International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 12(2020) pp. 1-20
https://doi.org/10.70917/ijcisim-2026-6153
Article

An Exact Algorithmic Pipeline Certifies the Minimum Binary Seed for Rebuilding Software Package Ecosystems from Source

Luis Eduardo Muñoz Guerrero*

Facultad de Ingenierías, Universidad Tecnológica de Pereira, Colombia, lemunozg@utp.edu.co

Abstract: Rebuilding a software ecosystem from source requires accepting some packages as binaries whenever dependency cycles would otherwise leave no build order. We ask how many packages this binary seed must contain in language package ecosystems and which kinds of dependency force them. The minimum seed is a minimum directed feedback vertex set, an NP-hard problem that admits no constant-factor approximation under the Unique Games Conjecture, so we built an exact algorithmic pipeline that reduces each dependency graph, solves the remaining components by integer programming with lazily generated cycle constraints, and certifies every answer with lower and upper bounds. On the Libraries.io dump of 15 June 2017, with 1,237,945 packages, all 33 instances (11 ecosystems under runtime, build, and development and test scenarios) were solved to proven optimality in 120 s, including reading the extracted dependencies, and the reductions left only 6 components for the integer program. Seeds range from 1 to 2,206 packages when development and test dependencies are required; npm needs 394 binary packages for its runtime dependencies alone and 2,206 with them, and, in an exploratory count, 50% of all runtime seed packages declare a dependency on themselves. A pre-specified hypothesis that development and test dependencies account for most of the seed was positive in the primary specification (7 of 8 ecosystems, sign test = 0.0352) and passed a label-permutation placebo check in 6 of 8 ecosystems, but it held in only 8 of 16 pre-specified specifications, below the 80% required, and is reported as not valid. A greedy heuristic stayed within 6.4% of the optimum in every ecosystem (exploratory), and suspending at most 3.1% of the development and test dependencies (up to 68.7% of those on a cycle) brought each seed down to its size without them (in npm, by an exploratory rerun).
Keywords: feedback vertex set, integer programming, algorithm engineering, software package ecosystems, build from source, software supply chain

1. Introduction

Rebuilding software from its source code is how a distribution reaches new hardware and how its users establish that a binary corresponds to the source it claims to come from. Bootstrapping a binary distribution means recompiling all of its packages for an architecture on which it does not yet exist [1], and recompiling from source was proposed long ago as a countermeasure against trap doors hidden in object code [2]. The protection has a limit: a compromised compiler binary can reinsert a Trojan horse each time it recompiles its own clean source [3], so every build from source starts from some binaries that have to be trusted. Functional package managers make that starting point explicit. Software is bootstrappable when it does not depend on a binary seed that cannot be built from source [4], and in Nix the build environment, which must itself be built, was bootstrapped on i686-linux from statically linked tools injected as sources [5].

The binary seed is not only the toolchain. Packages need one another to be built and tested, and the difficulty of bootstrapping comes from dependency cycles between packages that must be broken before a build order exists [1]. Language ecosystems make the problem tangible: a packager who brought the npm build tools into Guix reported that they pull in over a thousand dependencies, often cyclic, and used binary npm packages as stepping stones [6]. How

many packages an ecosystem must accept as binaries to rebuild everything else from source, and which kinds of dependency force them, has not been measured. The closest quantification is for Debian, where the source packages to build in reduced form were chosen with a greedy feedback vertex set heuristic [1], the analogue of suspending dependencies rather than of accepting binaries; those sets are heuristic, described as close to minimal, and come with neither an exact optimum nor a lower bound [1]. Studies of language ecosystems, for their part, often keep only runtime dependencies and omit development and test dependencies because not every package declares them completely and reliably [7, 8], although in the npm data of Libraries.io, development dependencies outnumbered runtime dependencies [9].

This paper measures that seed exactly. A package can be built from source once everything it depends on has been built, so a set of packages accepted as binaries suffices if and only if removing them leaves the dependency graph acyclic: the minimum seed is a minimum directed feedback vertex set, a problem whose decision version is NP-complete [10] and which, under the Unique Games Conjecture, is not approximable within any constant factor [11]. We build the dependency graph of every language ecosystem with cycles in the Libraries.io dump of 15 June 2017, with one version per package, under three nested scenarios: runtime dependencies only (K_1), plus build dependencies (K_2), and plus development and test dependencies (K_3). An exact algorithm that reduces each graph and solves the remaining components by integer programming with lazily generated cycle constraints certifies, for every instance, an interval $[L, U]$ that contains the optimum, and closes it when the optimum is proven.

The contributions are the following. First, a formulation of the minimum binary seed with suspendable dependencies, its complexity, and an algorithm with a certificate: the algorithm never reports a seed without a certified lower bound, and it proves the seed optimal when the two meet. Second, a census: all 33 instances of the 11 ecosystems were solved to proven optimality in 120 s in total, including reading the extracted dependencies and building the graphs, with seeds that range from 1 to 2,206 packages when development and test dependencies are required. Third, a confirmatory test, fixed in an analysis plan before any seed was computed, of the hypothesis that development and test dependencies account for most of the seed. The test was positive (7 of 8 ecosystems above one half, sign test p = 0.0352), and the observed shares exceeded a label-permutation placebo in 6 of 8 ecosystems; but the result held in only 8 of the 16 specifications of the pre-specified multiverse, below the 80% the plan required, so by the plan's own rule it is reported as not valid rather than as a finding. Fourth, the distance between a greedy heuristic and the certified optimum, the trade-off between seeds and suspended development and test dependencies, the part of each seed made of packages that depend on themselves, and a measured scaling curve of the exact method.

The rest of the paper reviews related work (Section 2), describes the data, the problem, the algorithm and the analysis plan (Section 3), reports the results (Section 4), discusses them (Section 5) and the threats to their validity (Section 6), and concludes (Section 7).

2. Related Work

Bootstrapping distributions from source. Debian bootstraps a new architecture by cross-compiling a minimal build system and then compiling the rest natively [12]. Its build graphs contain build-dependency cycles, and in these graphs strongly connected components reach about a thousand vertices [1]; breaking those cycles by hand used to take months to a year of work [1]. Schauer cast the choice of build dependencies to drop as a minimum feedback arc set problem [1] and evaluated heuristics against the smallest feedback arc set found by any of the tested algorithms, not against a certified optimum [1]; the resulting tool computes a build order that breaks compilation loops by pruning build dependencies and was adopted by Debian [12]. Functional package managers attack the toolchain end of the problem: Guix halved its bootstrap binary seed, and its authors' estimate of the seed of Debian ignored the cycles that show up higher in the package graph [4]; Nix substitutes pre-built outputs because building the whole dependency graph from source is impractical for most users [5]. Diverse double-compiling checks a compiler binary against its source with a second, trusted compiler [13], and reproducible builds have gained adoption in repositories such as Debian [14]. None of these works sizes the binary seed of a language ecosystem.

Dependency networks of language ecosystems. Decan et al. compared the evolution of the dependency networks of seven packaging ecosystems using the Libraries.io dataset [7] and left Maven and PyPI out because their dependency information there was incomplete [7]. Kikas et al. analysed the dependency networks of the JavaScript, Ruby and Rust ecosystems [15]. In Libraries.io, some of the Maven records contain unresolved variable references because the Maven project files are scanned without being interpreted [16], and the Maven Dependency Graph offers a separate snapshot of Maven Central with one edge per dependency and its scope [17]. Development and test dependencies are usually set aside: in Maven, dependencies with provided or test scope are not shipped with the released artifact [18], and developers dismissed a vulnerability advisory because the library was only a test dependency [19]. For bootstrapping, the opposite holds: a package cannot be rebuilt and tested from source without them.

Feedback sets. The decision versions of the minimum feedback vertex and arc set problems on digraphs are NP-complete [10]; polynomial-time algorithms approximate them within $O(\log n \log\log n)$ [20], and, for the vertex version, no constant factor is possible under the Unique Games Conjecture [11]. Directed feedback vertex set is fixed-parameter tractable in the size k of the solution, with running time $4^k k!$ times a polynomial in the size of the graph [21], which makes its bound impractical for seeds of a hundred packages or more. Exact methods rely on reductions and on integer programming. Levy and Low's contraction algorithm finds a minimum cutset on the graphs it contracts completely [22]; its contractions remove vertices that lie on no cycle and merge vertices with a single predecessor or successor without losing optimality [23], and Lin and Jou's additional contraction operations fully reduced most of the benchmark circuits that the earlier ones left unreduced [24]. Directed feedback vertex set can be written as a set covering problem with one constraint per directed cycle [25], and when the cycles are too many to list it can be solved as an implicit hitting set problem with an oracle that returns an unhit cycle [26]; the separation of dicycle inequalities is polynomial [27]. Funke and Reinelt studied the polytope of the complementary problem, the maximum weight node-induced acyclic subdigraph, and reported computational results with a branch-and-cut algorithm [28]. Heuristics are fast, but on large graphs they may be compared only with each other: for instance, the large-scale evaluation of Simpson et al. reports feedback arc set sizes without optimal values or lower bounds [29]. Exact optimisation has a precedent in package management: Tucker et al. solved Debian package installation with off-the-shelf solvers [30] and found that on real installation traces the default installer sometimes missed a solution that existed [30]. Table 1 summarises how the closest works relate to this one.

