Forklift: Fitting Zygote Trees for Faster Package Initialization

WoSC 2024 : 10th International Workshop on Serverless Computing
Dec 2rd, 2024

Yuanzhuo Yang, KJ Choi, Keting Chen, Tyler Caraza-Harter
{yyang682, kchoi, kchen, tharter}@wisc.edu
University of Wisconsin, Madison

Index

1. Introduction

Reducing Serverless Startup Latency through Hierarchical Zygote Trees

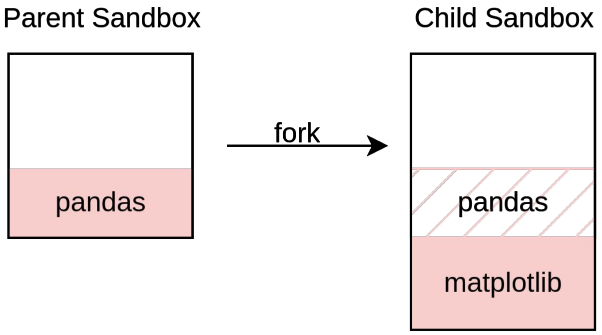
Introduction

Approaches towards reducing the startup latency:

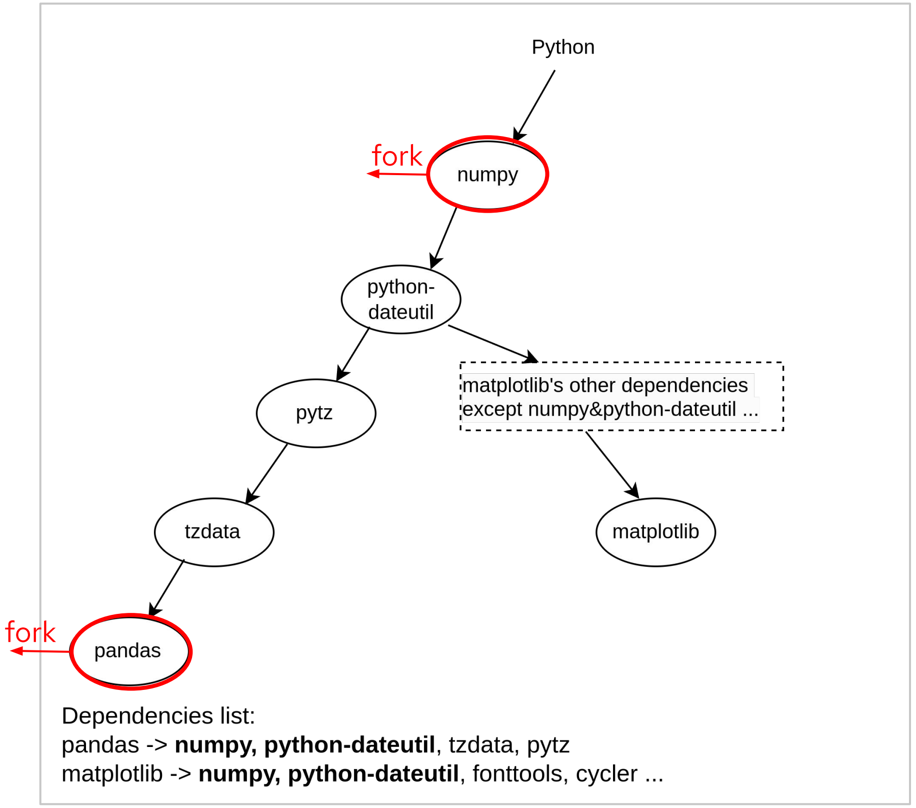
1. Lightweight sandboxes: containers, VM, unikernels

kata containers Firecracker

2. Initializing processes inside the sandboxes:
sock zygote initialization[1]



Go one step further:
organize zygotes in hierarchical tree structure
(known as Hierarchical Zygotes)



Dependencies list:
pandas -> **numpy, python-dateutil**, tzdata, pytz
matplotlib -> **numpy, python-dateutil**, fonttools, cycler ...

