Conflict-aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph

Wei Cheng
State Key Laboratory for Novel
Software Technology
Nanjing University, China
wchengcs.nju@gmail.com

Xiangrong Zhu
State Key Laboratory for Novel
Software Technology
Nanjing University, China
xrzhu.nju@gmail.com

Wei Hu*
State Key Laboratory for Novel
Software Technology
Nanjing University, China
whu@nju.edu.cn

ABSTRACT

Code sharing and reuse is a widespread use practice in software engineering. Although a vast amount of open-source Python code is accessible on many online platforms, programmers often find it difficult to restore a successful runtime environment. Previous studies validated automatic inference of Python dependencies using pre-built knowledge bases. However, these studies do not cover sufficient knowledge to accurately match the Python code and also ignore the potential conflicts between their inferred dependencies, thus resulting in a low success rate of inference. In this paper, we propose PyCRE, a new approach to automatically inferring Python compatible runtime environments with domain knowledge graph (KG). Specifically, we design a domain-specific ontology for Python third-party packages and construct KGs for over 10,000 popular packages in Python 2 and Python 3. PyCRE discovers candidate libraries by measuring the matching degree between the known libraries and the third-party resources used in target code. For the NP-complete problem of dependency solving, we propose a heuristic graph traversal algorithm to efficiently guarantee the compatibility between packages. PyCRE achieves superior performance on a real-world dataset and efficiently resolves nearly half more import errors than previous methods.

KEYWORDS

Python, Runtime environment inference, Knowledge graph, Conflict resolution, Dependency solving, Configuration management

1 INTRODUCTION

With the rise of programming communities such as GitHub and StackOverflow, code sharing and reuse have become a common

*Corresponding author

practice for programmers [25]. Python is one of the most popular high-level programming languages today, due in part to its massive third-party package resources, which also often cause environment configuration issues. An empirical research [24] observed that the usability rate of all the Python code snippets on StackOverflow is 76% parsable and 25% runnable, and the work in [9] further found that 75.6% of the Python code snippets shared through GitHub are not executable with over half of failures due to missing dependencies in a clean Python 2 environment.

Executing a Python code that contains third-party resources in a clean Python environment triggers dependency errors, which raise the built-in exception `ImportError`. To resolve this, programmers need to specify the packages and desired versions in a configuration script such as *requirements.txt*. However, this is not a trivial work. The study in [9] observed that programmers usually spend between 20 minutes and two hours to set up the environment, and in some cases, they even cannot restore a correct execution environment. Therefore, automatically inferring the runtime environments of Python code helps, which can free up the time of programmers spent on dependency issues, and thus is significant for code reuse and automated software configuration management.

However, there are several challenges to automatic inference of Python runtime environments. Let us see a Python code snippet shown in Figure 1a. The imported top-level modules, namely *redcap*, *influxdb*, *openfisca_core* and *gpkit*, are not built-in modules in the Python standard library. Executing `pip install redcap` receives the error message `ERROR: No matching distribution found for redcap`, because *redcap* does not exist on Python Package Index (PyPI) and the package corresponding to it is *pycap*. In fact, it is common in practice that the name of a module imported in Python code does not match the name of the Python package it belongs to. Moreover, after a successful installation with `pip install influxdb`, another error message `ImportError: cannot import name InfluxDBClusterClient` appears, which indicates that the latest version *influxdb-5.3.1* does not contain this attribute. In fact, the last version containing attribute *influxdb.InfluxDBClusterClient* is *3.0.0*.

A naive approach installs the Python packages with the same names as the imported top-level modules. However, this approach fails to infer correct dependencies in many cases, as illustrated in the above example. The challenges are essentially the lack of sufficient domain knowledge. DockerizeMe [10] builds an offline knowledge base semi-automatically to infer the environment dependencies for Python code snippets. V2 [11] searches working environments with proper versions based on the error messages of code execution. SnifferDog [21] builds an API bank of Python packages to infer the

```
1   import urllib2
2   import redcap
3   from influxdb import InfluxDBClusterClient as cli
4   import openfisca_core.simulations
5   from gpkit import units, Variable, Model
6   from gpkit.tools.autosweep import autosweep_1d
7
8   client = cli.from_DSN('influxdb://usr:pwd@host1:8086')
9
10  A = Variable("A", "m**2")
11  l = Variable("l", "m")
12  m1 = Model(A**2, [A >= l**2 + units.m**2])
13  tol1 = 1e-3
14  bst1 = autosweep_1d(m1, tol1, l, [1, 10], verbosity=0)
15  print("Solved after %2i passes, cost logtol +/-%.3g" % (bst1.nsols, bst1.tol))
```

(a) Python code: snippet.py

```
1   numpy==1.15.4
2   openfisca-core==25.2.5
3   pycap==1.1.1
4   gpkit==0.9.9.2
5   influxdb==3.0.0
```

(b) requirements.txt

```
1   FROM   python:2.7.18
2
3   COPY   requirements.txt /
4   RUN    pip install -r /requirements.txt
5
6   COPY   snippet.py /snippets/snippet.py
7   CMD    python /snippets/snippet.py
```

(c) Dockerfile

Figure 1: A motivating example.

specific versions and restore the execution environments of Jupyter notebooks. These previous studies have achieved good performance by using pre-built knowledge bases, but they do not cover sufficient knowledge to accurately match more complex Python code. In this paper, we design an elaborated ontology for Python third-party packages and automatically construct Python package knowledge graphs (KGs) by installing and analyzing the releases on PyPI.

After building the domain KGs, we are still challenged by how to match the target code with Python dependencies. To cope with this, we design a novel metric of matching degree and treat all attributes under a top-level module as a whole to better discover required libraries. Furthermore, dependency conflicts occur when different inferred packages depend on the same package, but specify different and incompatible versions of that package. Continuing to consider the code shown in Figure 1a, installing the latest versions of packages *openfisca-core* and *gpkit* causes a dependency conflict, because *openfisca-core-25.2.5* requires *numpy<1.16,≥1.11* and *gpkit-0.9.9.1* requires *numpy≥1.16.4*. Thus, we should choose an older version *gpkit-0.9.9.2*, which requires *numpy≥1.13.3* and is compatible with other packages. Dependency solving should ensure that all direct dependencies and transitive dependencies (i.e. dependencies of dependencies) are compatible with the rest of inferred environment. To the best of our knowledge, all previous studies [10, 11, 21] have not considered the compatibility of the inferred Python environments yet. Due to the NP-completeness of dependency solving [14], we propose a heuristic graph traversal algorithm to infer a compatible environment, which efficiently selects the newer versions and prunes the traversal paths.

The main contributions of this paper are listed as follows:

- We design an ontology for Python third-party packages and an automatic approach to KG construction. As a result, we create the Python package KGs for Python 2 and Python 3, each of which contains the knowledge of over 10,000 Python packages and nearly 300,000 versions. (Sections 3)
- We define a new metric of matching degree between Python libraries and third-party resources in target code to discover required libraries. Moreover, we consider the compatibility of the inferred Python environments, and design an efficient heuristic algorithm for dependency solving. (Sections 4)

Figure 2: Overview of our approach.

