Improving Dependency Resolution of Python Packages

Bachelorarbeit

von

Alexander Schneider

aus

Krasnyj Jar

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

Juli 2013

Betreuer:
Philipp Hagemeister M. Sc.

Acknowledgments

Firstly I want to thank gravity, because writing a thesis in midair would have been really annoying.
Secondly I thank Stannis Baratheon, the First of his name, King of the Andals and the Rhoynar, Lord of the three Kingdoms, Protector of the Realm and Champion of the one true God.
Last but not least I want to thank my fellow students from the *Fachschaft Informatik* and my advisor Philipp Hagemeister for their help.

Contents

List of Figures

Chapter 1

Introduction

Every big software project usually needs a number of dependencies on external software and packages. Despite the extensive built-in libraries of languages like Python the need for external packages, from now on called modules, often arises.

However most modules have dependencies on external modules themselves. This introduces the so called *dependency resolution problem*, which is defined defined by Burrows in [Bur05]. Current Python installation tools like *buildout* or *pip* can't solve complex conflicts between those dependencies and the modules already installed. For example *adhocracy* version 1.1 has 24 direct dependencies and over 700 possible indirect caused by the direct dependencies.
Furthermore there are no tools available which can perform a quick check whether a newly introduced dependency to a module breaks the module because of conflicts with other dependencies. For this thesis a Python module called *PyDYN* was designed and implemented. *PyDYN* adapts existing generic solvers for pseudo boolean optimization to compute solutions for the dependency resolution problem. For further research several analyses of python installation files and their structure were made. In the context of designing *PyDYN* several other needs for a metadata database or an automated downloader for packages arose and were catered to in scope of the thesis.

The second chapter examines the theoretical background for the thesis. The following chapters three and four describe the practical implementation of a tool to solve depen-

dencies and some additional work done in scope of this thesis; For example a downloader tool and analysis of setupfiles. The fifth chapter shows related work and the last chapter ends the thesis with a conclusion.

The second chapter examines the theoretical background for the thesis. The following chapter three discusses related work. Chapters four and five describe the practical implementation of a tool to solve dependencies and some additional work done in scope of this thesis; For example a downloader tool and analysis of setupfiles. The sixth chapter ends the thesis with a conclusion.

Chapter 2

Theoretical Background

This chapter states some theoretical definitions needed to understand the methods used for solving the dependency resolution problem. All given definitions are similar to the definitions introduced by Trezentos et al. in [TLO10] but specific to Python.

2.1 Definitions

1. **Module**: A module m is a tuple (id, v) where id is a simple name or identifier and v is the version.
   $\mathscr{M}$ is the finite set of all modules.

2. **Dependency**: A dependency d is a set of modules $d_1, \ldots, d_n$ with the possible version constraints $>, <, \geq, \leq, ==, \neq$ for every d_i. Every module has a finite number of dependencies.

3. **Conflict**: A conflict is a function $c: \mathscr{M} \times \mathscr{M} \rightarrow \{True, False\}$, that states if two modules can not be used simultaneously. Typically two modules in Python only conflict if they have the same id, because Python only allows unique module names in its namespace. The Python aliases for imports are not a solution because only the contents of Pythons `sys.modules` variable matter, which uses the original module names.

For example:

$$m_1 = (adhocracy,\ 1.1),\ m_2 = (pylons,\ 0.7.9),\ m_3 = (adhocracy,\ 2.0)$$

$$c(m_1,\ m_2)\ =\ False\ ,\ \ c(m_1,m_3)\ =\ True$$

Furthermore two modules can conflict indirectly through their dependencies for example: m_a has dependency d_1 = "pylons>2" and m_b has dependency d_2 = "pylons=1.1" $\rightarrow c(m_a,m_b) = True$.

