Balancing the Quality and Cost of Updating Dependencies

Damien Jaime, Pascal Poizat, Joyce El Haddad, Thomas Degueule

Balancing the Quality and Cost of Updating Dependencies

Damien Jaime
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
SAP France S.A.
F-92300, Levallois-Perret, France
damien.jaime@lip6.fr

Pascal Poizat
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
Université Paris Nanterre
F-92000, Nanterre, France
pascal.poizat@lip6.fr

Joyce El Haddad
Université Paris Dauphine-PSL, CNRS, LAMSADE
F-75016, Paris, France
joyce.elhaddad@lamsade.dauphine.fr

Thomas Degueule
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France
thomas.degueule@labri.fr

ABSTRACT

Keeping dependencies up to date is a crucial software maintenance task that requires significant effort. Developers must choose which dependencies to update, select appropriate target versions, and minimize the impact of updates in terms of breaking changes and incompatibilities. Several factors influence the choice of a new dependency version, including its freshness, popularity, absence of vulnerabilities, and compatibility.

In this paper, we formulate the dependency update problem as a multi-objective optimization problem. This approach allows for updating dependencies with a global perspective, considering all direct and indirect dependencies. It also enables developers to specify their preferences regarding the quality factors to maximize and the costs to minimize when updating. The update problem is encoded as a linear program whose solution provides an optimal update strategy that aligns with developer priorities and minimizes incompatibilities.

We evaluate our approach using a dataset of 107 well-tested open-source Java projects using various configurations that reflect real-world update scenarios and consider three quality metrics: dependency freshness, a time-window popularity measure, and a vulnerability score related to CVEs. Our findings indicate that our approach generates updates that compile and pass tests as well as the naive approaches typically implemented in dependency bots. Furthermore, our approach can be up to two orders of magnitude better in terms of freshness. By considering a more comprehensive concept of quality debt, which accounts for freshness, popularity, and vulnerabilities, our approach is able to reduce quality debt while maintaining reasonable memory and time consumption.

KEYWORDS

software maintenance, dependency graph, dependency update

1 INTRODUCTION

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails using external *software libraries*. Instead of creating new systems from scratch, developers incorporate libraries that provide the desired functionalities into their projects. These libraries expose their features through Application Programming Interfaces (APIs), which dictate the interactions between client projects and libraries. The way dependencies are managed varies across different software ecosystems [9]. Typically, it involves using a package manager or build system to automatically retrieve specific versions of dependencies from remote software repositories—along with their own (transitive) dependencies—in order to build a so-called *dependency graph* [28]. For example, JavaScript and TypeScript developers can use npm or Yarn to fetch dependencies from the npm registry, while Java developers can use Maven or Gradle to retrieve dependencies from the Maven Central repository.

Libraries continuously evolve to incorporate new features, bug fixes, security patches, refactorings, *etc.* [1, 4]. Clients must stay up to date with the libraries they use to benefit from these improvements and to avoid technical lag and the associated technical debt [5, 11, 29]. However, when a library evolves, it may introduce changes that break the contract previously established with its clients, resulting in syntactic and semantic errors [35]. Developers are thus faced with the challenge of maximizing the freshness and quality of their dependencies while minimizing the costs associated with updating them. This challenge is further complicated by the nature of dependency graphs: updating a single dependency can cause a snowball effect and result in incompatibilities with other indirect dependencies. As a result, clients sometimes hesitate to update their dependencies, raising security concerns [21] and making future updates even more difficult.

The problem of assisting developers in updating their dependencies has therefore attracted significant interest. This includes efforts to identify client code that is affected by breaking changes [24, 25], automatically migrate client code [36], or find versions that minimize the impact on the dependency graph. UPCY [8], in particular, is a novel approach that takes a library and a target version as input to construct a migration plan that minimizes the number of breaking changes within the graph induced by the update. While this approach is particularly suitable for updating a single dependency to a specific version (*e.g.*, to avoid a particular vulnerability, as exemplified by the recent log4shell mayhem), it is much less suitable for updating the entire dependency graph at once, which is critical for managing technical debt in decaying projects. Besides, breaking changes may not accurately reflect the actual impact an update has, as it has been empirically shown that many breaking releases do not impact client projects in practice [19, 25].

Migrating to the latest available version of each dependency may not always be the optimal choice. Developers must juggle various criteria to find a satisfactory solution, ranging from ensuring license consistency across projects [32] to minimizing security vulnerabilities [21] and easing the migration process [25]. In this paper, we propose to model the problem of updating a dependency graph as a custom multi-objective optimization problem. We formulate the optimization problem as a linear programming problem on a project-rooted extended dependency graph. Our approach is generic regarding the quality and cost metrics considered. As different developers prioritize these criteria differently, our multi-objective problem incorporates weights for each, hence supporting updates tailored to organizational rules or individual developers' preferences. While other criteria can be considered, we focus in our experimental evaluation on the joint use of three quality metrics: dependency freshness (to minimize the cost of future updates), a time-window popularity measure (as a proxy for community support), and a vulnerability score based on CVEs (as a proxy for security concerns). For the cost of change, we estimate the impact breaking changes introduced in a release have on the project.

We develop a new tool, GoblinUpdater, that automatically proposes an update plan from developer-defined preferences. GoblinUpdater targets the Java programming language and the Maven ecosystem and leverages the Maven Dependency Graph [2] and the enrichment capabilities offered by Goblin [16] to incorporate quality and cost metrics into dependency graphs. GoblinUpdater also leverages Maracas to mesure the impact of breaking changes on client code [25]. Our experiments evaluate the correctness, effectiveness, and scalability of GoblinUpdater on a dataset of 107 well-tested open-source Maven projects. We show that GoblinUpdater outperforms naive approaches while maintaining reasonable memory and time consumption.

The remainder of this paper is organized as follows. Section 2 introduces the problem of balancing the quality and cost of dependency updates and the existing approaches. Section 3 gives a general overview of our approach, Section 4 discusses the construction of project-rooted dependency graphs, and Section 5 the encoding of the problem as a linear program. Section 6 evaluates the benefits of our approach on various update configurations. Section 7 presents the threats to validity. Finally, Section 8 discusses some lessons learned in using linear programming for dependency update and Section 9 concludes the paper.

2 PROBLEM STATEMENT & RELATED WORK

To illustrate the challenges of balancing the quality and cost of updating dependencies, consider the dependency graph G shown in Figure 1. The figure depicts a simple root project p that declares two direct dependencies towards libraries l1 (in version l1-1) and l2 (in version l2-1), and inherits indirect dependencies towards libraries l3–l4. Each library offers a set of releases that act as candidates for replacing existing dependency versions (l1, for instance, offers releases l1-1, l1-2, and l1-3, with l-3 the most recent release). Migrating from one library version to another incurs a change cost, depicted with dotted arrows in Figure 1.