- We implement our approach called PyCRE (https://github.com/nju-websoft/PyCRE) and evaluate it with 10,250 real-world Python code snippets on Gistable [9]. Our experiments show that PyCRE efficiently resolves dependency issues for both Python 2 and Python 3, leaving only 1,524 ImportError, which is significantly superior to 2,654 ImportError of the state-of-the-art approach [11].

2 OVERVIEW OF PYCRE

Figure 2 depicts an overview of PyCRE, which automatically infers Python compatible runtime environments with the pre-built Python package KGs. The fundamental requirement of PyCRE is to cover as many third-party resources used in the target code as possible. Additionally, PyCRE directly specifies the packages and their versions in a feasible installation order to avoid the potential conflicts between inferred dependencies.

PyCRE consists of two phases, where the upper portion of the figure shows the construction of our Python package KGs. All we can know from target code are the imported modules and the called attributes, including the names of variables, classes, functions and even hidden submodules. See Figure 1a for example, matching the third-party resources in target code to the correct libraries requires a great deal of domain knowledge. According to our designed ontology, we offline construct two Python package KGs for Python 2 and Python 3, respectively.

The bottom portion shows the automatic inference of compatible runtime environments. With the Python package KGs, PyCRE

parses a target Python code, discovers its candidate libraries according to matching degrees, and generates the compatible runtime environment through dependency solving.

The output of PyCRE is a *requirements.txt* containing a list of required Python packages with specific versions in a correct order and a *Dockerfile* containing the inferred Python version. The dependencies in *requirements.txt* work together, reducing the risk of dependency conflicts compared to installing dependencies individually. For example, Figure 1b shows the *requirements.txt* generated by PyCRE for the Python code shown in Figure 1a. The *Dockerfile* shown in Figure 1c installs all Python packages with the command pip install -r requirements.txt.

3 PYTHON PACKAGE KNOWLEDGE GRAPH

Domain knowledge of Python packages is essential for automatic inference of compatible runtime environments. We devote our efforts to designing an ontology for Python third-party packages and a method to automatically construct the corresponding KGs.

3.1 Python Package Ontology Design

As shown in Figure 3, we define an ontology to represent relationships between entities and properties for describing entities:

- **Package node.** Each package node represents a Python package and stores the package's *name* as a property. The stored package names are normalized and unified, making no two package entities having an identical name.
- **Version node.** Each version of a Python package is stored as a distinct version node. A version node contains its standard *version* identifier and *install_status* of the corresponding release. There are three values for *install_status*: *Success*, *Fail* and *Unknown*, where *Unknown* means that the version has not been installed yet.
- **Module node.** Each module node corresponds to a specific module of a version and has two properties. Property *import_status* takes the value *True* or *False*, indicating whether the module can be successfully imported or not. Another property is the fully-qualified *name*, e.g., *client* is a submodule of module *redis*, and its name is stored as *redis.client*. However, module names are not unique, as different versions of a package may contain homonymous modules that have different attributes, or even different packages may have homonymous modules.
- **Attribute node.** Each attribute node stores the attribute's *name* as a property. Unlike module nodes, attributes with an identical name are defined as a single entity in the ontology. For example, attribute *redis.client.Redis* is saved as attribute *Redis* of module *redis.client*. In our ontology, attribute can be variable, function, class or any content available in the corresponding module.
- **Package → Version:** *has_version* edge, which indicates the relationship between the package and its version.
- **Version/Module → Module:** *has_module* edge. Each successfully installed version has modules in principle, and modules may have their submodules.
- **Module → Attribute:** *has_attribute* edge, which shows that the attribute is available in the module.
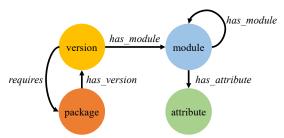

Figure 3: Entities and relationships defined in the Python package ontology.

- **Version → Package:** *requires* edge, which represents that the package is a direct dependency of the version, and has a *requirement* property to store the version specifier.

3.2 Knowledge Acquisition

To extract knowledge, we first determine the Python packages that we want to analyze and get all available versions. Then, we record the installation status by installing each version of the packages. At last, we acquire the modules, attributes and direct dependencies of those versions that are successfully installed. The whole process of knowledge acquisition is automated.

- **Packages.** A list of available distributions on PyPI can be obtained according to PyPISimple.[1] Due to the expense of time and storage, we usually specify a list of packages for further knowledge acquisition. With the knowledge of commonly used Python packages in a certain domain, the success rate of inferred Python compatible runtime environments can be greatly improved.
- **Versions.** We get all available versions of each package by executing pip install <package>==<version*>, since many packages are not available in exactly the same versions under different Python releases. <version*> is a special version identifier that does not exist, which makes pip output all available versions.
- **Installations.** Some releases have strict requirements for the supported Python versions or even require system-level dependencies that pip cannot handle. Thus, not every release on PyPI can be installed successfully. We attempt to install each version of a package with pip install <package>==<version> and record the installation status.
- **Requirements.** Most releases require certain direct dependencies to be installed before their installations, which are stored in their metadata. We get the dependency requirements of each successfully installed release from its *META-DATA* file. Dependency requirements prompt a lot of unknown Python packages, due to the incompleteness of our KG. We create Package nodes for these packages, but without doing further knowledge acquisition.
- **Modules and attributes.** A Python distribution usually has multiple modules. For each successfully installed distribution, we first attempt to get its top-level modules from *top_level.txt*. If that file does not exist, we try to check all created directories and files to obtain its top-level modules. We then find all

---

[1]https://wiki.python.org/moin/PyPISimple

submodules of each top-level module recursively. Finally, we try to import each module and use Python built-in function *dir()* to attain all attributes of the imported modules. We remove the modules and attributes that start with the underscore because they are conventionally intended for internal use.

After constructing the Python package KGs, we can periodically check for new packages and versions based on the above process to incrementally upgrade our KGs. Note that PyPI doesn't support the replacement of existing releases but only deletion (we can also delete the corresponding entities and relations in our KGs), thus ensuring the consistency of our KGs.

4 ENVIRONMENT INFERENCE

For environment inference, we first obtain the imported third-party modules and called attributes by parsing the target code. Then, we query our KGs to discover candidate libraries that best match these modules and attributes. Finally, we expand a dependency graph with transitive dependencies of candidate libraries, and infer the installation instructions of compatible dependencies in order by dependency solving.