4. **Installation**: An installation $\mathscr{I}$ is a typically small subset of $\mathscr{M}$. An installation is *consistent* if $\forall m_i, \neq m_j \in \mathscr{I} :\ c(m_i,m_j) = False$, where $i, j \in \mathbb{N}, m_i \in \mathscr{M}$.
   In simpler terms an installation is consistent if all modules of the installation do not conflict with each other pairwise.

2.2 Dependency Resolution Problem

The dependency resolution problem is defined using a definition alike that of Burrows in [Bur05]:
Given an installation i_0 that is not consistent, we want to find a similar installation i_c that is consistent. In other words, all *id*s corresponding to modules contained in i_0 have to be present in i_c. This problem is NP-complete as burrows shows in [Bur05] using a reduction from CNF-SAT.
Although the problem is NP-complete, a pseudo boolean optimization formula can be contrived for a given dependency problem. The PBO formula in turn can be used to solve the problem heuristically for different optimization criteria.

2.3 Pseudo Boolean Optimization

Pseudo boolean functions are of the form

$$f(x_1,\ldots,x_n) \;=\; \sum_i c_i \;\cdot x_i$$

where $i \in \mathbb{N},\ x_i \in \{0,1\},\ c_i \in \mathbb{Z}$ is a constant.
For the problem at hand integer constraints are added to the formula as seen in [TLO10]. The formula then has the generic form

$$\sum_i c_i \;\cdot x_i \geq n,\ \ n \in \mathbb{Z}.$$

The constraint is not limited to the $\geq$ operator. The operator can be replaced by $\leq$, $>$, $<$, $=$. A generic PBO formula can also have an optimization formula, for example $min(f_1)$, *or* $max(f_2)$. While the constraints are mandatory the optimization formula is not. A PBO formula without the optimization part is alike to a SAT problem, where all solutions are equally good.

Figure 2.1: PBO Formula Example

$$\begin{aligned} min : 20 \cdot x_1 \;+\; 15 \cdot x_2 \;+\; 25 \cdot x_3 \\ 1 \cdot x_1 \;+\; 2 \cdot x_2 \;\geq 1 \\ 1 \cdot x_3 \;+\; 1 \cdot x_2 \;\leq 1 \end{aligned}$$

Chapter 3

Related Work

Other researchers tried to solve the dependency resolution problem before.
[Bur05] proposed a best First Search in combination with techniques to reduce the branching factor. However this solution was not evaluated in practice.
[LBR09] introduced an metadata format and a corresponding constraints encoding for dependency relationships to resolve upgradeability problems with *Eclipse* plugins.
A similar approach was made by [ABL$^+$10]. The paper proposed using boolean optimization and SAT to resolve the upgradeability problem in Linux distributions. The corresponding implementation of a tool used *Sat4J-PB* and *P2CUDF*, which did not always perform at the top level compared to other solvers.
Finally Trezentos et al. [TLO10] used an component based approach in combination with pseudo boolean encoding. The main contribution was showing that every step of the dependency resolution (pbo translation, resolving the pbo formula, fetching the files) is interchangeable if encapsulated.

Chapter 4

Implementation

The main work of this thesis was the creation of a Python dependency resolver tool called *PyDYN*. *PyDYN* and its corresponding API and implementation are described in this chapter. When standard installation tools for Python, like *pip* or *buildout*, use *PyDYN* they can potentially experience a big speedup of execution. *PyDYN* calculates all needed packages beforehand so the installation tool can use concurrent downloads. Without *PyDYN* the installation tool downloads one module then calculates its dependencies and downloads those sequentially. Then the procedure is repeated for the newly downloaded modules until all dependencies are resolved.

4.1 Structure

PyDYN was structured in three modules. The modules are responsible for metadata generation, PBO translation, graphical output and provide an API. Every part is described in detail in the following sections. For the correct use of *PyDYN* a PBO solver and the metadata file *meta.json* is needed.