1. https://www.usenix.org/system/files/conference/atc18/atc18-oakes.pdf

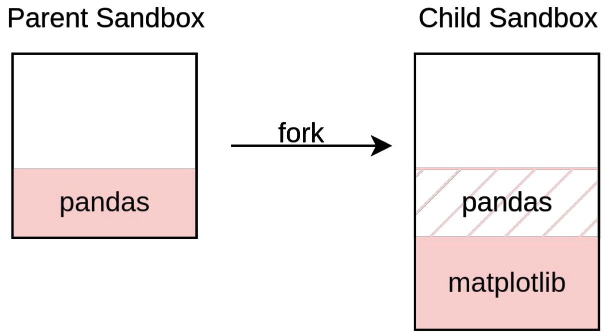
Introduction

Approaches towards reducing the startup latency

1. Lightweight sandboxes: containers, VM, unikernels

kata containers Firecracker

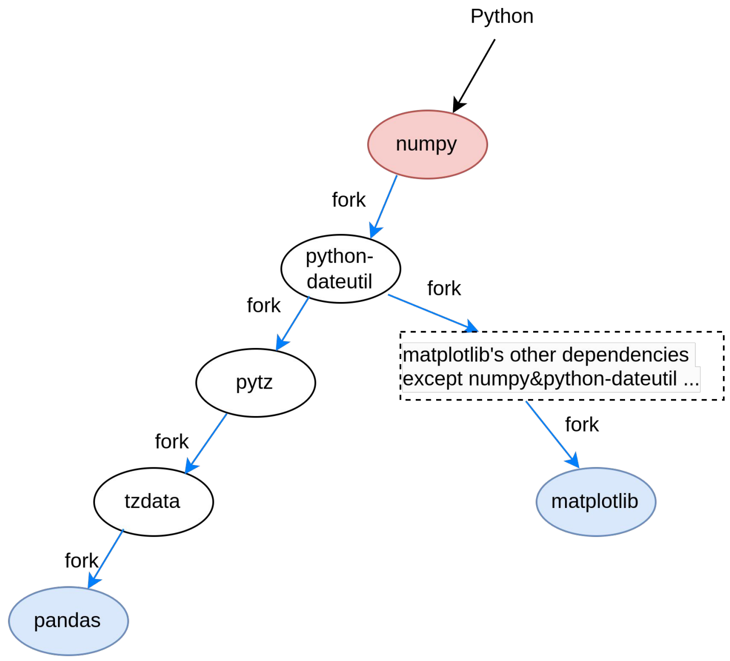
2. Initializing processes inside the sandboxes: sock zygote initialization[1]


Parent Sandbox
pandas
fork
Child Sandbox
pandas
matplotlib

1. https://www.usenix.org/system/files/conference/atc18/atc18-oakes.pdf

Go one step further:

organize zygotes in hierarchical tree structure
(known as Hierarchical Zygotes)


Python
numpy
fork
python-dateutil
fork
fork
pytz
matplotlib's other dependencies except numpy&python-dateutil ...
fork
fork
tzdata
matplotlib
fork
pandas

Dependencies list:
pandas -> **numpy, python-dateutil**, tzdata, pytz
matplotlib -> **numpy, python-dateutil**, fonttools, cycler ...

2. GitHub PyPI[2] Dependency Study

- Background
- Requirement Counts
- Popularity Distribution

3. Pronounced "pie pee eye".

Background

A requirements.txt example:

```
numpy
pandas
```

`pip install -r requirements.txt` installs *numpy, pandas* and their dependencies recursively.

We extracted 9,678 unique requirements.txt files from the BigQuery public dataset[3] and analyzed.