Table 1. Closest works: the graph each one studies, the dependency kinds it covers and whether it reports an optimum.

Work	Graph	Kinds	Optimum
Schauer [1]	Debian build graph	build	heuristic
Abate and Di Cosmo [12]	Debian build order	build	heuristic
Nieuwenhuizen [4]	Guix toolchain seed	toolchain	–
Decan et al. [7]	language ecosystems	runtime	cycles not studied
Tucker et al. [30]	Debian installation	install	exact
Lin and Jou [24]	circuit graphs	–	exact
This work	language ecosystems	all three	certified

3. Materials and Methods

Data. The study uses version 1.0.1 of the Libraries.io open data [31], a dump dated 15 June 2017 that lists the versions of the packages of many package managers and the dependencies declared by each version. Its versions table (1.08 GB) and dependencies table (4.96 GB) were read as streams and checked against the checksums of the archive. Each package keeps one version: the newest release published on or before the dump date, with ties in the publication time broken by the larger identifier of the version's row in the dump. This choice reduces 9,023,105 versions to

1,237,945 packages and 51,239,986 dependency rows to the 5,358,073 rows of the chosen versions. An arc $p \to q$ joins two packages of the same ecosystem when the chosen version of p declares a dependency that the dump resolves to q; a package that depends on itself forms a self-loop, because it can only be built with a binary of itself. Each arc keeps the dependency kinds of all its rows. Table 2 gives the size of the data of every ecosystem studied; coverage is uneven, as the dump records few dependency rows for PyPI and leaves about a third of the rows of the Comprehensive Perl Archive Network (CPAN) and of the Comprehensive R Archive Network (CRAN) unresolved.

Table 2. Data of the eleven ecosystems with cycles in the dump: packages with a chosen version, their dependency rows, the rows left unresolved, the arcs with all kinds and the largest strongly connected component.

Ecosystem	Packages	Dep. rowsa	Unresolvedb	Arcsc	SCCd
npm	516,447	3,773,436	8,359	3,761,981	7,110
Packagist	101,310	307,601	16,222	291,088	249
RubyGems	134,381	490,449	746	489,688	300
CPAN	34,308	283,737	94,906	188,801	428
CRAN	11,068	45,840	14,693	31,136	275
Maven	109,146	243,198	9,343	233,855	14
Cargo	9,838	29,555	8	29,535	72
Pub	2,665	8,263	29	8,234	63
NuGet	91,621	151,682	675	150,990	2
PyPI	92,295	4,406	113	4,293	2
Haxelib	1,054	720	0	720	2

[a] Dependency rows of the chosen versions. [b] Rows whose target the dump does not resolve to a project of the same ecosystem. c Arcs with all dependency kinds, including excluded and optional ones; $G_e^{K_3}$ has slightly fewer. d Vertices of the largest strongly connected component of the graph with all kinds.

Scenarios. The kinds that each package manager declares were mapped, before any seed was computed, to three classes (Table S1): runtime (K_1), needed only to build (K_2), and needed only to develop or test (K_3). With $\kappa_e(p, q)$ the kinds declared on the dependency of p on q and $\mathcal{T}_e(K)$ the kinds in the classes of scenario K, the graph G_e^K of ecosystem e keeps the arcs A_e^K in Equation (1), and the scenarios are nested: K_1 alone, K_1 with K_2, and all three. The suspendable arcs H_e are those that only K_3 contains: removing them from $G_e^{K_3}$ leaves $G_e^{K_2}$, so the share θ_e defined below is the part of the seed that disappears without them. Optional dependencies stay out of the primary specification.

$$A_e^K = \{(p, q) \in A_e : \kappa_e(p, q) \cap \mathcal{T}_e(K) \neq \emptyset\}, \qquad (1)$$
$$H_e = A_e^{K_3} \setminus A_e^{K_2}$$

The 11 ecosystems whose graph with all kinds has a cycle form the study set: npm, Packagist, RubyGems, CRAN, Maven, Cargo, Pub, CPAN, PyPI, NuGet and Haxelib. Of the 23 package managers in the dump, 15 declare dependencies in their chosen versions; Atom has no arc inside its own ecosystem, because all its rows point to another one, and Dub, Elm and Hex have arcs but no cycle (Table S2). PyPI, NuGet and Haxelib record only runtime dependencies in the dump, so their three graphs coincide by construction of the data. For one comparison (H4) we also use the Maven Dependency Graph, a snapshot of Maven Central with the scope of every dependency edge [17], keeping the newest version of each library and its compile and runtime edges.

Problem. A package can be built from source once all its dependencies are available, so a set S of packages accepted as binaries lets the rest be built in some order if and only if $G - S$ is acyclic. In the bicriteria variant, some arcs $H \subseteq A$ (in the ecosystem graphs, the arcs H_e of Equation (1)) can be suspended at a cost, as when a package is built without its test suite. Given $G = (V, A)$, weights $w_v \geq 0$ and costs $c_a \geq 0$, the minimum seed with suspendable arcs is the integer program in Equation (2), with one covering constraint per directed cycle C of G, where $\mathcal{C}(G)$ is the set of directed cycles and x_v and y_a mark the packages in the seed and the suspended arcs; Table 3 collects the notation.

$$\begin{aligned} \min \ & \sum_{v \in V} w_v \, x_v + \sum_{a \in H} c_a \, y_a \\ \text{s.t.} \ & \sum_{v \in V(C)} x_v + \sum_{a \in A(C) \cap H} y_a \geq 1 \\ & \qquad \text{for all } C \in \mathcal{C}(G), \\ & x \in \{0,1\}^V, \ y \in \{0,1\}^H \end{aligned} \tag{2}$$

Table 3. Notation used in the equations and in the algorithm.

Symbol	Meaning	Domain
$G = (V, A)$	dependency graph: packages and dependency arcs	digraph
$H \subseteq A$	suspendable arcs; in $G_e^{K_3}$, the arcs H_e that only K_3 contains	set of arcs
w_v, c_a	weight of a seed package, cost of a suspended arc	$\mathbb{R}_{\geq 0}$
S, F	seed and suspended arcs	$S \subseteq V$, $F \subseteq H$
$w(S)$, $c(F)$	total weight of a seed, total cost of suspended arcs	$\mathbb{R}_{\geq 0}$
x_v, y_a	indicators of $v \in S$ and $a \in F$ ($y_a = 0$ for $a \notin H$)	$\{0,1\}$; $[0,1]$ in the relaxation
$\mathcal{C}(G)$, $\mathcal{C}'$	directed cycles of G; cycles collected so far	set of cycles
$V(C)$, $A(C)$	vertices and arcs of a cycle C	$V(C) \subseteq V$, $A(C) \subseteq A$
$d(s, t)$	shortest-path length from s to t with arc costs $x_v + y_{uv}$	$\mathbb{R}_{\geq 0}$
Σ, ϵ	sources examined in a round of the linear phase; tolerance of the violation test	$\Sigma \subseteq V$; $\epsilon > 0$
$d^-(v)$, $d^+(v)$	in-degree and out-degree of v	$\mathbb{N}$
K_1, K_2, K_3	runtime class, build-only class, development and test class	kind classes
G_e^K	graph of ecosystem e with the kinds of scenario K	digraph
A_e^K, $\kappa_e(p, q)$, $\mathcal{T}_e(K)$	arcs of G_e^K; kinds of the dependency of p on q; kinds in the classes of scenario K	set of arcs; sets of kinds
$s^*(e, K)$	size of a minimum seed of G_e^K	$\mathbb{N}$
OPT; L, U	optimum; certified lower bound and cost of the returned pair, also per component (L_Q, U_Q) and per ecosystem and scenario ($L_{e,k}$, $U_{e,k}$)	$L \leq \text{OPT} \leq U$

θ_e	share of the seed added by K_3	$[0,1]$
$\theta_e^{lo}, \theta_e^{hi}$	certified lower and upper ends of θ_e	$[0,1]$
τ	time limit per component	seconds
n, k, j, α	number of ecosystems whose θ_e^{lo} is not one half, number of shares above one half, summation index, significance level	$\mathbb{N}$; $(0,1)$
p_e, B_e, b	placebo p-value, number of permutations, permutation index	$(0,1]$; $\mathbb{N}$
$\tilde{\theta}_{e,b}^{hi}$	upper end of the share in permutation b	$[0,1]$
$\mathcal{Q}_e$	strongly connected components of $G_e^{K_3}$ that contain a cycle	set of subgraphs
$g_Q, g_e, U_Q^{\mathrm{gr}}$	relative gap of the greedy heuristic in component Q and in ecosystem e; cost of its seed in Q	$\mathbb{R}_{\geq 0}$

With $H = \emptyset$ and unit weights, the program is the minimum directed feedback vertex set, and $s^*(e, K)$ denotes its optimum on G_e^K. Its decision version is NP-complete [10], and the identity maps each of its instances to an instance of the seed problem with the same optimum, so the seed problem is NP-hard; under the Unique Games Conjecture it cannot be approximated within any constant factor [11]. Unless P = NP, no polynomial method can promise the optimum, nor, under that conjecture, a constant ratio, so the algorithm below reports a certificate instead: an interval $[L, U]$ with $L \leq \mathrm{OPT} \leq U$.

