# Build System Analysis with Link Prediction

Xin Xia[1]*, David Lo[2], Xinyu Wang[1], and Bo Zhou[1]
[1]College of Computer Science and Technology, Zhejiang University, China
[2]School of Information Systems, Singapore Management University, Singapore
xxkidd@zju.edu.cn, davidlo@smu.edu.sg, {wangxinyu, bzhou}@zju.edu.cn

## ABSTRACT

Compilation is an important step in building working software system. To compile large systems, typically build systems, such as make, are used. In this paper, we investigate a new research problem for build configuration file (e.g., Makefile) analysis: how to predict missed dependencies in a build configuration file. We refer to this problem as *dependency mining*. Based on a Makefile, we build a dependency graph capturing various relationships defined in the Makefile. By representing a Makefile as a dependency graph, we map the dependency mining problem to a link prediction problem, and leverage 9 state-of-the-art link prediction algorithms to solve it. We collected Makefiles from 7 open source projects to evaluate the effectiveness of the algorithms.

## Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids

## Keywords

Build System, Link Prediction, Makefile

## 1. INTRODUCTION

Software build process converts source code, libraries and other data into executable programs by orchestrating the execution of compilers and other tools. The whole building process is managed by a build system, such as ***make, ant, scon,*** *or* ***cmake***. The build system ***make*** reads a build configuration file known as a Makefile to build a system, and it is one of the most widely used build system.

In this paper, we investigate a new research problem for software build systems: automatic mining of missed dependencies in build systems, and we mainly focus on the build system **make**. We notice that for a large-scale software project, such as Linux, the dependencies among the source code files are complex. It is easy to miss some dependencies, which is hard to detect.

To solve the dependency mining problem, we first convert a Makefile into a dependency graph. Nodes in the graph corresponds to entities in the Makefile and edges in the graph correspond to relationships among these entities. The dependency mining problem is then reduced to the problem of predicting missing edges (links) in this graph. We propose the usage of link prediction algorithms [1] to find these missing edges. Nine state-of-the-art link prediction algorithms are investigated in this paper, which are: common neighbors (CN), cosine similarity (CS), Jaccard Index (JI), Adamic-Adar (AA), Resource Allocation (RA), Leicht-Holme-Newman (LHN1), preferential attachment (PA), Katz, and local path index (LP).

## 2. PRELIMINARIES

Figure 1 presents a simple Makefile which specifies the build process of a calculator. We define 6 *targets* in this Makefile: *add.o, subtract.o, mult.o, divide.o* (object files), *calculator* (executable file), *clean* (a label). The target *calculator* depends on several *prerequisites*: *add.o, substract.o, mult.o, divide.o* and a static library *lib.a*. The object files depend on their own source code files and a common header file *num.h*.

Using MAKAO [2], we can convert a Makefile into a dependency graph. Figure 2 shows the dependency graph derived from the simple Makefile in Figure 1. The *targets* and *prerequisites* in the Makefile become nodes, and the dependency relationships become edges (links) in the graph.

```
calculator :add.o subtract.o mult.o divide.o lib.a
     gcc add.o subtract.o mult.o divide.o -L . lib.a -o
calculator

add.o:  add.c num.h add.h
     gcc -c add.c

subtract.o:  subtract.c num.h add.h
     gcc -c subtract.c

mult.o:  mult.c num.h
     gcc -c mult.c

divide.o:  divide.c num.h
     gcc -c divide.c

clean:
     rm -rf *.o
```

**Figure 1: An Example Makefile which Specifies the Build Process for** `calculator`**.**

*The work was done while the author was visiting Singapore Management University.

