

CS 4530: Fundamentals of Software Engineering

Module 16 (alternate): Dependency Management

Adeel Bhutta and Mitch Wand (with material by Donald Pinckney)

Khoury College of Computer Sciences

© 2024, released under [CC BY-SA](#)

Learning Objectives for this Module

- By the end of this module, you should be able to:
 - Explain why you need dependencies
 - Explain the major risks of dependencies
 - Explain the principles of semantic versioning
 - Explain what a package manager does
 - Understand that different package managers may solve dependencies differently

Software isn't written in a vacuum

- Writing a JS app?
 - you depend on: React, 100s of small JS packages, Node, V8, ...
- Writing ML code in Python?
 - You depend on: PyTorch, Numpy, CUDA, C libraries, compilers, ...
- And so on for nearly all software

Our context:

- You are writing an application in JS/TS
- You need some services
- Is there a dependency you can use, or should you build your own?

Risks of Dependencies

- You are reliant on the designer's choices
 - (but: they may have done a better job than you would)
- Security risks
- Upstream risks (transitive dependencies)
- May need multiple copies of some dependencies
- How to keep them all up to date??

Dependency Management Isn't Easy

- Too many dependencies to manage manually
 - Often 100+ for JavaScript projects when considering transitive dependencies
- Too frequent dependency updates to apply manually
 - Even though they may be very important, e.g. critical security patches!
- Dependency updates can't be done in isolation: you may have to update other dependencies to match

We can control the direct dependencies, but not the transitive dependencies

- We declare our immediate dependencies in a *manifest*: eg package.json
- But we don't/can't control our dependencies' dependencies

What the *#!?

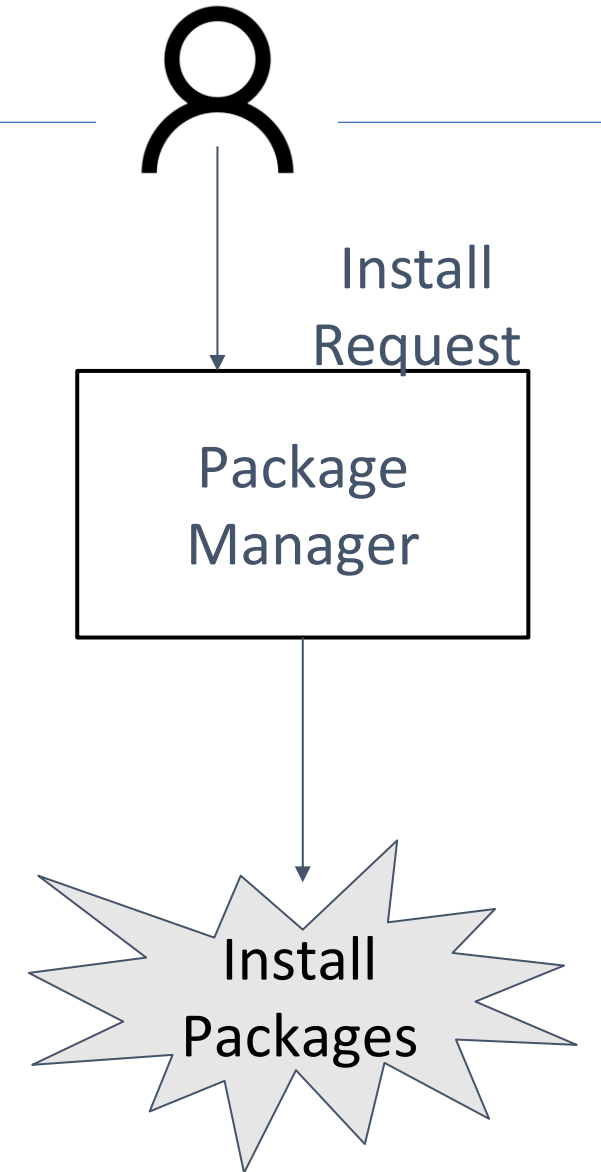
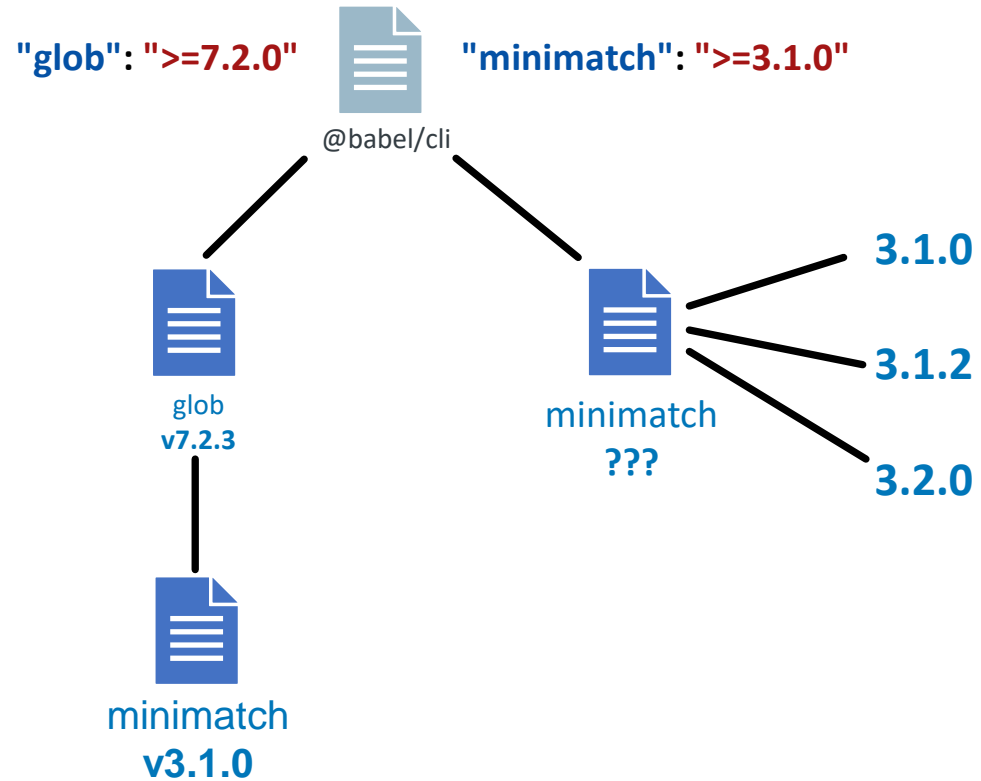
```
$ npm install
npm WARN deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do
not use it. Check out lru-cache if you want a good and tested way to coalesce async
requests by a key value, which is much more comprehensive and powerful.
npm WARN deprecated @humanwhocodes/config-array@0.13.0: Use @eslint/config-array instead
npm WARN deprecated rimraf@3.0.2: Rimraf versions prior to v4 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated @humanwhocodes/object-schema@2.0.3: Use @eslint/object-schema instead
npm WARN deprecated eslint@8.57.1: This version is no longer supported. Please see
https://eslint.org/version-support for other options.
```

```
added 750 packages, and audited 751 packages in 1m
```