BigQuery

---

[3] https://console.cloud.google.com/marketplace/product/github/github-repos

Requirement Counts

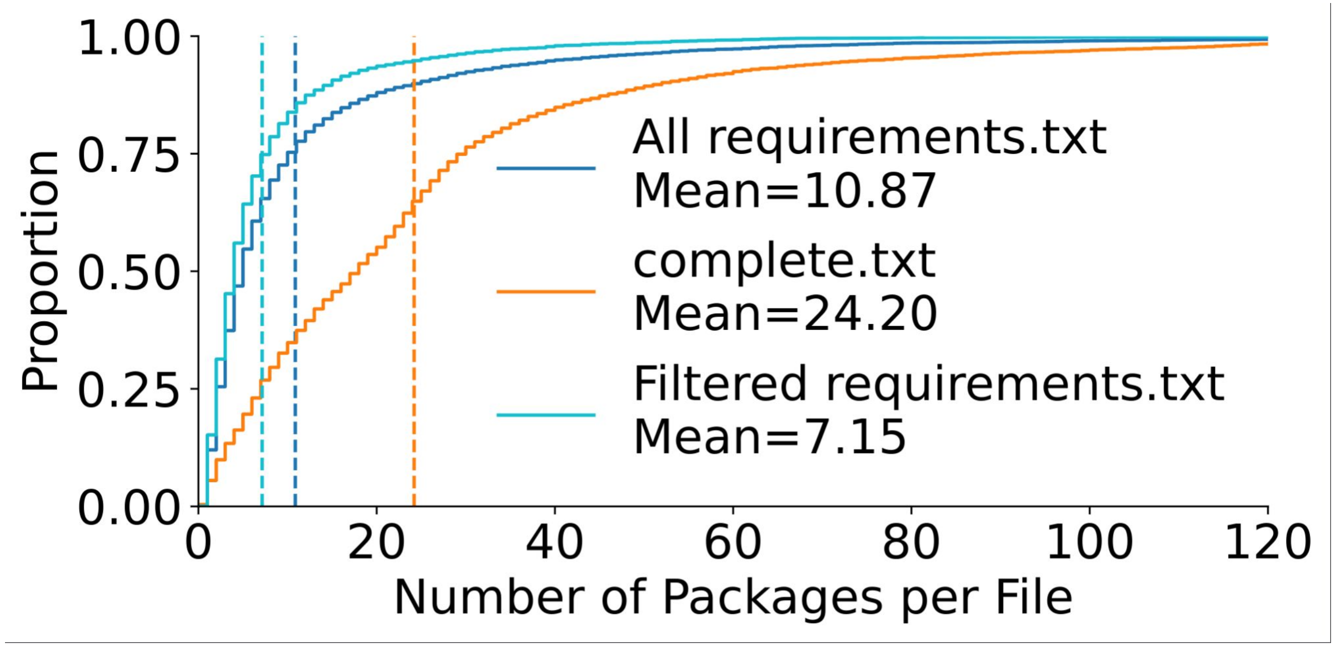
> Direct dependencies: packages that are explicitly listed in requirements.txt.

> Indirect dependencies: packages that are not directly required by the project but required by direct dependencies.

we try to pip-compile each requirements.txt to get complete.txt, which contains:

a) indirect dependencies

b) precise package versions



Background: pip-compile

requirements.txt example:

```
numpy
pandas
```

“pip-compile *requirements.txt* -o *complete.txt*”

complete.txt example:

```
numpy==2.1.3
        # via
        #   -r requirements.txt
        #   pandas
pandas==2.2.3
        # via -r requirements.txt
python-dateutil==2.9.0.post0
        # via pandas
pytz==2024.2
        # via pandas
six==1.16.0
        # via python-dateutil
tzdata==2024.2
        # via pandas
```

Background: pip-compile

requirements.txt example:

```
numpy
pandas
```

"pip-compile *requirements.txt* -o *complete.txt*"

complete.txt example:

indirect dependencies

```
numpy==2.1.3
    # via
    #   -r requirements.txt
    #   pandas
pandas==2.2.3
    # via -r requirements.txt
python-dateutil==2.9.0.post0
    # via pandas
pytz==2024.2
    # via pandas
six==1.16.0
    # via python-dateutil
tzdata==2024.2
    # via pandas
```

pandas depend on *six*

requirements.txt contains only *direct* dependencies,
complete.txt contains *direct+indirect* dependencies

Background: pip-compile

requirements.txt example:

```
numpy
pandas
```

“pip-compile *requirements.txt* -o *complete.txt*”

complete.txt example:

precise package versions

```
numpy==2.1.3
    # via
    #   -r requirements.txt
    #   pandas
pandas==2.2.3
    # via -r requirements.txt
python-dateutil==2.9.0.post0
    # via pandas
pytz==2024.2
    # via pandas
six==1.16.0
    # via python-dateutil
tzdata==2024.2
    # via pandas
```

Requirement Counts

> Direct dependencies: packages that are explicitly listed in requirements.txt.