Algorithm. Algorithm 1 lists the method. It first splits the graph into strongly connected components and applies reductions until none applies: a vertex with a hard self-loop goes to the seed, and a vertex with a single predecessor or successor is merged into it, as in Levy and Low's contractions [22, 23], provided the arcs involved are hard and the vertex weighs no less than that neighbour; on a graph without suspendable arcs it also applies Lin and Jou's PIE, CORE and DOME operations [24], with the weighted condition on CORE proved in the supplementary material. Each remaining component receives an upper bound from a greedy heuristic (which repeatedly removes the vertex with the largest product of in- and out-degree per unit of weight and then drops from the seed every vertex whose removal is unnecessary) and a lower bound from a packing of cycles that share no vertex. A linear phase then solves the relaxation of Equation (2) with the cycles collected so far and adds, from every examined source s, the cycle of minimum weight through s when that weight is below $1 - \epsilon$, computed by shortest paths as in Equation (3), which separates the cycle inequalities exactly from each examined source. The integer phase solves the program with the collected cycles; if the solution leaves a cycle, the cycles that remain are added and the program is solved again, as in an implicit hitting set method [26].

$$\min_{t\,:\,(t,s)\in A,\; t\neq s} (x_s + d(s,t) + y_{ts}) < 1 - \epsilon \qquad (3)$$

where $d(s,t)$ is the length of a shortest path from s to t when every arc (u,v) costs $x_v + y_{uv}$, with $y_{uv} = 0$ for an arc that cannot be suspended; the examined sources Σ are the at most 4,000 vertices with the smallest x_s below $1 - \epsilon$ that have a predecessor, with $\epsilon = 10^{-6}$, and a self-loop is added when $x_v + y_{vv} < 1 - \epsilon$. An optimal solution of the integer phase that breaks every cycle is optimal for the full program, because the program with a subset of the cycles is a relaxation; when the time limit τ runs out, the best bound of the solver and of the linear phase gives L, and the best integer solution, completed by the greedy heuristic and made minimal, competes with the greedy solution for U. The guarantee, proved with its lemmas in the supplementary material, is that the algorithm returns a feasible pair with $L \leq \mathrm{OPT} \leq w(S) + c(F)$, and $L = \mathrm{OPT}$ whenever every component is closed, because its integer phase finishes or because its lower and upper bounds meet.

Implementation. The pipeline is written in Python 3.8 with NumPy 1.18.0 and SciPy 1.4.1, whose Dijkstra routine computes the separation. The integer and linear programs are solved by CBC 2.9.0, the binary distributed with PuLP 2.0, called as a separate process with one thread. On the test machine, CBC's integer preprocessing, its primal heuristics and its cut generators made the solver process fail intermittently with memory faults, so all three are disabled; a failed process is retried and counted, and a component whose lower bound reaches the cost of its feasible solution is certified by the bounds. The time limit is τ = 3,600 s per component, with the linear phase stopped after 200 rounds or at $\tau/2$, whichever comes first. Every run records the commit, the digest of the data, the tag of the analysis plan, the seed of the random number generator and the library versions. Property-based tests written with Hypothesis 4.57.1 from the formulation, without access to the implementation, check every lemma and the guarantee against a brute-force oracle on small instances, and an independent replica of the primary analysis was written from the plan and the data sheet, with its own graph construction and reductions but the same integer solver. All experiments ran on a desktop computer with an 8-core processor and 16 GB of memory under 64-bit Windows. Figure 1 shows the stages of the pipeline, the size of the data at each one and the check that protects it.

Algorithm 1 Certified minimum seed with suspendable arcs	
	Input: digraph $G = (V, A)$, suspendable arcs $H \subseteq A$, weights w, costs c, time limit τ
	Output: seed S, suspended arcs F, lower bound L, with $L \le \mathrm{OPT} \le w(S) + c(F)$
1:	$S \leftarrow \emptyset$, $F \leftarrow \emptyset$, $L \leftarrow 0$; drop vertices and arcs that lie on no cycle
2:	**repeat** until no rule applies
3:	a vertex v with a hard self-loop: $S \leftarrow S \cup \{v\}$, $L \leftarrow L + w_v$
4:	a vertex v with one predecessor (or successor) u, hard arcs, $w_u \le w_v$: merge v into u
5:	**if** the graph has no suspendable arc **then** apply PIE, CORE and DOME
6:	**for each** strongly connected component Q of the reduced graph **do**
7:	$U_Q \leftarrow$ greedy by $d^-(v)\, d^+(v)/w_v$, then drop every unneeded vertex from its seed
8:	$L_Q \leftarrow$ weight of a packing of vertex-disjoint cycles
9:	$\mathcal{C}' \leftarrow$ self-loops, 2-cycles and short cycles through high-degree vertices
10:	**repeat** until no examined source gives a violated cycle inequality, 200 rounds pass or $\tau/2$ elapses
11:	solve the relaxation over $\mathcal{C}'$; add the cycles found by Equation (3); raise L_Q
12:	**repeat** until τ elapses
13:	solve the integer program over $\mathcal{C}'$; raise L_Q to its bound
14:	**if** the program is solved to optimality and its solution leaves no cycle **then** $U_Q \leftarrow L_Q \leftarrow$ its cost; **break**
15:	add cycles that the solution leaves in the graph
16:	**if** not closed **then** $U_Q \leftarrow \min(U_Q,$ repaired best integer solution$)$
17:	add the solution of Q to S and F; $L \leftarrow L + L_Q$
18:	**return** S, F, L

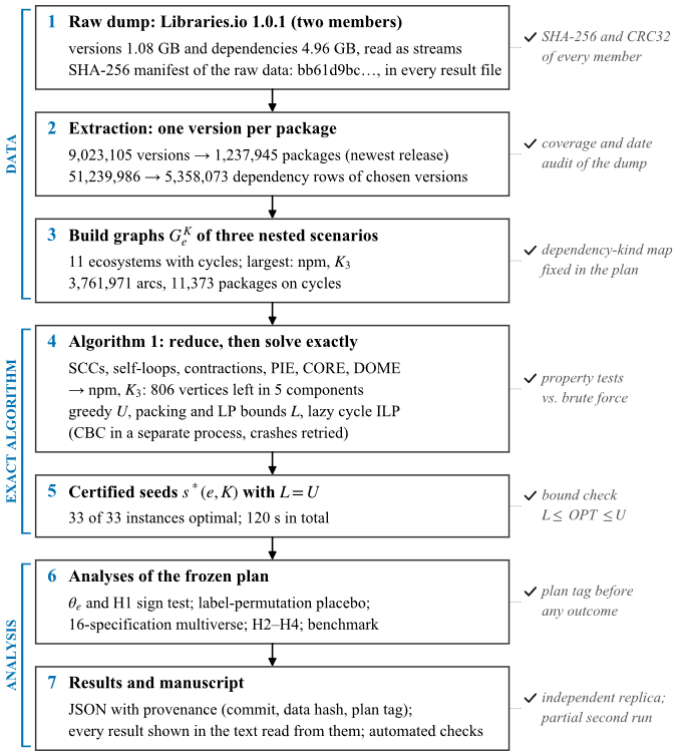


Figure 1. Architecture of the pipeline. Each box gives the stage and the size of its data, and on the right the check that protects it; the brackets group the stages into three phases, and the SHA-256 digest of the manifest of the raw data is the one recorded in every result file. Every result shown in the figure and in the text is read from the result files.

Outcome and confirmatory hypothesis. The share of the seed added by development and test dependencies is θ_e in Equation (4), computed from certified bounds: its lower end uses the upper bound of $s^*(e, K_2)$ and the lower bound of $s^*(e, K_3)$, and both ends coincide when both seeds are optimal.

$$\theta_e = 1 - \frac{s^*(e,K_2)}{s^*(e,K_3)}, \qquad (4)$$

$$\theta_e^{lo} = \max\left(0,\ 1 - \frac{U_{e,2}}{L_{e,3}}\right), \quad \theta_e^{hi} = 1 - \frac{L_{e,2}}{U_{e,3}}$$

H1 states that, in the ecosystems that record development and test dependencies, they account for most of the seed: the median of θ_e exceeds one half. The H1 set holds the ecosystems of the study with at least one arc of class K_3 and a positive $s^*(e, K_3)$, and the test is the exact one-sided sign test in Equation (5) at α = 0.05, where k is the number of ecosystems with θ_e^{lo} above one half among the n whose θ_e^{lo} differs from one half. A sign test fits because the units are few and heterogeneous and each θ_e is an exact measurement; nothing is assumed about its distribution.

$$p = \sum_{j=k}^{n} \binom{n}{j} 2^{-n} \qquad (5)$$

A positive result is valid only if it passes three checks fixed in the plan. In the placebo, the label of the arcs that only K_3 contains is reassigned at random among the arcs of $G_e^{K_3}$, keeping their number, 99 times per ecosystem, and s^* is recomputed with a limit of 600 s; the observed share passes in an ecosystem when the permutation p-value in Equation (6), which compares the lower end of the observed share with the upper end $\tilde{\theta}_{e,b}^{hi}$ of the share recomputed in

permutation b (with the observed $s^*(e, K_3)$ as denominator), is at most 0.05, and the check passes when it holds in more than half of the ecosystems.

$$p_e = \frac{1 + \#\{\, b : \tilde{\theta}_{e,b}^{hi} \geq \theta_e^{lo} \,\}}{1 + B_e} \tag{6}$$

A multiverse analysis repeats the analysis across the data sets that reasonable alternative processing choices produce [32]; robustness requires H1 to remain positive in at least 80% of a multiverse of 16 specifications, each with a limit of 600 s, that combine two options for each choice the plan considered doubtful: the newest release or the newest one without a pre-release mark, optional dependencies out or in, the planned kind map or an alternative one (CRAN suggestions as build dependencies and Maven system scope as runtime), and unresolved dependencies left out or resolved by exact name. The certificate check requires $L \leq \mathrm{OPT} \leq U$ to hold in every instance, and every returned seed to leave its graph acyclic, as checked by a topological order; the decision rule uses the lower end θ_e^{lo}. If H1 were null, a one-sided sign test on the upper ends would decide whether the null is informative.