Updating the dependencies of project p involves finding a subgraph G' of G that satisfies ecosystem-specific well-formedness

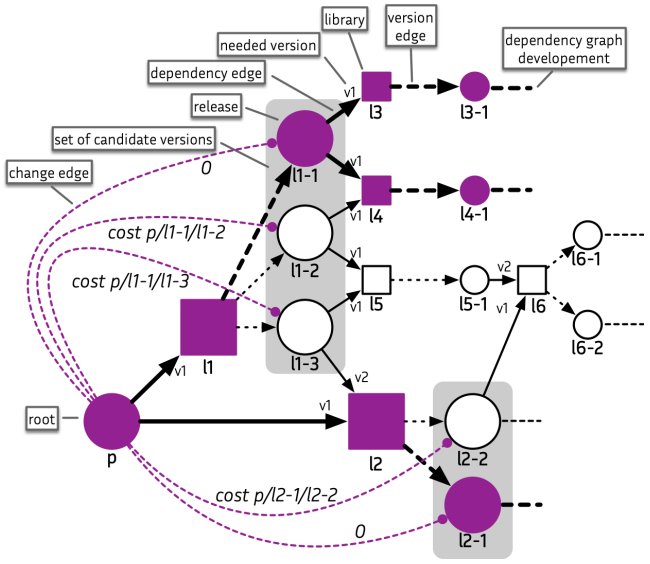


Figure 1: An extended dependency graph rooted in p. This dependency graph for p (purple/dark squares and circles) incorporates all alternative versions of p's direct dependencies (grey rounded boxes) and change edges with associated costs. The direct and indirect dependencies of these alternative versions are depicted with white squares and circles. The current dependencies of p are {l1-1, l2-1, l3-1, l4-1}. Alternative dependency graphs for p are {l1-1, l2-2, l3-1, l4-1, l6-1}, {l1-1, l2-2, l3-1, l4-1, l6-2}, {l1-2, l2-1, l4-1, l5-1, l6-1}, {l1-2, l2-1, l4-1, l5-1, l6-2}, {l1-2, l2-2, l4-1, l5-1, l6-1}, {l1-2, l2-2, l4-1, l5-1, l6-2}, {l1-3, l2-1, l5-1, l6-1}, {l1-3, l2-1, l5-1, l6-2}, {l1-3, l2-2, l5-1, l6-1}, and {l1-3, l2-2, l5-1, l6-2} (changes are underlined).

constraints (*e.g.*, one can only directly depend on a single version of a given library), maximizes the quality of each dependency in the graph rooted in p, and minimizes the cost of migrating to new versions. A given solution must specify the version of each dependency, whether direct or transitive, to ensure that no version is left open for the dependency resolver to pick arbitrarily. The goal is to find an optimal solution with respect to specific user-defined quality and cost preferences. Even in the simple example of Figure 1, combining every candidate version of each library yields ten candidate solutions G'. In real-world settings with dozens or hundreds of dependencies, the number of possible solutions quickly becomes unmanageable with naive algorithms and approaches.

Several factors can be considered when selecting a dependency. These can range from its freshness and the compatibility of its license with the project, to the absence of known security vulnerabilities (CVEs) or its overall popularity and community support. In this paper, we choose to focus on three quality metrics: freshness, popularity, and the number of known vulnerabilities. We choose these metrics because they have been used in prior work and reflect real-world concerns [21, 32]. However, our approach is both generic and extensible, allowing for the integration of additional criteria, such as licensing constraints, into the solution.

In Figure 1, consider the simple case where a developer wants to maximize the freshness of their dependencies while minimizing the cost of migrating to these versions. Updating would mean choosing {l1-3, l2-2, l5-1, l6-2}—assuming that these do not incur significant costs in terms of breaking changes. Another developer may choose to prioritize maximizing the freshness of dependencies and minimizing the presence of vulnerabilities, regardless of the cost of migrating to this new configuration. A successful approach should enable developers to (i) express their quality and cost preferences precisely and (ii) determine an optimal solution based on these preferences within a reasonable time.

Several tools have been developed to assist developers in managing their dependencies (*e.g.*, Dependabot [13], Renovate [26], and Greenkeeper [27]). These tools monitor the release of new dependencies and automatically create pull requests submitted for approval to project maintainers. They operate at the level of individual releases and do not aim to find a global solution that satisfies user-defined criteria. UPCY [8], on the other hand, is a novel approach to dependency update that takes as input one library to update and one target version to construct a migration plan that minimizes the number of breaking changes induced by version updates across the entire dependency graph. While this approach is well-suited for updating a single dependency to a specific version (*e.g.*, to avoid a specific vulnerability), it is much less suitable for updating the entire dependency graph at once. Besides, the number of breaking changes is a poor indicator of the actual impact of an update, as it has been shown empirically that many breaking releases have no practical impact on client projects [19, 25]. In contrast, our approach considers the *impact* of breaking changes on the root project p and allows for updating all project dependencies simultaneously. Hejderup et al. studied to which extent test suites of client projects can detect regression caused by dependency updates and found that tests can only detect 47% of artificial faults injected in direct dependencies and 35% of those injected in transitive ones [14]. They advocate that a successful approach to dependency update should incorporate static analysis to compensate the inadequacy of tests. Our approach follows this path by incorporating the precise cost of dependency updates directly within the dependency graph.

3 APPROACH OVERVIEW

In this section, we give an overview of our approach for updating a project's dependencies. Our approach is depicted in Algorithm 1. It takes as input:

- the project of interest, including its POM file (to retrieve direct dependencies) and its source code (to compute the cost of change),
- a non-empty set Q of user-chosen quality metrics, to which we add a specific cost of change metric (`cost`), yielding a set Q^+,
- a function w associating to each q in Q^+ a user-defined weight in $[0, 1]$, with w_q denoting the weight for q. We further require that $\sum_{q \in Q} w_q = 1$ (the sum of weights for quality metrics is 1). Note that this means that $\sum_{q \in Q^+} w_q$ is in $[1, 2]$.

The algorithm consists of two main steps: constructing the dependency graphs and solving the dependency update using linear programming.

Constructing the dependency graphs. In the first step, we start (line 1) by constructing a dependency graph called the *rooted dependency graph* (rDG) from p's direct dependencies. In addition to containing all direct and indirect dependencies for p, this graph also contains additional libraries and releases corresponding to a *potential for update*. This information is extracted from the whole dependency graph (*e.g.*, Maven Central). Values for the quality metrics of interest for the user are also computed and associated with each release in the rDG. Then, the rDG is extended into a *rooted extended dependency graph* (rEDG) with information related to the cost of change when switching from a library version to another one, based on the practical use of the library by project p (line 2).

Solving. In the second step, we formulate the update problem as a linear program using the rEDG, quality metrics, cost of change, and weight function. The objective is to determine the *optimal set of dependency updates* that balance quality improvement and cost of change. Since quantitative information (*i.e.*, values for quality metrics and change cost) can vary significantly in scale, we need to perform normalization first (line 3) to ensure a consistent basis for comparison when using linear programming. Then, we can proceed with the encoding itself (line 4). Finally, we use a linear programming solver (line 5) to find the optimal solution. This solution always exists, although in the worst case, it can be p's current set of dependencies. The solution indicates which part of the rEDG should be considered to update p's dependencies (line 6).

4 MODELS & GRAPH CONSTRUCTION

4.1 Models

Dependency graphs. Our first model is used to represent dependencies and versioning between libraries and their releases. Such models can be retrieved from software ecosystems to address ecosystem-wide research questions and support software-related maintenance processes. Our model is a formalization of the graph database used by Goblin [16].