4.2 Dependency Database

For a complete calculation of an optimal installation, a powerset of all dependencies of the to be installed module must be examined. For example *adhocracy version 1.1* has 24 dependencies. If the recursive possible dependencies are considered the number rises to over 700. Far too much packages to be downloaded for every calculation.
Because the *Python Package Index* (*PyPI*) provided no means to obtain metadata, a database was constructed which holds all parseable dependencies for modules which were available on *PyPI* at 2013-05-03. For the construction 112,600 modules were downloaded with a tool created for this purpose. The downloader is described further in chapter 5.1.
A Python tool parses all modules heuristically for the creation of the database. All modules on *PyPI* are required to have a setup.py file which typically uses a *setup()* function. The *setup()* function then provides a list of all dependencies needed for the installation of mentioned module. Three different database formats were considered for the task at hand.

1. A *.tar.gz* file which holds directories corresponding to module names, which in turn hold *.json* files corresponding to versions of that module, with the metadata as content.

2. A *SQLite* database where every row in a table holds the metadata.

3. A single *.json* file which represents a dictionary with the module names as keys and another dictionary as value. This second dictionary has the module versions as key and the corresponding dependencies as value.

For every of the three formats benchmarks were obtained to measure performance. Using the dependency resolution calculation for *adhocracy v. 1.1* the time needed for execution was measured.

1. **.tar.gz-file**: 79 sec

2. **SQLite database**: 50 sec

3. **Single .json file**: $<$ 1 sec

The .tar.gz file performed poorly because .tar is designed for sequential access and the algorithm utilizes a lot of random access.
The fact that select-statements with string comparisons are fairly slow in SQL explains the bad performance of the SQLite database. Furthermore an index was not defined at the time of the database creation, which probably slowed it down further. Other databases like *PostgreSQL* could potentially be a lot faster because of advanced features ,like the definition of own field types, not available in *SQLite*. They were however not tested for the sake of simplicity and because the *.json* file was fast enough at the time of implementation.
Because of the good performance of the *.json* file it was chosen as the database format in the final version of *PyDYN*. The file has a size of 5 MB for 112000 packages, so concerns that the program may slow down if the file gets too big to load in RAM are unwarranted.
Currently the developers of Python are working on a metadata format which will provide online metadata for all packages on *PyPI* in the future. The metadata format is described further in PEP 426 [CHS13]. A proof of concept implementation of the online metadata for *PyPI* has been made by the red-dove.com developers [Tea13].
To demonstrate compatibility with the new format *PyDYN* has a *new meta mode* where it uses the red-dove.com database to retrieve metadata. This mode is just a proof of concept and is very slow at the moment, because every dependency request results in one *HTTP* request.

4.3 OPB Translation

The *solver.py* module is part of *PyDYN* and mainly responsible for generation of metadata and translation from dependencies to *OPB* and back. *solver.py* was designed as the core of *PyDYN*.

4.3.1 OPB file format

PyDYN utilizes modern PBO solvers for the resolution of dependencies. *PyDYN* is compatible to all solvers that were part of the *Pseudo Boolean Competition* [RM13b]. These solvers were supported because they all have to comply to the OPB file format, which is described in [RM13a].
The OPB format describes how a PBO formula can be represented in a machine-readable fashion.

4.3.2 Dependency dictionary

The main part of the solver.py is the *OPBTranslator* class which is responsible for the generation of metadata, the translation of a dependency dictionary to OPB and the translation from the output of a PBO solver back to a human or machine-readable format.
`OPBTranslator` is instantiated with a module name and optionally a version of the module that should be installed.
`generateMetadata()` is then called to generate a dependency dictionary. The algorithm uses the module name to retrieve the dependencies for that module. If those were not already processed their dependencies are retrieved as well. This step is iterated until no new dependencies can be retrieved.
If more than one module has to be installed the process is slightly different. For every module `addDependency()` then `generateMetadata()` has to be called.