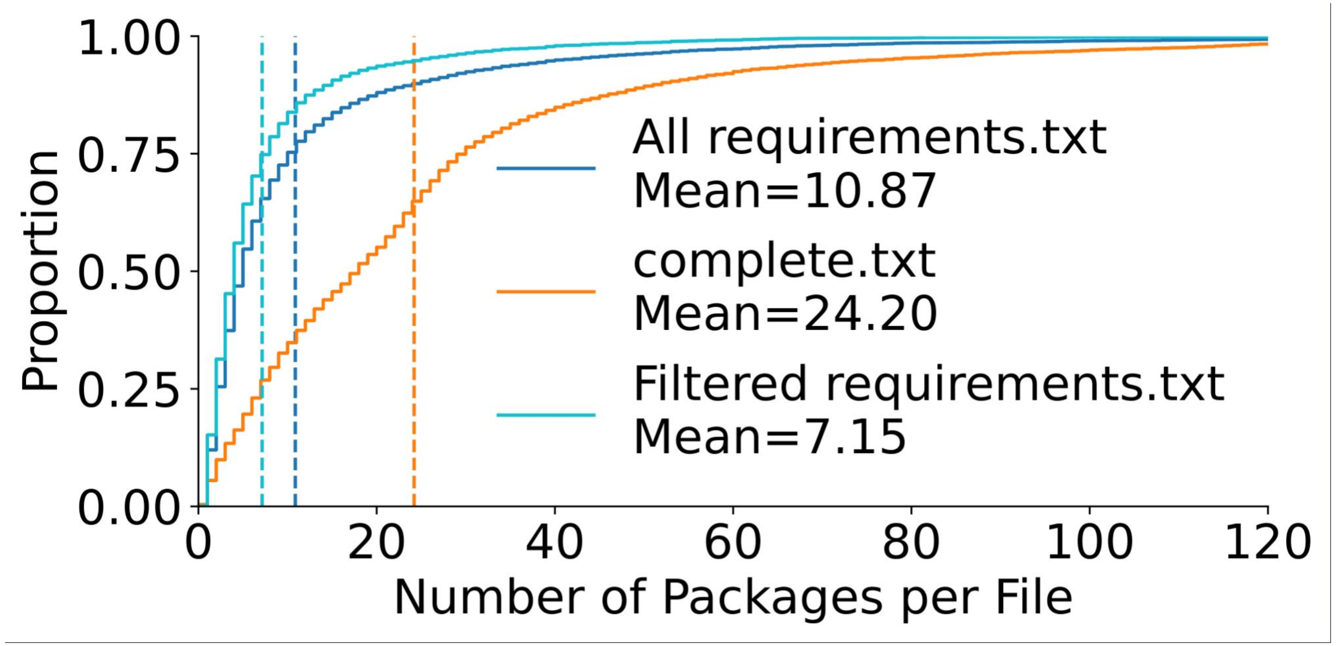
> Indirect dependencies: packages that are not directly required by the project but required by direct dependencies.

we try to pip-compile each requirements.txt to get complete.txt, which contains:

a) indirect dependencies

b) precise package versions

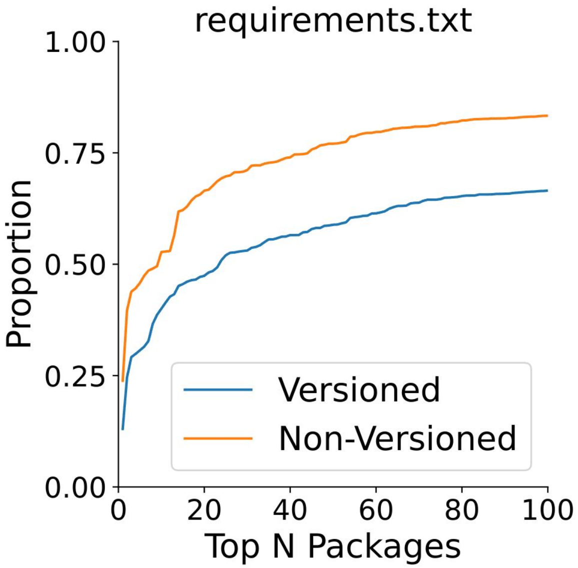
Implication: Most package requirements are indirect, package initialization may be costlier than expected