Definition 1 (Dependency Graph). *A Dependency Graph (DG) G is a tuple* $(N_L, N_R, E_D, E_V, \text{req})$ *where N_L is a set of library nodes, N_R is a set of release nodes, $E_D \subseteq N_R \times N_L$ is the dependency relation (edges), $E_V \subseteq N_L \times N_R$ is the version relation (edges), and* req *is a version constraint function associating to each edge in E_D a version in a set Ver denoting semantic versions. We also define $N = N_L \cup N_R$ and $E = E_D \cup E_V$.*

Algorithm 1 Update project dependencies

Inputs: the project p (code and POM file), the set of metric values $Q^+ = Q \cup \{\texttt{cost}\}$, the weight function w
Output: p' an update of p

1: $G \leftarrow$ computeRDG($p.dependencies$, Q)
2: $G \leftarrow$ extendRDG(p, G)
3: $G \leftarrow$ normalize(G, Q^+)
4: $program \leftarrow$ generateLinearProgram(G, Q^+, w)
5: $solution \leftarrow$ solve($program$)
6: $p' \leftarrow$ update(p, $solution$)
7: **return** p'

An edge $e = (\mathsf{r}, \mathsf{l})$ in E_D denotes a dependency relation between release r and library l, with required version being $\text{req}(e)$. A preliminary experiment on our data revealed that, on Maven Central, only approximately 1% of dependency relations use range version requirements (*e.g.*, `[1.0, 2.0)`). Hence, we assume that Ver strictly corresponds to semantic versions, not ranges. An edge (l, r) in E_V denotes a version relation between l and r meaning that r is a version of l.

Rooted dependency graphs. We say that a DG G is *rooted* when G has a distinct node p in N_R and G contains only nodes and edges that are reachable from p. In this paper, we only use rooted DGs, or rDGs, since our objective is to update the dependencies of a project of interest that acts as the root of the graph. There are *different strategies to compute rDGs* when it comes to the versions of libraries. For libraries that are direct dependencies of the root, a strategy could, for instance, keep all versions, only versions newer than the one currently required by the root, or only non-patch versions. The same choice applies to libraries that are indirect dependencies of the root; one could decide to keep only required versions. The choice of a strategy to compute an rDG has implications on its size, the possible updates, and the time/memory required to compute the best update plan.

Extended dependency graphs. To support dependency updates, we extend rDGs with a new kind of edge denoting the cost of changing from one version to another. This is done using *change edges* and a cost function associated with them.

Definition 2 (Extended Dependency Graph). *An Extended Dependency Graph (EDG) G, is a tuple* $(N_L, N_R, E_D, E_V, E_C, \text{req}, \text{cost})$ *such that* $(N_L, N_R, E_D, E_V, \text{req})$ *is a DG and* $E_C \subseteq N_R \times N_R$ *is the change relation (edges), and* cost *is a cost function associating to each edge in* E_C *a cost in some abstract set Cost (a measure of change debt is typically used here, see in the sequel). Further, we require that there can only be an edge* $(\mathsf{r}_1, \mathsf{r}_2)$ *in* E_C *when there is an edge* $(\mathsf{r}_1, \mathsf{l})$ *in* E_D *and an edge* $(\mathsf{l}, \mathsf{r}_2)$ *in* E_V.

Rooted extended dependency graphs. As for DGs and rooted DGs, we have EDGs and rooted EDGs, or rEDGs. As for the computation of rDGs, the computation of change edges is a matter of *strategy*. The *global* strategy is to compute the maximal set of possible change edges (*i.e.*, all that fulfill the requirements in Definition 2). The *local* strategy, on the other hand, is to compute only change edges outgoing from the root.

Illustration. An example of an rEDG is given in Figure 1. There, the strategy for the computation of the rDG is to include all versions for direct dependencies. For indirect dependencies, the strategy is to include only the versions that are required (*e.g.*, the two versions of l6). Even if l3, l4, or l5 had more than one version, only one would be present in the graph. Finally, the strategy for computing the change edges to obtain the rEDG is to include all possible change edges for the root and none for the other nodes. Other combinations of strategies would have produced different rEDGs.

4.2 Rooted dependency graph construction

To construct the rDG for a project p, we rely on features provided by Goblin [16]: the whole Maven Central dependency graph (stored in a graph database), the possibility to extract sub-graphs using predefined REST routes or Cypher queries [23], and the possibility to compute and insert additional metrics on the nodes and edges of the graphs.

A release strategy dictates where dependencies are expanded into candidate versions: only for direct dependencies (the "local" strategy) or for all direct and indirect dependencies (the "global" strategy). Even with a local strategy, multiple versions can co-exist for an indirect dependency, for instance l6 in Figure 1.

Regarding the additional metrics required in our experiments (Section 6), we reuse Goblin's "CVE" and "Freshness" metrics associated with release nodes [16]. For the latter, we reuse as-is the data computed by Goblin. For CVEs, we perform a post-treatment since Goblin only provides us with the list of CVEs that impact a release. To make this usable for updating, we compute the number of CVEs in each of four criticality categories (low, moderate, high, critical) and aggregate them using coefficients from the Fibonacci suite. We also extend Goblin with a new popularity metric we rely on.

4.3 Rooted extended dependency graph construction

Once the rDG is obtained, it must be extended with the edges in E_C and the values in cost to integrate the cost of change in the update process. The first step is to determine the desired change edges. This decision is based on strategy, as previously discussed. The local strategy involves computing change edges only between the root and its direct dependencies, as shown in Figure 1, while the global strategy involves computing change edges whenever a library in the rDG has multiple versions to consider indirect change costs. For example, in Figure 1, the global strategy would involve adding six additional change edges: (l2-2, l6-1), (l2-2, l6-2), (l1-3, l2-1), (l1-3, l2-2), (l5-1, l6-1), and (l5-1, l6-2).

The idea behind the computation of the cost of change is to accept possible breaking changes in exchange for a better overall quality of the dependencies. The cost of change is computed as follows for each change edge. Suppose a release r that depends on a library l and uses its version r_i (as specified by req). Suppose we want to compute the cost on the change edge between r and another version of l, say r_j. To compute the cost, we invoke the tool Maracas [25] with: the jar file of r_i, the jar file of r_j, and the source code of r. Maracas computes all breaking changes (removed methods, changed exceptions, altered visibilities, *etc.*) between r_i and r_j, and the impact these changes would have on the code of r (unresolved methods, uncaught exceptions, *etc.*) as a set of *broken uses*. The number of broken uses, *i.e.*, the number of code locations in r that would be impacted by the update from r_i to r_j, constitutes the cost on the corresponding change edge.

It should be noted that in the case one accepts possible breaking changes in indirect dependencies (*e.g.*, for l5-1 (resp. l2-2) using l6-1 (resp. l6-2) instead of l6-2 (resp. l6-1) one cannot use Maracas to compute the cost of change as it not suited to measure the impact of indirect dependencies. Instead, we use the Japicmp tool [22]. Contrary to Maracas, Japicmp can only compute the list of breaking

changes between two releases, not their impact on client code, so the extracted cost of change is more pessimistic in this case.

5 LINEAR PROGRAMMING FOR DEPENDENCY UPDATE

In this section, we present how dependency update can be encoded as a linear program that maximizes the quality of dependencies while minimizing the cost of change.

In a linear program [6] three elements are required: a set of *decision variables*, a set of *constraints*, and an *objective function*, where both the constraints and the objective function must be linear. The output of a linear program is the optimal value of the objective function (maximum or minimum) and the corresponding values of the decision variables that achieve this optimum.

5.1 Graph-based decision variables

We use the following decision variables:

- for each library l in N_L, a binary variable v_l^{lib} representing whether l is present (equals 1) or not (equals 0) in the solution;
- for each release r in N_R, a binary variable v_r^{rel} representing whether r is present or not in the solution;
- for each change edge between release nodes r and r' in E_C, a binary variable $v_{rr'}^{\text{chg}}$ representing whether the edge is present or not in the solution.

These decision variables are used in the sequel to express constraints that an update solution must fulfill.