Package Managers Manage the Transitive Dependencies

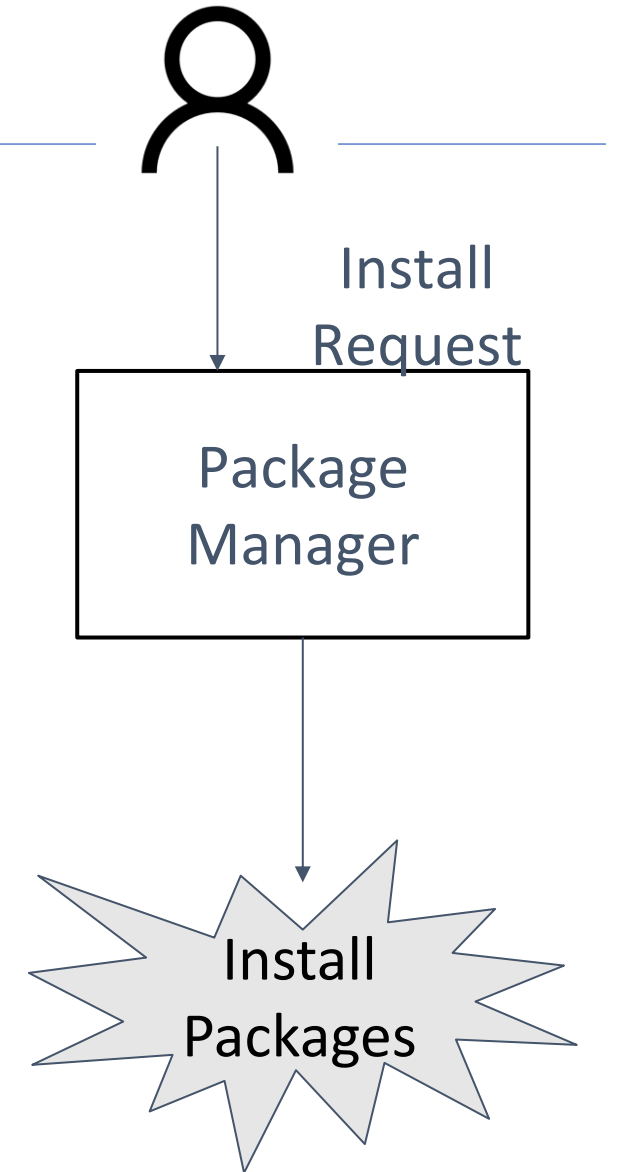
- NPM: 3 million+ packages
- Complex graph of dependencies
- 20TB+ of package code
- Fairly rich dependency specification language