*SAC'14* March 24-28, 2014, Gyeongju, Korea.
Copyright 2014 ACM 978-1-4503-2469-4/14/03 ...$15.00.
http://dx.doi.org/10.1145/2554850.2555134

We formalize the definition of a dependency graph and the task of dependency mining below:

Definition 1. (**Dependency Graph.**) *A dependency graph is graph $G(V,E)$, where each node $v \in V$ denotes a target or a prerequisite in a corresponding Makefile, each edge (link) $e \in E$ denotes a dependency between a target and a prerequisite node. We denote the edge from node $v'$ to node $v$ as $e(v',v)$.*

Definition 2. (**Dependency Mining.**) *Consider a dependency graph $G(V,E)$, and let us denote $U$ as the set containing all $\frac{|V|\times(|V|-1)}{2}$ possible edges (links) among nodes in $G(V,E)$. The dependency mining task is to mine the missed edges from $U-E$ edges.*

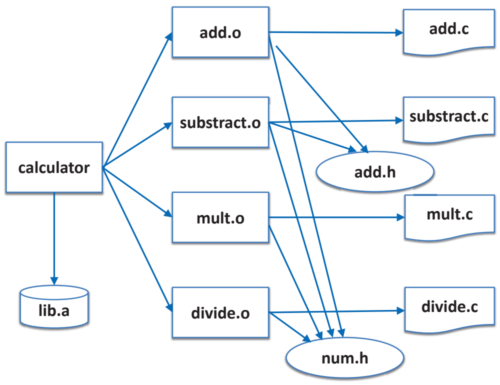

**Figure 2: The Dependency Graph for The Simple Makefile in Figure 1.**

## 3. LINK PREDICTION

In this section, we present 9 state-of-the-art link prediction algorithms (c.f., [1]) that is used in this paper. They are: common neighbors (CN), cosine similarity (CS), Jaccard Index (JI), Adamic-Adar (AA), Resource Allocation (RA), Leicht-Holme-Newman (LHN1), preferential attachment (PA), Katz, and local path index (LP). These algorithms compute similarities of pairs of nodes and return a sorted list of most similar pairs. The more similar the nodes in a pair is, the higher is the probability that a link (edge) exists between the nodes. Given a dependency graph $G(V,E)$, for each node pair $x,y \in V$, we denote the similarity score of nodes $x$ and $y$ as $scores(x,y)$. We can represent a dependency graph as an adjacency matrix $M$ where for any two different nodes $v$ and $v'$, if there is a link between them, $(M)_{vv'} = 1$. We denote neighbors of a node $v$ as $\Gamma(v)$, which represents a set of nodes $v'$ which are adjacent to $v$ in $G(V,E)$. We denote the degree of a node $v$ as $k(v)$, which represents the number of neighbors that $v$ has. Based on these notations, the 9 algorithms compute similarities of pairs of nodes in the following ways.

**Common Neighbors (CN).** Common neighbors (CN) computes a score based on the number of common neighbors that nodes $x$ and $y$ share as follows:

$$score_{CN}(x,y) = |\Gamma(x)\cap\Gamma(y)| \tag{1}$$

**Cosine Similarity (CS).** Cosine similarity (CS) computes a score based on the common neighbors that $x$ and $y$ share, and the degree of the nodes as follows:

$$score_{CS}(x,y) = \frac{|\Gamma(x)\cap\Gamma(y)|}{\sqrt{k(x)\times k(y)}} \tag{2}$$

**Jaccard Index (JI).** Jaccard Index computes a score based on the ratio of common neighbors to all neighbors that node $x$ or $y$ have as follows:

$$score_{JI}(x,y) = \frac{|\Gamma(x)\cap\Gamma(y)|}{|\Gamma(x)\cup\Gamma(y)|} \tag{3}$$

**Adamic-Adar (AA).** Adamic-Adar (AA) computes a score based on the degrees of the common neighbors that node $x$ and $y$ have as follows:

$$score_{AA}(x,y) = \sum_{z\in\Gamma(x)\cap\Gamma(y)} \frac{1}{log(k(z))} \tag{4}$$