4.3.3 Translation process

`generateOPB()` is used after a dependency dictionary was created and translates the resolution problem at hand into a PBO formula in OPB format.
First the minimization function is created for which the problem has to be optimized. Modules already installed get a big negative coefficient, because optimally they should not get uninstalled during the installation of a new module if possible.
All possible new modules get a small positive coefficient, so the solution is optimized for

the least number of newly installed modules. Installing the newest version of each module would be another optimization criteria which conflicts with the least number of new packages criteria and is not implemented in the current *PyDYN* version. For example:

Current installation:	*foo 3, bar 1*
Module to be installed:	*baz 5*
Dependency Dictionary:	{*baz 5: [pyrate 3, pyrate 2], pyrate 3: [], ...*}

The optimization formula then is:
$min: \ -100 \cdot (foo,3) \ -100 \cdot (bar,1) \ +20 \cdot (baz,5) \ +20 \cdot (pyrate,3) \ +20 \cdot (pyrate,2)$
The to be installed module is translated to a formula with a $\geq$ constraint. According to the first example: $(baz,0.5) \geq 1$
All conflicts caused by the same name in the dependency dictionary are translated to formulas with similar constraints.
For example $-1 \cdot (pyrate,3) \ -1 \cdot (pyrate,2) \geq -1$ is a conflict between two versions of the module *pyrate*
The dependencies for a module translate into several formulas, because every dependency d_i of one module and all possible valid candidates to satisfy d_i are translated into one formula. The procedure ensures that every dependency gets satisfied if possible. For example if the previous example had another dependency for *baz v. 0.5: pygments v. 1.3alpha*
$-1 \cdot (baz,0.5) \ +1 \cdot (pyrate,3) \ +1 \cdot (pyrate,2) \geq 0$
$-1 \cdot (baz,0.5) \ +1 \cdot (pygments,1.3alpha) \ \geq 0$
An `OPBTranslator` object maintains a symbol table which maps modules to variables, because the PBO format only allows variable names $\in \ x_1 \ldots x_n$

`OPBTranslator` also parses the output of a PBO-solver and uses the symbol table to generate two lists: one list which contains the elements that should be installed or should be kept installed and one list with elements that should not be installed or should be uninstalled.

4.4 Graphical Output

The *depgraph.py* module is part of *PyDYN* and provides methods for the visualization of dependency graphs.
depgraph.py generates dictionary objects from the current installation and uses those dictionaries to generate *.dot* [gra13] files. *depgraph.py* uses the graphviz package under *UNIX* systems to convert the *.dot* files to *PNG* or *SVG* graphics. Furthermore a terminal output can be generated.
depgraph.py can be used standalone from terminal or imported as a module.

4.5 API

The API provides two classes `Problem` and `Solution`.
A `Problem` object is instantiated with the name of the module to be installed. Optionally a solver and an installation path can be set. `Problem` provides `solve()` to solve a conventional dependency resolution problem or `checkFutureDependency()`, for checking if one or more newly introduced dependencies break the module. For example: If one wants to introduce *communism* as a new dependency to *adhocracy* `checkFutureDependency()` verifies if all dependencies of *adhocracy* could still get satisfied at the same time. If there is no possible consistent solution `checkFutureDependency()` prints a maximal subset of dependencies that is consistent. Continuing the previous example: If *communism* conflicts with three of *adhocracys* old dependencies the returned solution will suggest to not use *communism*.
Either way, both `solve()` and `checkFutureDependency()` return a `Solution` object.
`Solution` objects provide methods to draw a graph of the solution and get human readable strings or tuples of the modules which have to be installed for the consistent solution found. Another output option are *pip* compatible *requirements.txt* files.