Package Managers Manage the Transitive Dependencies



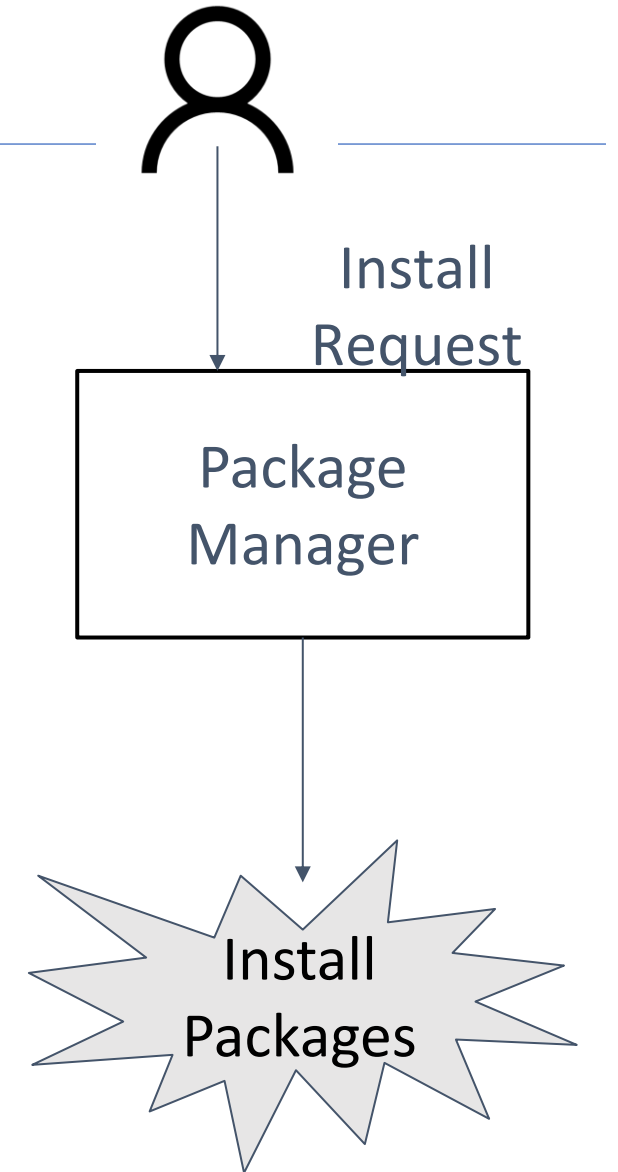
What Can Go Wrong?

- Dependency solving can fail
 - Conflicting constraints
 - Weaknesses in solving algorithms (old Pip, current NPM)



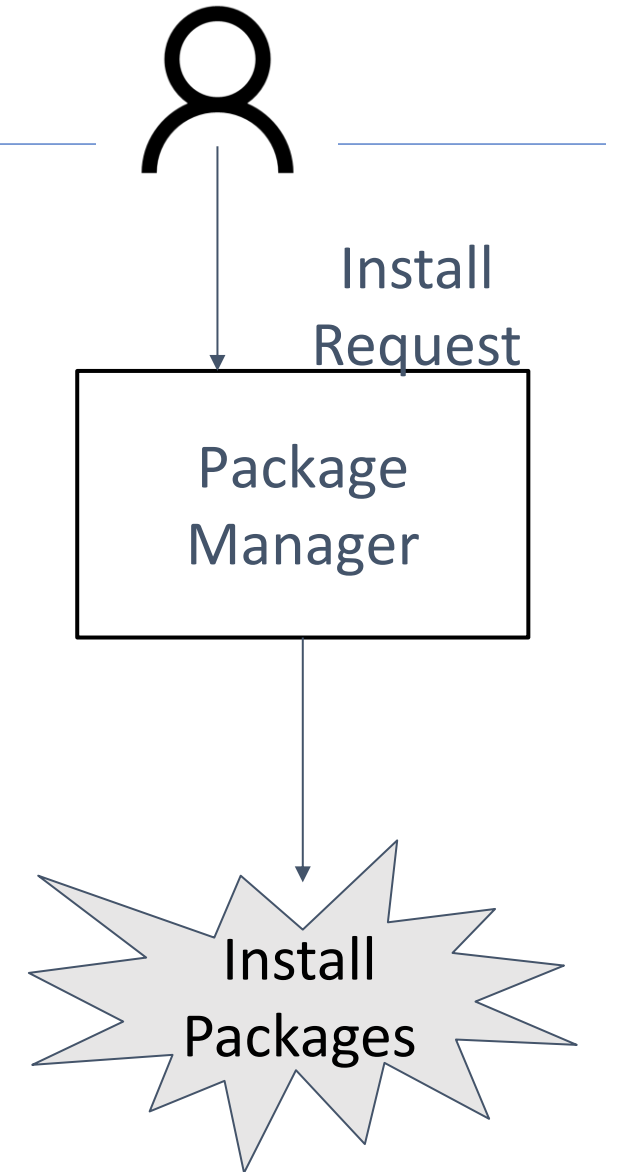
What Can Go Wrong?

- Dependency solving can fail
 - Conflicting constraints
 - Weaknesses in solving algorithms (old Pip, current NPM)
- Dependency solutions can induce code failures
 - Build failures, runtime crashes, runtime bugs, etc.



What Can Go Wrong?

- Dependency solving can fail
 - Conflicting constraints
 - Weaknesses in solving algorithms (old Pip, current NPM)
- Dependency solutions can induce code failures
 - Build failures, runtime crashes, runtime bugs, etc.
- Low-quality dependency solutions
 - Security vulnerabilities
 - Large code size
 - Old versions of packages



Semantic Versioning Can Help Keep Track of Breaking Changes

- Given a version number MAJOR.MINOR.PATCH, increment the:
 - MAJOR version when you make incompatible API changes
 - MINOR version when you add functionality in a backward compatible manner
 - PATCH version when you make backward compatible bug fixes
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

<https://semver.org/>

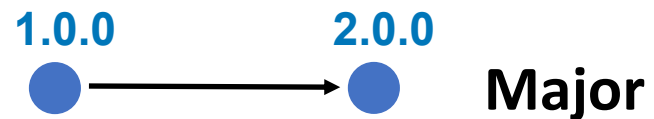
Semantic Versioning takes effort

"It's hard to follow semantic versioning—it takes significant effort to make backward-compatible changes, backward-compatible bug fixes, and to backport security patches to old release numbers. However, following semantic versioning is the best way to spread joy to your downstream users."