**Resource Allocation (RA).** RA also computes a score based on the degrees of the common neighbors that node $x$ and $y$ have as follows:

$$score_{RA}(x,y) = \sum_{z\in\Gamma(x)\cap\Gamma(y)} \frac{1}{k(z)} \tag{5}$$

**Leicht-Holme-Newman (LHN1).** Leicht-Holme-Newman Index (LHN1) computes a score based on the actual and expected number of common neighbors that nodes $x$ and $y$ have as follows:

$$score_{LHN1}(x,y) = \frac{|\Gamma(x)\cap\Gamma(y)|}{k(x)\times k(y)} \tag{6}$$

**Preferential Attachment (PA).** Preferential Attachment (PA) computes a score based on the idea that the probability of a new link to take node $x$ as one of its endpoint is proportional to the current number of neighbors that $x$ has, in the following way:

$$score_{PA}(x,y) = k(x)\times k(y) \tag{7}$$

**Katz.** Katz [1] computes a score based on the paths (i.e., series of nodes) between nodes $x$ and $y$, as follows:

$$score_{katz}(x,y) = \sum_{l=1}^{\infty} \beta^l \times |Path^l(x,y)| \tag{8}$$

where $Path^l(x,y)$ denote paths of length $l$ between two nodes $x$ and $y$, and $\beta^l > 0$ is a weight which controls the contribution of length $l$ paths to the similarity score. By default, we set $\beta = 0.0005$.

**Local Path Index (LP).** Local path index (LP) is a mix between common neighbors (CN) and Katz. It is defined as:

$$score(x,y) = (M^2)_{xy} + \beta(M^3)_{xy} \tag{9}$$

where, the first term in Equation (9) (i.e., $(M^2)_{xy}$) refers to the number of common neighbors between node $x$ and $y$, and the second term refers to the number of paths of lengths 2 between the two nodes. By default, we set $\beta = 0.0005$.

## 4. EXPERIMENTS AND RESULTS

We evaluate the 9 link prediction algorithms presented in Section 3 on the collected Makefiles whose statistics are presented in Table 1. The experimental environment is a

**Table 1: Statistics of Collected Build Systems.**

| Projects | # Nodes | # Links |
|---|---|---|
| Zlib | 91 | 233 |
| putty | 99 | 411 |
| vim | 146 | 1,144 |
| APR | 289 | 1,223 |
| Memcached | 227 | 2,443 |
| Nginx | 291 | 6,798 |
| Tengine | 320 | 8,258 |

Windows 7 64-bit, Intel(R) Xeon(R) 2.53GH server with 24GB RAM. One feature of our dependency mining problem is the imbalance data phenomenon, i.e., the number of actual dependencies is small compared to the total number of possible dependencies. For example, in **APR** project, the number of nodes is 289, and thus the number of possible dependencies is $\frac{289*288}{2} = 41,616$, but the number of actual dependencies is 1,223, which is only $\frac{1,223}{41,616} = 2.94\%$ of the number of possible dependencies. We evaluate the performance of the 9 algorithms in terms of the area under the ROC curve (AUC). AUC is a commonly used metric to evaluate the performance of link prediction algorithms [1]. The higher an AUC value is, the better performance an algorithm achieves. Moreover, if AUC is below 0.5, it means an algorithm is even worse than random guess.

**Table 2: AUC Scores for The 9 Algorithms. Z.=Zlib. P.=putty. V.=vim. A.=APR. M.=Memcached. N.=Nginx. T.=Tengine.**

| Algo. | Z. | P. | V. | A. | M. | N. | T. | Avg. |
|---|---|---|---|---|---|---|---|---|
| CN | 0.38 | 0.37 | 0.39 | 0.38 | 0.29 | 0.33 | 0.33 | 0.36 |
| CS | 0.38 | 0.37 | 0.38 | 0.38 | 0.29 | 0.33 | 0.33 | 0.35 |
| JI | 0.38 | 0.37 | 0.39 | 0.38 | 0.29 | 0.33 | 0.33 | 0.35 |
| AA | 0.38 | 0.37 | 0.39 | 0.38 | 0.30 | 0.33 | 0.33 | 0.36 |
| RA | 0.38 | 0.37 | 0.39 | 0.38 | 0.30 | 0.33 | 0.33 | 0.36 |
| LHN1 | 0.38 | 0.37 | 0.39 | 0.38 | 0.29 | 0.33 | 0.33 | 0.35 |
| PA | 0.71 | 0.82 | 0.85 | 0.86 | 0.87 | 0.84 | 0.85 | 0.83 |
| Katz | 0.58 | 0.59 | 0.64 | 0.71 | 0.50 | 0.64 | 0.64 | 0.61 |
| LP | 0.61 | 0.64 | 0.64 | 0.70 | 0.50 | 0.70 | 0.70 | 0.64 |

