DockerKG: A Knowledge Graph of Docker Artifacts

Jiahong Zhou
zhoujiahong17@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Wei Chen*
wchen@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Chang Liu
liuchang@buu.edu.cn
Beijing Union University
Beijing, China

Jiaxin Zhu
zhujiaxin@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Guoquan Wu
gqwu@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

Jun Wei
wj@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China

ABSTRACT

Docker helps developers reuse software artifacts by providing a lightweight solution to the problem of operating system virtualization. A Docker image contains very rich and useful knowledge of software engineering, including the source of software packages, the correlations among software packages, the installation methods of software packages and the information on operating systems. To effectively obtain this knowledge, this paper proposes an approach to constructing a knowledge graph of Docker artifacts, named DockerKG, by analyzing a large number of Dockerfiles in Docker Hub, which contains more than 3.08 million Docker repositories (up to February 2020). Currently, DockerKG contains the domain knowledge extracted from approximately 200 thousand Dockerfiles in Docker Hub. Besides, it contains the information on Docker repositories and their semantic tags. In future work, DockerKG can be used for Docker image recommendations and online Q&A service providing software engineering domain knowledge.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

Docker, Dockerfile, knowledge graph, software package

ACM Reference Format:
Jiahong Zhou, Wei Chen, Chang Liu, Jiaxin Zhu, Guoquan Wu, and Jun Wei. 2020. DockerKG: A Knowledge Graph of Docker Artifacts. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3387940.3392161

*The corresponding author.

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392161

1 INTRODUCTION

Docker provides a lightweight solution to the problem of operating system virtualization, which makes it easy for developers to reuse software artifacts. At present, using Docker to build, release and run applications has become a trend. In recent years, the Docker community has been thriving. Docker Hub is a specialized online community for storing Docker repositories. At present, there are more than 3.08 million Docker repositories on Docker Hub (up to February 2020). A Docker image contains very rich and useful knowledge of software engineering, e.g., the source of software packages, the correlations among software packages, the installation methods of software packages and the information on operating systems.

In recent years, DevOps (development and operations) [9] is advocated for achieving continuous integration, deployment, and delivery [16], by eliminating the barrier between software development and system operations. Docker, being the representative of the container technique, is an important impetus for DevOps.

However, adopting and implementing DevOps is complicated because a lot of domain knowledge is required for proposing optimal DevOps solutions and effectively reusing software artifacts (e.g., scripts and recipes, software packages, best practices). Therefore, one major challenge for adopting DevOps is how to systematically capture, link, and utilize DevOps knowledge as a foundation for informed decision-making during application design and deployment [17]. That is, DevOps knowledge is practically available at large scale, but there is not an approach to systematically capturing and managing it [17].

The Docker ecosystem is an important source for retrieving DevOps knowledge. It contains abundant reusable artifacts (especially Docker images and Dockerfiles) [4]. These artifacts imply a lot of domain knowledge, e.g., software configuration dependencies, software orchestrations [18], version compatibilities. Xu and Marinov [18] advocated for mining container image repositories for understanding configurations and use cases of software systems, and they envisioned that it helps fill the gap between software development and the operations of software systems.

This paper proposes an approach to constructing a knowledge graph of Docker artifacts for obtaining the domain knowledge. In summary, the main contributions of this work are as follows.

- **Approach.** We propose an approach to constructing a knowledge graph of Docker artifacts. The approach automatically extracts domain knowledge from a large number of Docker artifacts, including the source of software packages, the correlations among software packages, the installation methods of software packages and the information on operating systems.
- **Implementation.** With the approach, we build a knowledge graph, named *DockerKG*, which contains the domain knowledge extracted from nearly 200 thousand Docker images in Docker Hub, together with the information on Docker repositories and their semantic tags.

The remainder of this paper is organized as follows. Section 2 briefly introduces the background. Section 3 and Section 4 propose the details of our approach and the implementation of the knowledge graph, respectively. Section 5 presents the related work. Finally, Section 6 concludes and mentions the future work.

2 BACKGROUND

A knowledge graph is a structured semantic knowledge base that symbolically describes concepts and their relations in the physical world. The basic unit of a knowledge graph is a set of triplets. A triplet is represented as $(entity, relation, entity)$, containing the entities and the relations among them. The entities in a knowledge graph are connected through relations among them, forming a networked knowledge structure.