Secondary analyses. H2 measures the gap of the greedy heuristic of Algorithm 1, applied without the reductions to each strongly connected component Q of $G_e^{K_3}$ that contains a cycle (the set $\mathcal{Q}_e$), with the gaps of Equation (7), where U_Q^{gr} is the cost of its seed in Q, L_Q the certified lower bound of Q and $L_e = \sum_Q L_Q$; g_e, a mean of the gaps of the components weighted by their lower bounds, equals the exact gap when the optimum is certified and bounds it otherwise; the plan asks whether its median lies below 5%, with a sign test across ecosystems, a bootstrap interval for the median, and a bootstrap over components clustered by ecosystem.

$$g_Q = \frac{U_Q^{\mathrm{gr}} - L_Q}{L_Q}, \qquad g_e = \frac{\sum_{Q \in \mathcal{Q}_e} \left(U_Q^{\mathrm{gr}} - L_Q\right)}{\sum_{Q \in \mathcal{Q}_e} L_Q} = \sum_{Q \in \mathcal{Q}_e} \frac{L_Q}{L_e}\, g_Q \tag{7}$$

H3 traces the trade-off between seeds and suspended development and test dependencies by solving the bicriteria program with suspension costs between 0.25 and 4 times the cost of a seed, and computes the quantity it asks for, the fewest suspended arcs that bring the seed down to $s^*(e, K_2)$, by giving each package a weight above the number of suspendable arcs so that the seed is minimised first. H4 compares the runtime seed of Maven in Libraries.io with the one in the Maven Dependency Graph. H3 and H4 are descriptive; the two H2 tests form the exploratory family, corrected by Benjamini and Hochberg's procedure, which controls the false discovery rate for independent test statistics [33]; the two H2 tests share their data, so here the correction is approximate. Four descriptive analyses were added after the plan was fixed and are labelled as exploratory: how many packages of each seed declare a dependency on themselves, which by Lemma 2 of the supplementary material belong to every seed, so that the seed without them is obtained by subtraction and θ_e can be recomputed without them; whether the version requirement of each such dependency admits an earlier release of the package; which packages the two runtime seeds of Maven share; and the runtime seed of the Maven Dependency Graph cut at the date of the dump. The following analyses were added as well and are also labelled as exploratory: a check of the certificates of the primary analysis that does not use the integer solver (Threats to Validity); a comparison of the greedy heuristic with one that applies Levy and Low's reductions, but not Lin and Jou's PIE, CORE and DOME, before every choice (Results, H2); and a rerun of the bicriteria program of H3 with every suspendable arc on a cycle subdivided by a new vertex, with a limit of 600 s per component instead of τ (Results, H3).

Benchmark. The construction of each graph and each stage of the algorithm are timed in 5 repetitions, each in a fresh process with one thread, after a discarded warm-up run, and summarised by the median wall time and the peak resident memory (the interquartile range and the processor time are given in Table S17); processor time and memory are those of the main process, without the solver process, and stages under a second are indicative; the extraction from

the dump, which the plan also listed, was not timed. The machine was not dedicated: an unrelated workload was active, and the 2 load samples taken while the benchmark ran found the whole processor between 26% and 46% busy, whereas the benchmark used a single thread; the times are those of a shared machine and are expected to overstate those of a dedicated one. The scaling curve uses breadth-first balls of 64 to 4,096 vertices grown from random vertices of the largest component of npm in G^{K_3}, 3 per size, with a limit of 600 s; a run that does not finish within the limit is reported as censored and left out of the fitted slope. The slope β in Equation (8) is fitted by least squares over the set I of samples that finish and keep a vertex after the reductions, with $\bar{t}_i$ the median wall time of sample i and n_i its vertices left, and its bootstrap 95% interval resamples the pairs 9,999 times; the slope of the peak memory is fitted in the same way against the vertices of the sample. The same samples, solved with and without the PIE, CORE and DOME reductions ($t_{i,r}^{(1)}$ and $t_{i,r}^{(0)}$ in repetition r), give the speed-up σ_i of those reductions in the samples where both variants finish; its bootstrap interval resamples the 5 repetitions of each variant, so it reflects only the noise between repetitions and is rough.

$$\log \bar{t}_i = a + \beta \log n_i + e_i, \quad i \in I, \tag{8}$$

$$\sigma_i = \frac{\operatorname{med}_r t_{i,r}^{(0)}}{\operatorname{med}_r t_{i,r}^{(1)}}$$

Plan, scope and reporting. The analysis plan, with its hypotheses, decision rules, checks and multiverse, was fixed and versioned in the project repository before any seed was computed; before that point only the structure of the graphs with all dependency kinds, including the number of arcs of each kind and of self-loops, was examined. The departures from the plan, with their reasons and effects, are listed in the supplementary material. Each analysis writes its results with their provenance (commit, data hash and plan tag), and every result shown in the text, tables and figures is read from those files when the manuscript is assembled. By design, the data, the literature, the methods and the software versions named in this paper are restricted to material published up to December 2019.

4. Results

Table 4. Certified minimum seeds $s^*(e, K)$ of every ecosystem in the three nested scenarios, the share θ_e of the seed added by development and test dependencies, and the gap g_e of the greedy heuristic; every seed is optimal ($L = U$).

Ecosystem	Cyclica	K_1	K_2	K_3	θ_e	g_e[b]
npm	11,373	394	394	2,206	0.82	1.59%
Packagist	1,226	243	243	400	0.39	0.25%
RubyGems	694	92	92	261	0.65	0.38%
CPAN	530	4	6	110	0.95	6.36%
CRAN	409	0	0	87	1.00	0.00%
Maven	190	35	39	87	0.55	1.15%
Cargo	177	3	4	26	0.85	0.00%
Pub	108	7	7	23	0.70	0.00%
NuGet	19	18	18	18	–	0.00%
PyPI	4	2	2	2	–	0.00%
Haxelib	2	1	1	1	–	0.00%

[a] Packages on a cycle of $G_e^{K_3}$; –: no development or test kind in the dump.

[b] Gap of the greedy heuristic of Algorithm 1 to the certified optimum on $G_e^{K_3}$ (H2).

Certified seeds. All 33 instances were solved to proven optimality (Table 4); the whole primary run, including reading the extracted dependencies and building the graphs, took 120 s. The reductions did most of the work: of the 11,373 packages on a cycle of the K_3 graph of npm, 786 remained in the largest component handed to the integer program, which closed in 19.1 s in the primary run, and across all instances only 6 components needed the integer program at all (Table S12 and Table S13). Figure 2 shows one of these optima: the seed of Cargo inside the dependency cycles of its K_3 graph, with at least one seed package in every strongly connected component. Requiring development and test dependencies enlarges the seed: npm needs 394 packages accepted as binaries to rebuild its runtime graph and 2,206 to rebuild it with its tests, and CRAN, whose runtime graph is acyclic, needs 87 once its suggested packages are required. Build-only dependencies matter much less; they enlarge the seed only in the ecosystems whose kinds single them out: Maven (35 to 39), Cargo (3 to 4) and CPAN (4 to 6). In 8 of the 11 ecosystems (an exploratory count), part of the seed is made of packages whose chosen version declares a dependency on the package itself, a self-loop that puts the package in every seed: 190 of the 394 runtime seed packages of npm and 542 of its 2,206 with development and test dependencies, and almost the whole runtime seed of RubyGems (91 of 92) and of NuGet (17 of 18) (Figure 3a).

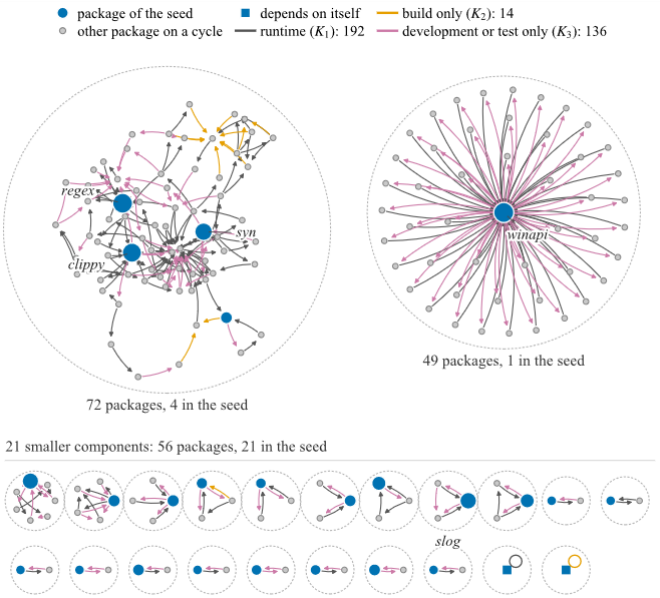


Figure 2. Dependency cycles of Cargo in $G_e^{K_3}$ and its certified minimum seed (descriptive). Dots are the 177 packages on a cycle, grouped by strongly connected component, the two largest above and the rest below by size; arrows go from each package to its dependencies inside its component, coloured by the smallest scenario that contains them: runtime (K_1, grey), build only (K_2, orange) or development or test only (K_3, purple), with their counts in the key; of the 77 cycles of two packages, 71 pair a runtime dependency with a development or test one.