4.6 Integration in pip

For demonstration purposes the API was integrated in the 1.3.X developer version of the *pip* installation tool. The API hooks before *pip* starts to download the needed modules. Before the hook *pip* only adds the module that should be installed to its download queue. At this point *PyDYN* calculates the dependencies and passes the result as fixed dependencies (for example *pyrate==0.1*) to *pip*. This method extends *pips* own dependency resolution, because the user still can add custom dependencies with a *requirements.txt* file or per command line parameter. *pip* then proceeds to download and install the packages. Because *PyDYN* heeds the, at the moment of the installation, current python library path it will not simply install conflicting versions or abort like standard *pip* does, but instead calculate an optimal solution for the configuration at hand if possible.
A pull request for the modded version against the master branch of *pip* was not made, because *PyDYN* needs a metadata database to work (for example *meta.json*, described earlier) and the efforts to construct a metadata-format and a database for python just have been started by the developers. For further information see PEP 426 [CHS13].

Chapter 5

Additional Work

This chapter describes additional work that was either necessary for the completion of the thesis or improved the quality of *PyDYN*.

5.1 Downloader Tool

For a fast analysis of all packages on the *Python Package Index (PyPI)* the packages need to be downloaded. A Python script which was written for this purpose is described in this section.
The script uses five threads to execute the downloads simultaneously. The number of threads can be adjusted in the source code in the `get_files()` method. A hashing method is responsible for the management of finished downloads. Every thread that is not busy requests a new download URL from a generator. The generator parses PyPI for available modules. If it encounters a new module it generates a hash *h* from the name of the module and crates a *h.part* file in the cache directory. The generator then parses all download links for the versions of the given module and saves them into *h.part*.
The links are returned to the threads. The thread hashes the link and checks if the hash is present in the cache directory. If not it creates a *hash.part* for the link and starts the download. If the download is ready the *hash.part* file gets renamed to *hash*. When all downloads for a module are ready *h.part* becomes *h*. This way every version of a

module only has to be downloaded exactly once. The .part files are used to prevent corrupted files. If network connectivity is interrupted during the download the tool repeats all downloads whose hash is still a .part file.
The downloader can be started in update-mode. Update-mode is slower, because it ignores the existing index-hashes and this way scans for new versions of all packages.

5.2 setup.py Analysis

PyDYN in its current state faces the problem, that an up-to-date metadata database is needed for the calculation of a solution. Furthermore the database file has to be present on the system for every calculation. Another problem of the Python installation routine is that currently a *setup.py* file has to be run for the installation of a Python module. The code in the *setup.py* file is potentially harmful and is not limited to a subset of the python language; For example see the *python-nation* module on *PyPI* [KM13].
Those problems would be obsolete if an online database that stores metadata, similar to the repositories of *Linux* distributions, would be provided. It is not realistic that every of the 112000 modules would be updated to the new metadata format by their maintainers. So an analysis of the setup.py had to be made to investigate if the code of the setup.py files is simple enough to be replaced by metadata.

5.2.1 Execution of the Analysis

The Analysis was done using a script written in Python. The algorithm iterates over all in chapter 5.1 previously downloaded modules. For every module the setup.py file is extracted and parsed into an *Abstract Syntax Tree (AST)*. For further analysis an *Analyzer* object is created for the given AST. The Analyzer object iterates over the trees body, which is constituted by all nodes that are direct children of the root. Figuratively speaking the body consists of all expressions, function definitions, class definitions, etc. The analyzer's main goal is to assess if the setup file is valid or not. The criteria for a valid file are:

- Not importing an invalid module.
- Not calling non-builtin functions (with the exception of the *setuptools*, *distribute* and *ez_setup* modules).
- Not instantiating invalid classes. A class is invalid if it calls non-builtin functions.