The origin of knowledge graph can be traced back to the 1960s. The Semantic Network proposed by American psychologist JR Quillian is a graph-based semantic representation [14]. In order to represent the knowledge graph, researchers have proposed standard models for semantic description, including RDF[1], RDFS[2], and OWL[3]. The concept of knowledge graph was proposed by Google in 2012[4] . Aiming to improve their search engines, Google created a large-scale knowledge graph. After that, the knowledge graph quickly became a hot topic, and "*knowledge graph*" has gradually become an academic concept. At present, there are many well-known, large-scale, universal and high-quality knowledge graphs, e.g., DBpedia [1][2][10], Yago [15], Yago3 [12] and HowNet [5]. DBpedia is a large-scale encyclopedia knowledge base that contains structured Wikipedia information; Yago is a multilingual knowledge base by combining Wikipedia, wordnet, and GeoNames; Yago3 is an extension of Yago by combining Wikipedias in multiple languages; HowNet is a Chinese semantic dictionary, and it presents the concepts as description objects in Chinese and English, and builds a common sense knowledge base containing the relations between concepts and their attributes.

A Dockerfile is a declarative definition of an environment that aims to enable reproducible builds of the containers [4]. A Dockerfile specifies how to build a Docker image with a set of instructions written in a kind of domain-specific language (DSL). As such, a Dockerfile can also be thought of as a recipe or a set of source code following the notion of IaC [7]. A Dockerfile contains several kinds

[1]https://www.w3.org/RDF
[2]https://www.w3.org/TR/rdf-schema/
[3]https://www.w3.org/OWL
[4]https://developers.google.com/knowledge-graph/

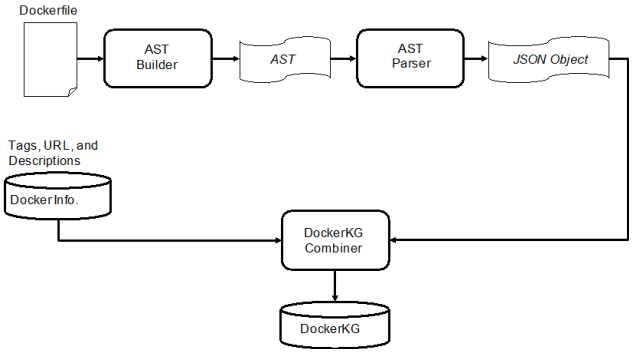

Figure 1: The overview of constructing DockerKG.

```
ENV VERSION 2.26
RUN mkdir /src
WORKDIR /src
RUN curl https://www.kernel.org/pub/linux/utils/util-linux/v$VERSION/util-linux-$VERSION.t
| tar -zxf-
RUN ln -s util-linux-$VERSION util-linux
WORKDIR /src/util-linux
RUN ./configure --without-ncurses
RUN make LDFLAGS=-all-static nsenter
RUN cp nsenter /usr/local/bin

ENV PATH /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/gopath/bin
```

Figure 2: A Dockerfile snippet of "contiv/ubuntu".

of instructions, e.g., "`FROM`", "`RUN`", "`ENV`", "`CMD`" and "`ENTRYPOINT`". "`FROM`" declares the dependent base image, and according to the layered union file system, all features (such as OS, software, etc.) of the ancestor images will be inherited. "`RUN`" is responsible for executing any shell commands, and its one important task is software package installations and updates. "`ENV`" sets environment variables, e.g., working directory and default path. "`ENV, ENTRYPOINT`" are similar since they usually declare the command to launch the specific services when starting a container instance.

A Dockerfile contains a lot of semantic information, including the source of software packages, the correlations among software packages, the installation methods of software packages and the operating system information. It is viable to obtain the knowledge of software packages by analyzing Dockerfiles.

In consequence, we are motivated to propose an approach to automatically constructing a knowledge graph of Docker artifacts.

3 APPROACH

As is shown in Figure 1, our approach comprises three parts, namely *AST Builder*, *AST Parser* and *DockerKG Combiner*. *AST Builder* takes a Dockerfile as the input and builds an abstract syntax tree (AST) of it. *AST Parser* generates a JSON object by parsing the abstract syntax tree. Finally, *DockerKG Combiner* combines the descriptive information of Docker images and the JSON objects and outputs DockerKG.