4.1 Target Code Parsing

As the only input, we assume that the code should be fully parsed. To determine which third-party libraries to be installed, we find all the imported modules that are not in the Python standard library and the called attributes in those modules. We first parse the target code into an abstract syntax tree (AST) in a clean Python environment, and then walk in the AST for both of the following information:

- **Imported modules**. The syntax for the `import` statement in Python is `import <module> as alias` or `from <module> import <name> as alias`. The modules in the Python standard library are ignored, and we store all possible third-party modules as *imported modules*. It is worth mentioning that `<name>` can be a submodule, function, class, variable or any attribute that can be accessed in the module. Additionally, programmers can bind an imported resource with an optional alias name and use the alias name directly in the code, but at the same time, the original name of the imported resource would no longer be used. We record the mappings between the imported resources and their alias names.
- **Called attributes**. In addition to the imported modules, the attributes of those modules used in the code are also crucial for discovering the candidate libraries. We visit each attribute node in the AST and record all the attribute names prefixed with an imported resource or its alias. According to the mapping obtained from the `import` statement, we then map the prefix back to the corresponding imported resource name, thus restoring its fully qualified name for each attribute. For example, the fully qualified name of attribute *from_DSN* is *influxdb.InfluxDBClusterClient.from_DSN* (Line 8 in Figure 1a). Due to the uncertainty in `from <module> import <name>`, we treat `<module>.<name>` as a possible attribute name as well. We store these attribute names as *called attributes*.
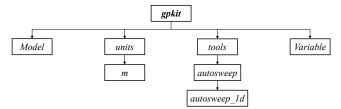

Figure 4: A parse tree generated by target code parsing.

Algorithm 1: $CalculatesMatchingDegree(S, L)$

Input: set S of resources, list L of resources in the library
Output: matching degree between S and L

1   $degree \leftarrow 0$;
2   **foreach** $res \in S$ **do**
3     $pre \leftarrow$ longest prefix reference path of res matched in L;
4     $degree \leftarrow degree + \frac{PathLength(pre)}{PathLength(res)}$;
5   **return** $degree$;

We represent the *imported modules* and *called attributes* as a forest. Each tree in the forest has a top-level module name as a root node, and contains all the submodules and attributes of that module in the code. For instance, by parsing the example Python code in Figure 1a, the parse tree of module *gpkit* is shown in Figure 4.

The results of target code parsing can be used to infer the candidate Python versions. We parse the code in Python 2 and 3, respectively, and the Python versions with the fewest *imported modules* and no syntax errors are the candidates. For example, the imported module *urllib2* (Line 1 in Figure 1a) is a standard library in Python 2 and has been split into several modules in Python 3 named *urllib.request* and *urllib.error*. The sample code imports fewer nonstandard modules in Python 2 than in Python 3, and thus its Python version is inferred. If the number of *imported modules* is equal, we would extrapolate further in the following.

4.2 Candidate Library Discovery

In this phase, for each candidate Python version, we query the corresponding Python package KG to find the candidate libraries that best match the forest obtained by code parsing.

For the resources (i.e. modules or attributes) used in the code, it is more reasonable to use partial matching than to precisely query the fully qualified names in the KG due to incomplete knowledge. However, widely-used string similarity metrics such as edit distance and Jaccard coefficient [5] are not applicable here, as a fully qualified name represents a reference path, e.g., *openfisca_core.simulations* and *sapphire.simulations* are two completely unrelated modules. We define a metric based on the longest prefix match of the reference path to calculate the matching degree between a list of resources and a library, as shown in Algorithm 1. Reference paths in Python are separated by dots. For example, the length of the reference path for resource *influxdb.InfluxDBClusterClient.from_DSN* is 3. Although the release *influxdb-5.3.1* has attribute *influxdb.InfluxDBClient*, its longest prefix reference path is *influxdb* and matching degree is $\frac{1}{3}$.

We discover candidate libraries separately for each parse tree in the forest according to the following steps:

S1. We query the root node of the parse tree in the KG, which refers to a top-level module. If there are module entities with the same name in the KG, we directly go to S2 with the query results. Otherwise, we attempt to install a Python package with the same name as the top-level module.

S2. There may be different versions or even different packages that have the same top-level module, so we filter these candidates further by submodules. We set max-hop as the depth of the parse tree and query our KG to obtain spanning trees reachable from the top-level modules following the *has_module* relationships to max-hop. Then, we calculate the matching degree between the imported modules and the modules with *import_status* as *True* in each spanning tree. The candidate spanning trees with the highest matching degree are retained.

S3. Since the version updates may cause the addition or removal of attributes, we need to select the proper versions. We enrich the spanning trees by querying the attributes of modules in the parse tree. Similarly, we calculate the matching degree between the called attributes and the resources in each spanning tree. The candidate trees with the highest matching degree are retained, and their corresponding libraries are optimal for the parse tree.

After our discovery process, we finally determine the candidate libraries of each parse tree in the forest, which may correspond to multiple versions of more than one package. Meanwhile, the Python version with the maximum matching degree is selected. Python 3 is the default if their matching degrees are equal.

4.3 Dependency Solving

To cover most third-party resources used in the target code, each top-level module should have at least one candidate library in the inferred runtime environment. Furthermore, we expect each package in the inferred environment to meet all its version constraints, which is called *dependency solving*.

Dependency solving is a hard problem in all non-trivial component models and has been proved NP-complete [1, 14]. Existing work treats dependency solving as a separate concern in package manager [2] and relies on generic dependency solvers based on the tried-and-tested techniques such as solvers of the Boolean satisfiability problem (SAT) [3]. There are efficient modern SAT solvers [7, 13, 18], but it is nearly impossible to control the answers provided by the solvers with many priorities. For optional versions of a Python package, we prefer the latest version that can be installed successfully, since the latest version is usually downward compatible with previous versions and contains new resources. For time efficiency and compliance with version selection priorities, we propose an efficient heuristic graph traversal algorithm.

We create a start node, which points to all the virtual nodes that represent the top-level modules. We query our KGs for all transitive dependencies of the candidate libraries, forming a heterogeneous directed graph called a *dependency graph*. The dependency graph would be iteratively extended until there are no more dependencies. The nodes in the dependency graph are divided into conjunction nodes and disjunction nodes. Conjunction nodes include the start node and version nodes, which depend on all its direct successors.

Disjunction nodes include module nodes and package nodes, which require at least one direct successor. Moreover, each edge pointing to a package node is attached with specific dependency requirements, which restrict the choice of its versions. The versions that fail to be installed during knowledge acquisition are not considered.

Definition 1 (Compatible dependency subgraph). Let $G = (V, E)$ be a dependency graph, where V denotes the vertex set including packages, versions and modules, and E denotes the directed edge set based on the dependency relationships. A compatible dependency subgraph $G' = (V', E')$ of G satisfies:

(i) $V' \subseteq V, E' \subseteq E$, and start node $s \in V'$.