During the iteration every node is checked for its type and passed to a corresponding processing method. If the passed node itself has a body, which is for example the case when the node is a class definition or function definition, a new analyzer instance is created with that node as root and processed further. The analyzer works with a combination of a white and a blacklist. If a module import is encountered and the module is not whitelisted the module itself and all of its functions are blacklisted. Furthermore if the module is not a built-in module in Python an *Analyzer* object is created for the source code of the imported module. The reason for this behavior is that in Python a module executes its source code upon import and could therefore execute harmful or undesirable code - in which case the importing setup.py would have to be labeled as not valid according to the previous requirements. However this method is heuristic, because it is possible to have an invalid module with the name of a builtin module in the directory of the setup-file. This would cause python to import this invalid module instead of the built-in module.
When a function definition is parsed its AST body is evaluated with an Analyzer object and saved in a symbol table for future reference. When that function is called in the future its content is validated again and blacklisted or respectively taken down from the blacklist according to the outcome. The body of the function has to be saved because in python function assignments are a mere binding to a namespace and can be changed during execution of a program. Consider the following example:

```
def evil():
        print("I'm a valid function")
def f():
        evil()
def evil():
        evilmodule.evilfunction()
f()
```

First the analyzer encounters the function definition for *evil()*. Then the analyzer saves the body of the corresponding AST but does not blacklist the function because the body is valid. Now the analyzer encounters *f()* and again saves the body and does not blacklist the function. On the second encounter of *evil()* the functions gets blacklisted. Now the call to *f()* is encountered and the saved body for *f* is evaluated anew. The analyzer encounters the call to evil, which is now blacklisted, and marks the source code as invalid. Without the saving mechanism for the function bodies f would not have been evaluated a second time and invalid code would not have been detected.
Objectfunctions are handled heuristically, because it is not trivial to detect which type or class a variable represents with only the given AST. Upon calls the analyzer checks if the call's `func.name` is of the type `ast.Attr` or `ast.Name`. Object functions in Python have a `func.name` that is an instance of `ast.Attr`. If it is a call to an object function a separate objectfunction-blacklist is checked. This heuristic method fails if two classes each implement a function with the same name. The probability of this happening is relatively low, because the setup-file is typically very small in the range of 10-20 kB and often has no class definitions.
Variable assignments are blacklisted according to the validity of the expression which is assigned to the variable.

5.2.2 Results of the Analysis

Every setup-file deemed valid by the analyzer can be automatically replaced by metadata because it only uses a small subset of the python language. An analysis of 112690 files showed the following results as seen in figure 5.1. 26% of the setup-files are invalid. Additionally the calls which caused a file to become invalid have been recorded. Following are the top 10 calls showing in figure 5.2.
Although this option has not been researched further it is likely that the calls to the `os` module are made to ensure multi platform compatibility. If that would be the case the percentage for invalid setup files would be lowered to 15%.

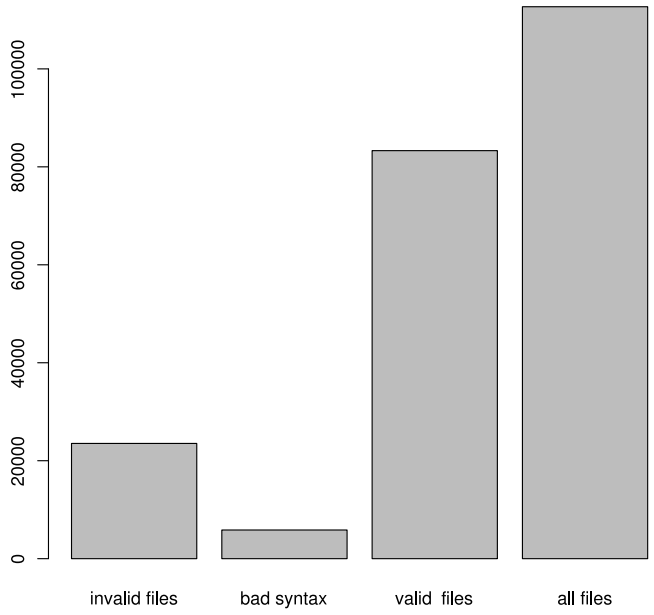

Figure 5.1: Barplot of setupfile Distribution

Figure 5.2: Top 10 calls causing invalid files

Name of call	**Nr. of files invalid because of call**
os	11991
sys	1793
os.path	998
use_setuptools	631
distribute_setup	556
re	507
ConfigParser	502
subprocess	337
glob	318
paver.tasks	316

The other "forbidden" calls are not present in a significant number and can thus be ignored from further research.