3.1 AST Builder

AST Builder builds an abstract syntax tree for a Dockerfile. First of all, *AST Builder* identifies each instruction in a Dockerfile according to the keywords and the DSL syntax. Then, *AST Builder* analyses

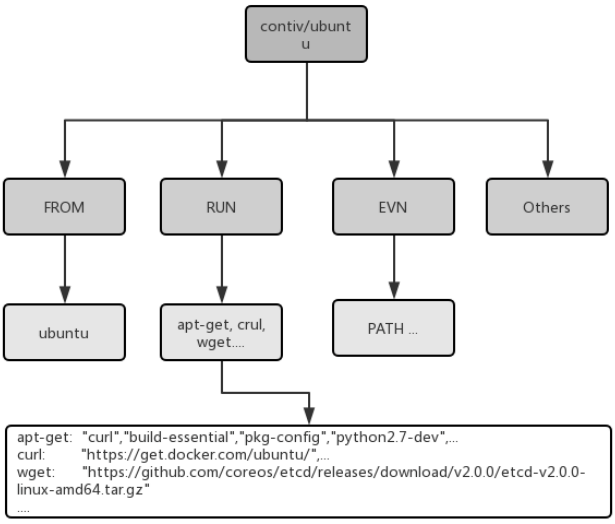

Figure 3: The Dockerfile AST of "contiv/ubuntu".

Algorithm 1 AST Builder algorithm

Input: AST of *Dockerfile*
Output: A JSON object *AST_Object* containing entities and relations

```
1: create an empty hash table as env_table
2: for every instruction in Dockerfile do
3:     if instruction is useless then // ignore comments and other useless commands
4:         continue // skip the current instruction
5:     end if
6:     cmd_type = parseCmdType(instruction)
7:     if cmd_type is "ENV" then
8:         < key, value > = parseEnv(instruction)
9:         replace value with env_table
10:        update env_table with < key, value >
11:    else
12:        replace instruction with env_table
13:    end if
14:    add instruction into AST_Object according to cmd_type
15: end for
16: return AST_Object
```

each instruction to obtain the instruction type, and meanwhile, it keeps instructions executed by "ENV", "RUN" and "FROM" commands and filters out the comments and the other types of commends. Finally, *AST Builder* creates a syntax tree for the extracted instructions of the Dockerfile.

When parsing Dockerfiles, there are 'ENV' instructions that define runtime environment variables for making Dockerfiles concise and configurable, where each variable is defined as a key-value pair. As Figure 2 shows, the first line of the Dockerfile snippet defines an environment variable "VERSION" and sets its value with "2.26". Furthermore, a variable can be defined by embedding some other variables, in other words, a variable might be assembled with some other variables. For example, "ENV NDK_HOME optandroid-ndk-$ANDORID_NDK_VERSION" defines a variable "NDK_HOME" whose value is a file path, with another embedded variable named "ANDORID_NDK_VERSION" in it. When parsing "NDK_HOME", "ANDORID_NDK_VERSION" must be parsed firstly and instantiates the variable (denoted as "$ANDORID_NDK_VERSION") in the value of "NDK_HOME". Such embedded variable definitions make it difficult to identify environment variables and determine their values.

To solve this problem, we propose an algorithm (see Algorithm 1). Given a Dockerfile, it first builds an empty hash table (line 1), and then it traverses the Dockerfile from up to down. When getting an ENV-type instruction, it puts the $< key, value >$ pair of each variable into the hash table. If the variable is an assembled one, before putting it into the hash table, the algorithm searches the hash table and instantiates the variable's value by replacing the embedded variables with their values (line 7-10). When getting the other type of instructions, the algorithm replaces the variables with their values by searching in the hash table (line 11-12). Finally, it adds the parsed instructions into the AST. Figure 3 shows an exemplary AST of a specific Docker repository "contiv/ubuntu"[5].

3.2 AST Parser

AST Parser extracts entities and relations from the constructed AST. For an AST, *AST Parser* obtains the following information: the image name, the repository hosted in Docker Hub, the operating system and the software packages installed in the Docker image.

AST Parser identifies the entities and the relations among them, by addressing the following challenges.

3.2.1 Software package identification. Docker images contain multiple software packages installed by using different software package managers, which makes the software package identification challenging. RPM (in RedHat Linux, installing software packages with "yum" command) and DPKG (in Debian Linux, installing software packages with "apt" and "apt-get" commands) are the mainstream package managers in Linux, and they have different package installation methods and software package libraries. The list below shows two exemplary commands that install software packages with RPM and DPKG, respectively.