Blue marks the 26 packages of the seed returned by the primary analysis, certified optimal (other seeds of the same size may exist); their area grows with the logarithm of their direct dependents in $G_e^{K_3}$, and the most depended-on are named. Squares are packages that depend on themselves. Removing the seed leaves no cycle.

Confirmatory result (H1). In the H1 set of 8 ecosystems, θ_e ranged from 0.39 in Packagist to 1.00 in CRAN, with a median of 0.76; 7 of the 8 shares exceeded one half, and the one-sided sign test gave $p = 0.0352$, a positive result by the rule of the plan (Figure 3b). The placebo check passed: the permutation p-value was at most 0.05 in 6 of the 8 ecosystems, all except RubyGems and Packagist, where the observed share was no larger than what random labels produce.

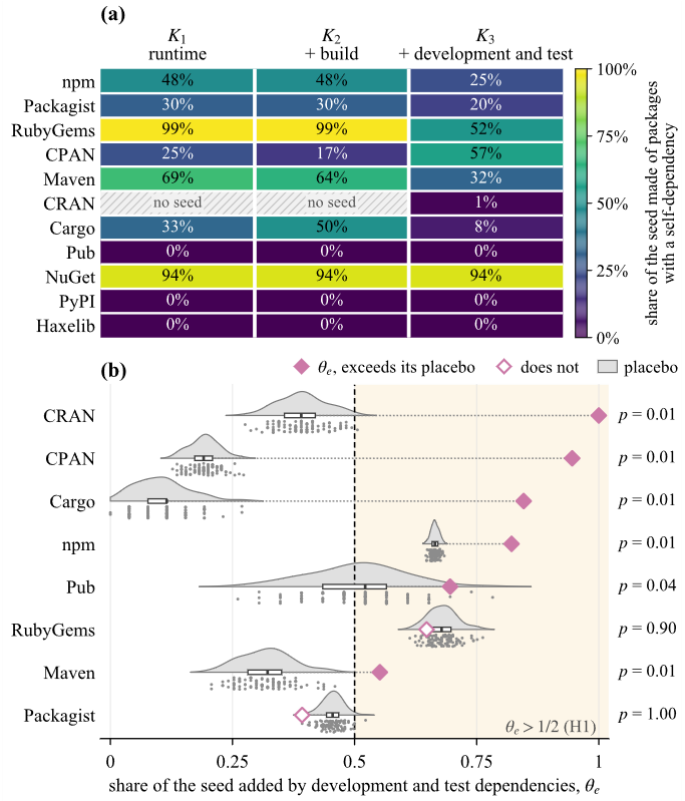


Figure 3. Composition of the certified seeds and the share added by development and test dependencies. (a) Exploratory: share of $s^*(e, K)$ made of packages that declare a dependency on themselves (colour and percentage in each cell) for the runtime (K_1), build (K_2) and development and test (K_3) scenarios, with the ecosystems ordered by $s^*(e, K_3)$; CRAN has no runtime or build seed (hatched), and the exact counts are in Table S6. (b) Lower end of θ_e (diamonds; filled where the permutation p-value of that ecosystem is significant) against the upper end of the share recomputed in each of 99 placebo permutations per ecosystem: the permutations (points, spread vertically for legibility), their kernel density (grey, scaled to the same height in every row), interquartile range (box) and median (bar), with a dotted line from the median to θ_e; the dashed line is the threshold of one half, the tinted region lies above it, and the permutation p-value of each ecosystem is on the right.

The robustness check failed (Figure 4): H1 was positive in 8 of the 16 specifications, below the 80% that the plan required. Including optional dependencies changed no seed and no test in this dump, so the specifications form 8 distinct analyses, 4 of them positive, the same proportion. The shares themselves barely moved across specifications. Every negative specification uses the alternative kind map, in which the suggested packages of CRAN count as build dependencies; CRAN then leaves the H1 set, and the sign test on the remaining 7 ecosystems, with Packagist below one half, gives $p = 0.0625$. The certificate held in every instance, with the lower bound equal to the cost of the returned seed and the acyclicity of the remaining graph checked by a topological order, as it did in all the instances of the multiverse and in 792 of the 792 placebo instances. By the rule fixed in the plan, H1 is therefore positive but not valid: the shares in Table 4 are measurements of this dump, not evidence that development and test dependencies account for most of the seed in general.

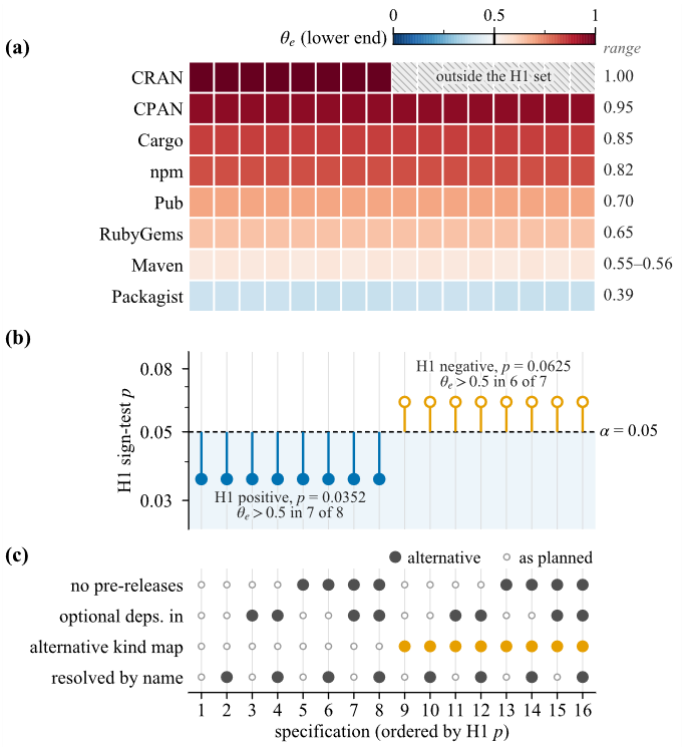


Figure 4. Multiverse of the 16 pre-specified specifications, ordered by the p-value of H1 (ties in the order of the plan) and numbered; thin vertical lines follow each specification down the panels. (a) Lower end of θ_e for each candidate ecosystem (rows, ordered by their mean) on a diverging colour scale centred on the H1 threshold of one half; hatched cells are ecosystems outside the H1 set of that specification, and the right-hand column gives the range of each row across specifications. (b) One-sided sign-test p-value of H1 on a logarithmic axis, with stems from α = 0.05 (dashed; the region below it is shaded): filled blue markers are positive specifications and open orange markers negative ones; each label gives the decision, p and the number of ecosystems of the H1 set whose share exceeds the threshold. (c) Choices of each specification: a filled dot marks the alternative option of a dimension and a small open dot the planned one; the alternative kind map, in orange, is used by every negative specification.

Greedy heuristic (H2, exploratory). Applied without the reductions to each strongly connected component of $G_e^{K_3}$, the greedy heuristic of Algorithm 1, with its minimality pass, returned a seed within 6.4% of the certified optimum in every ecosystem, the largest gap being that of CPAN, and it was optimal in 6 of the 11 ecosystems. The gap stayed below the 5% margin of the plan in 10 ecosystems (one-sided sign test p = 0.006, and 0.006 after the Benjamini and Hochberg correction), with a median of 0.0% (bootstrap 95% interval 0.0% to 1.1%). The aggregate hides the small components: over the 1,577 strongly connected components with at least two vertices, the heuristic missed the optimum in 14, by up to 100% of their minimum seed (Figure 5a). The bootstrap over components, clustered by ecosystem, is degenerate here: most components have no gap, so the median gap was 0.0% with a bootstrap 95% interval from 0.0% to 0.0%, and none of the 9,999 resampled medians reached the margin; it adds little to the test across ecosystems. Large components can exceed the margin: the largest of all, that of npm with 7,110 vertices, where the greedy seed was 5.5% above the optimum, and the largest of CPAN, with 428 vertices and a gap of 9.0%, which holds all 7 extra packages of the greedy seed of that ecosystem. A heuristic that applies Levy and Low's solution-preserving reductions, but not Lin and Jou's PIE, CORE and DOME, before every greedy choice, as the randomised construction of Pardalos et al. interleaves them with its choice of a vertex [25], did better in 5 ecosystems and worse in none: it was optimal in 9 of the 11, and at most 0.9% above the optimum (CPAN); in npm it returned 2,215 packages against 2,241 for the greedy heuristic and an optimum of 2,206 (exploratory; supplementary

material). Since the greedy heuristic already splits each component again after every removal and takes self-loops first, the two differ mainly in the contraction of vertices with a single predecessor or successor; the comparison does not separate a first application of the reductions from the later ones.