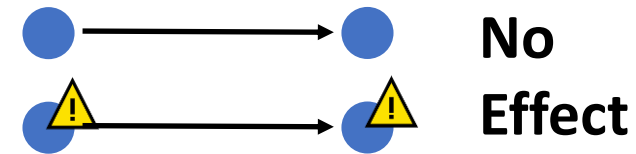
<https://semver.org/>

Characterizing Updates

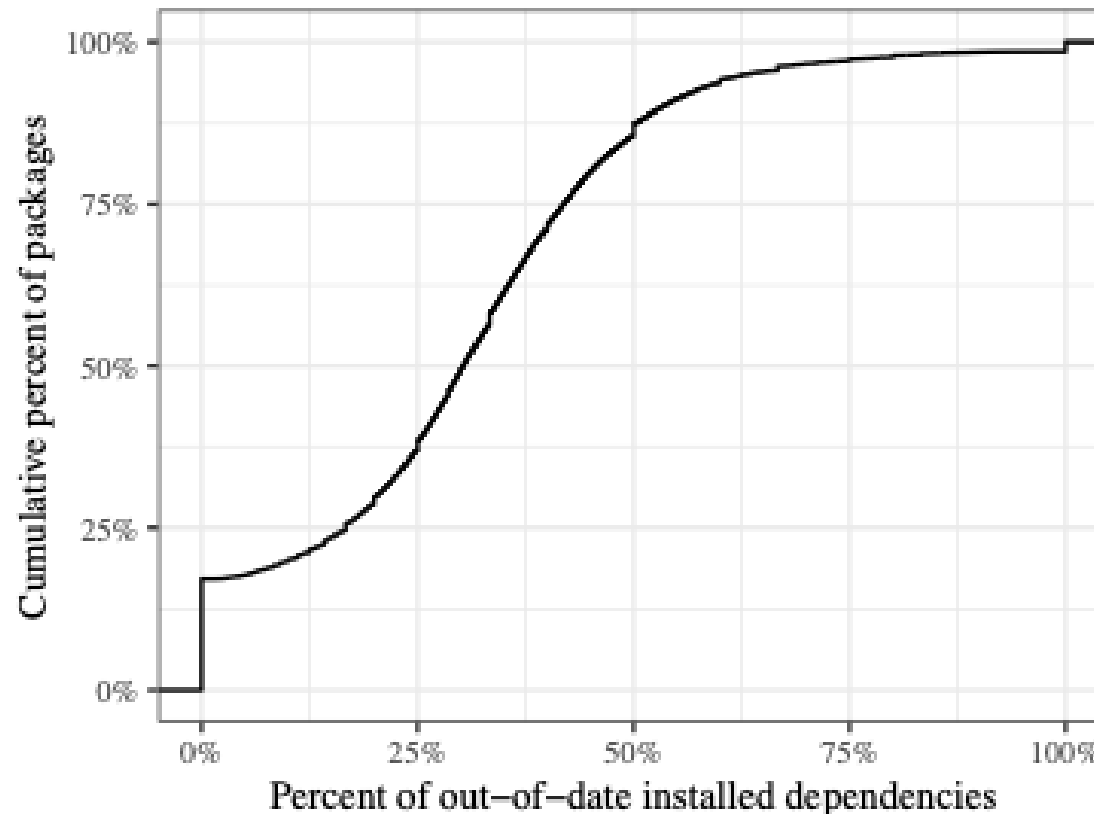
Semver Increment



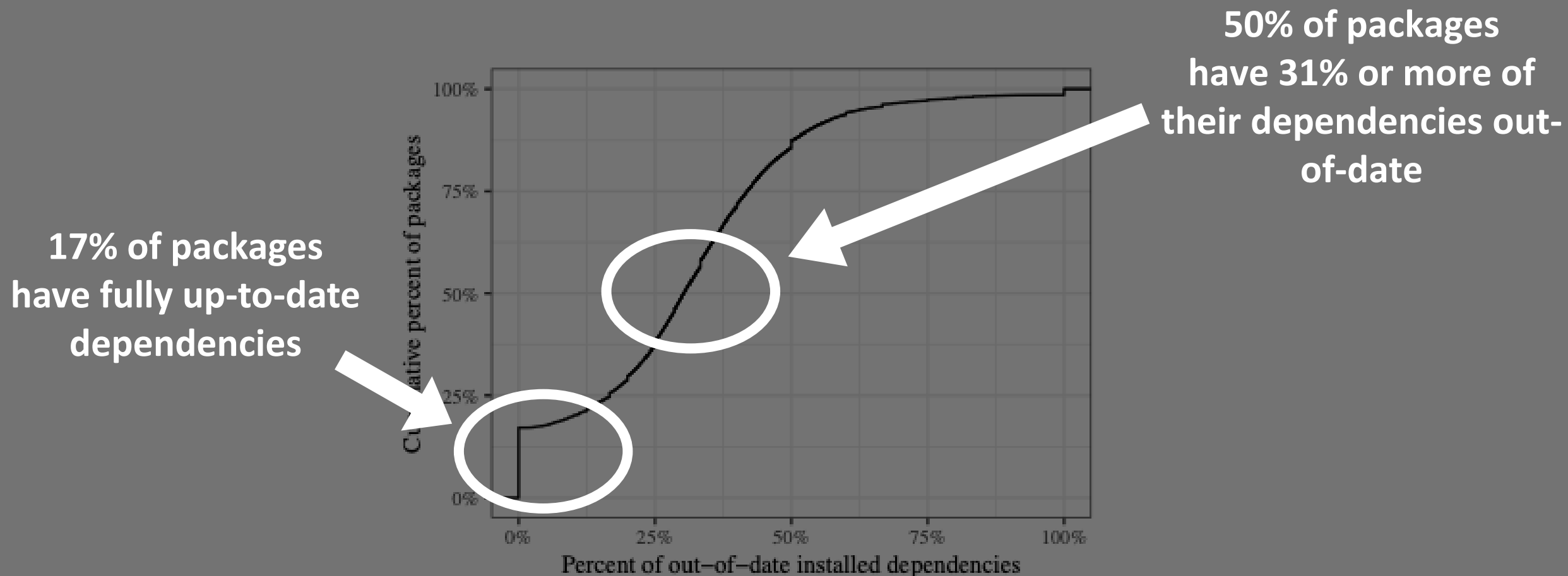
Security Effect



Most Packages Have Out-of-Date Dependencies



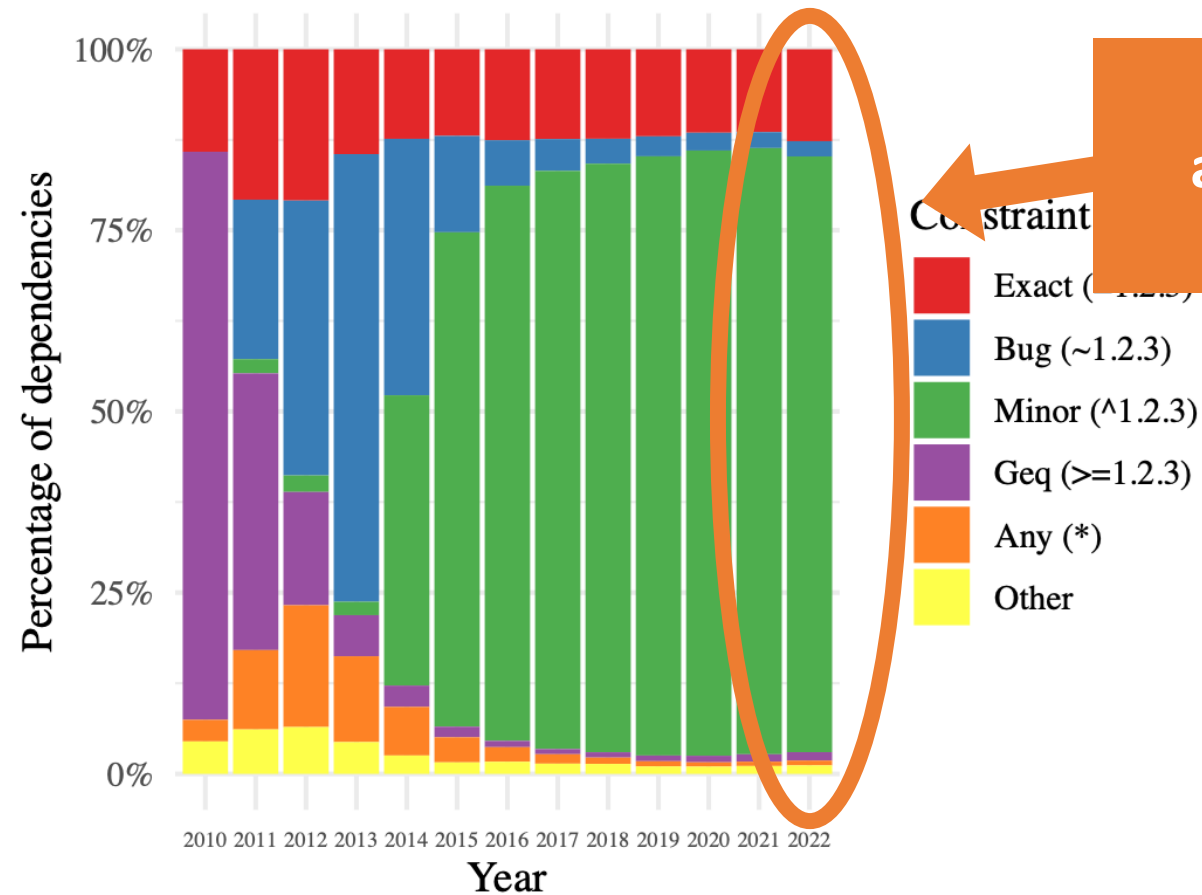
Most Packages Have Out-of-Date Dependencies



Semantic Versioning starts with your package.json

```
1 "dependencies": {  
2   "react": "18.2.0", // Exactly this version  
3   // Any patch updates (7.2.5, 7.2.6, etc.)  
4   "protobufjs": "~7.2.5",  
5   // Any patch or minor updates (4.17.21, 4.18.0, etc.)  
6   "lodash": "^4.17.21",  
7   // Arbitrary conjunctions and disjunctions  
8   "moment": ">=1.0.0 <1.2.0 || ^2.3.1"  
9 }
```

Developers Rarely Distinguish Bug vs. Minor Updates



Most constraints are either exact or minor-flexible

Probably because npm --save defaults to "^"