(1) An example of RPM-based installation: "yum install -y zsh-5.0.2-33.el7" installs version "5.0.2-33.el7" of package "zsh" on CentOS.
(2) An example of DPKG-based installation: "apt-get install -y zsh=5.4.2-3ubuntu3" installs version "5.4.2-3ubuntu3" of package "zsh" on Ubuntu.

One problem is how to filter out the low-quality software packages (including packages with typo names and unofficial packages). To solve the problem, *AST parser* builds software package name lists for RPM and DPKG (it executes 'yum –showduplicates list' on CentOS and 'apt-cache pkgnames' on Ubuntu), and then it filters out the packages whose name not in the lists when parsing the installation commands.

DPKG uses '=' as the joint mark between a software package name and its version and thus *AST Parser* can get them directly. In contrast, RPM specifies a software package by using '-' as the

[5]https://hub.docker.com/r/contiv/ubuntu/dockerfile

```
FROM debian:stretch

RUN set -eux; \
        apt-get update; \
        apt-get install -y --no-install-recommends \
                bzip2 \
                ca-certificates \
                libffi-dev \
                libgmp-dev \
                libssl-dev \
                libyaml-dev \
```

Figure 4: A Dockerfile snippet of "library/ruby:2.4".

```
FROM ruby:2.4

MAINTAINER thinkbot@outlook.de

ENV VERSION=0.1.9

RUN gem install clenver --version ${VERSION} --no-format-exec

WORKDIR /tmp

ENTRYPOINT ["clenver"]
CMD ["--help"]
```

Figure 5: A Dockerfile snippet of "rubygem/clenver".

joint mark between its name and version, but the package name itself might contain '-' (e.g., package 'zsh-common'). As such, it is infeasible to directly split the package name and the version by using '-'. To solve the problem, *AST Parser* splits and aligns the names and the versions by matching them to the RPM-based package name list.

3.2.2 Operating system identification. It is known that one Docker image might be built on another image, and the inheritances are transitive. Therefore, all the images form a forest and each root of a tree is an image containing a specific operating system, and the other Docker images are built directly or indirectly on those operating system images. For example, Figure 4 illustrates a case where the base image of "`library/ruby:2.4`" is "`debain:stretch`", a Linux operating system image. In contrast, the example in Figure 5 shows that the base image of "`rubygem/clenver`" is "`ruby:2.4`", not an operating system image. By recursively parsing the Dockerfile (see also Figure 4 and Figure 5), *AST Parser* can determine that "`rubygem/clenver`" also depends on operating system image "`debain:stretch`". In consequence, the difficulty is how to identify what the operating system of a Docker image exactly is.

AST Parser constructs an operating system name list by crawling the official operating system repositories in Docker Hub, for determining whether an image is an operating system one. A given Docker image is thought of as an operating system image if its name is in the list. However, there are still some exceptions in which the dependent operating systems cannot be identified. According to the different software package managers, *AST Parser* defines them as "`RPM_Based_OS`", "`DPKG_Based_OS`" or "`Unknown_OS`". The first one represents the operating system based on RPM software package manager, the second one denotes the operating system based on DPKG software package manager, and the last one is the operating system that *AST Parser* cannot determine which software package manager is based on.

3.3 DockerKG Combiner

To enrich DockerKG, DockerKG Combiner adds some other information on Docker repositories into DockerKG, including URL of the Docker repositories in Docker Hub, URL of the Dockerfiles in Docker Hub, and semantic tags recommended by SemiTagRec [3].

4 DOCKERKG

4.1 Ontology of DockerKG

Figure 6 shows the ontology of DockerKG, which references the ontology definition of DockerPedia [13]. Some main classes in the ontology are, *Dockerfile*, *Image* (the Docker image built by the Dockerfile), *OperatingSystem* (the operating system in the Dockerfile), *PackageURL* (the download URL of a software package in the Dockerfile), *PackageVersion* (the version of software package in the Dockerfile), *Repository* (the Docker repository on Docker Hub) and *Tag* (the semantic tags of the Docker repository recommended by SemiTagRec [3]). DockerKG also represents the relationships between the classes, including *builds* (a Dockerfile builds an entity Image), *isBasedOn* (an image is based on another image), *dependsOn* (an image depends on some entities of PackageVersion and PackageURL) and *hasTag* (an entity of Repository has some tag entities of Tag).

