csp.take(downloadAndParsePackage(pkg))
      let errCount = 0
      while(res instanceof Error) {
        // error handling/printing omitted
        res = yield csp.take(downloadAndParsePackage(pkg))
      }
      yield csp.put(resChan, res)
    }
  }

  for (let i = 0; i < nrConnections; i++) {
    csp.go(spawnWorker)
  }

  // pkfInfoGetter - pkg is either a name or an url to tarball
  return (pkg) => {
    return csp.go(function*() {
      if (pkg in registry) {
        if (registry[pkg]) {
          return yield csp.peek(registry[pkg])
        }
      }
      // resChan has to have buffer of a size 1 to be peekable
      let resChan = csp.chan(1)
      registry[pkg] = resChan
      yield csp.put(ch, [pkg, resChan])
      return yield csp.peek(resChan)
    })
  }
}
```

Listing B.2 The function to transform a callback function to a one using csp.

```
// returns a channel that blocks until function callback is called
// the channel yields either an error or csp.CLOSED
export function cspy(fn, ...args) {
  let ch = csp.chan()
  fn(...args, (err) => {
    if (err) csp.putAsync(ch, err)
    ch.close()
  })
  return ch
}
```

Listing B.3 The function that handles Node.js stream and returns the result in csp channel.

```
// returns contents of any stream that emits 'data' and 'end'
// events as a single string
export function cspyDataStream(stream) {
  let ch = csp.chan()
  let str = []
  stream.on('data', (chunk) => {
    str.push(chunk)
  })
  stream.on('error', (e) => {
    csp.offer(ch, e)
    ch.close()
  })
  stream.on('end', () => {
    csp.offer(ch, str.join(''))
    ch.close()
  })
  return ch
}
```

Listing B.4 Our implementation of the missing peek function

```
// patch csp with a peek method:
// obtain a value from channel without removing it
csp.peek = function(ch) {
  return csp.go(function*() {
    let res = yield csp.take(ch)
    yield csp.put(ch, res)
    return res
  })
}
```

Listing B.5 The recursive checkNode algorithm.

```
const checkNode = (visitedSet = new Map()) => {
  if (visitedSet.has(self)) return visitedSet.get(self)
  visitedSet.set(self, [])
  let publicExports = []
    .concat(...self.getDependencyNodes(true)
    .map(d => d.checkNode(visitedSet)), self)
  visitedSet.set(self, publicExports)
  let privateImports = []
    .concat(...self.getDependencyNodes(false)
    .map(d => d.checkNode(visitedSet)), self)
  let checkMap = {}
  for (let dep of privateImports) {
    if (checkMap[dep.name] &&
      (checkMap[dep.name].version !== dep.version)) {
      conflictingNodes.add(dep)
      conflictingNodes.add(checkMap[dep.name])
    }
    checkMap[dep.name] = dep
  }
  return publicExports
}
```

Listing B.6 The simulated annealing algorithm

```
export function annealing(root) {
  return csp.go(function*() {
    debugroot = root
    for (let j = 0; j < TEMP_ITERATIONS; j++) {
      let oldState = 0
      for (let i = 0; i < ANNEAL_ITERATIONS; i++) {
        oldState = checkDependencies(root)
        if (!oldState) break
        let muts = mutatibleNodes()
        let candidate = sample(muts)
        let undoMutation = yield candidate.mutate()
        let newState = checkDependencies(root)
        if (!annealTransition(oldState, newState, j)) {
          undoMutation()
        }
      }
      if (!oldState) break
    }
  })
}
```

Bibliography

[1] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, J'erome Vouillon, Berke Durak, et al.. *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*. 21st IEEE/ACM International Conference on Automated Software Engineering, Sep 2006, Tokyo, Japan. ACM, pp.199-208, 2006

[2] Apt-get manual,
https://www.debian.org/doc/manuals/aptitude/ch02s03s01.en.html

[3] Ian murdock blogpost (discussion),
https://www.techdirt.com/articles/20160112/16582733316/ian-murdock-his-own-words-what-made-debian-such-community-project.shtml

[4] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner *OPIUM: Optimal Package Install/Uninstall Manager*. In Proceedings of the 29th international conference on Software Engineering (ICSE '07). IEEE Computer Society, Washington, DC, USA, 178-188, 2007

[5] Opensuse documentation,
http://doc.opensuse.org/projects/libzypp/HEAD/

[6] J. Austen *Pride and Prejudice*. CreateSpace Independent Publishing Platform (November 29, 2014).

[7] The end of yum discussed,
http://dnf.baseurl.org/2015/05/11/yum-is-dead-long-live-dnf/

[8] Yum documentation,
http://yum.baseurl.org/api/yum/yum/depsolve.html

[9] Debian manual - dependency types,
https://www.debian.org/doc/debian-policy/ch-relationships.html

[10] Portage manual,
http://dev.gentoo.org/ zmedico/portage/doc/pt02.html

[11] Package management blog,
http://blog.ezyang.com/2014/08/the-fundamental-problem-of-programming-language-package-management/

[12] Semantic versioning,
`http://semver.org/`

[13] Number of modules within different environments,
`http://www.modulecounts.com/`

[14] V. Černý *Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm*. Journal of Optimization Theory and Applications In Journal of Optimization Theory and Applications, Vol. 45, No. 1. (1 January 1985), pp. 41-51, 1985

[15] Molinillo documentation,
https://github.com/CocoaPods/Molinillo/blob/master/ARCHITECTURE.md

[16] Pip issue - conflict resolution,
`https://github.com/pypa/pip/issues/988`

[17] Alsuwaiyel, M. H. *Algorithms: Design Techniques and Analysis*. World Scientific Publishing Company (August 30, 1999).

[18] Library re-exports is NP-complete,
https://http://wiki.apidesign.org/index.php?title=LibraryReExportIsNPComplete

[19] D. Burrows *Modelling and Resolving Software Dependencies*.
`https://people.debian.org/ dburrows/model.pdf`

[20] CSP model of asynchronicity and transducers,
`http://phuu.net/2014/08/31/csp-and-transducers.html`

[21] Node.js documentation
`https://nodejs.org/api/modules.html`

[22] Node.js module system explained
http://fredkschott.com/post/2014/06/require-and-the-module-system/

[23] NPM peer dependency discussion,
`https://github.com/npm/npm/issues/5080`

[24] Javascript csp library documentation,
`https://github.com/ubolonton/js-csp`

[25] more on simulated annealing,
http://mathworld.wolfram.com/SimulatedAnnealing.html