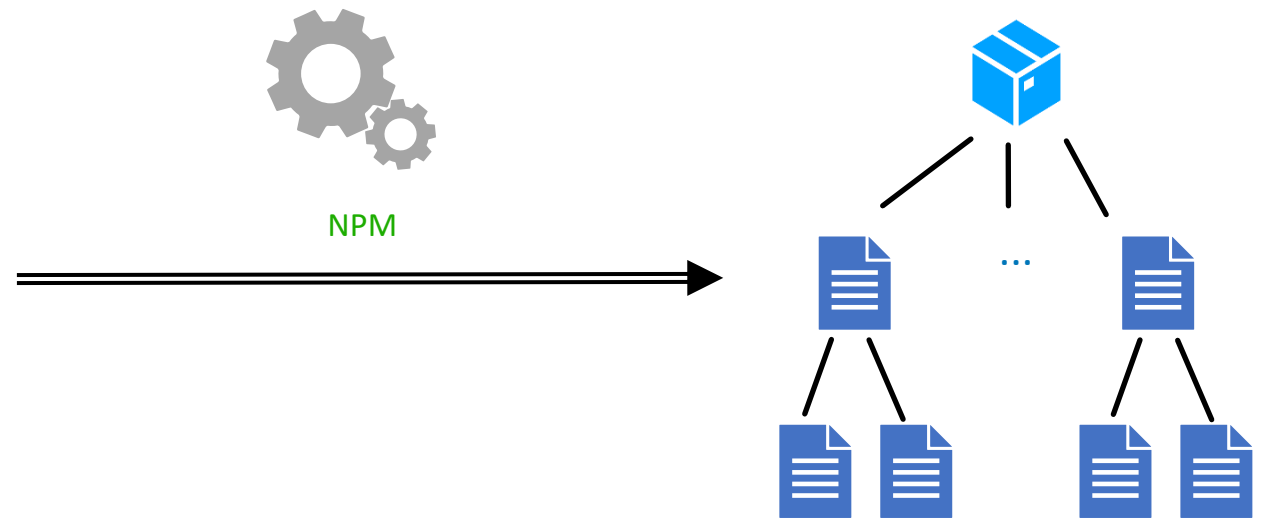
```
"dependencies": {  
  "@chakra-ui/next-js": "^2.2.0",  
  "@chakra-ui/react": "^2.8.2",  
  "@emotion/styled": "^11.11.0",  
  "framer-motion": "^11.0.3",  
  "jsdoc": "^4.0.2",  
  "next": "14.1.0",  
  "react": "^18",  
  "react-dom": "^18",  
  "react-icons": "^5.0.1",  
  "what-props-changed": "^1.0.2"  
},
```

Implications For Developers & Researchers

- Consider using ~ constraints (bug updates) instead of ^ (bug + minor updates)
 - At the cost of technical lag
 - And forcing the technical lag on clients
- Alternatively, allow developers to specify preferences outside of constraints
 - what would that even mean? 🤔

What is dependency solving?

```
"dependencies": {  
  "commander": "^2.8.1",  
  "convert-source-map": "^1.1.0",  
  "fs-readdir-recursive": "^1.1.0",  
  "glob": "^7.0.0",  
  "lodash": "^4.17.10",  
  "mkdirp": "^0.5.1",  
  "output-file-sync": "^2.0.0",  
  "slash": "^2.0.0",  
  "source-map": "^0.5.0"  
}
```

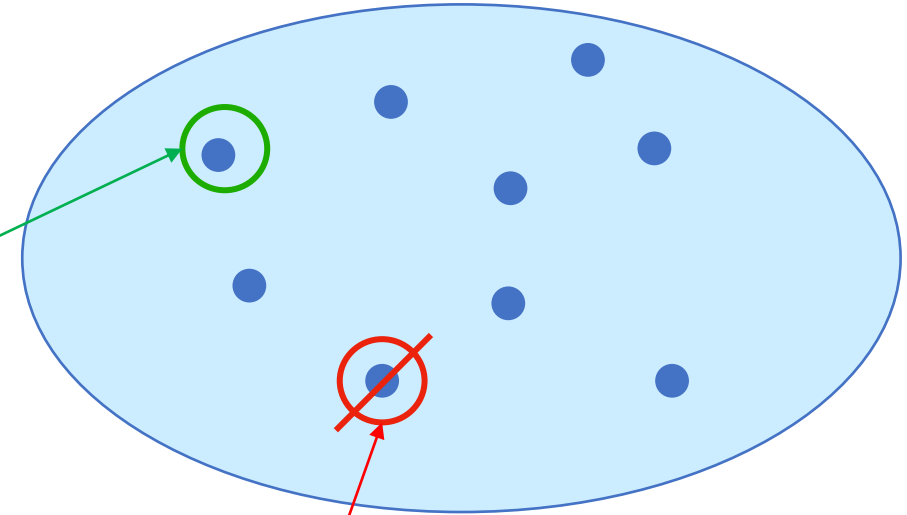


Consider the solution space

```
"dependencies": {  
  "commander": "^2.8.1",  
  "convert-source-map": "^1.1.0",  
  "fs-readdir-recursive": "^1.1.0",  
  "glob": "^7.0.0",  
  "lodash": "^4.17.10",  
  "mkdirp": "^0.5.1",  
  "output-file-sync": "^2.0.0",  
  "slash": "^2.0.0",  
  "source-map": "^0.5.0"  
}
```

```
{  
  "commander" @ "^2.8.1",  
  "convert-source-map": @ "^1.1.1",  
  ...  
}
```

```
{  
  "commander" @ "^2.8.1",  
  "convert-source-map": @ "^2.1.1",  
  ...  
}
```