4.2 The Implementation of DockerKG

We crawled and parsed approximately 200 thousand Dockerfiles from Docker Hub. The DockerKG contains more than 900 thousand entities and nearly 2,900 thousand relations, among which there are more than 200 entities labeled with "`docker_OperatingSystem`", more than 4,900 entities labeled with "`docker_Tag`", nearly 48 thousand entities labeled with "`docker_PackageURL`", and about 12 thousand entities labeled "`docker_Pakcage`". The data of DockerKG is stored in Neo4j[6], a popular graph database. Figure 7 shows a subgraph of DockerKG, which relates to the Docker project "`contiv/ubuntu`".

Currently, DockerKG can be used for querying. For example, we can query for the specific Docker images with a Cypher Query: `MATCH (os:docker_OperatingSystem name: "ubuntu:18.04"), (images)-[docker_isBasedOn]->(os) RETURN images, os LIMIT 100`.

A set of nodes will be returned, representing a subgraph relating to Ubuntu 18.04 (see Figure 8). More applications of DockerKG will be explored in the future, e.g., querying the dependencies among software packages, Docker image recommendations, and automatic creations of user-specific Docker images and Dockerfiles.

[6]https://neo4j.com/

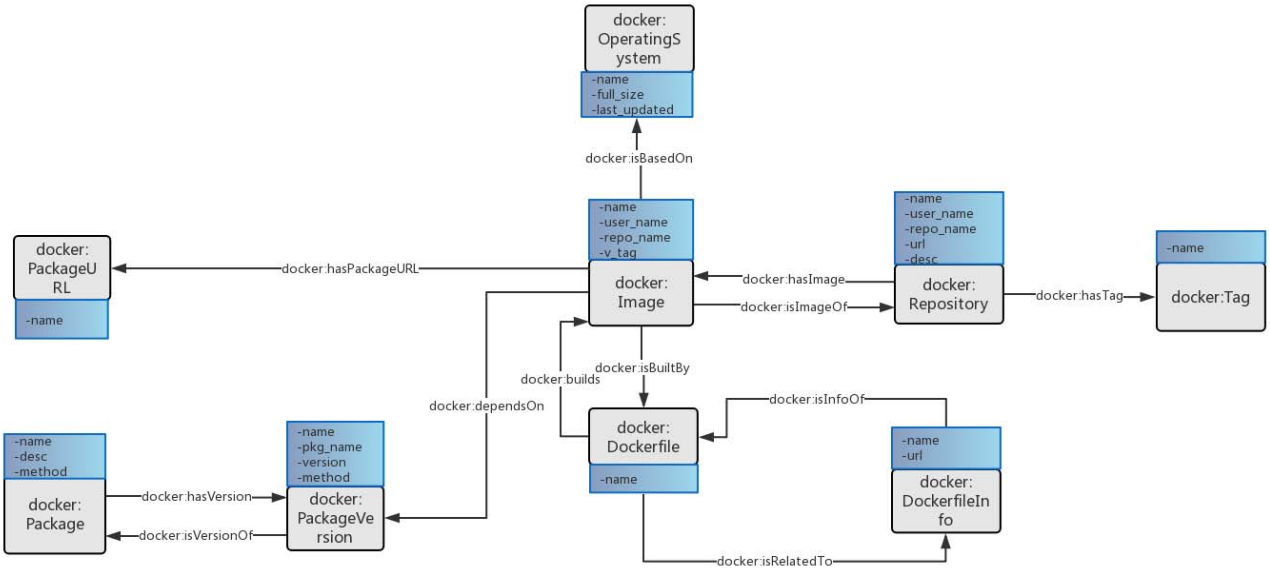

Figure 6: The ontology of DockerKG.

