Resolver.jl

A New SAT-based Resolver

Stefan Karpinski

https://JuliaHub.com

Package Version Resolution Problem

The setup:

- There are *packages*
- Each package has *versions*
- Each version *depends* on a set of packages
- A subset *incompatible* pairs of versions
  - only makes sense for versions of different packages

Package Version Resolution Problem

A valid solution:

- Every dependency is satisfied
- No incompatible version pairs

A solution satisfies a set of requirements (packages) when:

- The solution contains a version of each required package

Package Version Resolution Problem

The decision problem:

- *Is there a valid solution that satisfies a set of requirements?*

This problem is somewhat famously NP-complete

- If you can solve this you can solve SAT problems
- You might as well use a SAT solver (or something stronger)

Otherwise you'll end up badly implementing a buggy, slow SAT solver

Not Just Any Solution

Obviously just knowing if there is a solution is not that useful

- That’s just how computational complexity classes are defined

Optimal Solutions

We also don't just want any solution — we want an optimal solution

- Requires a notion of some solutions

Start by putting a preference ordering on versions of each package

- Extend this to a preference ordering on solutions
- Actually more tricky and subtle than expected

What We Do Now

Pkg.jl includes a version resolver

- Uses belief propagation
- Heuristic: may not find solutions nor necessarily optimal ones
- But, works remarkably well

Implemented and maintained by Carlo Baldassi

- Thank you, Carlo!

Version Numbers

The biggest issue with the existing resolver:

- Structure & meaning of version numbers deeply baked into logic
- Bakes in that higher version numbers are better
- Can’t support pre-release or build numbers

Newer ≠ Better?

When would you prefer older versions over newer ones?

- Version fixing — prefer current version
  - minimize changes to manifests
- Download avoidance — prefer pre-installed versions
  - avoid installing new versions if possible
- Downgrade resolution — prefer oldest allowable versions
  - useful for testing lower compat bounds

We do all of these in hacky ways currently

Resolver.jl Approach

- Avoid coupling with details of packages, versions, registries
  - Resolver.jl doesn't know about any of these
- Use an actual SAT solver (PicoSAT)
  - how does optimization work? (you'll see)
- SAT solvers are very sensitive to problem size
  - significant preprocessing to minimize SAT problem size

Ends up being fast & scalable

Emergent Features

- Since it doesn’t know about version numbers
  - can give it arbitrary preference ordering
- Generates optimal solutions in lexicographical order
  - user can specify priority of packages
- Semi-internal SAT problem API is more generally useful
  - can be used for other related problems

Also quite flexible

SAT

A SAT problem consists of:

- A number of variables
- A number of clauses — all must be satisfied
- Each clause is a *disjunction* of variables and negated variables

Example: $(a \vee \neg b \vee c) \wedge (\neg a \vee b)$

- Equivalent: $(b \Rightarrow (a \vee c)) \wedge (a \Rightarrow b)$

Encoding Package Problems

Variables:

- One for each package: p
  - use p and q for packages
- One for each version: v
  - use v and w for versions

Some Notation

Properties:

- Package of a version: $p = P(v)$
- Versions of a package: $v \in V(p)$
- Dependencies of a version: $q \in D(v)$
- Set of conflicts: $\{v, w\} \in C$
- Conflicts of a version: $w \in C_v$

Clauses

meaning	quantifiers	clause
version implies its package	$\forall\, v$	$v \Rightarrow P(v)$
package implies some version	$\forall\, p$	$p \Rightarrow \bigvee V(p)$
only one version of a package	$\forall\, p, \{v, w\} \subseteq V(p)$	$\neg v \vee \neg w$
versions imply dependencies	$\forall\, v, q \in D(v)$	$v \Rightarrow q$
conflicts are exclusive	$\forall\, \{v, w\} \in C$	$\neg v \vee \neg w$

Code

Loop over packages

```
for p in names
    info_p = info[p]
    n_p = length(info_p.versions) # number of versions of p
    v_p = vars[p]                 # lookup variable for p

    # generate clauses for p
end
```

Code

Version implies its package

```
for i = 1:n_p
    PicoSAT.add(pico, -(v_p + i)) # ¬(p@i)
    PicoSAT.add(pico, v_p)        # p
    PicoSAT.add(pico, 0)          # end clause
end
```

Code

Package implies some version

```
PicoSAT.add(pico, -v_p)          # ¬p
for i = 1:n_p
    PicoSAT.add(pico, v_p + i) # ¬(p@i)
end
PicoSAT.add(pico, 0)             # end clause
```

Code

Only one version of a package

```
exclusive &&
for i = 1:n_p-1, j = i+1:n_p
    PicoSAT.add(pico, -(v_p + i)) # ¬(p@i)
    PicoSAT.add(pico, -(v_p + j)) # ¬(p@j)
    PicoSAT.add(pico, 0)          # end clause
end
```

Code

Versions imply dependencies

```
for i = 1:n_p
    for (j, q) in enumerate(info_p.depends)
        info_p.conflicts[i, j] || continue # deps & conflicts
        PicoSAT.add(pico, -(v_p + i)) # ¬(p@i)
        PicoSAT.add(pico, vars[q])    # q
        PicoSAT.add(pico, 0)          # end clause
    end
end
```

Code

Conflicts are exclusive

```
for i = 1:n_p
    for q in names; v_q = vars[q]
        for j = 1:length(info[q].versions)
            info_p.conflicts[i, b+j] || continue
            PicoSAT.add(pico, -(v_p + i)) # ¬(p@i)
            PicoSAT.add(pico, -(v_q + j)) # ¬(p@j)
            PicoSAT.add(pico, 0)          # end clause
        end
    end
end
```

Preprocessing

SAT is sensitive to problem size

- Pays off to generate as small a problem as you can

Two strategies to reduce the size of problems:

1. Only include “reachable” packages and versions
2. Eliminate redundant versions that can’t be picked

Reachability

Concept:

- Start with best versions of all required packages
- Only consider next version if better version has potential conflict
- Also consider dependencies of any reachable versions

Reachability

Recursive definition

condition	implies
p in requirements	p_1 reachable
p_i reachable and depends on q	q_1 reachable
p_i reachable with reachable conflict	p_{i+1} reachable
p_i reachable and depends on q and q_n reachable with reachable conflict	p_{i+1} reachable

Redundancy

Suppose v and w are versions of the same package

- Preference: $v > w$ (v is preferrable to w)
- Dependencies: $D(v) \subseteq D(w)$
- Conflicts: $C_v \subseteq C_w$

There can never be a reason to choose w over v

- We can delete w

Constructing a SAT instance

The process:

- Parse registry data for explicitly required packages
- Parse registry data for recursive dependencies of all versions
- Reachability analysis to find candidate package versions
- Redundancy analysis to eliminate unchoosable versions
- Construct SAT problem for remaining package versions

Finding Optimal Solutions

SAT solvers can find some solution (or tell us there is none)

- How do we find an optimal solution?

Finding Optimal Solutions

PicoSAT allows pushing and popping sets of clauses

- Given a solution, add clause to improve some package version
- When no longer improvable, last package version is optimal
- Pop last clause, leaves optimial solution locked in
- Repeat for each package

Finding Multiple Solutions

There can be multiple Pareto-optimal solutions

- When an optimal solution is found
- Add clause forcing future solutions not to be dominated by it
  - In other words, must be strictly better w.r.t. some package
- Stop when no longer satisfiable
  - Or you have enough solutions

Changing Version Preferences

This is completely straightforward now:

- The resolver doesn't care what order versions are ordered
- You just give it an ordering of versions for each package
  - Downgrade resolution — reverse normal version order
  - Download avoidance — sort pre-installed versions first
  - Version fixing — sort current version first
- This can be done on a per-package basis
  - *E.g.* Downgrade for some, normal for others

Changing Package Priority

Solutions are optimized one package at a time

- Sorting by popularity might be a good default
- User can modify this order however they want
  - Move certain packages to the front

Can’t really prioritize indirect dependencies over direct ones

- This is actually an inherent feature of the problem
- Subtle and took me a long time to understand

Improved Error Feedback?

How to give better feedback when resolution is impossible?

- Open problem — if you have ideas, let me know!

Possible approach:

- Don't try to explain why it's impossible
- Tell people how they could fix the situation
  - Requirements they could remove
  - Conflicts they could eliminate