Versions, Constraints, and Constraint Semantics

$\mathcal{V} := (x\ y\ z)$ Version numbers

$\mathcal{C} :=$

| | |
|--|-------------|
| $(= x\ y\ z)$ | Exact |
| $(\leq x\ y\ z)$ | At most |
| $(\geq x\ y\ z)$ | At least |
| $(\text{and } \mathcal{C}_1\ \mathcal{C}_2)$ | Conjunction |

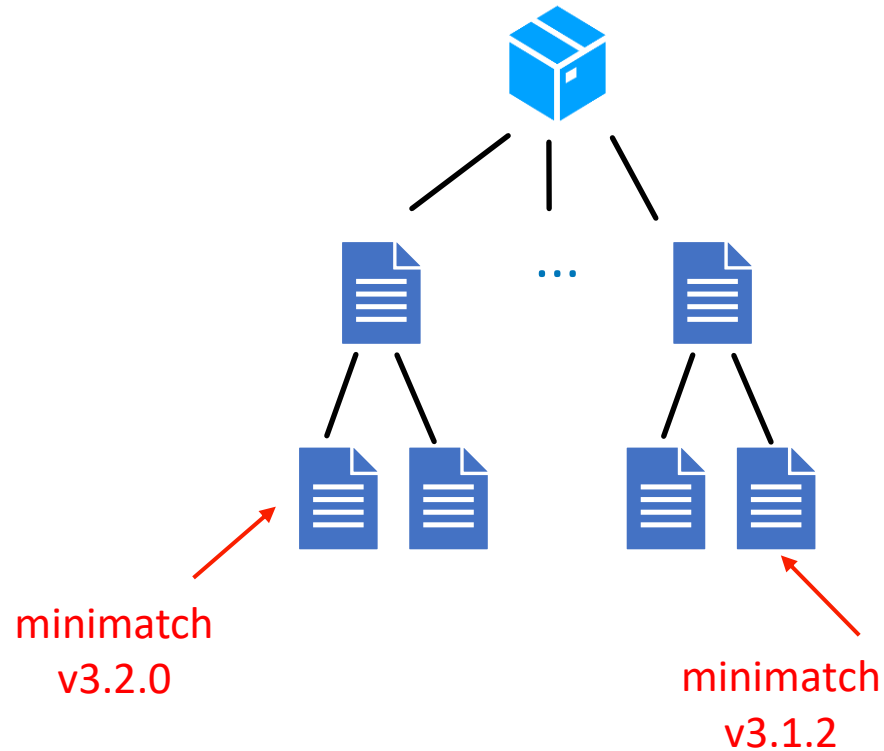
$\text{sat} : \mathcal{C} \rightarrow \mathcal{V} \rightarrow \mathbf{Bool}$

Does version v
satisfy
constraint c ?

```
(define (sat c v)
  (match `(,v ,c)
    [ `(,x ,y ,z) (= ,x ,y ,z))      #true]
    [ `(,x ,y ,z1) (<= ,x ,y ,z2))   (<= z1 z2)]
    [ `(,x ,y1 ,z1) (<= ,x ,y2 ,z2)) (< y1 y2)]
    [ `(,x1 ,y1 ,z1) (<= ,x2 ,y2 ,z2)) (< x1 x2)]
    [ `(_ (and ,c1 ,c2))
      (and (sat c1 v) (sat c2 v))]
    [ `(,x1 ,y1 ,z1) (>= ,x2 ,y2 ,z2))
      (sat `(,x2 ,y2 ,z2) `( <= ,x1 ,y1 ,z1))]
    [ _                                     #false]))
```

Are Multiple Versions of a Package Allowed?




- NPM: Yes
- PIP: No
- Cargo: Partially

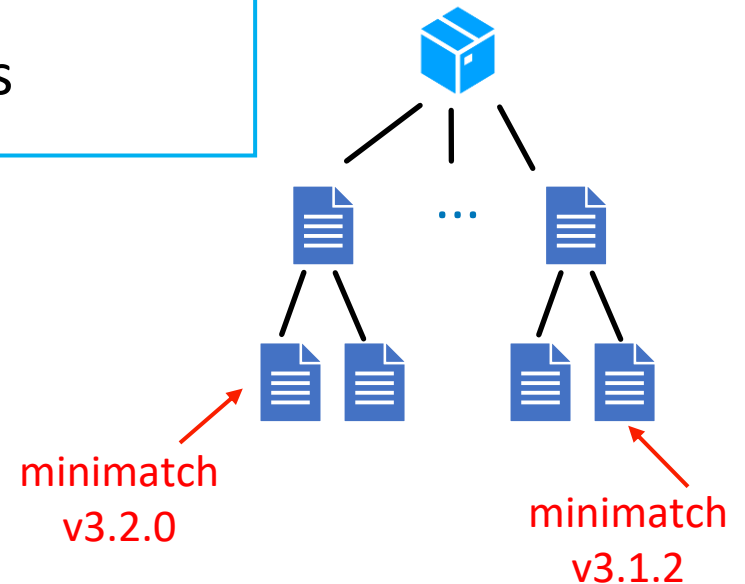


If multiple versions are allowed, which ones are consistent with each other?