5.2 Conditions & constraints for a valid update

Several conditions are required for an update solution to be correct (whether optimal or not):

- for releases, (a) the root is present, (b) if a release (including root) is present then all its dependencies (libraries) are present, and (c) if a release (but for root) is present then the library it is a version of is present.
- for libraries, if a library is present, then (d) exactly one of its versions (releases) is present and (e) at least one of its dependants (releases) is present.
- for change edges, (f) if a change edge (r, r') is present then both r and r' are present, and (g) conversely, if two nodes r and r' connected by a change edge are present then the change edge is present.

Here, "present" means that some node or edge is included in the solution, *i.e.*, the corresponding variable is set to 1.

The set of linear constraints that encode these conditions is the following one, the correspondence being (1) ⇔ (a), (2)⇔ (b), (3)⇔ (c) ∧ (d), (4)⇔(e), and (5) ⇔ (f) ∧ (g).[1]

$$v_p^{\text{rel}} = 1 \tag{1}$$

$$\forall d = (r, l) \in E_D, v_l^{\text{lib}} \geq v_r^{\text{rel}} \tag{2}$$

$$\forall l \in N_L, \sum_{l_i \in \{l_i \mid (l,l_i) \in E_V\}} v_{l_i}^{\text{rel}} = v_l^{\text{lib}} \tag{3}$$

$$\forall l \in N_L, \sum_{r \in \{r \mid (r,l) \in E_D\}} v_r^{\text{rel}} \geq v_l^{\text{lib}} \tag{4}$$

$$\forall e = (r, r') \in E_C, v_{rr'}^{\text{chg}} = v_r^{\text{rel}} \times v_{r'}^{\text{rel}} \tag{5}$$

which, using linearization, becomes $\forall e = (r, r') \in E_C$,

$$v_{rr'}^{\text{chg}} \leq v_r^{\text{rel}} \tag{6a}$$

$$v_{rr'}^{\text{chg}} \leq v_{r'}^{\text{rel}} \tag{6b}$$

$$v_{rr'}^{\text{chg}} \geq v_r^{\text{rel}} + v_{r'}^{\text{rel}} - 1 \tag{6c}$$

[1] A proof is available in our reproduction package [18].

5.3 From MO-MC to SO-SC optimization

Now that we have defined what a correct update solution is, we would like to find one solution that is indeed optimizing conflicting criteria, *i.e.*, quality and cost. Multi-Objective Multi-Criteria Decision-Making is the field concerned with solving such problems [33]. The difficulty there stems from the presence of more than one criterion, with some to be maximized and some to be minimized. Multiple Pareto optimal solutions usually exist in such a case. Therefore, many methods to solve Multi-Objective Multi-Criteria (MO-MC) optimization problems proceed by transforming them into a Single-Objective Single-Criterion (SO-SC) problem.

From MC to SC. We use one of these methods, called Simple Additive Weighting (SAW) [10]. This method is based on *weights* assigned by the developer to each criterion. SAW consists in combining multiple criteria values into a single criterion value using a weighted sum. Before this phase, SAW requires a *normalization* phase to scale criteria values and compare the ratings of all existing solutions. Some of the criteria are positive (w.r.t. maximizing an objective function), *i.e.*, the higher the value, the higher the quality. This includes popularity metrics such as stars or downloads. Other criteria are negative (w.r.t. maximizing an objective function), *i.e.*, the higher the value, the lower the quality. This includes criteria such as the cost of change or the vulnerability score related to CVEs that we use in our experiments. There are several normalization techniques [30]. To choose one of them, we need to take into account the objective function (should it be maximized or minimized) and the nature of the criteria (positive or negative w.r.t. the objective function).

From MO to SO. The linear programming solver attempts either to maximize or minimize the value of the objective function by adjusting the values of the decision variables while enforcing the constraints. When updating dependencies, our goal is to maximize certain quality metrics (*e.g.*, popularity) while minimizing change cost and other quality metrics (*e.g.*, vulnerabilities). To make this amenable to a SO problem, we adopt the following perspective: we consider that *quality metrics are a measure of a form of quality debt*, and thus they should also be minimized. This allows us to focus solely on criteria to minimize. Therefore, positive metrics like popularity are treated as negative criteria for minimization. Conversely, negative metrics like CVE-based vulnerabilities or release age are treated as positive criteria for minimization. This inversion is necessary because we are working with an objective function that must be minimized rather than maximized.

Normalization. We can now define our normalization process. Since we use an objective function to be minimized, values q_i for a positive metric q are scaled according to $\frac{q^{\max}-q_i}{q^{\max}-q^{\min}}$ where $q^{\min}$ and $q^{\max}$ are, respectively, the minimal and maximal possible values for q. For example, suppose that release popularity ranges from 10 stars to 100 stars. The normalized value for a release with 80 stars (which is quite popular) is $\frac{100-80}{100-10} = 0.22$. Accordingly, values for a negative metric are scaled using $1 - \frac{q_i^{\max}-q_i}{q_i^{\max}-q_i^{\min}}$. For example, suppose that release age ranges between 5 days and 365 days. The normalized value for a release that is 300 days old (which is quite old) is $1 - \frac{365-300}{365-5} = 0.82$.

To increase the solver efficiency [3] and to make the possible feedback of quality debt enhancement more legible for developers, we apply a multiplicative factor $k = 1000$ when normalizing. For the examples above, one would thus get values of 220 and 820.

5.4 Optimization objective function

We must now establish the objective function of the linear program. This function will be minimized to identify the optimal solution for updating dependencies, referred to as *sol*. Our proposed objective function is outlined as follows:

$$Min\Big(\big(1 - w(cost)\big)\times Quality_{sol} + w(cost) \times Cost_{sol}\Big) \tag{7}$$

where, $w(cost)$ is the weight assigned by the developer to the cost of change metric, and $1 - w(cost)$ is the weight of the overall quality of the solution, referred to as $Quality_{sol}$. This overall quality is in turn computed using the following aggregation function:

$$Quality_{sol} = \sum_{\forall q \in Q} w(q) \times f_q \tag{8}$$

where $w(q)$ is the weight assigned by the developer to the quality metric q, and f_q is the aggregation function for the computation of the quality metric q of solution *sol*. The aggregation function of any metric depends on the nature of the criterion it seeks to aggregate. For instance, the aggregation function for vulnerabilities related to CVEs is defined by the sum of the vulnerability values of each release within the solution *sol*, as illustrated below:

$$f_q = \sum_{\forall r \in sol} q(r) \times \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \tag{9}$$

where $q(r)$ is the vulnerability value of release r and $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$ is used to prune releases that are not in the solution (remind that $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$ is 1 if release r is present in the solution, else it is 0).

Similarly, the cost of change metric values must be aggregated. The cost of change for the solution *sol* is computed as the sum of the cost of change values associated with change edges within *sol* as follows:

$$Cost_{sol} = \sum_{\forall (r,r') \in sol} cost(r, r') \times \mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} \tag{10}$$

again, with $\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}}$ being used to prune change edges that are not present in the solution.

5.5 Retrieval of the updated set of dependencies

Given the solution, the optimal set of (direct and indirect) dependencies correspond to all nodes r in N_R such that $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$=1 in the solution. Yet, a specificity of the Maven package manager is that whenever there are several paths from the project to some library l, the shortest path is used to discriminate the version of l to be selected. In Figure 1, for example, there are two paths from the root to l6: one ending with l5-1, requiring version 2, and one ending with l2-2, requiring version 1, the latter being the shortest. This means that if the best solution is (l1-3, l2-2, l5-1, l6-2), one cannot just update p's dependency file using l1-3 instead of l1-1 and l2-2 instead of l2-1 because l6-1 would be used and not l6-2. A simple solution to this is to update p's dependency file with all releases in the solution, here l1-3, l2-2, l5-1, and l6-2. Although this approach may lock the versions and complicate the dependency file by adding indirect dependencies to the root, it ensures the quality we commit to.