Table 2 presents the AUC scores for the 7 Makefiles and 9 algorithms. AUC scores for CN, CS, JI, AA, RA, and LHN1 are extremely low ($< 0.5$). And AUC scores for PA, Katz and LP are better ($> 0.5$). Among the 9 algorithms, PA achieves the best performance – the AUC scores vary from 0.71 to 0.87 with an average of 0.83.

## 5. RELATED WORK

There have been a number of studies studies on build system maintenance. McIntosh et al. investigate version histories of one proprietary and nine open source projects [3]. Adams et al. [4] analyze changes to the Linux kernel build system from its inception up to version 2.6 using MAKAO. They conclude that a good balance between obtaining a fast, correct build system and migrating in a step by step way is the general approach followed by developers maintaining the Linux build system. A similar conclusion is also observed for ant-based build systems [5]. Suvorov et al. perform an empirical study on build system migration [6]. Neitsch et al. analyze build systems for programs which are developed in multiple programming languages [7]. They identify major issues in building multi-language software, and explore the reasons why these issues occur. Tu and Godfrey perform a case study on build-time software architecture, and introduce the "code robot" architectural style [8]. Tamrawi et al. propose SYMAKE which processes Makefiles and produce symbolic build graphs (SDGs) [9, 10]. Xia et al. analyze and categorize bugs in Make, Ant, CMake, Maven, Scons, and QMake [11]. Zhao et al. investigate bugs in the build processes of 5 software systems including CXF, Camel, Felix, Struts2, and Tuscany [12].

## 6. CONCLUSION AND FUTURE WORK

In this paper, we investigate a new research problem: automatic inference of missed dependencies in build configuration files. To solve this research problem, we first construct a dependency graph using MAKAO [2], and then use 9 state-of-the-art link prediction algorithms to infer missed dependencies. The experiment results show that on average the preferential attachment (PA) algorithm performs best in terms of AUC which is a commonly used evaluation metric. PA can achieve AUC scores of 0.71-0.87 when analyzing the 7 systems.

In the future, we plan to investigate more build configuration files from large software systems to reduce threats to external validity. We also plan to develop another algorithm which can achieve better AUC scores.

**Acknowledgment** This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B03).

## 7. REFERENCES

[1] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: Statistical Mechanics and its Applications*, pp. 1150–1170, 2011.

[2] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *ICSM*, 2007, pp. 114–123.

[3] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan, "An empirical study of build maintenance effort," in *ICSE*, 2011, pp. 141–150.

[4] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," *EASST*, 2008.

[5] S. McIntosh, B. Adams, and A. Hassan, "The evolution of ant build systems," in *MSR*, 2010.

[6] R. Suvorov, M. Nagappan, A. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on kde and the linux kernel," in *ICSM*, 2012.

[7] A. Neitsch, K. Wong, and M. Godfrey, "Build system issues in multilanguage software," in *ICSM*, 2012.

[8] Q. Tu and M. Godfrey, "The build-time software architecture view," in *ICSM*, 2001.

[9] A. Tamrawi, H. Nguyen, H. Nguyen, and T. Nguyen, "Build code analysis with symbolic evaluation," in *ICSE*, 2012, pp. 650–660.

[10] ——, "Symake: a build code analysis and refactoring tool for makefiles," in *ASE*. ACM, 2012, pp. 366–369.

[11] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *QSIC*, 2013, pp. 200–203.

[12] X. Zhao, X. Xia, P. Kochhar, D. Lo, and S. Li, "An empirical study of bugs in build process," in *SAC*, 2014.