(ii) We denote edge $a \rightarrow b$ by (a, b), and regard a specific version requirement as a collection of versions that satisfy the requirement. Compatibility requires:
- For each conjunction node $c \in V'$, $\forall e = (c, n) \in E, e \in E'$.
- For each disjunction node $d \in V'$, let $S = \{e \mid e = (d, n) \in E\}$, $|E' \cap S| \geq 1$ ($= 1$, if d is a package node).
- For each package node $p \in V'$, let $R = \{$requirement of $e \mid e = (i, p) \in E'\}$, $\exists (p, n) \in E', n \in \bigcap_{r \in R} r$.

SAT solver. With the dependency graph, each node corresponds to a Boolean variable and the constraints are encoded as a Boolean formula in conjunctive normal form (CNF). The constraints of a dependency graph are translated into Boolean clauses in CNF according to the following rules:

- Start node s is set to *True*, denoted by (s).
- A conjunction node c decides all its direct successors $x_1, x_2, \ldots, x_n$: $c \rightarrow (x_1 \wedge \cdots \wedge x_n) \equiv \bigwedge_{1 \leq i \leq n} (\neg c \vee x_i)$. Conjunction nodes include the start node and version nodes.
- A disjunction node d decides at least one of its direct successors $x_1, x_2, \ldots, x_n$: $d \rightarrow (x_1 \vee \cdots \vee x_n) \equiv (\bigvee_{1 \leq i \leq n} (x_i) \vee \neg d)$. Disjunction nodes include module nodes and package nodes.
- A virtual module node m requires at least one of its candidate versions $v_1, v_2, \ldots, v_n$: $m \rightarrow (v_1 \vee v_2 \vee \cdots \vee v_n) \equiv (\bigvee_{1 \leq i \leq n} (v_i) \vee \neg m)$. Meanwhile, each candidate version v_i requires its package p_i: $\bigwedge_{1 \leq i \leq n} (\neg v_i \vee p_i)$.
- A package node only has one of its versions $v_1, v_2, \ldots, v_n$: $\bigwedge_{1 \leq i < n, i < j \leq n} (\neg v_i \vee \neg v_j)$.
- A version node v is incompatible with versions $v_1, v_2, \ldots, v_n$ of its direct dependencies that do not meet the specific requirements: $v \rightarrow (\neg v_1 \wedge \neg v_2 \wedge \cdots \wedge \neg v_n) \equiv \bigwedge_{1 \leq i \leq n} (\neg v \vee \neg v_i)$.

If the formula is satisfiable, the resulting compatible dependency subgraph would consist of all the nodes with value *True* and the edges between them.

Our heuristic algorithm. The details of our algorithm are shown in Algorithm 2 and we start traversing from the start node with an empty subgraph. With the depth-first search (DFS), our algorithm has two heuristic strategies: (i) Priority: we prefer the latest version (Line 13) that can be successfully installed by sorting all compatible versions (Line 8); (ii) Pruning: if one version of a package is incompatible, we skip the other versions of that package with identical dependency requirements (Line 15), because they would cause the same conflicts. When the currently selected version of a package does not satisfy the newly added version constraints, the current version and all its direct dependencies and transitive dependencies need to be removed from the subgraph (Line 11). We

Algorithm 2: $ExtendsSubgraph(G', n, p)$

Input: subgraph G' to be extended, current traversed node n, direct predecessor p of n in this visit

Output: *True* or *False* (indicating whether a compatible subgraph can be found.)

1   $tmpG \leftarrow G'$, and add node n and edge (p, n) to $tmpG$;

2   **if** *n is a conjunction node* **then**

3   **foreach** *direct successor c of n* **do**

4   **if** $\neg ExtendsSubgraph(tmpG, c, n)$ **then**

5   **return** *False*;

6      $G' \leftarrow tmpG$; **return** *True*;

7   **else**

8      $C \leftarrow$ version-sorted direct successors of n which are compatible in $tmpG$;

9   **if** *exists edge* $(n, d) \in tmpG$ **then**

10   **if** $d \in C$ **then** $G' \leftarrow tmpG$; **return** *True*;

11   **else** remove (n, d), and recursively remove the dependencies of version d from G';

12   **while** *C is not empty* **do**

13   **if** $ExtendsSubgraph(tmpG, C[0], n)$ **then**

14           $G' \leftarrow tmpG$; **return** *True*;

15   **else** remove $C[0]$ and the elements that have the identical dependency requirements in C;

16   **return** *False*;

---

only backtrack on the current search path when we encounter a dependency conflict, which is highly efficient but in rare cases may miss solutions. To address it, we call a SAT solver as the fallback to ensure the completeness of our algorithm when it claims that there is no solution.

Let us see the dependency graph in Figure 5. According to DFS, our algorithm first traverses to *openfisca-core-25.2.5*, depending on *numexper-2.6.8* that requires *numpy≥1.7*. Thus, the latest version *1.16.6* is chosen for *numpy*. When considering the direct dependency of *openfisca-core-25.2.5* on *numpy*, it is found that *numpy-1.16.6* does not meet the new version requirements *<1.16,≥1.11*. So, *numpy-1.16.6* is removed (it does not depend on other packages) and a compatible version *1.15.4* is reselected. When traversing to the package *gpkit*, we prefer the latest version *0.9.9.9.1*. However, its version requirement ≥ *1.16.4* for *numpy* conflicts with the previous requirement *<1.16,≥1.11*, so we reselect the version of *gpkit*. We skip version *0.9.9.9*, which has the identical requirement for *numpy* as the incompatible version *0.9.9.9.1*, and traverse to version *0.9.9.2*. The version requirements of *gpkit-0.9.9.2* are compatible and the current version of *numpy* meets all the requirements. Now, we find a compatible dependency subgraph using our heuristic algorithm.

At last, we identify the packages that need to be explicitly installed as well as the installation order. After removing the start node and module nodes in the subgraph, all packages with an indegree of 0 are required to be installed explicitly, since they are not the dependencies of any packages. Moreover, for a package, if the latest version that meets its version constraints is different from the selected version, it would probably lead to an unexpected version,
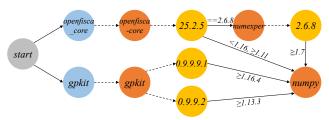

Figure 5: Partial display of a dependency graph. Node colors are in accord with the colors of entity types in Figure 3 except for the start node. Nodes with dashed outgoing edges are conjunction nodes, and nodes with solid outgoing edges are disjunction nodes. Best viewed in color.

so we install this package explicitly as well. The installation order of all packages is generated by topological sorting, which ensures that all dependencies of each package are installed ahead of time.

5 EVALUATION

5.1 Experiment Settings

Dataset. We conduct our experiments on the Gistable [9] dataset, which is a real-world dataset built on the GitHub's gist system and contains 10,250 single-file Python code snippets. DockerizeMe and V2 have also been evaluated on this dataset.