6 EXPERIMENTAL EVALUATION

In this section, we report on the empirical evaluation of our approach, driven by the following research questions:

RQ1 (correctness) Does our approach generate correct updates? For zero-cost updates, does our approach generate updates that compile and pass tests?

RQ2 (effectiveness) Which quality gain and update cost can be expected when using our approach and how does it compare to naive approaches?

RQ3 (performance and scalability) How does our approach perform on different graph sizes and with different strategies?

In this evaluation, we compare our solution to three naive approaches to update direct dependencies that are typically implemented in popular dependency management bots. The *max version* approach (MMP) always picks the latest release, the *max version with same major* approach (mMP) picks the latest release within the same major version, and the *max version with same major and minor* (mmP) picks the latest release within the same major and minor versions. The data and results discussed in this section are available in our replication package [18].

6.1 Subject application & methodology

Choice of metrics. Different quality metrics can be used to measure the quality of releases, each representing some quality aspect. Although our approach is open-ended and can accommodate new metrics beyond the ones discussed in this paper, we consider the following quality metrics in our evaluation. We remind the reader that these are normalized before being fed to the LP solver.

CVEs Our vulnerability score is based on the set of CVEs that (directly) impact a release. This set, denoted as C_{r} for a release r, is computed using Goblin. To obtain the score, we use the formula $\sum_{c \in C} k(c) \times nb(c, C_{\mathrm{r}})$ where $C = \{low, moderate, high, critical\}$ denotes the criticality of a CVE, $nb(c, C_{\mathrm{r}})$ counts the CVEs of criticality c in C_{r}, and k is a function associating a coefficient in the Fibonacci suite (2, 3, 5, and 8) to each element c in C.

Freshness Freshness [7] materializes the age of a release. Given some release r of a library l, freshness can be computed using

Table 1: Configurations used for experiments. F = freshness, P = popularity, CVE = vulnerability score.

id	weights			cost	strategy	
	F	P	CVE		for releases	for cost
cfg1	1.0	–	–	0.0	global	local
cfg2	1.0	–	–	0.5	global	local
cfg3	1.0	–	–	≤ 0.0	global	local
cfg4	1.0	–	–	≤ 0.0	local	global
cfg5	0.4	0.4	0.2	0.5	global	local
cfg6	0.4	0.4	0.2	0.5	local	local

the release date of r and a more recent date of reference, *e.g.*, the present day, the day the ecosystem information was last extracted, or the date of the latest release of l. In our experiments, we use the latter which is directly computed by Goblin.

Popularity There are several possibilities for popularity too [34]: "stars" on social development forges, number of downloads, *etc.* The former requires all libraries in the ecosystem to be present on the social network, which is not always the case. Further, stars and other metrics often apply to libraries, not releases. The number of downloads is influenced by inflation issues [20] and the passage of time, with older releases naturally accumulating more downloads. A mitigation for the first issue is to use the number of dependants and a mitigation for the second issue is to use a time window. Thus, in our experiments, we use the number of dependants in the ecosystem over a 1-year window and have extended Goblin accordingly.

Configurations. Our approach exposes a high degree of variability regarding graph construction strategies and the weights one can assign to the quality metrics. To evaluate our approach accurately in different situations, we define six representative configurations, depicted in Table 1. Each configuration specifies weights for each of the quality metrics, as well as for cost. Although our tool supports expressing an upper bound on the number of vulnerabilities in the solution, we do not evaluate it here. Similarly, the cost metric allows expressing both weights (*e.g.*, 0.5 for cfg2) and hard constraints (*e.g.*, ≤ 0.0 in cfg3, which forces the cost of the solution to be zero). While weights and constraints can be mixed, we do not evaluate this scenario here. The last two columns specify the strategies used in the configurations. For releases, it means either developing alternatives to direct dependencies only (local) or for all libraries in the graph (global). For cost, it means computing change edges and costs either only at the root level (local) or whenever a library has more than one version (global). Costs are computed either using Maracas at the root level or Japicmp at any other level. Figure 1 is an example where a local strategy is used for both releases and costs. With a global strategy for releases, possibly more versions of l3–l6 would be present. With a global strategy for costs, there would be change edges between the two dependants of l6 and the two versions of this library (hence, four more change edges). In addition, all the configurations here use a strategy that only considers releases newer than the ones currently used in the project to be updated, *i.e.*, our approach will never downgrade a dependency.

Table 2: Demographics of our dataset of 107 Java projects

	Q1	Q2	Q3	min	max
p's direct dependencies	4.5	7	9	2	22
releases (N_R)	138	336	2771	16	39474
libraries (N_L)	12.5	40	134	4	960
dependency edges (E_D)	114.5	394	7066	9	141429
versions edges (E_V)	137	335	2770	15	39473

We use configurations cfg1 to cfg4 for **RQ1** and **RQ2** under various update strategies. These configurations only use freshness as a quality metric to provide a fair comparison with naive approaches that only aim to update dependencies to the latest available version. cfg1 does not account for cost and is thus the one closest to naive approaches. cfg2 gives an average relevance to cost. cfg3 enforces the absence of cost for direct changes. cfg4 enforces the absence of cost for both direct and indirect changes. We use configurations cfg5 and cfg6 for **RQ3**. These two configurations exploit the full potential of our approach by combining different quality metrics and weights with different strategies.

Dataset. To evaluate our approach, we reuse the dataset of 462 well-tested Java projects from Hejderup and Gousios [14]. Additionally, we filter projects that cannot be cloned (8), lack a root `pom.xml` file (13), or feature multi-module `pom.xml` hierarchies which are not supported in our tool (216). We also filter stall projects that have not been updated since 2020 (77), that cannot be analyzed by Maracas (8), that have missing direct dependencies (14), or that take prohibitively long time to analyze (19). In the end, we obtain a subset of 107 well-tested and active Java Maven projects that can be analyzed with our tools. Table 2 details descriptive statistics of the resulting dataset in terms of rEDG size for cfg1 (different configurations yield different graph sizes).

Comparing solutions. Comparing solutions requires a common referential. The space of all possible solutions, *i.e.*, the rEDG and the normalized values for the different quality and cost metrics, play this part. As an illustration, Figure 1 contains both the solution obtained with our approach and other solutions that could be obtained with naive strategies such as using the most recent releases of the direct dependencies ({l1-3, l2-2, l5-1, l6-1} due to Maven's shortest path semantics) or using the most recent zero-cost releases ({l1-2, l2-1, l4-1, l5-1, l6-2} assuming that cost p/l1-1/l1-2 is zero and cost p/l1-1/l1-3 and cost p/l2-1/l2-2 are non-zero). To compute the aggregated quality value, the cost value, and the global value of a solution is then just using the formulas $Quality_{sol}$, $Cost_{sol}$, and their weighted sum, as given in Section 5.4. This does not require the use of the solver, only the rEDG and normalized values for its release nodes and change arcs.

Methodology. For **RQ1** (correctness), we select each project that can be successfully compiled and tested in our environment before updating dependencies. For these projects, we launch our tool with configurations 1 to 4 to retrieve the proposed updates and compare them with those obtained from the three naive approaches (MMP, mMP, and mmP). We update the `pom.xml` file according to the solution before building and testing the project again to check whether compilation and tests are still successful.

For **RQ2** (effectiveness), we compare the quality and cost obtained from our approach and the three naive ones. All the values given in this section are normalized on the same graph so that they can be compared.