consistent : $\mathcal{V} \rightarrow \mathcal{V} \rightarrow \mathbf{Bool}$

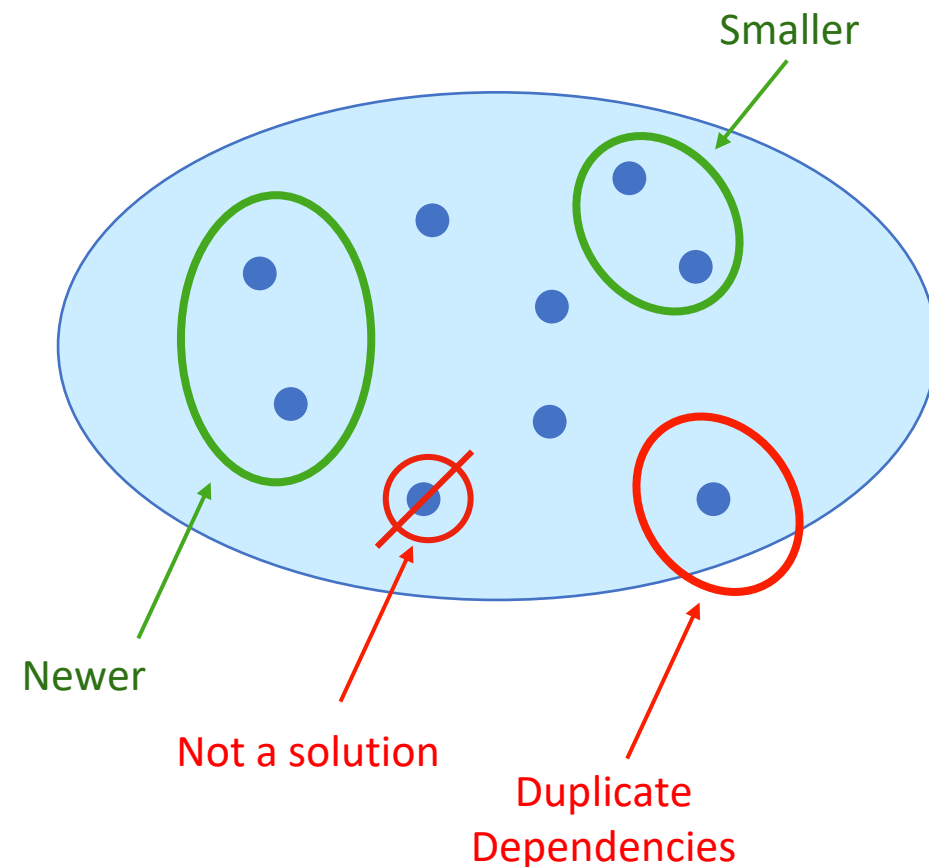
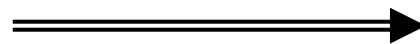
Different package managers may allow different combinations

```
1 (define (npm-consistent v1 v2) )
2   #true)
3
4 (define (pip-consistent v1 v2) )
5   (equal? v1 v2))
6
7 (define (cargo-consistent v1 v2) )
8   (match `(,v1 ,v2)
9     [ `((0 0 ,z1) (0 0 ,z2)) #true]
10    [ `((0 ,y ,z1) (0 ,y ,z2)) (= z1 z2)]
11    [ `((0 ,y1 ,z1) (0 ,y2 ,z2)) #true]
12    [ `((x ,y1 ,z1) (x ,y2 ,z2)) (and (= y1 y2) (= z1 z2))]
13    [ _ #true]))
```



If there are many possible solutions, which one should we choose?

```
"dependencies": {  
  "commander": "^2.8.1",  
  "convert-source-map": "^1.1.0",  
  "fs-readdir-recursive": "^1.1.0",  
  "glob": "^7.0.0",  
  "lodash": "^4.17.10",  
  "mkdirp": "^0.5.1",  
  "output-file-sync": "^2.0.0",  
  "slash": "^2.0.0",  
  "source-map": "^0.5.0"  
}
```



$$\text{minGoal} : \mathcal{G} \rightarrow \mathbb{R}^n$$

So this is an optimization problem!

Minimize # of Dependencies

```
(define (minGoal-num-deps g)
  (length (graph-nodes g)))
```

Different package managers may have different optimization goals

Prefer Newer Versions

```
1 (define (minGoal-oldness g)
2   (apply +
3     (map
4       (lambda (n)
5         (get-oldness
6           (node-package n)
7           (node-version n)))
8       (graph-nodes g))))
9
10 (define (get-oldness p v)
11   ; The get-sorted-versions retrieve
12   ; a list of all versions of p
13   (define all-vs
14     (get-sorted-versions p))
15   (if (= (length all-vs) 1)
16     0
17     (/ (index-of all-vs v)
18        (sub1 (length all-vs)))))
```

Tunable knobs for a package manager

$sat : \mathcal{C} \rightarrow \mathcal{V} \rightarrow \mathbf{Bool}$

Constraint satisfaction semantics

$consistent : \mathcal{V} \rightarrow \mathcal{V} \rightarrow \mathbf{Bool}$

Version consistency versions

$cycles_ok \in \mathbf{Bool}$

If cycles are permitted in solution graphs

$minGoal : \mathcal{G} \rightarrow \mathbb{R}^n$

Objective functions

Luckily, most projects are robust to different dependency solutions

| MAXNPM | Pass | Fail |
|--------|------------|------------|
| | Pass | Fail |
| Pass | 563 77% | 7 1% |
| Fail | 38 5% | 127 17% |
| NPM | | |