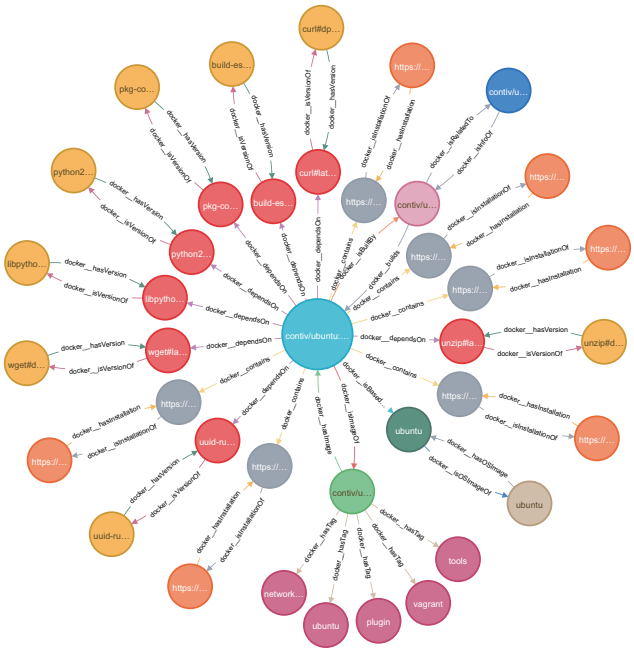

Figure 7: A subgraph relating to "contiv/ubuntu".

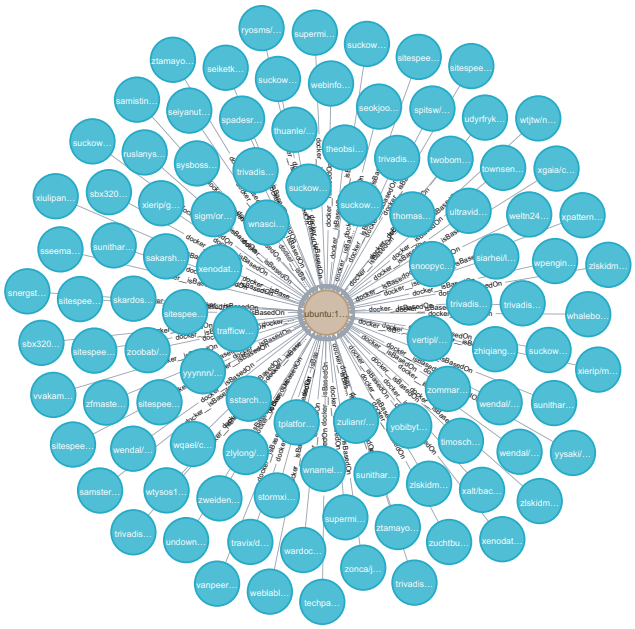

Figure 8: The query result relating to Ubuntu 18.04.

5 RELATED WORK

Researches on Docker and knowledge graph are the most related ones.

In the Docker field, there are studies devoted to mining Docker repositories and Docker project evolution analysis. Xu et al. [18] envisioned that abundant Docker repositories encapsulate a plethora of reusable artifacts and knowledge, e.g., configurations, dependent libraries, and components. The authors advocated that mining container image repositories are valuable and showcased the opportunities of benefitting from mining the repositories. Some work conducts empirical studies by analyzing and mining the repositories, focusing on the impact of Dockerfile evolutionary trajectories [20][19]. RUDSEA [6] recommends updates of Dockerfiles to developers by analyzing changes in software environment assumptions

and their impacts. The most related work is DockerPedia [13], Osorio et al. built it from the instances of deployed Docker images, to extract the information of the installed packages, and then they used Clair[7] to detect the vulnerabilities of them, which is a resource that publishes information of the packages within Docker images as Linked Data. DockerPedia provides information about the software dependencies and its vulnerabilities for environment reproductions and security checks. To our best knowledge, DockerPedia is the only one knowledge graph of Docker images. However, constructing DockerPedia is time-consuming and costly, because the Docker images must be downloaded, deployed and instantiated. DockerPedia is based on a lightweight ontology [8] that shapes the relations between the different concepts. DockerKG is different from DockerPedia in several aspects, (1) the former is constructed by extracting information from Dockerfiles instead of Docker images, which makes our approach more lightweight; (2) the schema model of DockerKG is different, on the one hand, besides the key elements extracted from Dockerfiles, it contains semantic tags [3] generated by mining the descriptions and Dockerfiles. On the other hand, DockerKG does not contain the security information of the Docker project, without analyzing the vulnerabilities; (3) DockerPedia did not contain the source of unofficial software packages; (4) The data of DockerKG is stored and managed based on graph database Neo4j rather than RDF.