Finally, for **RQ3** (performance and scalability), we run our tool with cfg5 and cfg6 on all projects within our dataset. For each execution, we retrieve various operational metrics such as execution times, graph size, memory usage, and the obtained solutions. Between each launch, we clean the Goblin database to avoid memoization and caching biases between subsequent runs.

Experimental setup and reproducibility. All the experiments presented in the rest of this section were carried out on a Windows Server 2019, 64 GB memory, 8 CPUs Intel(R) Xeon(R) CPU E7-8880 v4 @2.19GHz on a Docker instance using the openjdk:17-jdk image (Oracle Linux 8). The Maven Central dependency graph is the version dated April 12, 2024 [15] and the CVE database is dated May 07, 2024 [18]. We used version 1.0.0 of our GoblinUpdater tool [17] in the experiments, version 2.1.0 of Goblin, version 0.5.0 of Maracas, version 0.17.2 of Japicmp, and version 9.8 of OR-tools.

6.2 Results

RQ1 (correctness). To address post-update correctness, it is mandatory to use projects that compile and pass tests before any update attempt. Among the 107 projects in our dataset, only 48% compiled and passed tests in our Docker environment, yielding a total of 51 projects. This is due to several factors, *e.g.*, Java version, OS compatibility, and license requirements. The difficulty of reproducing passing builds is a common issue that has been reported in other studies with similar failure rates [12]. We excluded two more projects because cfg4, which calculates transitive costs, could not be completed in two days. This highlights the substantial computational resources required to implement a full-cost approach. The analysis in this section considers the remaining 49 projects.

Figure 2 shows the percentage of projects that still compile and pass tests after the update suggested by naive approaches and different configurations of our approach. Note that, for cfg3, our approach did not find a zero-cost solution for 17 projects, so their solution graphs remain unchanged and the projects continue to compile and pass tests successfully. The same situation arises for 28 projects in configuration cfg4. We can see that the local zero-cost approach (cfg3) does not always yield a solution that compiles and passes tests. This is because this configuration attempts to update all dependencies in the graph (not just direct dependencies like naive approaches) but only considers the cost associated with direct dependencies. To maximize the chances of finding a zero-cost solution, we use cfg4, which combines Maracas to compute the cost for direct dependencies and Japicmp to compute the cost for indirect dependencies. With this zero-cost configuration, we achieve better results than any other approaches. However, it does not achieve 100% success due to arbitrary constraints in the configuration of one project that arbitrarily enforce specific versions and licenses to be used. This stresses the importance of taking license consistency into account when updating software dependencies. It is also interesting to note that naive approaches that update direct dependencies to the latest release within the current major version or current major/minor version also yield projects that cannot be compiled or tested. This confirms that one cannot fully trust semantic versioning within the Maven ecosystem [25].

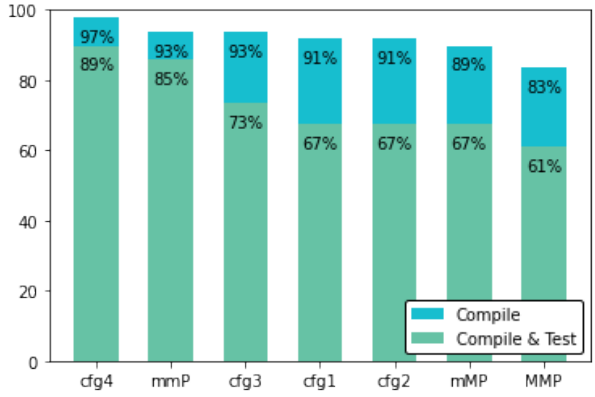


Figure 2: Correctness of the solutions generated by different approaches and configurations

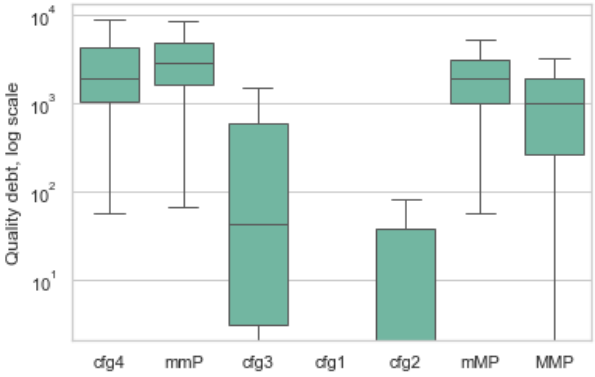


Figure 3: Freshness of the solutions generated by different approaches and configurations

RQ2 (effectiveness). The data presented in this section is derived from the same 49 projects discussed above, illustrating the correlations among compilation, testing, quality, and cost. cfg1, cfg2, and cfg3 share identical graphs and metrics, thus ensuring uniform normalization in terms of quality and cost values. In contrast, cfg4 employs a different graph, and naive approaches lack quality and cost values. To standardize the normalization process for all solutions and enable comparison, we have applied normalization to cfg4 and the naive approaches' graphs using values from the graph of one of the earlier configurations.

Figure 3 shows the varying levels of freshness across solutions generated by different approaches. Since quality is assessed over the entire graph, even the naive approach of setting all direct dependencies to their most recent version fails to deliver optimal quality in the majority of cases. This highlights the importance of taking transitive dependencies into account when assessing the quality of software dependency graphs. Configuration cfg1 consistently yields solutions of maximum quality as it ignores cost constraints.

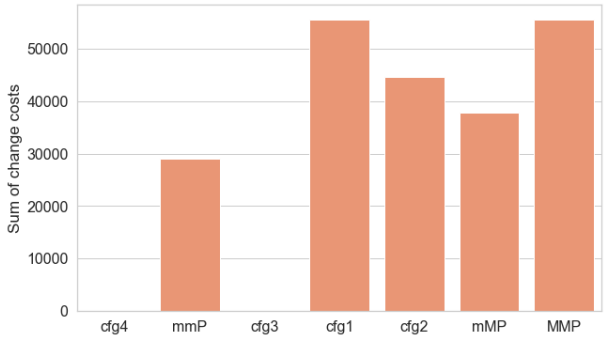


Figure 4: Cumulative cost of change of the solutions generated by different approaches and configurations

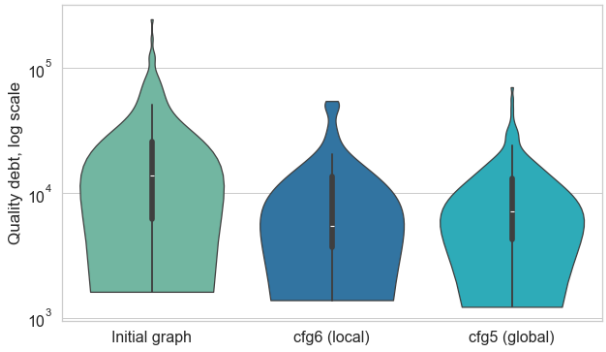
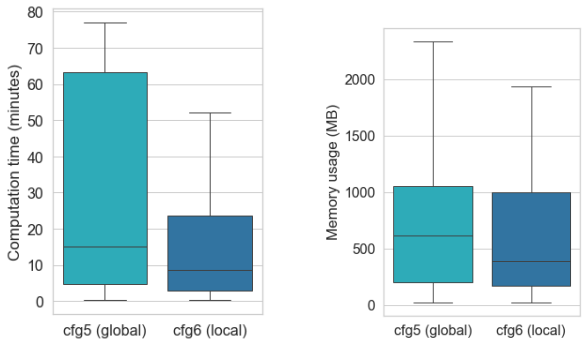


Figure 5: Quality debt before and after update



(a) Execution time **(b) Max memory usage**

Figure 6: Time and memory for updating projects

This approach selects the most recent versions for all nodes in the graph, thereby ensuring the highest possible quality. Configuration cfg2, which maintains a quality/cost balance of 50%, provides solutions of average quality and cost. Finally, configurations cfg3 and cfg4 prioritize zero-cost solutions and are paying the price in terms of quality.