Seeds against suspended dependencies (H3, descriptive). Suspending a development or test dependency, that is, building a package before the development or test tool it declares, is the alternative to seeding it. A suspended arc can always be replaced by seeding one of its endpoints (Proposition in the supplementary material), so suspension pays only when it costs less than a seed: all 24 certified solutions with $c/w \geq 1$ cost exactly $s^*(e, K_3)$, and none of the 16 with $c/w > 1$ suspended an arc. Below that ratio the seed falls quickly (Figure 5b): at c/w = 0.25 it fell to between 7% of $s^*(e, K_3)$ in CRAN and 64% in Packagist. The quantity that H3 asks for, the fewest suspended dependencies that bring the seed down to $s^*(e, K_2)$, was certified in 7 of the 8 ecosystems: from 54 arcs in Pub to 1,480 in CPAN, at most 3.1% of the development and test arcs of an ecosystem but from 22.6% to 68.7% of those that lie on a cycle, the only ones whose suspension can break one (Table S14). In npm, the solver closed within τ neither the two programs with c/w below one, which stopped within 0.62% of their lower bounds, nor the lexicographic one, whose best solution, with a seed of 863 packages, is worse than the trivial one that suspends every development and test arc and keeps the runtime seed of 394. An exploratory rerun added after the plan was fixed replaced every development and test arc on a cycle by a path through a new vertex that weighs c/w (one in the lexicographic program, where each package outweighs all new vertices together), which leaves the optimum unchanged and lets every reduction of Algorithm 1 apply (supplementary material). With a limit of 600 s per component instead of 3,600 s, it reproduced the cost of all 45 programs that H3 certified and left the same 3 open, but narrowed their bounds: the two with c/w below one stopped within 0.23% of their lower bounds, and the lexicographic one reached the runtime seed of 394, below which no solution can go, with 5,607 suspended arcs, 0.26% of the development and test arcs of npm; its lower bound shows that at least 5,542 are needed. By the Proposition, the ratios of one or more in the plan could only reproduce $s^*(e, K_3)$, which the plan did not anticipate.

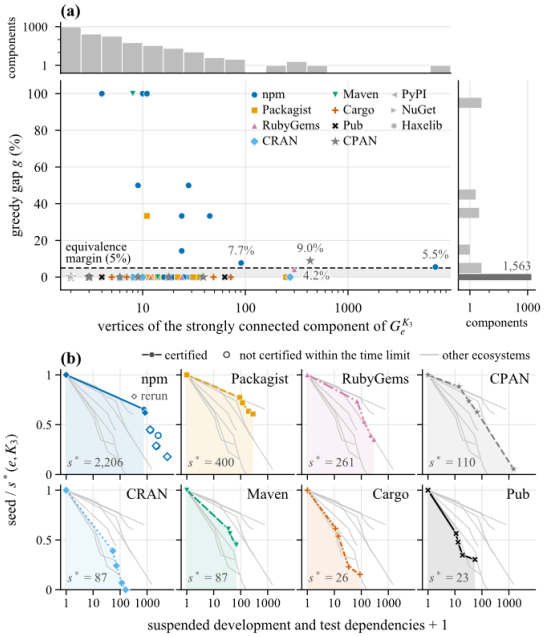


Figure 5. Secondary analyses. (a) Relative gap g_Q (Equation (7)) of the greedy heuristic for each of the 1,577 strongly connected components of $G_e^{K_3}$ with at least two vertices, against its size (log scale), with the equivalence margin of the plan, a relative gap of 0.05 (dashed line; shaded band below it); gaps between zero and twice the margin are printed. Top: histogram of component sizes (bins of one fifth of a decade); right: histogram of g_Q, with the 1,563

components with $g_Q = 0$ in the darker bar and positive gaps in bins of ten points; both counts on log scales. (b) Trade-off between seeds and suspended development and test dependencies, one panel per ecosystem of the H1 set, ordered by $s^*(e, K_3)$: the seed as a fraction of $s^*(e, K_3)$ against the number of suspended arcs plus one, for suspension costs between 0.25 and 4 times the cost of a seed and for the lexicographic solution, which first minimises the seed. Lines and the shaded area under them join certified solutions; open markers are solutions not certified within the time limit; for npm, diamonds mark the exploratory rerun with subdivided suspendable arcs (filled when certified); grey lines repeat the certified frontiers of the other ecosystems for comparison.

Computational cost. With one thread, reading the arcs of npm and building its K_3 graph took 27 s and Algorithm 1 took 16 s, with a peak of 3.4 GB of memory; in every other ecosystem the algorithm took at most 0.7 s (Figure 6c and Table S17), and all 11 were again solved to optimality.

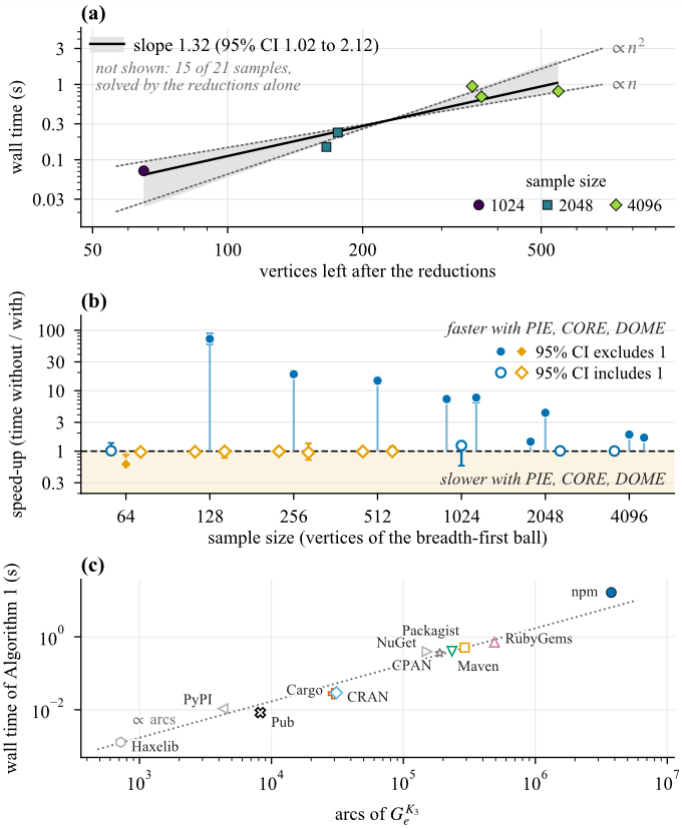


Figure 6. Scaling of Algorithm 1 on breadth-first balls of the largest component of npm in G^{K_3}. (a) Wall time (median and interquartile range of 5 repetitions; the ranges are narrower than the symbols) against the number of vertices left by the reductions, with colour and symbol by sample size. The black line is the log-log least-squares fit, through the centroid of the points in log scale; the grey fan spans the lines through the same centroid with the slopes at the ends of the bootstrap 95% interval of the slope (an interval for the slope, not a confidence band for the mean), and the dotted lines are references proportional to n and n^2. Samples that the reductions solved entirely leave no vertex and are not shown, and triangles would mark runs that did not finish within 600 s. (b) Speed-up of the PIE, CORE and DOME reductions, as the ratio of the median times without and with them: each stem runs from 1 to the ratio, and the thin bar is its bootstrap 95% interval; circles mark ratios above 1 and diamonds ratios below 1, filled when the interval excludes 1 and open when it includes it, and the tinted region lies below 1. The three samples of each size are spread horizontally for legibility. (c) Wall time of Algorithm 1 on the graph $G_e^{K_3}$ of each of the eleven ecosystems (median and interquartile range of 5 repetitions; open markers: below one second, indicative) against its

arcs, with the marker of each ecosystem; the dotted line, proportional to the arcs, is a guide, not a fit. The time of every stage and the peak memory are in Table S17.

On the 21 breadth-first samples of 64 to 4,096 vertices drawn from the largest component of npm, none reached the limit: the reductions alone solved 15 samples, and in the rest at most 546 vertices reached the exact phase, which closed in at most 0.95 s (Figure 6a; Table S18). Over those 6 samples, the time grew with the number of vertices left by the reductions, with a log-log slope of 1.32 (bootstrap 95% interval 1.02 to 2.12), and the peak memory grew with the size of the sample, with a slope of 0.20 (bootstrap 95% interval 0.17 to 0.27). Both fits are descriptive, cover a narrow range and include the reductions as well as the exact phase; for an exact method for an NP-hard problem no polynomial exponent is expected, so the slopes describe only this range of sizes.

Without the PIE, CORE and DOME reductions every sample still closed within the limit. The ratio of the median times without and with them ranged from 0.61 to 72, with a median of 1.01: it was above one, with its bootstrap interval, in 9 of the 21 samples and below one in 1 (Figure 6b). The three reductions pay off where cycles survive the simpler rules and change little elsewhere.