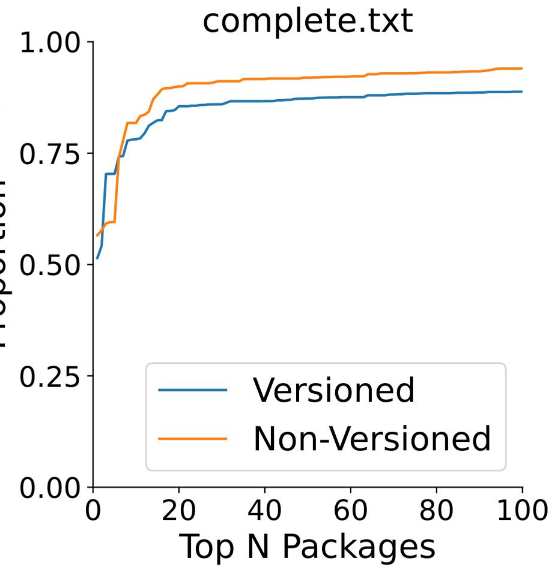


Note: 1) requirements.txt contains only *direct* dependencies,
complete.txt contains *direct+indirect* dependencies
2) filtered requirements.txt: the files on which pip-compile ran successfully

Popularity Distribution

We count how many requirements.txt/complete.txt files specify at least one of *Top N* (with or without version) packages.





Implications: package usage is highly skewed,

relatively few zygotes could provide substantial benefit on cold startup.

3. Forklift Zygote Trees

- Forklift: Zygote Trees Construction Algorithm
    - Basic idea
    - Example
    - Optimizations
- Deploy the Zygote Tree in OpenLambda

Forklift: Basic Idea

Construct a tree based on historical call data.
Commonly used packages added to the tree first.
Adding nodes gradually until #nodes reaches the limit.

Restriction

Before a package can be imported in a node, all of its dependencies should be imported in the node's ancestors.

Define the Forklift Algorithm: Input/Output

input: a binary call matrix:

	A_1	B_1	B_2	C_1	D_1
$\boxed{fn1}$	$\boxed{1}$	$\boxed{1}$	0	$\boxed{1}$	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0

Define the Forklift Algorithm: Input/Output

input: a binary call matrix:

$$\begin{array}{c|ccccc} & A_1 & B_1 & B_2 & C_1 & D_1 \\ \hline \boxed{fn1} & \boxed{1} & \boxed{1} & 0 & \boxed{1} & 0 \\ fn2 & 1 & 0 & 1 & 1 & 1 \\ fn3 & 1 & 1 & 0 & 1 & 1 \\ fn4 & 0 & 1 & 0 & 1 & 0 \end{array}$$

dependencies:

$$D_1 \rightarrow A_1$$

$$\begin{aligned} C_1 &\rightarrow B_1 \\ or\ C_1 &\rightarrow B_2 \end{aligned}$$

Define the Forklift Algorithm: Input/Output

input: a binary call matrix:

$$
\begin{array}{c|ccccc}
 & A_1 & B_1 & B_2 & C_1 & D_1 \\
\hline
\boxed{fn1 \quad 1} & & \boxed{1} & 0 & \boxed{1} & 0 \\
fn2 & 1 & 0 & 1 & 1 & 1 \\
fn3 & 1 & 1 & 0 & 1 & 1 \\
fn4 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

dependencies:

$$D_1 \rightarrow A_1$$

$$
\begin{aligned}
C_1 &\rightarrow B_1 \\
or\ C_1 &\rightarrow B_2
\end{aligned}
$$

size limit: <=6

Define the Forklift Algorithm: Input/Output

input: a binary call matrix:

	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0

dependencies:

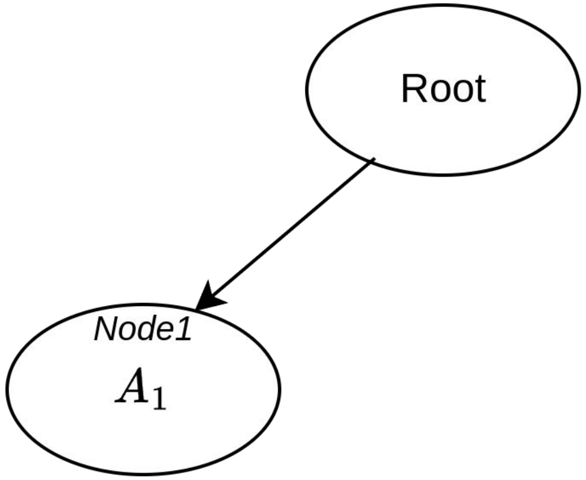
$$D_1 \rightarrow A_1$$

$$\begin{aligned} C_1 &\rightarrow B_1 \\ or\ C_1 &\rightarrow B_2 \end{aligned}$$

size limit: <=6

output: A hierarchical tree, the path from the root to a node represents the packages imported by this node.
Requests for these packages can be initiated from this node.

e.g.,



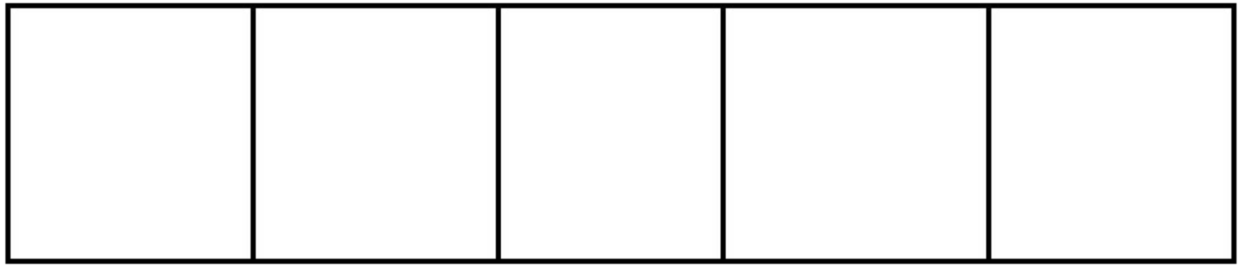
requests for fn_1, fn_2, fn_3 can be initialized frorm *Node1*

Each node only import one package for simplicity.

Forklift Example: bootstrap the construction

CandidateQ:

A priority Queue



$$utility(P) = \sum(column(P))$$

enqueue the highest utility packages at Root to candidateQ

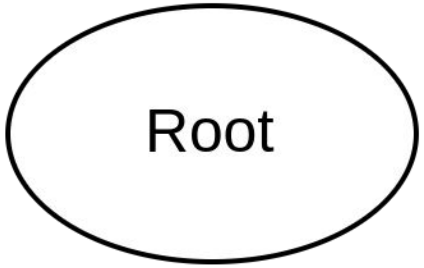
dependencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0
	3	3	1		



utilities at Root: $A_1 : 3, B_1 : 3, B_2 : 1; C_1, D_1$ pre-requisite not satisfied

Forklift Example: bootstrap the construction

CandidateQ:

A priority Queue

enqueue the highest utility package A_1 at Root to candidateQ

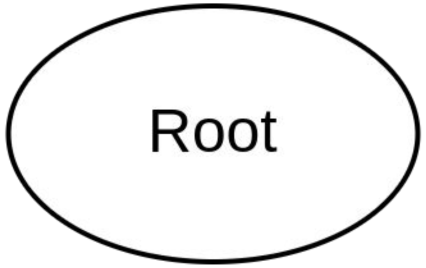
dependencies:

$D_1 \rightarrow A_1$

$C_1 \rightarrow B_1$

$or\ C_1 \rightarrow B_2$

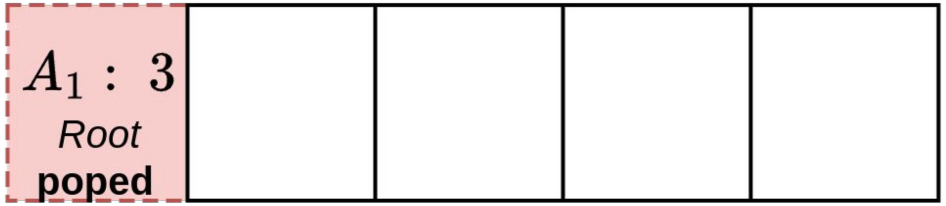
	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0



utilities at Root: $A_1 : 3, B_1 : 3, B_2 : 1$; C_1, D_1 pre-requisite not satisfied

step 1: add a child by popping the CandidateQ

CandidateQ:
A priority Queue



pop the A_1 at *Root*, then add A_1 to the child(*Node1*)