Figure 4 shows the varying costs associated with the solutions generated by different approaches. Here, the results are normalized based on the local cost metrics between root and direct dependencies. Figure 4 presents the aggregate costs for updating projects to fit with the new direct dependencies, as calculated by Maracas, across different approaches. Configurations cfg3 and cfg4 consistently provide solutions with the minimum cost because they constrain the solver to achieve zero-cost solutions. Configuration cfg1 gives the same cost as the naive MMP method, as the cost is only calculated locally. The same direct dependencies are used, with the transitive dependencies being the point of variation. Finally, the balanced configuration cfg2 yields update costs between mMP and MMP but delivers much better quality (Figure 3). Our different configurations deliver results that are in line with expectations regarding quality and cost. It is up to the developer to select the optimal configuration to meet specific requirements by adjusting the tool's parameters accordingly.

RQ3 (performance and scalability). In this experiment, we use the entire dataset of 107 projects as successful compilation and testing of the projects is not required. We start by comparing the difference in expected quality gain between a global (cfg5) and a local (cfg6) approach. As a reminder, the local approach considers all versions of direct dependencies, while the global approach considers all versions of (possibly transitive) dependencies. Therefore, the global approach considers additional solutions but increases the size of the solution graph. Figure 5 shows the distribution of initial project quality debt and that of a local and a global approach. Surprisingly, the difference in quality between the local and global approaches is small. The local approach delivers on average a reduction in quality debt of 57% and the global approach 60%.

Let us now compare the distribution of average execution time (Figure 6.a) and memory used (Figure 6.b) for these two approaches. Even if memory consumption does not vary significantly between the two approaches, the execution time differs greatly, going from an average of 19 min locally to 120 min globally (+533%) when considering outliers. The global approach yields much longer calculation times for very little potential quality benefit. Therefore, we believe that the average user should most likely opt for the local approach, though the global approach may still be useful in certain scenarios.

Figure 7 dives deeper into different phases of our approach. In the global case, the generation of the graph and the solving phase dominate execution times. The graph generation includes the creation of the rDG and the weaving of metrics onto nodes and edges (realized using Goblin). In fact, the popularity metric consumes a lot of time as it requires retrieving the associated library, identifying the dependents using the appropriate version, and retrieving these dependents to apply the 1-year time window. This process is extremely time-consuming for popular libraries. In normal circumstances, Goblin memoization would help greatly, but we deactivated it in our experiments to ensure fair comparisons. The computation time of our approach varies greatly with the chosen quality metrics and their individual computation times. Also, solving time represents a significant cost, and as we would expect, it increases with the size of the graph and the number of considered constraints. The generation of change edges and the computation of costs remain consistent in terms of time across both configurations. Since both

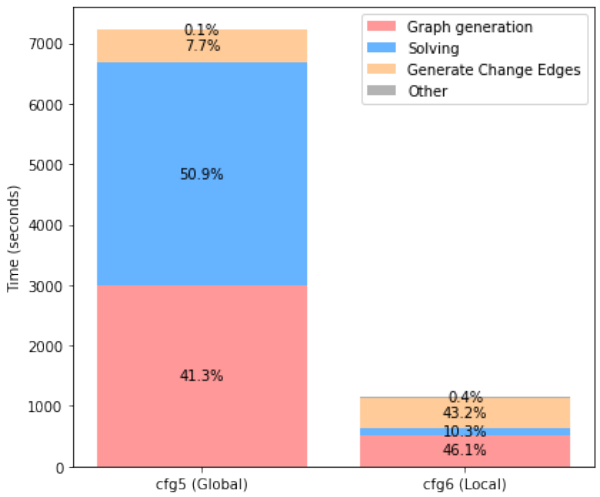
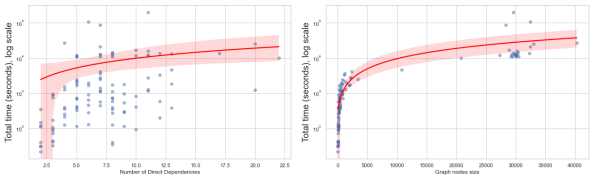


Figure 7: Execution time distribution



(a) Time (log) / direct dependencies **(b) Time (log) / graph nodes**

Figure 8: Relation between size and computation time

configurations rely on a local cost, they produce the exact same change edges.

Finally, we look at how to estimate the computation time of our approach on a project. In Figure 8.a, we can see that the number of direct dependencies does not significantly impact the total computation time. On the other hand, we can see from Figure 8.b that the total number of nodes in graphs does. This means that, unfortunately, computation times cannot be reliably estimated from the number of direct dependencies. To conclude, our tool's execution times mainly depend on the size of the graph, the number of constraints in the problem, and the chosen quality metrics.

7 THREATS TO VALIDITY

For the construction of our threats to validity, we follow the structure proposed by Wohlin et al. [31].

7.1 Internal & construct validity

In our approach, we operate under the hypothesis that the external tools we rely on (Goblin, Maracas, Japicmp, and OR-tools) yield accurate results and that the ecosystem dependency graph and metric values associated with release nodes, provided by Goblin, are both complete and accurate. This is reasonable as these tools have been independently evaluated by other researchers. As we used tests to check for update correctness, it should be noted that Hejderup and Gousios have shown that tests may have a low coverage of calls to dependencies (58% for calls to direct dependencies and 21% for calls to indirect ones [8, 14]). To mitigate this risk, we have used a dataset of well-tested projects. Furthermore, for calls to direct dependencies, we alleviated the issue with our cost of change computation using Maracas, which plays the role of the Updatera tool proposed in [14] for finding semantic breaks but goes further in being integrated *within* the update process itself. Yet, this still means that correctness computed with tests is a lower bound [8].

Our approach assumes that dependency versions are fixed, thereby excluding version ranges (*e.g.*, `[1.0, 2.0)`). This is not an issue in the Java/Maven context since the use of dependency ranges is very rare. To verify this, we have examined the whole Maven Central dependency graph as of January 26, 2024. It revealed that only 1.12% of dependencies (*i.e.*, 1,173,629 dependencies out of 104,949,615) employ ranges. Our tool uses a POM file to infer the list of direct dependencies of a project. The adaptation to other file formats (*e.g.*, Gradle's build files) should be straightforward. At present, the cost of change is based on counting the number of broken uses identified by Maracas and/or Japicmp. This can be simplistic, as one can easily imagine that the cost associated with a renamed method differs from that of a removed method, or that n identical changes yield a lesser cost compared to n distinct changes. Enhancing the cost function is a priority for future evolution perspectives. Finally, for the experimental evaluation part, we have discarded certain projects from the study that were prohibitively long to analyze, thus reducing average calculation times.

7.2 External validity

Our approach does not account for libraries that are not hosted on Maven Central, such as those available through GitHub Package or proprietary dependencies internal to a company. In practice, Goblin could be extended for this, provided specific miners are developed. While our approach is adaptable to different software ecosystems, the primary concerns regarding generalization stem from the Java/Maven specifications. First, in contrast with Maven, ecosystems such as npm allow for multiple versions of a library to coexist in a project. The LP encoding can be modified for this, relaxing the single library version constraint. The absence of consideration for version ranges may be a challenge when applying our tool to ecosystems like npm where version ranges are used extensively.