Comparative methods. In addition to our PyCRE, several comparative methods are assessed in the experiments:

- Gistable [9], which attempts to install the Python packages with the same names as the imported top-level modules. It is a baseline method and does not use any knowledge.
- DockerizeMe [10], which infers the runtime environments based on a pre-built knowledge base, without considering Python versions nor specific dependency versions.
- V2 [11], which reuses the inference results of DockerzieMe as the starting environments in Python 2 and Python 3, and guides the version changes by the error messages of code execution until finding a working environment.
- SnifferDog [21], which uses a pre-built API bank to restore the execution environments of Jupyter notebooks.

Prior knowledge. For the fairness of our experiments, all methods should acquire knowledge for the same Python packages. We obtain a list of Python packages from the knowledge base built by DockerizeMe [10]. According to Python Enhancement Proposals (PEP) 508 [6], Python package names are case-insensitive and do not distinguish between dash, dot and underscore. So, we merge the duplicate packages such as *Flask-RESTful* and *flask_restful*. Eventually, we have 10,765 unique Python packages for knowledge acquisition.

The knowledge base built by DockerizeMe for the 10,765 Python packages is available. V2 uses the inference results of DockerizeMe as the starting environments, so it can be ignored. It is worth mentioning that the knowledge acquisition of DockerizeMe is not fully automated but requires manual work. SnifferDog only acquires knowledge for a special selection of 488 Python packages. In this paper, we build its API bank for the 10,765 packages using its publicly available code. We also automatically construct our KGs for Python 2 and Python 3 by the method described in Section 3.2. The

Table 1: Statistics of the exit status of code execution on the Gistable dataset.

Exit status	Python 2	Python 3
Success	2,112	1,293
Timeout	132	71
ImportError	5,515	3,536
SyntaxError	719	4,388
Other Exceptions	1,772	3,212

knowledge of PyCRE and other methods is stored in a Neo4j database, except for SnifferDog, which is loaded directly into memory according to its design.

Validation environment. We leverage Docker to ensure that each Python code executes in an isolated environment. For Python 2 and Python 3, our Docker images are initialized on the official images *python:2.7.18* and *python:3.8.11*, respectively. We update pip to the latest version, currently 20.3.4 and 21.2.4 for Python 2 and Python 3, respectively. We configure pip to use its new dependency resolver by default, which refuses to install the packages with incompatible requirement combinations. We conduct all experiments on Ubuntu 18.04 LTS with one Intel Xeon Gold 5117 CPU and 16GB memory.

Experiment procedure. Given that only our PyCRE and V2 can infer the Python version for target code and V2 cannot specify a fixed Python version, we conduct the experiments in three environments: Python 2, Python 3 and the inferred Python versions.

For each method, we first infer dependencies for each code in the dataset and record the time spent on the inference to analyze the efficiency. The timeout of inference is set to one hour, since V2 is a dynamic inference method, which needs to repeatedly roll back the versions and generate the validation environments. The code to infer dependencies in SnifferDog is incomplete, so we replicate it according to their paper for enabling SnifferDog to infer the dependencies of Python code.

Then, we record the inferred Python dependencies in *requirements.txt* in order and build the Docker image according to the experimental settings. To avoid the failures due to network fluctuation or other factors, we give ten minutes for each Python package installation when building each Docker image for validation. Failures to install dependencies when building images are ignored, which is in line with the behavior of programmers.

Finally, we run the code in the corresponding Docker container. The code snippets that run successfully are marked as `Success`. We mark the code snippets running for more than one minute as `Timeout`, because some code may not stop running due to awaiting input, encountering dead loops, etc. The others are marked as `Exception` and the names of their exceptions are recorded. Since V2 only returns dependencies when it finds a working environment, for each failed code, we select the exception name of the last validation from its output logs. Built-in exception `ImportError` is raised when an `import` statement fails to find the module definition or when a `from ... import` statement fails to find the name to import. We also include Python 3's built-in exception `ModuleNotFoundError` as `ImportError`, which is a subclass of `ImportError` and raised by `import` when a module cannot be located.

Table 2: Statistics of our Python package KGs.

Entity	Python 2	Python 3	Relationship	Python 2	Python 3
Package	12,948	13,439	*has_version*	291,805	307,841
Version	291,805	307,841	*has_module*	11,316,114	11,240,182
Module	11,316,114	11,240,182	*has_attribute*	125,554,883	124,215,617
Attribute	772,774	853,389	*requires*	717,357	946,092
Total	12,393,641	12,414,851	Total	137,880,159	136,709,732

Dataset analysis. We run each code in the Gistable dataset in the validation environment and the exit status is shown in Table 1. There are 4,388 code snippets with `SyntaxError` in Python 3, much more than 719 in Python 2, which include 454 code snippets that have syntax errors in both versions. The difference in the number of code snippets that run successfully also indicates the bias towards Python 2 in the dataset. All automatic inference methods only focus on dependencies, i.e. `ImportError`, because it is almost impossible to automatically resolve errors in the code itself or missing external inputs such as database [16]. So, our experiments only validate the inferred environments of code snippets with `ImportError`.

5.2   Evaluation of Knowledge Graph

Our knowledge graphs. Knowledge acquisition is indeed time-consuming, which takes about 135 hours using 20 normal CPU cores. The rest of the building process is negligible in comparison. The whole process can be easily accelerated by parallelization, thus reducing time and effort to a great extent. Table 2 shows the scale of our Python package KGs and the quantities of each type of entities and relationships. Excluding those packages for which no version is available in the corresponding Python environments, we analyze 10,623 and 10,564 packages in Python 2.7.18 and Python 3.8.11, respectively. The extra 2,325 and 2,375 packages are added due to package dependencies, but we do not perform knowledge acquisition for these packages to ensure the fairness of knowledge. By further analyzing our KGs, we find that 74,657 (25.6%) and 68,281 (22.2%) versions fail to be installed, and 2,337,425 (20.7%) and 2,360,264 (21.0%) modules fail to be imported in Python 2 and Python 3, respectively, due to various issues such as missing dependencies and incompatible Python versions. This indicates that the releases on PyPI cannot always be installed successfully, and the modules and attributes obtained by statically parsing the code cannot always be used successfully. Moreover, nearly 74% of the successfully installed versions have at least one direct dependency, showing the necessity to consider the compatibility of inferred dependencies.

Domain knowledge. To compare the knowledge acquired by different methods, we count the quantities of versions, modules and attributes, which are shown in Table 3. The modules and attributes with the same full-qualified names are counted only once, and the packages have at least one version. DockerizeMe analyzes the latest version of each package to get the name of top-level modules, containing the fewest domain knowledge. SnifferDog downloads all releases on PyPI for each package and statically parses the Python code to get all the defined APIs. It fails to get any top-level modules for 4,831 packages, mainly because it cannot infer the Python versions needed for static parsing. In contrast to our PyCRE, SnifferDog gets slightly more versions and attributes, but cannot determine the

Table 3: Comparison of domain knowledge acquired by different methods.

Method	Package	Version	Module	Attribute
DockerizeMe	10,441	11,254	9,517	-
SnifferDog	**10,638**	**316,376**	5,764	**4,580,920**
PyCRE (Python 2)	10,623	291,805	**338,709**	4,343,530
PyCRE (Python 3)	10,564	307,841	302,908	3,949,917

Table 4: Comparison of knowledge coverage on the Gistable dataset.

Method	Top-level module	Submodule	Attribute
Dataset	1,721	7,083	8,704
DockerizeMe	727	-	-
SnifferDog	513	-	1,924
PyCRE (Python 2)	**800**	**1,622**	**3,486**

availability of this knowledge, which may cause unexpected errors in the inferred environments. Additionally, we add submodules into the Python package ontology. The acquisition of submodules provides a large amount of module knowledge for PyCRE, which are useful to infer appropriate dependencies.

Knowledge coverage on dataset. We analyze the coverage of the knowledge acquired by each method on the Gistable dataset, which significantly influences the results of inference. Considering that the dataset is more biased towards Python 2, we parse each code under Python 2.7.18 by the approach presented in Section 4.1. We divide the imported modules into top-level modules and submodules, since only our method supports queries for submodules. The analytical results are shown in Table 4. Compared to DockerizeMe and SnifferDog, our KG has the maximum coverage on all types of knowledge, but still lacks a large amount of knowledge about the dataset. Only 800 (46.5%) top-level modules are covered in our KG, which means that in many cases PyCRE can only choose to install the Python package with the same name. Submodules and attributes are covered with 1,622 (22.9%) and 3,486 (40.1%), respectively. It increases the difficulty of discovering candidate libraries, which reflects the realistic limitations of exact matching and the necessity of our matching degree.

5.3 Evaluation of Inference

We evaluate the effectiveness and efficiency of PyCRE in inferring Python compatible runtime environments. Table 5 shows the validation results of the inferred environments by each method.

ImportError. The most intuitive assessment is the ability to resolve ImportError, which reflects the effectiveness of the runtime environment inference. As shown in Table 5, PyCRE resolves the most ImportError in all three different settings and is significantly better than the comparative methods.

Gistable, the baseline method that is consistent with the programmer's behaviors in solving dependency issues, fails to resolve ImportError for 2,592 gists and 1,751 gists in Python 2 and Python 3, respectively. DockerizeMe fails to resolve ImportError in more gists than the baseline Gistable's method, mainly because of its

Table 5: Validation of the inferred environments generated by each method in different Python releases.

Method	ImportError	Success	Timeout	Others
Dataset	5,515	2,112	132	2,491
Gistable	2,592	2,988	422	4,248
DockerizeMe	2,624	2,986	415	4,225
SnifferDog	2,296	3,086	466	4,402
PyCRE	**1,645**	**3,309**	**499**	**4,797**

(a) Environment validation in Python 2.

Method	ImportError	Success	Timeout	Others
Dataset	3,536	1,293	71	5,350
Gistable	1,751	1,934	218	6,347
DockerizeMe	1,965	1,903	183	6,199
SnifferDog	1,632	1,960	254	6,404
PyCRE	**1,302**	**2,114**	**270**	**6,564**

(b) Environment validation in Python 3.

Method	ImportError	Success	Timeout	Others
V2	2,654	3,073	379	4,144
PyCRE	**1,524**	**3,410**	**579**	**4,737**

(c) Environment validation with the inferred Python versions.

method of matching target code and Python dependencies. DockerizeMe uses partial matching for imported modules to find libraries in the knowledge base, which is based on the longest prefix of the module name. For example, statement *from pyspark.sql.functions import udf* in the code snippet[2] is mapped to the package *py*, because there is no module *pyspark* and the longest matched prefix is module *py* in Dockerizeme's knowledge base. However, the Python package corresponding to this module is *pyspark*, which is the package that other methods choose to install. In fact, DockerizeMe uses domain knowledge to solve 482 code for which baseline cannot solve ImportError, but fails to solve 514 code for which baseline can solve ImportError in Python 2. Although these two methods do not specify versions for the inferred packages, which minimizes the version restrictions, they still encounter dozens of dependency conflicts.

Since V2 uses the inference results of DockerzieMe as the starting environments, its performance is similarly affected. V2 claims to find working environments for 3,206 code, but 133 of them do not run successfully in our validation, and even have dependency issues. One major reason is that V2 does not consider the compatibility between the packages in the inferred environments. Additionally, DockerizeMe and V2 install some packages incidentally according to their association rules, which leads to many redundant packages and is also more likely to cause dependency conflicts.

SnifferDog achieves relatively good performance, mainly because it has a large amount of API knowledge to assist its inference. However, SnifferDog is designed for Jupyter notebooks and ignores some issues in Python, such as the inference of Python versions, which affects its applicability on more general Python code.

---

[2]https://gist.github.com/samuelsmal/feb86d4bdd9a658c122a706f26ba7e1e

Table 6: Average and longest inference time of each method for 5,655 code snippets with at least one third-party package and have no inference timeout in all methods.

Method	DockerizeMe	V2	PyCRE
Avg. time (s)	**5.0**	128.0	7.0
Max. time (s)	**120.6**	3,570.7	215.2

Working environments. Another valuable metric is the number of working environments, which indicates the ability of an inference method to restore the runtime environments of Python code. As shown in Table 5, PyCRE infers a successfully runnable environment for the most Python code snippets in all three different settings. Moreover, the code marked as Timeout usually has a dead loop or awaits inputs, and can be treated as Success at least until they time out. In this sense, it can be assumed that PyCRE infers the working environments for 3,989 (38.9%) gists, which is also the best result among all methods.

Inference time. Inference time is a crucial metric for the efficiency of inference methods and represents the user's waiting time, which is one of the most important performance metrics for software. As listed in Table 6, we evaluate the average inference time and the longest inference time for each method. We only consider the 5,655 gists that have at least one third-party package in Python 2 or Python 3 and have no inference timeout in all methods. Since the API bank of SnifferDog is loaded in memory and does not query an external database, we do not discuss its inference time. The inference time of DockerizeMe and PyCRE, which fully uses pre-built knowledge bases, are short and comparable, whereas V2, which involves online execution, has a much longer inference time. It is worth noting that only the inference of V2 for 353 gists does not finish in one hour.