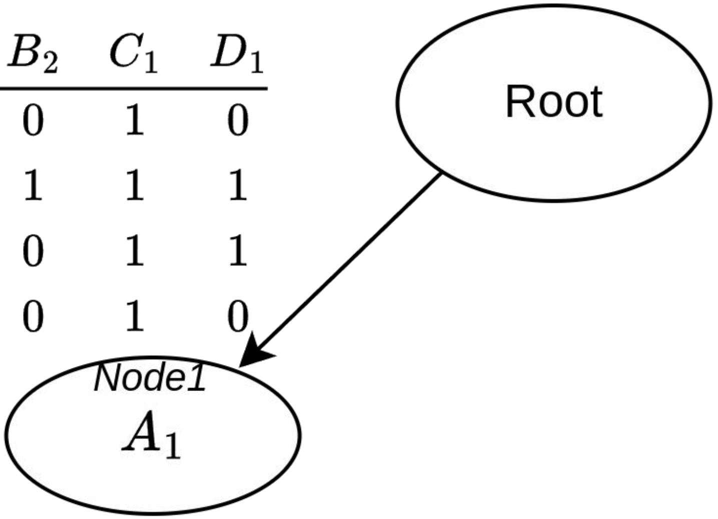
Dependencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

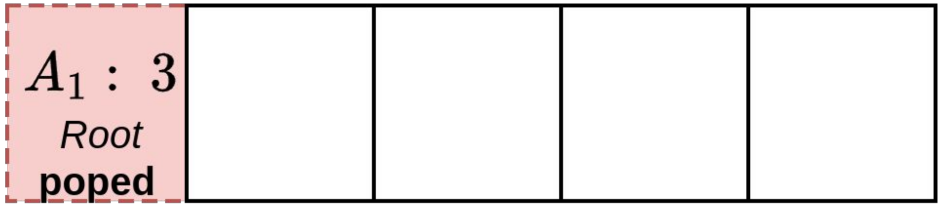
$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0



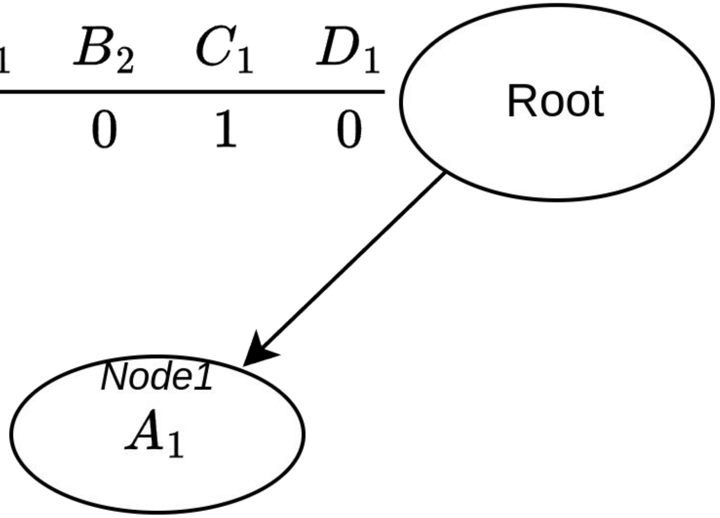
step 1: add a child by popping the CandidateQ

CandidateQ:
A priority Queue



pop the A_1 at *Root*, then add A_1 to the child(*Node1*)

	A_1	B_1	B_2	C_1	D_1
$fn4$	0	1	0	1	0



Dependencies:

$$D_1 \to A_1$$

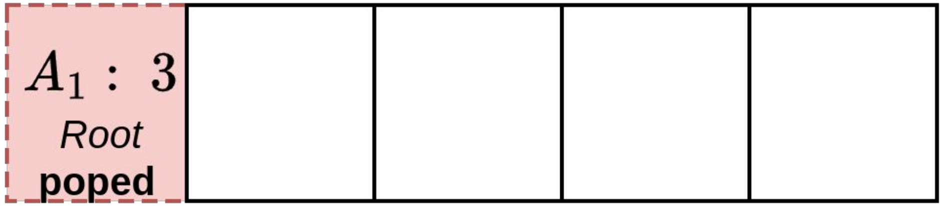
$$C_1 \to B_1$$

$$or\ C_1 \to B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1

step 1: add a child by popping the CandidateQ

CandidateQ:
A priority Queue



pop the A_1 at *Root*, then add A_1 to the child(*Node1*)