8 DISCUSSION

In this section, we present lessons learned in the use of our GoblinUpdater tool. Our approach and tool offer different degrees of freedom:

- DF1: the strategy to retrieve an rDG from a project p, *i.e.*, where to compute sets of candidate versions (locally at the root direct dependencies or globally) and how to compute them (all versions, more recent versions),
- DF2: the strategy to obtain an rEDG from an rDG, *i.e.*, where to compute change edges (locally at the root direct dependencies or globally) and how to estimate cost (Maracas and/or Japicmp),
- DF3: where to compute the quality metrics (on the Goblin side, on the GoblinUpdater side, or both),

- DF4: the chosen set of quality metrics, associated weights, and possible hard constraints for CVEs and cost.

We observed that global and version-exhaustive strategies for DF1 are costly, while using a local strategy and limiting the set of versions of a library to the ones newer than the current one is more efficient. This is reasonable considering the use one would make of our tool, a notable counter-example being, *e.g.*, the recent `xz` backdoor (CVE-2024-3094) which required downgrading from versions 5.6.0/5.6.1 to an earlier one.

The number of change edges selected by DF2 is also an important parameter. The computation of change edges is quite efficient. Yet, for each of them a JAR file must be retrieved (which can be done once, offline) and Maracas or Japicmp has to check for all possible breaking changes, which is costly.

Metrics can be computed, following DF3, either on the Goblin side, on the GoblinUpdater side, or a combination of both. The first solution requires forking and extending the Goblin tool, but then the memoization mechanisms available in Goblin speed up the computation of metrics on releases that have already been analyzed. This is the solution we have used for our popularity metric. The second solution is easier but does not benefit from Goblin's Neo4J-based algorithms nor from memoization. The last one, where basic "atoms" are computed by Goblin and metrics are computed on top of these provides a good balance.

DF4 enables a personalized update solution with selected metrics and their respective weights chosen either at the organization or developer level. This choice is also connected to DF3: if a particular metric is of interest, one could incorporate it in Goblin to benefit from memoization. Finally, RQ1 made clear the importance of license consistency. To account for this in updates, one could select a set of compatible acceptable licenses, check each dependency against them, and set a limit constraint on the solution (as is currently done for CVE and cost).

9 CONCLUSION

Updating dependencies is a crucial practice in software development for projects relying on external libraries. While some developers might resist updating their project's dependencies to avoid the cost of change, others might prioritize finding a middle ground between the immediate costs of updating and the long-term benefits of maintaining up-to-date dependencies. This paper advocates for balancing the benefits of maintaining up-to-date dependencies and the costs of updating. We introduce a multi-objective optimization approach to the dependency update dilemma, aiming to identify the most beneficial update solution based on criteria such as popularity, freshness, vulnerability, and the minimization of breaking changes. We implement our optimization approach in a new tool GoblinUpdater, available online [17], and show using a dataset of 107 well-tested open-source Maven projects that it successfully finds update solutions and outperforms the naive approaches typically implemented in dependability bots.

ACKNOWLEDGMENTS

This work was partially funded by the French National Research Agency through grant ANR ALIEN (ANR–21–CE25–0007) and by PhD grant 2021/0047 from ANRT. We thank the anonymous reviewers for their precious comments.

REFERENCES

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *29th International Conference on Software Maintenance (ICSM)*. 280–289.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The maven dependency graph: a temporal graph-based representation of maven central. In *16th International Conference on Mining Software Repositories (MSR)*. 344–348.

[3] Timo Berthold and Gregor Hendel. 2021. Learning to scale mixed-integer programs. In *35th AAAI Conference on Artificial Intelligence*, Vol. 35. 3661–3668.

[4] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 255–265.

[5] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.

[6] Vašek Chvátal. 1983. *Linear programming*. Macmillan.

[7] Joel Cox, Eric Bouwers, Marko C. J. D. van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *37th International Conference on Software Engineering (ICSE)*. 109–118.

[8] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely Updating Outdated Dependencies. In *45th International Conference on Software Engineering (ICSE)*. 233–244.

[9] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[10] Matthias Ehrgott. 2005. *Multicriteria optimization*. Vol. 491. Springer Science & Business Media.

[11] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*. 182–192.

[12] Foyzul Hassan, Shaikh Mostafa, Edmund S.L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 38–47.

[13] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* (2023).

[14] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? A case study of Java Projects. *Journal of Systems and Software* 183 (2022), 111097.

[15] Damien Jaime. 2024. Goblin: Neo4J Maven Central dependency graph. https://doi.org/10.5281/zenodo.10605656

[16] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2024. Goblin: A Framework for Enriching and Querying the Maven Central Dependency Graph. In *21st International Conference on Mining Software Repositories (MSR)*.

[17] Damien Jaime, Joyce El Haddad, Pascal Poizat, and Thomas Degueule. 2024. *Dupdater code*. https://github.com/Goblin-Ecosystem/goblinUpdater

[18] Damien Jaime, Joyce El Haddad, Pascal Poizat, and Thomas Degueule. 2024. Experiments data. https://zenodo.org/records/13285362

[19] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *32nd International Symposium on Software Testing and Analysis (ISSTA)*. 1433–1444.

[20] Jeremy Katz. 2018. Don't believe the download numbers when evaluating open source projects. https://blog.tidelift.com/dont-believe-the-download-numbers-when-evaluating-open-source-projects [Accessed: 2024-04-09].

[21] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *30th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 201–211.

[22] Martin Mois. 2024. japicmp. https://siom79.github.io/japicmp/ [Accessed: 2024-04-09].

[23] Inc. Neo4j. 2024. Neo4j Cypher Query Language. https://neo4j.com/product/cypher-graph-query-language/ [Accessed: 2024-04-09].

[24] Lina Ochoa, Thomas Degueule, and Jean-Rémy Falleri. 2022. BreakBot: Analyzing the Impact of Breaking Changes to Assist Library Evolution. In *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results ICSE (NIER) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 26–30. https://doi.org/10.1109/ICSE-NIER55298.2022.9793524

[25] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. *Empirical Software Engineering* 27, 3 (2022), 1–42.

[26] Renovate. 2024. Renovate: Universal dependency automation tool. https://github.com/renovatebot/renovate.

[27] Benjamin Rombaut, Filipe R Cogo, Bram Adams, and Ahmed E Hassan. 2023. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the greenkeeper dependency bot in npm. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–40.

[28] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

[29] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical Lag of Dependencies in Major Package Managers. In *27th Asia-Pacific Software Engineering Conference (APSEC)*. 228–237.

[30] Nazanin Vafaei, Rita A Ribeiro, and Luis M Camarinha-Matos. 2016. Normalization techniques for multi-criteria decision making: analytical hierarchy process case study. In *7th advanced doctoral conference on computing, electrical and industrial systems (DoCEIS)*. 261–269.

[31] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[32] Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M German, and Katsuro Inoue. 2017. Analysis of license inconsistency in large collections of open source projects. *Empirical Software Engineering* 22 (2017), 1194–1222.

[33] K Paul Yoon and Ching-Lai Hwang. 1995. *Multiple attribute decision making: an introduction*. Sage publications.

[34] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. 2019. On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 589–593.

[35] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–12.

[36] Hao Zhong and Na Meng. 2024. Compiler-directed Migrating API Callsite of Client Code. In *46th International Conference on Software Engineering (ICSE)*.