Ablation experiment. To validate the effectiveness of our proposed heuristic algorithm, we exclusively use the SAT solver to conduct an ablation experiment (i.e. disable our heuristic algorithm), which is called PyCRE (SAT only). In the inferred Python version, PyCRE (SAT only) infers 3,337 environments marked as Success and 536 marked as Timeout, leaving 1,597 ImportError. PyCRE with the heuristic algorithm, shown in Table 5c, is superior to PyCRE (SAT only) in the validation of inferred results. The average solving time of the heuristic algorithm is 0.2 seconds while the SAT solver is 3.9 seconds. However, the longest time of the heuristic algorithm is 8.4 seconds, which is much lower than 219.3 seconds of the SAT solver, and the difference becomes more significant as the size of the dependency graph increases. Moreover, compared to our heuristic algorithm, only using the SAT solver has problems caused by randomness. The SAT solver may choose old versions as long as the dependency requirements are met, but it usually leads to a loss of compatibility with newer versions. Also, the environments inferred by the SAT solver are variable and have the potential to introduce redundant dependencies, which may cause troubles for users. Based on our analysis, our heuristic algorithm solves 5,602 (99.4%) of the 5,637 code snippets for which have at least one dependency and a compatible solution. We believe the approximation of our heuristic algorithm is good. Besides, while specifying a version for each inferred package can better match the target code, such
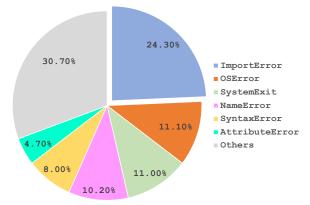

Figure 6: Proportions of 6,261 ($=1,524+4,737$) exceptions thrown by the environment validation inferred by PyCRE.

version requirements are more likely to cause dependency conflicts. In contrast, our approach is effectively conflict-aware through dependency solving, with dependency conflicts occurring only in 18 inferred environments.

Exception statistics. We further analyze the exceptions in Table 5c for the validated environments inferred by PyCRE. We include the subclasses of an exception in the count of that exception and the results are shown in Figure 6. Except for ImportError, the three most common Python built-in exceptions are OSError, SystemExit and NameError, which are mainly caused by code design or missing extra inputs, and cannot be solved by installing dependencies. SyntaxError is raised largely due to the target code itself. Moreover, AttributeError is raised when an attribute reference fails, mainly due to the reference on NoneType and some third-party resource. Therefore, we can assume that in most cases, the packages are also compatible in the inferred environments where ImportError is resolved by PyCRE.

Practical significance. The effectiveness and efficiency of PyCRE contributes to software engineers solving dependency issues of Python code in practice. Beginners often have difficulty building a runtime environment for sample code, and professional programmers also waste much time with complicated versions and dependencies. As in the manual evaluation on the Gistable dataset, it takes software engineers between 20 minutes and 2 hours to build the runtime environment [9], while the average inference time of our approach on this dataset is only 7 seconds, which can significantly improve the efficiency of development. Considering the performance of PyCRE, it is also important for the development of automated configuration management.

5.4 Threats to Validity

The results of our experiments may suffer from several threats to validity. The first is the fairness of knowledge, which can largely affect the inferred results of individual methods. For this reason, all methods in our experiments acquire knowledge based on the same 10,765 packages following their own methods. However, the knowledge base shared by DockerizeMe and V2 was previously constructed, and the latest versions of Python packages in it are sometimes not the latest versions at now. We think that this has little

impact on the experimental results because the dataset was proposed earlier than the construction of the DockerizeMe knowledge base, which means that the code actually relies on older versions of the packages. Oppositely, more new versions in PyCRE also increase the difficulty of inference. Another threat is Timeout in the validation results. The code that is marked as Timeout does not throw any exceptions in a minute, but this does no guarantee that there are no problems in subsequent execution. Considering that the import statements for resources are usually at the beginning of the Python code, code that executes with a timeout usually does not have dependency issues, which mitigates this threat.

5.5 Case Study

In addition to resolving the compatibility of inferred environments, the following Python code snippets illustrate several other unique capabilities of PyCRE.

Skip unusable releases. PyCRE obtains the real status of package installations and module imports with knowledge acquisition, which can guide PyCRE to skip the releases that are actually unusable. See the code snippet excerpted from a real-world gist:[3]

```
1  from pytube import YouTube
2  def download_video(video_url, output):
3          youtube = YouTube(video_url)
4          video = youtube.filter('mp4')[0]
5          return video.download(output)
```

Note that this code is only compatible with Python 2. Gistable and DockerizeMe generate pip install pytube to install the dependency, which means that the latest version *pytube-9.6.0* in Python 2 would be installed by default. However, after successfully installing the package, it fails to import *pytube* in the code, which raises an exception due to the problem in this version itself. V2 fails to find a working environment for this code, because it encounters timeout while verifying the candidate environment, resulting in no information available to guide the version changes. SnifferDog statically parses all versions of *pytube* and the latest version *11.0.0* contains the attribute *pytube.YouTube*, so it selects *pytube-11.0.0*, which fails to be installed and is actually only available in Python 3. In the knowledge acquisition phase, we already know that the module cannot be successfully imported in these versions, so based on our pre-built KGs, PyCRE skips the unusable versions and installs *pytube==9.5.2*.

Avoid useless downloads. If one version of a package fails to be installed or is incompatible with other packages, pip would attempt to install other versions instead, which is also known as the backtracking behavior.[4] Since pip does not have full package dependency information before downloading the package, it may lead to a large number of unnecessary downloads, which increases the time and system memory spent on building the runtime environment. Skipping unusable releases as described above can avoid the useless downloads caused by failed installations. The import statements below exemplify the strength of our approach in another aspect.

---

³https://gist.github.com/miratcan/4cd70e9515ab722b2bce
⁴https://pip.pypa.io/en/stable/user_guide/#dependency-resolution-backtracking

```
1  import numpy as np
2  from deepwalk.graph import Graph
```

The dependencies that are inferred to be explicitly installed by all the methods except PyCRE are *numpy* and *deepwalk*. Pip first downloads *numpy-1.16.6*, and then downloads *deepwalk-1.0.3*. However, *deepwalk-1.0.3* depends on *gensim-3.8.3*, which requires *numpy≤1.16.1* and is incompatible with *numpy-1.16.6*, so *numpy-1.16.1* is downloaded again. This causes a redundant download for *numpy*. Based on the unique knowledge of dependencies, PyCRE infers that only *deepwalk-1.0.3* needs to be installed explicitly and that compatible *numpy-1.16.1* would be installed automatically.

6 RELATED WORK

6.1 Python Runtime Environment Inference

There are many studies related to software KG in software engineering [12, 17, 22, 23]. DockerizeMe [10] is a pioneering work, which offline builds a knowledge base to infer the language/system-level environment dependencies required for Python code to execute without import errors. Compared with it, our PyCRE has several significant differences. First, DockerizeMe only considers the latest version of each package, which cannot handle removal or renaming that may accompany with version changes. Second, DockerizeMe maps the top-level modules to Python packages, and does not obtain submodules and attributes. However, there may be multiple versions of different packages containing modules with the same name, which needs further decision based on the attributes called in the code.

V2 [11] enhances DockerizeMe by exploring the possible configuration space for a Python code snippet. It validates candidate environments iteratively through code execution and applies environment mutation to generate new candidate configurations according to the failure messages. While V2 can find successful runtime environments for some Python code, its feedback-directed search is quite time-consuming, and even fails when an incorrect version does not manifest as a crash. On the contrary, PyCRE infers the appropriate Python libraries using the pre-built KGs, which is an efficient approach.