Second source for Maven (H4, descriptive). On the Maven Dependency Graph [17], with the newest version of each library and its compile and runtime edges, the certified runtime seed of Maven had 35 packages, as many as in Libraries.io (35), although the Maven Dependency Graph, a later snapshot (2018), has 178,705 libraries with a dependency edge against 54,966 in Libraries.io. The agreement is in size only: the two returned seeds share 4 packages, and of the libraries that depend on themselves, which every seed of their graph contains, 2 are common to the two sources. Cut at the date of the dump, the Maven Dependency Graph gives a smaller certified seed, 27 packages, of which 4 are in that of Libraries.io: at a common date the two sources agree neither on the size nor on the packages (both exploratory; supplementary material).

Exploratory: self-dependencies and the packages of the seeds. Summed over ecosystems, packages that depend on themselves make up 50% of the runtime seeds and 27% of the seeds with development and test dependencies (Table S6). A package that needs a release of itself to be built or tested is in the position of a compiler written in its own language [3], but some of these declarations may be metadata errors. Discounting them changes the sizes but not the pattern behind H1: computed without them, the share exceeded one half in 7 of the 8 ecosystems, with Packagist again the lowest (0.47). In npm and RubyGems, the packages of the returned K_3 seeds with the most direct dependents are testing and build tools; in Packagist, Pub, CPAN and CRAN such tools head lists that also hold widely used libraries, and in Cargo and Maven the lists are mostly libraries (Table S4). The minimum seed is not unique, so these lists describe one optimal seed among possibly several of the same size; only the size is certified. Summed over ecosystems, 72.5% of the packages of the K_3 seeds lie in strongly connected components of at most ten packages (Figure S1).

5. Discussion

What the seeds measure. A certified seed is the number of packages that a from-source rebuild of an ecosystem cannot avoid accepting as binaries under a given set of required dependencies, and no rebuild with the same requirements can do with fewer. Requiring development and test dependencies enlarged the seed in every ecosystem that records them, by a factor of 5.6 in npm. The difference is the price of building every package with its development and test dependencies, such as the frameworks its test suite runs on, during the bootstrap; Debian avoids part of it by building some source packages in a reduced form that drops optional features, test suites or documentation [1], and the trade-off of H3 quantifies the same option for language ecosystems.

Self-dependencies. Packages that declare a dependency on themselves are a large part of several seeds. With one version per package they cannot be avoided, but a rebuild can break these self-dependencies outside the model: by building the package with a different implementation, as Steel Bank Common Lisp can be built by several other Common Lisp implementations [34], or, when an earlier release did not depend on itself, by building that release first. The version requirements leave the second route open for most of them: in $G_e^{K_3}$, 69% of the packages that depend on

themselves accept an earlier release of themselves and have some earlier release in the dump (whether that release meets the requirement or itself depends on the package was not checked), while 18% accept only their own release or later ones, as do 23 of the 24 in the runtime graph of Maven when its plain versions are read as exact (exploratory; supplementary material). Where such a declaration is a metadata error, the seed overstates the work, and the seed without self-dependencies, reported for every ecosystem, bounds that error.

Why the confirmatory result is not a finding. The shares were large in most ecosystems and stable across specifications, but the conclusion of H1 rested on the classification of a single kind in a single ecosystem: when the suggested packages of CRAN count as build dependencies, CRAN leaves the H1 set and the sign test on the remaining ecosystems cannot reach significance unless all of them exceed one half, which Packagist does not. The robustness threshold of the plan exists for this case: a confirmatory claim should not hinge on one defensible choice among several. The threshold of one half does not adjust for the number of development and test arcs: in RubyGems and npm every share obtained with random labels already exceeds one half, and in Pub most of them do (Figure 3b), so the placebo, not the threshold, is what tests where those arcs sit. The placebo adds a qualification of its own: in RubyGems and Packagist the development and test arcs form no more of the seed than the same number of arcs labelled at random, so where the share is large it is not always because those arcs sit where the cycles are.

Exact optimisation at ecosystem scale. Feedback vertex set is hard in the worst case, yet the dependency graphs of this dump were easy: after the reductions, the largest component handed to the integer program had 786 vertices and closed in seconds. The value of the method lies in the certificate, which removes the doubt about how far a heuristic choice is from the minimum, a doubt that the heuristic sets of the Debian work could not settle [1]. For the minimum seed, the point where exactness stops being cheap was not reached: on subgraphs of the largest component of npm the reductions did most of the work and the exact phase closed in under a second. The bicriteria program reached it: in npm, the two programs with c/w below one and the lexicographic one did not close within τ, nor in an exploratory rerun in which every reduction applied but the limit was 600 s per component, although that rerun narrowed their bounds.

Engineering of the pipeline. Two design choices proved necessary. Isolating the integer solver in its own process turned intermittent memory faults of the solver into recoverable events, and checking every returned seed with a topological order, together with lower bounds that do not depend on the solver finishing, kept those faults from reaching any reported number. Tests written from the formulation, without access to the code, found the faults in a first solver interface, a fault in the solver’s preprocessing and an imprecise statement of a reduction rule before any seed was computed. Further faults, in the solver’s heuristics and cut generators, appeared during the analysis; the heuristics and cut generators were disabled, the whole analysis was run again twice, and the certified values did not change. One run of the secondary analyses was discarded because the solver could not run on an overloaded machine, and was repeated.

6. Threats to Validity

Construct validity. The seed is defined on declared metadata: a dependency of a scenario is assumed to be needed to build or test its package, and a declared dependency that a build never uses would inflate the seed. A build that only needs its compile-time inputs does not need runtime-only dependencies first, such as the runtime scope of Maven, so for such a build the runtime seed is an upper bound; it is exact when every runtime dependency must be present to build and test a package. Libraries.io relies on package metadata alone and has material errors in some complex cases [35]. The kinds that each package manager declares do not mean the same thing everywhere: npm, RubyGems, Packagist and Pub have no build-only kind, so tools that build a package, such as compilers and task runners, are declared as development dependencies there, and θ_e measures the share added by the kinds labelled development and test, not by testing alone. A package that declares a dependency on itself counts as needing a binary of itself, even when the declaration is a mistake; the seeds without such packages are reported alongside. We fixed the map from kinds to scenarios in the plan before computing any seed and put its most doubtful entries, CRAN suggestions and Maven system scope, in the multiverse; the result of H1 depends on the first of them, which is why it

is not reported as a finding. A single version per package removes the version choice from the problem: npm allows several versions of a package in one dependency chain [15], so a cycle between the newest versions may not exist between the versions an actual build would pick; a package that depends on an earlier release of itself is the extreme case, a self-loop here but not a cycle in a model with several versions per package, and the requirements of most self-dependencies admit such a release. The multiverse varies the version rule only between the newest release and the newest one without a pre-release mark.

Internal validity. Every seed is computed by an exact method whose output is checked independently of the solver: a topological order confirms that the returned set breaks every cycle, and the lower bound comes from the cycle packing, the linear relaxation and, where these fall short, the integer solver's own proof of optimality; both bounds coincide in every instance of the primary analysis. A check that does not use the integer solver at all, added after the plan (exploratory; supplementary material), certifies 32 of the 33 seeds of the primary analysis: in 31 instances the reductions leave no component, and in CPAN with development and test dependencies a dual solution of the linear relaxation, which can be checked with additions alone, closes the remaining component. In npm with development and test dependencies, exhaustive search settles its 4 small components, but in the largest one the relaxation, rounded up, reaches only 116 of the 118 packages of its seed; without the solver, the seed of npm with development and test dependencies is therefore known to lie between 2,204 and 2,206 packages, which keeps its share at 0.82 or more. The rest of its certificate is the solver's proof, which the topological order cannot check and on which the independent replica also relied. In early runs the solver process failed intermittently on the test machine; such failures cannot produce a wrong seed, because a failed process returns nothing and its component keeps the bounds it already had. The failures are counted, and their number in the reported runs of the primary and secondary analyses was 0. The reductions and the guarantee are covered by property-based tests written from the formulation alone, and an independent replica reimplemented the primary analysis from the text of the plan.

External validity. The data are one dump of one service, dated 15 June 2017, and the seeds describe the packages that the dump resolves within each ecosystem: where the dump records few dependencies, as for PyPI, or leaves many unresolved, as for CPAN and CRAN (Table 2), the seed describes the dump rather than the registry. Unresolved dependencies are left out under the plan's assumption that whether a dependency is resolved does not depend on its role in the cycles, and resolving them by exact name, a dimension of the multiverse, moved θ_e by at most 0.01. Dependencies that cross ecosystems, the toolchain below each ecosystem and later states of the registries are outside the study. In the dump, 3 ecosystems record only runtime dependencies, so the comparison between scenarios covers the other 8. The Maven Dependency Graph, the second source used here, records almost no test-scoped edges (1,341 of 9,715,669), so it can only be compared with Libraries.io on the runtime seed of Maven (H4), not on the share θ_e.

Conclusion validity. With 8 ecosystems the sign test has low power, and its discreteness matters: without CRAN, 7 ecosystems remain and the test can only be positive if all of them exceed one half. The confirmatory family has a single hypothesis, the exploratory tests are corrected by Benjamini and Hochberg's procedure, and every decision rule, threshold and specification was fixed before the analysis; the result that failed its robustness check is reported as such.