The Bad Syntax cases refer to analysis cases where python threw a *SyntaxError*. Most of those errors - 5687 - are caused by Python 3 incompatible syntax, for example *print*

"python2 print statement". 103 of the errors are due to invalid tokens for example padding decimal numbers *a = 09*. 45 errors are caused by repeated keyword arguments in calls or function declaration for example *examplefunction(argument=foo, bar=baz, argument=pyrate)*. 13 come from inconsistent indentation, 3 are caused by an assignment to a reserved keyword and another 3 by invalid utf-8 encoding.

5.3 Solver comparison

A valid question to ask in the scope of this thesis is how good available PBO solver perform in the context of solving python module dependencies.
All available solvers that participated in the *Pseudo Boolean Competition 2012* [RM13b] were used in this comparison. The test set is a collection of the 100 most popular python modules on *PyPI* (and *adhocracy*) according to [htt13]. Because there is no ground truth data available for python dependency resolution problems and because a "good" solution for a dependency problem is to a grade subjective a custom metric was used.
The *(p)wbo* solver provided "good" solutions in practice and is used as a reference point. Every solver starts with a score of 0. For every dependency instance that was deemed unsolvable by that solver 50 points are deducted from the score of the solver. For every instance every solver had at maximum 10 seconds, after which the process running the instance was killed and the instance was deemed unsolvable. If the instance is declared solvable then *dpoints* are calculated as following and added to the score:

$$xweight = \frac{100}{nr.\ of\ changes\ proposed\ by\ pwbo}$$

$$dpoints = 100 - xweight \times (nr.\ of\ deviations\ from\ pwbo\ solution)$$

Te results are shown in 5.3 and 5.4. *wbo* and *npSolver-pbo* seem to be the only solver, which are capable of solving all of the 101 dependency resolution instances. The other solvers perform fairly well, but fail in 5 cases - typically if the number of clauses in the instance is too high. The score shows that all instances which *npSolver-pbo* solved were of equal quality as the ones *wbo* provided. The instances which were solved during the timeout period by the other solvers have to be comparable to those of *wbo*, because 250 points have to be deducted for the 5 missing cases and the maximum score for 96 solved

Figure 5.3: Cases solved and corresponding solver

Solver	Instances solved out of 101
(p)wbo	101
npSolver-pbo	101
sat4j-pb	96
clasp-2.1.3	96
minisat+	96
pb2sat+zchaff	96

Figure 5.4: Solver and points after tests

Solver	Score
(p)wbo	10100
npSolver-pbo	10100
pb2sat+zchaff	9450
sat4j-pb	9350
clasp-2.1.3	9350
minisat+	9350

cases is 9600. Solver speed was not measured for the score because it did not differ much in the solvable cases.

5.4 Synthethic generation of pbo Instances

Another question that was asked is if it is possible to generate PBO instances which can not be solved by the previously tested solvers. If synthetic generation is possible hints can be drawn from the generated data to improve the solver.
However there is currently no formula which iteratively generates PBO instances that increase in difficulty with each iteration. The procedure that was used instead works with PBO data obtained from Python dependency resolution problems. 96 solvable PBO instances are available from the solver comparison in the last chapter. The main idea is to create a metapackage and then iteratively add one of those packages as a dependency to determine a boundary where the formula is not solvable in under 10 sec anymore.
The test was done with the *minisat+* solver and the CPU time for obtaining a solution was measured. Surprisingly the needed CPU times were not getting larger with increas-

ing number of dependencies added to the metapackage. This shows that adding more clauses to a PBO formula does not make it necessarily more complex or harder to solve - probably because adding more clauses and literals causes some (trivial) optimizations built into the solver to trigger. However 6 instances which can not be solved by all solvers were acquired during the solver comparison. To help future research those instances have been sent as a benchmark to the Pseudo Boolean Competition.