SnifferDog [21] is committed to restoring the execution environments of Jupyter notebooks and even reproducing the results. It builds an API bank to record mappings from popular Python libraries to their APIs by parsing the Python files. Several aspects affect the effectiveness of its environment inference. First, its API bank stores only public functions, but other public submodules and variables should also be considered. Second, the static analysis may fail to get any knowledge due to the uncertainty of Python versions used by the releases. Third, it does not guarantee that the defined APIs can actually be called, since the packages and modules where these APIs are located may fail to be installed or imported.

As far as we know, there is no study addressing the compatibility of the inferred environments like our work.

6.2 Dependency Solving

Dependency solving (and some of its variants) has been proved to be NP-complete, which can be easily encoded into a SAT solving problem using CNF [2, 3, 14]. Any solution of SAT is equally

valid, but practically some solutions are better than others for dependency solving. Trezentos et al. [19, 20] defined the software dependency problem as an extension of the SAT formulation called pseudo-Boolean optimization (PBO). There are several efficient PBO solvers, such as Open-WBO solver [15] and sat4j solver [4], and pseudo-Boolean constraints can be translated into clauses that can be handled by a standard SAT solver [8].

Abate et al. [1] reviewed proposals from the dependency solving field in recent years. They treat dependency solving as a separate concern in component evolution management [2]. Although a few popular package managers like Eclipse P2 use SAT solvers for dependency solving, the vast majority of package managers including pip still uses customized dependency graph traversals. The traditional dependency resolver of package managers receives a specific installation request given by the user, whereas dependency solving in PyCRE needs to determine the required installations, which is more difficult. Fortunately, we have global knowledge through the pre-built Python package KGs, which enables us to heuristically prune the search path for generating the compatible environments.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose an approach to automatically inferring Python compatible runtime environments with domain KG. We design a domain-specific ontology for Python third-party packages and propose an automatic approach to constructing the Python package KGs. Given a Python code, we discover candidate libraries by measuring the matching degree with third-party resources used in the code. Furthermore, we propose a heuristic graph traversal algorithm to infer the compatible runtime environment. Compared with existing approaches, we show the superior effectiveness, efficiency and compatibility of our approach in runtime environment inference. Our approach can contribute to automated software configuration management and facilitate code reuse.

In future work, we will acquire knowledge for more Python packages and improve the coverage. We also plan to add the deprecation information of modules and APIs into the KGs, and use it to further infer appropriate versions. Besides, we will extend the dependency inference to the entire project instead of single-file code and consider the compatibility with local dependencies, which is more general in practice. Finally, we will apply our approach to other languages with transitive dependencies such as Node.js.

ACKNOWLEDGMENTS

REFERENCES

[1] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *SANER*. IEEE, London, ON, Canada, 547–551.

[2] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software* 85, 10 (2012), 2228–2240. https://doi.org/10.1016/j.jss.2012.02.018

[3] Daniel Le Berre and Anne Parrain. 2008. On SAT Technologies for Dependency Management and Beyond. In *SPLC (2)*. Lero Int. Science Centre, University of Limerick, Ireland, Limerick, Ireland, 197–200.

[4] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 2-3 (2010), 59–6. https://doi.org/10.3233/sat190075

[5] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks. In *IIWeb*. AAAI, Acapulco, Mexico, 73–78.

[6] Robert Collins. 2015. PEP 508 - Dependency specification for Python software packages. https://www.python.org/dev/peps/pep-0508/. Accessed March 23, 2021.

[7] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *SAT (Lecture Notes in Computer Science, Vol. 2919)*. Springer, Santa Margherita Ligure, Italy, 502–518.

[8] Niklas Eén and Niklas Sörensson. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1-4 (2006), 1–26. https://doi.org/10.3233/sat190014

[9] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *ICSME*. IEEE, Madrid, Spain, 217–227.

[10] Eric Horton and Chris Parnin. 2019. DockerizeMe: automatic inference of environment dependencies for python code snippets. In *ICSE*. IEEE/ACM, Montreal, QC, Canada, 328–338.

[11] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *ASE*. IEEE, San Diego, CA, USA, 477–488.

[12] Zeqi Lin, Bing Xie, Yanzhen Zou, Junfeng Zhao, Xuan-Dong Li, Jun Wei, Hailong Sun, and Gang Yin. 2017. Intelligent development environment and software knowledge graph. *Journal of Computer Science and Technology* 32, 2 (2017), 242–249. https://doi.org/10.1007/s11390-017-1718-y

[13] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. 2004. Zchaff2004: An Efficient SAT Solver. In *SAT (Selected Papers (Lecture Notes in Computer Science, Vol. 3542)*. Springer, Vancouver, BC, Canada, 360–375.

[14] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. 2006. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *ASE*. IEEE, Tokyo, Japan, 199–208.

[15] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. 2014. Open-WBO: A modular maxSAT solver. In *SAT (Lecture Notes in Computer Science, Vol. 8561)*. Springer, Vienna, Austria, 438–445.

[16] Saikat Mondal, Mohammad Masudur Rahman, and Chanchal K. Roy. 2019. Can issues reported at stack overflow questions be reproduced?: an exploratory study. In *MSR*. IEEE/ACM, Montreal, Canada, 479–489.

[17] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-misuse detection driven by fine-grained API-constraint knowledge graph. In *ASE*. IEEE, Melbourne, Australia, 461–472.

[18] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *SAT (Lecture Notes in Computer Science, Vol. 5584)*. Springer, Swansea, UK, 244–257.

[19] Paulo Trezentos. 2010. Comparison of PBO solvers in a dependency solving domain. In *LoCoCo (EPTCS, Vol. 29)*. Edinburgh, UK, 23–31.

[20] Paulo Trezentos, Inês Lynce, and Arlindo L. Oliveira. 2010. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *ASE*. ACM, Antwerp, Belgium, 427–436.

[21] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *ICSE*. IEEE, Madrid, Spain, 1622–1633.

[22] Lu Wang, Xiaobing Sun, Jingwei Wang, Yucong Duan, and Bin Li. 2017. Construct bug knowledge graph for bug resolution: poster. In *ICSE (Companion Volume)*. IEEE, Buenos Aires, Argentina, 189–191.

[23] Min Wang, Yanzhen Zou, Yingkui Cao, and Bing Xie. 2019. Searching Software Knowledge Graph with Question. In *ICSR (Lecture Notes in Computer Science, Vol. 11602)*. Springer, Cincinnati, OH, USA, 115–131.

[24] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: An analysis of Stack Overflow code snippets. In *MSR*. ACM, Austin, TX, USA, 391–402.

[25] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina V. Lopes. 2017. Stack overflow in Github: Any snippets there?. In *MSR*. IEEE, Buenos Aires, Argentina, 280–290.