7. Conclusions

The minimum binary seed of every language package ecosystem with cycles in the Libraries.io dump was computed exactly. An algorithmic pipeline that reduces each dependency graph and solves what remains by integer programming with lazily generated cycle constraints certified the optimum of all 33 instances of these 11 ecosystems, and showed that requiring development and test dependencies enlarges the seed in every ecosystem that records them. A pre-specified hypothesis that these dependencies account for most of the seed was positive in the primary specification but did not survive the multiverse fixed in advance, and is reported as not valid. Two observations qualify the seeds: in an exploratory count, about 50% of the runtime seed packages declare a dependency on themselves, and suspending at most 3.1% of the development and test dependencies of an ecosystem (up to 68.7% of those on a cycle)

removed the extra seed that they require (in npm, 0.26% sufficed in an exploratory rerun, without a certified minimum). The certified seeds give a target against which heuristic bootstrapping plans can be measured; a greedy heuristic came within 6.4% of it in every ecosystem, and within 0.9% when Levy and Low's reductions were applied before each of its choices (both exploratory). Extending the model to several versions per package, to the toolchain below each ecosystem and to other registries would show how far these numbers carry.

8. Funding

This research received no external funding.

9. Conflicts of Interest

The author declares no conflict of interest.

10. Data Availability Statement

Includes data from Libraries.io. The data are public and released under the Creative Commons Attribution-ShareAlike 4.0 International License: version 1.0.1 of the Libraries.io open data (https://doi.org/10.5281/zenodo.833207) and the Maven Dependency Graph (https://doi.org/10.5281/zenodo.1489120). Every result reported here is derived from these files by the analysis pipeline, and derived data that are shared carry the same license.

11. Code Availability

The code of the pipeline, its tests and the analysis plan are available from the author on reasonable request.

12. Ethics Statement

The study uses only public metadata of software packages, whose names sometimes include the public account name of their publisher; it involves no human participants and no animals, and no personal data are collected or reported.

REFERENCES

1. J. Schauer. "Solving the Bootstrap Problem for Free and Open Source Binary Distributions", Master's Thesis, Jacobs University, School of Engineering and Science, 2013. Available at: https://mister-muffin.de/bootstrap/thesis.pdf
2. P. A. Karger, R. R. Schell. "Thirty Years Later: Lessons from the Multics Security Evaluation". In *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 119–126, 2002. https://doi.org/10.1109/csac.2002.1176285
3. K. Thompson. "Reflections on Trusting Trust", *Communications of the ACM*, 27 (8), pp. 761–763, 1984. https://doi.org/10.1145/358198.358210
4. J. Nieuwenhuizen, "Guix Reduces Bootstrap Seed by 50%", GNU Guix blog, 2019. Available at: https://guix.gnu.org/blog/2019/guix-reduces-bootstrap-seed-by-50/
5. E. Dolstra. "The Purely Functional Software Deployment Model", PhD Thesis, Universiteit Utrecht, 2006. Available at: https://dspace.library.uu.nl/handle/1874/7540
6. J. Nieuwenhuizen, "Re: npm (mitigation)", Message to the guix-devel mailing list, 2017. Available at: https://lists.gnu.org/r/guix-devel/2017-07/msg00195.html
7. A. Decan, T. Mens, P. Grosjean. "An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems", *Empirical Software Engineering*, 24 (1), pp. 381–416, 2019. https://doi.org/10.1007/s10664-017-9589-y
8. A. Decan, T. Mens, E. Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network". In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 404–414, 2018. https://doi.org/10.1109/icsme.2018.00050
9. A. Zerouali, E. Constantinou, T. Mens, G. Robles, J. González-Barahona. "An Empirical Analysis of Technical Lag in npm Package Dependencies". In *New Opportunities for Software Reuse*, pp. 95–110, 2018. https://doi.org/10.1007/978-3-319-90421-4_6
10. R. M. Karp. "Reducibility among Combinatorial Problems", in *Complexity of Computer Computations*, Springer US, pp. 85–103, 1972. https://doi.org/10.1007/978-1-4684-2001-2_9
11. O. Svensson. "Hardness of Vertex Deletion and Project Scheduling", *Theory of Computing*, 9 (1), pp. 759–781, 2013. https://doi.org/10.4086/toc.2013.v009a024

12. P. Abate, R. Di Cosmo. “Adoption of Academic Tools in Open Source Communities: The Debian Case Study”. In *Open Source Systems: Towards Robust Practices*, pp. 139–150, 2017. https://doi.org/10.1007/978-3-319-57735-7_14
13. D. Wheeler. “Countering Trusting Trust through Diverse Double-Compiling”. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC’05)*, pp. 33–48, 2005. https://doi.org/10.1109/csac.2005.17
14. Z. Ren, H. Jiang, J. Xuan, Z. Yang. “Automated Localization for Unreproducible Builds”. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 71–81, 2018. https://doi.org/10.1145/3180155.3180224
15. R. Kikas, G. Gousios, M. Dumas, D. Pfahl. “Structure and Evolution of Package Dependency Networks”. In *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112, 2017. https://doi.org/10.1109/msr.2017.55
16. J. Dietrich, D. Pearce, J. Stringer, A. Tahir, K. Blincoe. “Dependency Versioning in the Wild”. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 349–359, 2019. https://doi.org/10.1109/msr.2019.00061
17. A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, O. Barais. “The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central”. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, 2019. https://doi.org/10.1109/msr.2019.00060
18. I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, F. Massacci. “Vulnerable Open Source Dependencies: Counting Those That Matter”. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10, 2018. https://doi.org/10.1145/3239235.3268920
19. R. G. Kula, D. M. German, A. Ouni, T. Ishio, K. Inoue. “Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration”, *Empirical Software Engineering*, 23 (1), pp. 384–417, 2018. https://doi.org/10.1007/s10664-017-9521-5
20. G. Even, J. Naor, B. Schieber, M. Sudan. “Approximating Minimum Feedback Sets and Multicuts in Directed Graphs”, *Algorithmica*, 20 (2), pp. 151–174, 1998. https://doi.org/10.1007/pl00009191
21. J. Chen, Y. Liu, S. Lu, B. O’Sullivan, I. Razgon. “A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem”, *Journal of the ACM*, 55 (5), pp. 1–19, 2008. https://doi.org/10.1145/1411509.1411511
22. H. Levy, D. W. Low. “A Contraction Algorithm for Finding Small Cycle Cutsets”, *Journal of Algorithms*, 9 (4), pp. 470–493, 1988. https://doi.org/10.1016/0196-6774(88)90013-2
23. M. Lemaic. “Markov-Chain-Based Heuristics for the Feedback Vertex Set Problem for Digraphs”, PhD Thesis, Universität zu Köln, 2008. Available at: https://kups.ub.uni-koeln.de/2547/
24. H.-M. Lin, J.-Y. Jou. “On Computing the Minimum Feedback Vertex Set of a Directed Graph by Contraction Operations”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19 (3), pp. 295–307, 2000. https://doi.org/10.1109/43.833199
25. P. M. Pardalos, T. Qian, M. G. C. Resende. “A Greedy Randomized Adaptive Search Procedure for the Feedback Vertex Set Problem”, *Journal of Combinatorial Optimization*, 2 (4), pp. 399–412, 1998. https://doi.org/10.1023/a:1009736921890
26. K. Chandrasekaran, R. M. Karp, E. Moreno-Centeno, S. Vempala. “Algorithms for Implicit Hitting Set Problems”. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 614–629, 2011. https://doi.org/10.1137/1.9781611973082.48
27. M. Grötschel, M. Jünger, G. Reinelt. “On the Acyclic Subgraph Polytope”, *Mathematical Programming*, 33 (1), pp. 28–42, 1985. https://doi.org/10.1007/bf01582009
28. M. Funke, G. Reinelt. “A Polyhedral Approach to the Feedback Vertex Set Problem”. In *Integer Programming and Combinatorial Optimization*, pp. 445–459, 1996. https://doi.org/10.1007/3-540-61310-2_33
29. M. Simpson, V. Srinivasan, A. Thomo. “Efficient Computation of Feedback Arc Set at Web-Scale”, *Proceedings of the VLDB Endowment*, 10 (3), pp. 133–144, 2016. https://doi.org/10.14778/3021924.3021930
30. C. Tucker, D. Shuffelton, R. Jhala, S. Lerner. “OPIUM: Optimal Package Install/Uninstall Manager”. In *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*, pp. 178–188, 2007. https://doi.org/10.1109/icse.2007.59
31. A. Nesbitt, B. Nickolls, “Libraries.io Open Source Repository and Dependency Metadata”, Zenodo, version 1.0.1, 2017. https://doi.org/10.5281/zenodo.833207
32. S. Steegen, F. Tuerlinckx, A. Gelman, W. Vanpaemel. “Increasing Transparency through a Multiverse Analysis”, *Perspectives on Psychological Science*, 11 (5), pp. 702–712, 2016. https://doi.org/10.1177/1745691616658637
33. Y. Benjamini, Y. Hochberg. “Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing”, *Journal of the Royal Statistical Society. Series B (Methodological)*, 57 (1), pp. 289–300, 1995. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x
34. C. Rhodes. “SBCL: A Sanely-Bootstrappable Common Lisp”. In *Self-Sustaining Systems*, pp. 74–86, 2008. https://doi.org/10.1007/978-3-540-89275-5_5
35. E. Bommarito, M. Bommarito, “An Empirical Analysis of the Python Package Index (PyPI)”, arXiv:1907.11073, 2019. Available at: https://arxiv.org/abs/1907.11073v1