Chapter 6

Conclusion

This thesis researched improvements of Python packaging. Weighted boolean optimization formulas, a special case of constraint solving, where the theoretical background for the implementation of a Python dependency solver called *PyDYN* and its embedding into the installation tool *pip*. *PyDYN* solves the dependency Resolution Problem for Python modules, using a database filled with metadata acquired from all modules online on *PyPI* in May 2013. *PyDYN* improves the workflow of installation tools like pip or buildout. During the implementation of *PyDYN* a downloader tool with cashing abilities and a metadata database for all modules available at the python package index was written.
Furthermore an analysis of all publicly available python packages was done to determine whether it is feasible to abandon an installation process dependent on the setup.py file, which can run arbitrary, potentially harmful, code.
The main contribution of this thesis was presenting the fact that, with the help of a proof of concept, weighted boolean optimization is a fast and reliable way to compute dependency resolution problems for Python packages.

Future work in the field of improving Python dependencies could be the creation of a metadata format [CHS13], which would render the setup.py installation process obsolete. Also a new method for storing the module files on the local machine could be thought of, so that usage of virtual environment tools like *virtualenv* would not be necessary anymore if two projects on one machine need the same module in different versions.

Bibliography

[ABL+10] ARGELICH, Josep; BERRE, Daniel L.; LYNCE, Inês; MARQUES-SILVA, Joao; RAPICAULT, Pascal: Solving Linux upgradeability problems using boolean optimization. In: *arXiv preprint arXiv:1007.1021* (2010).

[Bur05] BURROWS, Daniel: Modelling and resolving software dependencies. In: *Conferenza Italiana sul Software Libero (CONFSL 2010)*, 2005.

[CHS13] COGHLAN, Nick; HOLTH, Daniel; STUFFT, Donald: *PEP 426 Metadata for Python Software Packages 2.0*. `http://www.python.org/dev/peps/pep-0426/`. Version: 2013.

[gra13] GRAPHVIZ.ORG: *The DOT Language Reference Guide*. `http://www.graphviz.org/doc/info/lang.html`. Version: Mai 2013.

[htt13] HTTP://TAICHINO.COM: *PyPI Ranking*. `http://pypi-ranking.info`. Version: 2013.

[KM13] KAPLAN-MOSS, Jacob: *Download of the python-nation package*. `https://pypi.python.org/simple/python-nation/`. Version: 2013.

[LBR09] LE BERRE, Daniel; RAPICAULT, Pascal: Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. In: *Proceedings of the 1st international workshop on Open component ecosystems* ACM, 2009, S. 21–30.

[RM13a] ROUSELL, Olivier; MANQUINHO, Vasco: *Input/Output For-*

mat and Solver Requirements for the Competitions of Pseudo-Boolean Solvers. `http://www.cril.univ-artois.fr/PB12/format.pdf`. Version: Mai 2013.

[RM13b] ROUSELL, Olivier; MANQUINHO, Vasco: *Pseudo Boolean Competition Homepage*. `http://www.cril.univ-artois.fr/PB12/`. Version: Mai 2013.

[Tea13] TEAM, Red Dove D.: *Python Metadata Online*. `http://www.red-dove.com/pypi/projects/`. Version: 2013.

[TLO10] TREZENTOS, Paulo; LYNCE, Inês; OLIVEIRA, Arlindo L.: Apt-pbo: solving the software dependency problem using pseudo-Boolean optimization. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering* ACM, 2010, S. 427–436.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 15.Juli 2013 Alexander Schneider

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- **[anpassen]** die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software XYZ
- **[anpassen]** den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen