Dependency Resolution Predictability

Qianxi Chen

2024.04.02

PART 01

Dependency Resolution

What is it?

What is a dependency?

Some libraries/packages you used when building your software

Direct Dependency: explicitly required dependency by developer

Transitive Dependency: dependencies required by other dependencies

Example

A package could depend on multiple packages with specific version range

1. require blas * (any version)
2. require libcxx >=14.0.6 (any version larger than 14.0.6)
3. require libgfortran 5.* (any version with major version of 5)
4. require openssl >=1.1.1t,<1.1.2a (any version between 1.1.1 and 1.1.2a)
5. require foo ==1.0.0 (only 1.0.0)

```
└─ conda-tree depends -t numpy
numpy==1.23.5
  ├─ blas 1.0 [required: *, openblas]
  ├─ libcxx 14.0.6 [required: >=14.0.6]
  ├─ libopenblas 0.3.21 [required: >=0.3.20,<1.0a0]
  │  ├─ libcxx 14.0.6 [required: >=12.0.0]
  │  ├─ libgfortran 5.0.0 [required: 5.*]
  │  │  └─ libgfortran5 11.3.0 [required: any]
  │  │     └─ llvm-openmp 14.0.6 [required: >=8.0.0]
  │  └─ libgfortran5 11.3.0 [required: >=11.2.0]
  │     └─ dependencies of libgfortran5 displayed above
  ├─ numpy-base 1.23.5 [required: 1.23.5, py310haf87e8b_0]
  │  ├─ blas 1.0 [required: *, openblas]
  │  ├─ libcxx 14.0.6 [required: >=14.0.6]
  │  ├─ libopenblas 0.3.21 [required: >=0.3.20,<1.0a0]
  │  │  └─ dependencies of libopenblas displayed above
  │  └─ python 3.10.9 [required: >=3.10,<3.11.0a0]
  │     ├─ bzip2 1.0.8 [required: >=1.0.8,<2.0a0]
  │     ├─ libffi 3.4.2 [required: >=3.4,<4.0a0]
  │     ├─ ncurses 6.4 [required: >=6.4,<7.0a0]
  │     ├─ openssl 1.1.1w [required: >=1.1.1t,<1.1.2a]
  │     │  └─ ca-certificates 2024.2.2 [required: any]
  │     ├─ readline 8.2 [required: >=8.1.2,<9.0a0]
  │     │  └─ ncurses 6.4 [required: >=6.3,<7.0a0]
  │     ├─ sqlite 3.40.1 [required: >=3.40.1,<4.0a0]
  │     │  ├─ readline 8.2 [required: >=8.1.2,<9.0a0]
  │     │  │  └─ dependencies of readline displayed above
  │     │  └─ zlib 1.2.13 [required: >=1.2.13,<1.3.0a0]
  │     ├─ tk 8.6.12 [required: >=8.6.12,<8.7.0a0]
  │     │  └─ zlib 1.2.13 [required: >=1.2.12,<1.3.0a0]
  │     ├─ tzdata 2022g [required: any]
  │     ├─ xz 5.2.10 [required: >=5.2.10,<6.0a0]
  │     ├─ zlib 1.2.13 [required: >=1.2.13,<1.3.0a0]
  │     └─ pip 22.3.1 [required: any]
  │        ├─ python 3.10.9 [required: >=3.10,<3.11.0a0]
  │        │  └─ dependencies of python displayed above
  │        ├─ setuptools 65.6.3 [required: any]
  │        │  ├─ certifi 2024.2.2 [required: >=2016.9.26]
  │        │  │  └─ python 3.10.9 [required: >=3.7]
  │        │  │     └─ dependencies of python displayed above
  │        │  └─ python 3.10.9 [required: >=3.10,<3.11.0a0]
  │        │     └─ dependencies of python displayed above
  │        └─ wheel 0.38.4 [required: any]
  │           └─ python 3.10.9 [required: >=3.10,<3.11.0a0]
  │              └─ dependencies of python displayed above
  └─ python 3.10.9 [required: >=3.10,<3.11.0a0]
```

NYU

Dependency Resolution

"**project**" depends:

1. **bar** (any version)
2. **baz** (>=2.0)
3. **qux** (>=2.0)

3 Versions of bar available:

bar 3.0 depends:
1. **qux** (>=2.0)
2. **baz** (<2.0)

bar 2.0 depends:
1. **qux** (<2.0)

bar 1.0 depends:
1. **qux** (any)

Resolve the dependency:
1. Try with bar 3.0? Failed with baz
2. Try with bar 2.0? Failed with qux
3. Try with bar 1.0? Success!

Yes, different version of same package could depend on completely different packages

Dependency Resolution is the process of finding a set of packages that fulfill all the requirements

Dependency Conflict: Cannot find a solution that fulfill all the requirements (dependency resolution failed)

PART 02

Dependency Resolution

How Hard is it?

Four assumptions

1. A package can list zero or more packages or specific package versions as dependencies.
2. To install a package, all its dependencies must be installed.
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

Boolean satisfiability problem

Also called “SAT” problem

Assign **TRUE** or **FALSE** to these variables (i.e. **A**, **B**, **C**, **D**) to make the entire statement TRUE

(**A** **OR** **B**) **AND** ((**NOT** **A**) **OR** **C**) **AND** (**B** **OR** (**NOT** **C**)) **AND** ((**NOT** **C**) **OR** (**NOT** **D**))

Boolean satisfiability problem

(**A** OR **B**) AND ((NOT **A**) OR **C**) AND (**B** OR (NOT **C**)) AND ((NOT **C**) OR (NOT **D**))

There are $2^4 = 16$ possible combinations

As the number of variables goes up, possible combinations increases exponentially
# of combinations = 2^n

One possible answer: A=False, B=True, C=False, D=True

Boolean satisfiability problem

There is no existing algorithm that can solve it with worst time complexity less than $O(2^n)$

NP problem
Actually, NP-Hard

Which means if you find an efficient algorithm that has a polynomial time complexity, you proved NP=P

The “VERSION” Problem

Let’s call our dependency resolution issue the “VERSION” problem

Someone may already noticed that the SAT problem is pretty similar to the “VERSION” problem

Yes, they are actually reducible (or convertible)

The “VERSION” Problem

“**project**” depends:

1. **bar** (any version)
2. **baz** (>=2.0)
3. **qux** (>=2.0)

3 Versions of bar available:

bar 3.0 depends:
1. **qux** (>=2.0)
2. **baz** (<2.0)

bar 2.0 depends:
1. **qux** (<2.0)

bar 1.0 depends:
1. **qux** (any)

VERSION(project)=
(**bar_3.0** -> ((**qux_2.0** OR **qux_2.1** OR **etc**) AND (**baz_1.9** OR **baz_1.8** OR **etc**))
AND (**bar_2.0** -> (**qux_1.9** OR **qux_2.8** OR **etc**))
AND (**bar_1.0** -> (**qux_2.1** OR **qux_1.9** OR **etc**))

(**A** -> **B**) = (**NOT A OR B**)

The “VERSION” Problem

Conclusion: The “VERSION” problem, or the dependency resolution problem, is also a NP-hard problem, which means they are very hard to be solved efficiently

PART 03

Dependency Resolution

How to solve it efficiently?

Four assumptions

1. A package can list zero or more packages or specific package versions as dependencies.
2. To install a package, all its dependencies must be installed.
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

The hardness of “VERSION” problem is based on these four assumptions, what if they are not true?

Four assumptions

1. A package can list zero or more packages or specific package versions as dependencies. **(Yes, we need dependencies within the version range)**
2. To install a package, all its dependencies must be installed.
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

The hardness of “VERSION” problem is based on these four assumptions, what if they are not true?

Four assumptions

1. A package can list zero or more packages or specific package versions as dependencies. **(Yes, we need dependencies within the version range)**
2. To install a package, all its dependencies must be installed. **(Definitely Yes)**
3. Each version of a package can have different dependencies.
4. Two different versions of a package cannot be installed simultaneously.

The hardness of “VERSION” problem is based on these four assumptions, what if they are not true?

Four assumptions

1. A package can list zero or more packages or specific package versions as dependencies. **(Yes, we need dependencies within the version range)**
2. To install a package, all its dependencies must be installed. **(Definitely Yes)**
3. Each version of a package can have different dependencies. **(Of course)**
4. Two different versions of a package cannot be installed simultaneously.

The hardness of “VERSION” problem is based on these four assumptions, what if they are not true?

Rust Cargo

If multiple packages have a common dependency with semver-incompatible versions, then Cargo will allow this, but will build two separate copies of the dependency. For example:

```
# Package A
[dependencies]
rand = "0.7"

# Package B
[dependencies]
rand = "0.6"
```

The above will result in Package A using the greatest `0.7` release (`0.7.3` at the time of this writing) and Package B will use the greatest `0.6` release (`0.6.5` for example). This can lead to potential problems, see the Version-incompatibility hazards section for more details.

From “The Cargo Book”

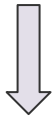
Version-incompatibility hazards

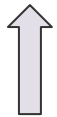
Module version 1.0

```rust
// in module version 1.0
pub struct Person {
    pub name: String,
    pub age: u32,
}

pub fn greet(person: &Person) {
    println!("Hello, {}!", person.name);
}
```

Module version 2.0

```rust
// in module version 2.0
pub struct Person {
    pub name: String,
    pub age: u32,
}

pub fn greet(person: &Person) {
    println!("Hello, {}!", person.name);
}
```





Error!

```rust
// get Person type from module version 1
let person: module::Person = somewhere...
```

```rust
// call great function from module version 2
module::greet(&person);
```

Version-incompatibility hazards

Module version 1.0

```rust
// in module version 1.0
pub struct Person {
    pub name: String,
    pub age: u32,
}

pub fn greet(person: &Person) {
    println!("Hello, {}!", person.name);
}
```

Module version 2.0

```rust
// in module version 2.0
pub struct Person {
    pub name: String,
    pub age: u32,
}

pub fn greet(person: &Person) {
    println!("Hello, {}!", person.name);
}
```

Although the Person type from both version has exactly same definition, they are still considered as different types!

Version-incompatibility hazards

Trick exists to “solve” this problem, partially: **The semver trick**

Basic idea: link the types so that they point to the same type

But not universal, i.e, the trick cannot entirely solve the problem

Backtracking Algorithm

A simple and straightforward algorithm to resolve dependencies

1. Try with the version
2. If not fulfill requirement, backtrack and try another one
3. If fulfill all requirements, return current result

```python
# A highly simplified backtracking algorithm pesudo-code
def backtracking(index):
    if !check(selected): # check if selected package fulfill the requirement
        return
    if index >= len(packages):
        print(f"find solution: {selected}")

    # try each version of the package, from newest to oldest
    for version in packages[index].versions:
        # append the version of the package
        selected.append(version)
        # append sub-dependencies of the package selected
        packages.append(packages[index].dependencies)
        # dfs
        backtracking(index + 1)
        # revert changes
        selected.pop()
        packages.pop()
```


Further heuristics algorithms could be applied to reduce the search tree

SAT Solver

SAT problem is so popular and important that there are a lot of people trying to make efficient algorithm to solve SAT problem

The boundary of $O(2^n)$ in **worst time complexity** is too hard to cross

But we could focus on: **Average time complexity** in solving SAT problems with **certain patterns**

SAT Solver: CNF form

We need a standard form of a boolean equation, so that it would be easier to process

conjunctive normal form (CNF):
Transform the boolean expression into a set of sub-expressions that **AND** together

(an expression only contains **OR** or **NOT**) **AND** (an expression only contains **OR** or **NOT**) **AND** ...

Example: (**A** **OR** **B**) **AND** (**NOT** **A** **OR** **C**)

Then our task is to make each sub-expressions **True**

It could be proved that every boolean expression could be converted into CNF form

SAT Solver: DPLL algorithm

Unit propagation

Consider the one of the sub-expression in CNF: (**A** **OR** **B**)

If **A** is assigned to be **False** (while trying), then **B** must be **True**

> So every time when we tried a variable with a value, we can look up each sub-expressions to see if there is any variable must be True/False, and assign directly.

SAT Solver: DPLL algorithm

Pure literal elimination

If a variable in each sub-expressions in CNF always has same polarity, then we assign the value directly.

(**A** **OR NOT** **B**) **AND** (**A** **OR** **B** **OR NOT** **C**) **AND** (**NOT** **B** **OR NOT** **C**)

We can notice that **A** always appears without **NOT** in front of it, so we can assign **True** to **A** directly
Similar to **C**, where we can assign **False** directly

SAT Solver

DPLL is introduced at 1961, which is a very old algorithm

Now SAT solvers algorithms have evolved a lot with a lot of amazing optimizations

1. Conflict-Driven
2. Real Occurrence Lists
3. Random Restart

There is even a SAT competition every year that people from all over the world submitting their SAT solvers to see who's algorithm is faster in solving the test cases

SAT Solver

We have already seen that the “VERSION” problem could be reduced to SAT problem, so we definitely could use SAT Solver to solve our dependency resolution problem

But how does that work exactly?

1. How to keep the installed packages with newest version possible?

PART 04

Dependency Resolution

Anaconda Deep Dive

Anaconda Dependency Resolution

Collect Information about the dependency we want to install (What package information we pass to SAT Solver?)

1. Packages already present in your environment
2. Past actions you have performed in that environment (Extra protection to packages installed by it)
3. aggressive updates list (packages that should be always up-to-date)
4. Packages pinned to specific version in pinned_packages in .condarc
5. Default_packages if in new environment
6. The packages user is asking for

Anaconda Dependency Resolution

We have the list of all packages with their dependencies version requirements, we can convert them to SAT boolean equations

We have already seen an example of converting “VERSION” problem to SAT problem

Besides dependency relationship, we also want each package is only installed exactly one version

We can add this constraints by adding more expressions to SAT

Only **exactly one** version of **bar** could be **True**:

(**NOT bar_0.3 OR NOT bar_0.2**) **AND** (**NOT bar_0.3 OR NOT bar_0.1**) **AND** (**NOT bar_0.2 OR NOT bar_0.1**)

Anaconda Dependency Resolution

But how to make the solution we found has the newest possible versions of all packages?

One way is to assign a weight to each package:
For each package, the newer the version is, the smaller its weight is. Then we add them together and try to minimize the value

Version Equation =
0*bar_3.0+**1***bar_2.0+**2***bar_1.0+**0***baz_2.1+**1***baz_2.0+...+**0***qux_2.5+**1***qux_2.4+...

We want to minimize the Version Equation while solving SAT

Pseudo-Boolean SAT

Such equation is called Pseudo-Boolean SAT:
$C_0 * p_0 + C_1 * p_1 + \ldots + C_{n-1} * p_{n-1} \geq C_n$

There is a systematic way to translate a pseudo-boolean equation into a set of SAT expressions

So we can solve Pseudo-Boolean SAT with SAT Solver

Anaconda Dependency Resolution

Version Equation =
0*bar_3.0+1*bar_2.0+2*bar_1.0+0*baz_2.1+1*baz_2.0+...+0*qux_2.5+1*qux_2.4+...

We want to minimize the Version Equation while solving SAT

Binary Decision Tree

Construct pseudo-boolean equation: 0 <= Version Equation <= 100
Then Solve using SAT Solver
Found any solution? Let's try 0 <= Version Equation <= 50
Found any solution? Let's try 0 <= Version Equation <= 25
No Solution Found? Let's try 25 <= Version Equation <= 50
.....

NYU

We can also make the number of the installed package smallest using similar method

Anaconda Dependency Resolution

Ok, everything looks really good so far, we use SAT Solver to be able to efficiently find a dependency resolution result with newest possible version and smallest number of package installed

But there is a minor issue, that could cause huge problem

Anaconda Dependency Resolution

I've made 9 packages as follows:

Foo 1.0: depends nothing
Foo 2.0: depends Bar<2.0, Baz<2.0

Bar 1.0: depends nothing
Bar 2.0: depends Foo<2.0, Baz<2.0

Baz 1.0: depends nothing
Baz: 2.0: depends Foo<2.0, Bar<2.0

Other two set of packages (Qux, a, b) and (c, d, e) has the same relationship

From our previous knowledge, each set has a 3 different resolution results that has least version equation value and smallest installed package. We have 3 sets, so there are 27 possible solutions treated same by anaconda

Which one will be returned determined by the SAT Solver

Anaconda Dependency Resolution

```
conda install --use-local foo bar baz qux a b c d e
```

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    a-1.0                      |                 0           5 KB  local
    b-2.0                      |                 0           5 KB  local
    bar-2.0                    |                 0           5 KB  local
    baz-1.0                    |                 0           5 KB  local
    c-2.0                      |                 0           5 KB  local
    d-1.0                      |                 0           5 KB  local
    e-1.0                      |                 0           5 KB  local
    foo-1.0                    |                 0           5 KB  local
    qux-1.0                    |                 0           5 KB  local
    ------------------------------------------------------------
```

Anaconda Dependency Resolution

Let’s create another package “dumb”

Dumb 1.0: depends on a<2.0 and foo<2.0

Remember, in our previous resolution result, a is already 1.0, and foo is also already 1.0

Anaconda Dependency Resolution

```
conda install --use-local foo bar baz qux a b c d e dumb
```

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    a-1.0                      |                0           5 KB  local
    b-1.0                      |                0           5 KB  local
    bar-2.0                    |                0           5 KB  local
    baz-1.0                    |                0           5 KB  local
    c-2.0                      |                0           5 KB  local
    d-1.0                      |                0           5 KB  local
    dumb-1.0                   |                0           5 KB  local
    e-1.0                      |                0           5 KB  local
    foo-1.0                    |                0           5 KB  local
    qux-2.0                    |                0           5 KB  local
    ------------------------------------------------------------
                                           Total:          54 KB
```

NYU

Anaconda Dependency Resolution

Without “dumb”

```
a-1.0
b-2.0
bar-2.0
baz-1.0
c-2.0
d-1.0
e-1.0
foo-1.0
qux-1.0
```

What’s different?
Package b version
Package qux version

With “dumb”

```
a-1.0
b-1.0
bar-2.0
baz-1.0
c-2.0
d-1.0
dumb-1.0
e-1.0
foo-1.0
qux-2.0
```

Anaconda Dependency Resolution

Is this expected?

We just add another package that has no conflict with current dependencies at all, and we get a completely different resolution result

Anaconda Dependency Resolution

Why would this happen?

It’s SAT Solver Magic

SAT Solver is so well optimized that there are just too many things happening inside the algorithm, and we can never predict what result would SAT Solver return

PicoSAT

PycoSAT is the SAT Solver used by Anaconda internally

PycoSAT is a python binding to PicoSAT (a C program)

What is inside PicoSAT?

It’s over 5000 lines of code (Remember our backtracking pseudo-code has less than 10 lines of code)

What’s even worse?

PicoSAT

Random Restart algorithm: Randomly select another search tree while searching

Which means they literally have a random number generator inside the algorithm

The result of PicoSAT is not predictable by nature

```
static unsigned
rng (void)
{
  unsigned res = srng;
  srng *= 1664525u;
  srng += 1013904223u;
  NOLOG (fprintf (out, "c rng () = %u\n", res));
  return res;
}

static unsigned
rrng (unsigned low, unsigned high)
{
  unsigned long long tmp;
  unsigned res, elements;
  assert (low <= high);
  elements = high - low + 1;
  tmp = rng ();
  tmp *= elements;
  tmp >>= 32;
  tmp += low;
  res = tmp;
  NOLOG (fprintf (out, "c rrng (%u, %u) = %u\n", low, high, res));
  assert (low <= res);
  assert (res <= high);
  return res;
}
```

PicoSAT

All of these optimizations inside the algorithm makes the SAT Solver becomes a chaos system:
Given a slightly different input, you will get a completely different output

Adding constraint: -1

```
└─ ./picosat uf250-02.cnf
s SATISFIABLE
v -1 -2 -3 4 5 6 7 -8 9 10 11 12 -13 14 -15 16 -17 -18 -19 -20 -21 22 23 24 25
v 26 -27 -28 -29 -30 -31 -32 33 -34 35 -36 -37 38 -39 40 -41 -42 43 44 45 -46
v -47 48 -49 50 51 -52 -53 -54 55 56 -57 -58 59 60 61 62 -63 64 -65 -66 -67 -68
v 69 70 -71 72 73 74 -75 76 -77 -78 79 -80 81 82 -83 84 -85 86 87 -88 89 90 -91
v 92 93 94 -95 96 -97 98 99 100 -101 102 -103 104 105 106 107 108 109 110 111
v 112 -113 -114 -115 -116 117 -118 119 -120 -121 122 -123 124 125 -126 -127
v -128 -129 -130 131 -132 133 -134 -135 136 137 -138 -139 140 -141 142 -143 144
v -145 -146 147 -148 -149 150 151 -152 -153 -154 155 156 -157 -158 159 -160 161
v 162 -163 -164 165 -166 -167 -168 -169 170 -171 -172 173 174 175 176 177 -178
v -179 -180 181 182 -183 184 -185 -186 -187 -188 189 190 -191 -192 193 -194
v -195 196 197 -198 -199 200 201 -202 -203 -204 -205 206 207 208 -209 -210 -211
v 212 -213 214 215 216 217 -218 -219 -220 221 -222 223 -224 -225 226 227 -228
v -229 -230 -231 -232 233 234 235 236 -237 238 -239 240 241 242 243 244 -245
v 246 247 -248 249 -250 0
```

```
└─ ./picosat uf250-02_2.cnf
s SATISFIABLE
v -1 -2 -3 4 5 6 7 -8 9 -10 11 12 -13 14 -15 16 -17 -18 19 -20 -21 22 23 24 25
v 26 -27 -28 -29 -30 -31 -32 33 -34 35 -36 37 38 -39 40 -41 -42 43 44 -45 -46
v -47 48 -49 50 51 -52 53 -54 55 56 -57 -58 59 60 -61 62 -63 64 -65 -66 -67 -68
v 69 70 -71 72 73 74 -75 76 -77 -78 79 -80 -81 82 -83 -84 -85 86 87 -88 -89 90
v -91 92 93 94 -95 96 -97 98 99 100 101 102 -103 104 105 106 107 108 109 110
v 111 112 -113 -114 -115 -116 117 118 119 -120 -121 122 -123 124 125 -126 -127
v -128 -129 -130 131 -132 133 134 -135 136 137 -138 -139 140 -141 142 -143 144
v -145 -146 147 148 -149 150 151 -152 -153 154 -155 156 -157 -158 159 -160 161
v 162 -163 164 165 -166 -167 168 -169 170 -171 -172 173 174 175 176 177 -178
v 179 -180 181 182 -183 184 -185 -186 -187 -188 189 190 -191 -192 193 -194 -195
v 196 197 -198 199 200 201 -202 -203 204 -205 206 207 208 -209 -210 -211 212
v -213 214 215 216 217 218 219 -220 221 -222 223 -224 -225 226 227 -228 -229
v -230 -231 232 233 234 235 -236 -237 238 -239 240 241 242 243 244 -245 246 247
v -248 249 -250 0
```

Why this could cause problem

We have already seen that given a slightly different package, SAT Solver based dependency resolution could give a different resolution result

How this could affect real life project?

Why this could cause problem

We usually have additional dependencies in different environments, like testing packages in testing environment, or develop packages in development packages.

Imagine you get a different dependency resolution result in testing environment and production environment due to minor dependency difference in these environments

Why this could cause problem

Different dependency resolution result means different version of some packages between these environment

Different version of packages means they may behave differently

Why this could cause problem

Placeholder: example of how different version could behave differently

We should use dependency resolution algorithm that's more predictable

Backtracking algorithm

Same experiment applied to backtracking algorithm

We use **pip** as example

Backtracking algorithm

```
pip install -i http://localhost:8080 foo bar baz qux a b c d e
```

```
└─ pip list
Package    Version
---------- -------
a          1.0
b          1.0
bar        1.0
baz        1.0
c          2.0
d          1.0
e          1.0
foo        2.0
pip        24.0
pipdeptree 2.16.2
qux        2.0
setuptools 65.5.0
```

```
└─ pipdeptree
c==2.0
├── d [required: <2.0, installed: 1.0]
└── e [required: <2.0, installed: 1.0]
foo==2.0
├── bar [required: <2.0, installed: 1.0]
└── baz [required: <2.0, installed: 1.0]
pipdeptree==2.16.2
└── pip [required: >=23.1.2, installed: 24.0]
qux==2.0
├── a [required: <2.0, installed: 1.0]
└── b [required: <2.0, installed: 1.0]
setuptools==65.5.0
```

Backtracking algorithm

```
└─ pip list
Package    Version
---------- -------
a          1.0
b          1.0
bar        1.0
baz        1.0
c          2.0
d          1.0
e          1.0
foo        2.0
pip        24.0
pipdeptree 2.16.2
qux        2.0
setuptools 65.5.0
```

Pip gives us a different result than anaconda, so we slightly modify requirements of dumb

Dumb 1.0: depends a<2.0, e<2.0

Backtracking algorithm

After uninstalling

```
pip install -i http://localhost:8080 foo bar baz qux a b c d e dumb
```

```
└─ pip list
Package    Version
---------- -------
a          1.0
b          1.0
bar        1.0
baz        1.0
c          2.0
d          1.0
dumb       1.0
e          1.0
foo        2.0
pip        24.0
pipdeptree 2.16.2
qux        2.0
setuptools 65.5.0
```

```
└─ pipdeptree
c==2.0
├── d [required: <2.0, installed: 1.0]
└── e [required: <2.0, installed: 1.0]
dumb==1.0
├── a [required: <2.0, installed: 1.0]
└── e [required: <2.0, installed: 1.0]
foo==2.0
├── bar [required: <2.0, installed: 1.0]
└── baz [required: <2.0, installed: 1.0]
pipdeptree==2.16.2
└── pip [required: >=23.1.2, installed: 24.0]
qux==2.0
├── a [required: <2.0, installed: 1.0]
└── b [required: <2.0, installed: 1.0]
setuptools==65.5.0
```

NYU

Backtracking algorithm

Exactly same result!

Backtracking algorithm is more predictable than SAT Solvers

Backtracking is good at making consistent result, making dependency resolution across different environment more safe