The knowledge graph is an attractive technique, and many knowledge graphs are constructed and applied in different domains, including software engineering. CodeWisdom publishes an online API knowledge graph that links API elements (e.g., libraries, classes, methods, parameters), their descriptive knowledge (e.g., functionalities and directives), and related background knowledge. Currently, this knowledge graph contains the knowledge of JDK and Android API. Lin et al [11] proposed a software knowledge graph construction approach that automatically creates knowledge graphs from software big data, e.g., source code files, version control data, mailing lists, issue tracks, online social question-answering pairs, etc. They used Neo4j to store the knowledge graph and Cypher to make queries. Specifically, they constructed a knowledge graph for Apache Luence by using the proposed knowledge graph construction approach. Comparing with the related work, DockerPedia contains different domain knowledge by automatically analyzing a large number of Dockerfiles.

6 CONCLUSION

This paper proposes an approach to constructing a knowledge graph of Docker artifacts, namely DockerKG, which obtains the sources of software packages, the correlations among software packages, the installation methods of software packages and the information on operating systems. Currently, DockerKG contains approximately 100 million entities and 300 million relations obtained from nearly 200 thousand Dockerfiles in Docker Hub.

In future work, we plan to explore the applications of DockerKG, e.g., querying the dependencies among software packages, Docker image recommendations, and automatic creations of user-specific Docker images and Dockerfiles.

[7] https://github.com/coreos/clair

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive comments. This work was partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1400602 and the National Natural Science Foundation of China under Grant No. 61732019.

REFERENCES

[1] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[2] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. 2009. DBpedia-A crystallization point for the Web of Data. *Journal of web semantics* 7, 3 (2009), 154–165.

[3] Wei Chen, Jia-Hong Zhou, Jia-Xin Zhu, Guo-Quan Wu, and Jun Wei. 2019. Semi-Supervised Learning Based Tag Recommendation for Docker Repositories. *Journal of Computer Science and Technology* 34, 5 (2019), 957–971.

[4] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 323–333.

[5] Zhendong Dong, Qiang Dong, and Changling Hao. 2010. Hownet and its computation of meaning. In *Proceedings of the 23rd international conference on Computational Linguistics: Demonstrations*. Association for Computational Linguistics, 53–56.

[6] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. 2018. RUDSEA: recommending updates of Dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 796–801.

[7] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 368–388.

[8] Da Huo, Jaroslaw Nabrzyski, and Charles Vardeman. 2015. Smart Container: an Ontology Towards Conceptualizing Docker.. In *International Semantic Web Conference (Posters & Demos)*.

[9] Michael Hüttermann. 2012. *DevOps for developers*. Apress.

[10] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. 2015. DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web* 6, 2 (2015), 167–195.

[11] Ze-Qi Lin, Bing Xie, Yan-Zhen Zou, Jun-Feng Zhao, Xuan-Dong Li, Jun Wei, Hai-Long Sun, and Gang Yin. 2017. Intelligent development environment and software knowledge graph. *Journal of Computer Science and Technology* 32, 2 (2017), 242–249.

[12] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M Suchanek. 2013. Yago3: A knowledge base from multilingual wikipedias.

[13] Maximiliano Osorio, Carlos Buil Aranda, and Hernán Vargas. 2018. DockerPedia: a Knowledge Graph of Docker Images.. In *International Semantic Web Conference (P&D/Industry/BlueSky)*.

[14] M Ross Quillan. 1966. *Semantic memory*. Technical Report. BOLT BERANEK AND NEWMAN INC CAMBRIDGE MA.

[15] Thomas Rebele, Fabian Suchanek, Johannes Hoffart, Joanna Biega, Erdal Kuzey, and Gerhard Weikum. 2016. YAGO: A multilingual knowledge base from wikipedia, wordnet, and geonames. In *International Semantic Web Conference*. Springer, 177–185.

[16] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.

[17] Johannes Wettinger, Vasilios Andrikopoulos, and Frank Leymann. 2015. Automated capturing and systematic usage of devops knowledge for cloud applications. In *2015 IEEE International Conference on Cloud Engineering*. IEEE, 60–65.

[18] Tianyin Xu and Darko Marinov. 2018. Mining container image repositories for software configuration and beyond. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 49–52.

[19] Yang Zhang, Huaimin Wang, and Vladimir Filkov. 2019. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences* 62, 1 (2019), 19101.

[20] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. 2018. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 138–143.