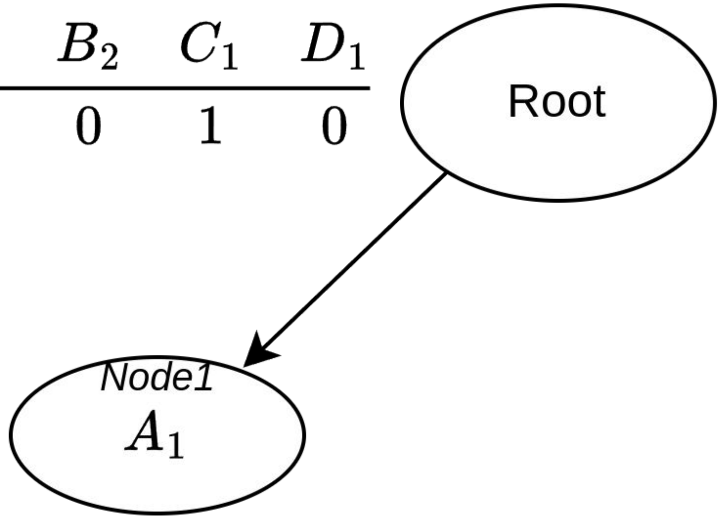
Dependencies:

$D_1 \to A_1$

$C_1 \to B_1$

$or\ C_1 \to B_2$

	A_1	B_1	B_2	C_1	D_1
$fn4$	0	1	0	1	0



	A_1	B_1	B_2	C_1	D_1
$fn1$	$\cancel{1} \to 0$	1	0	1	0
$fn2$	$\cancel{1} \to 0$	0	1	1	1
$fn3$	$\cancel{1} \to 0$	1	0	1	1

step 2: Enqueue for next branching

CandidateQ:
A priority Queue

$B_1 : 2$ *Node1*	$B_1 : 1$ *Root*			

enqueue the highest utility packages at Root and *Node1* seperately to candidateQ

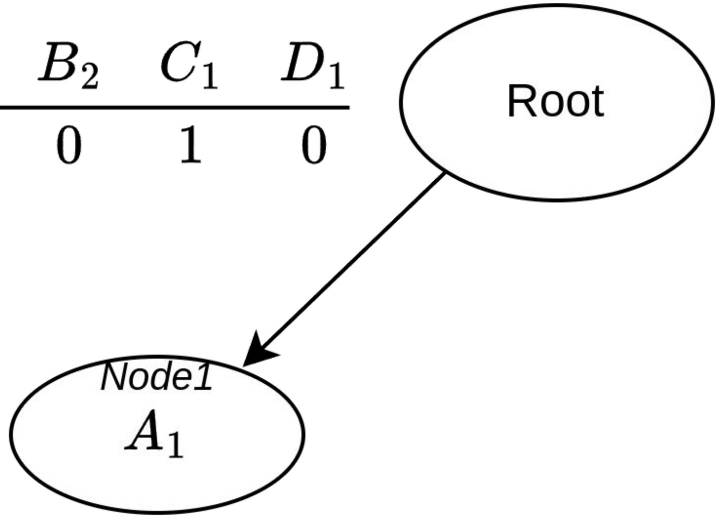
Dependencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn4$	0	1	0	1	0



	A_1	B_1	B_2	C_1	D_1
$fn1$	0	1	0	1	0
$fn2$	0	0	1	1	1
$fn3$	0	1	0	1	1

Do step 1*(add_child_node)*+2*(enqeue_top_child_candidate)* repeatedly ...

CandidateQ:
A priority Queue

$C_1 : 2$ *Node2* **poped**	$B_1 : 1$ *Root*	$B_2 : 1$ *Node1*	$D_1 : 1$ *Node3*	

pop C_1 at *Node2*, then enqueue the highest utility pkgs at *Node2* and *Node3* seperately

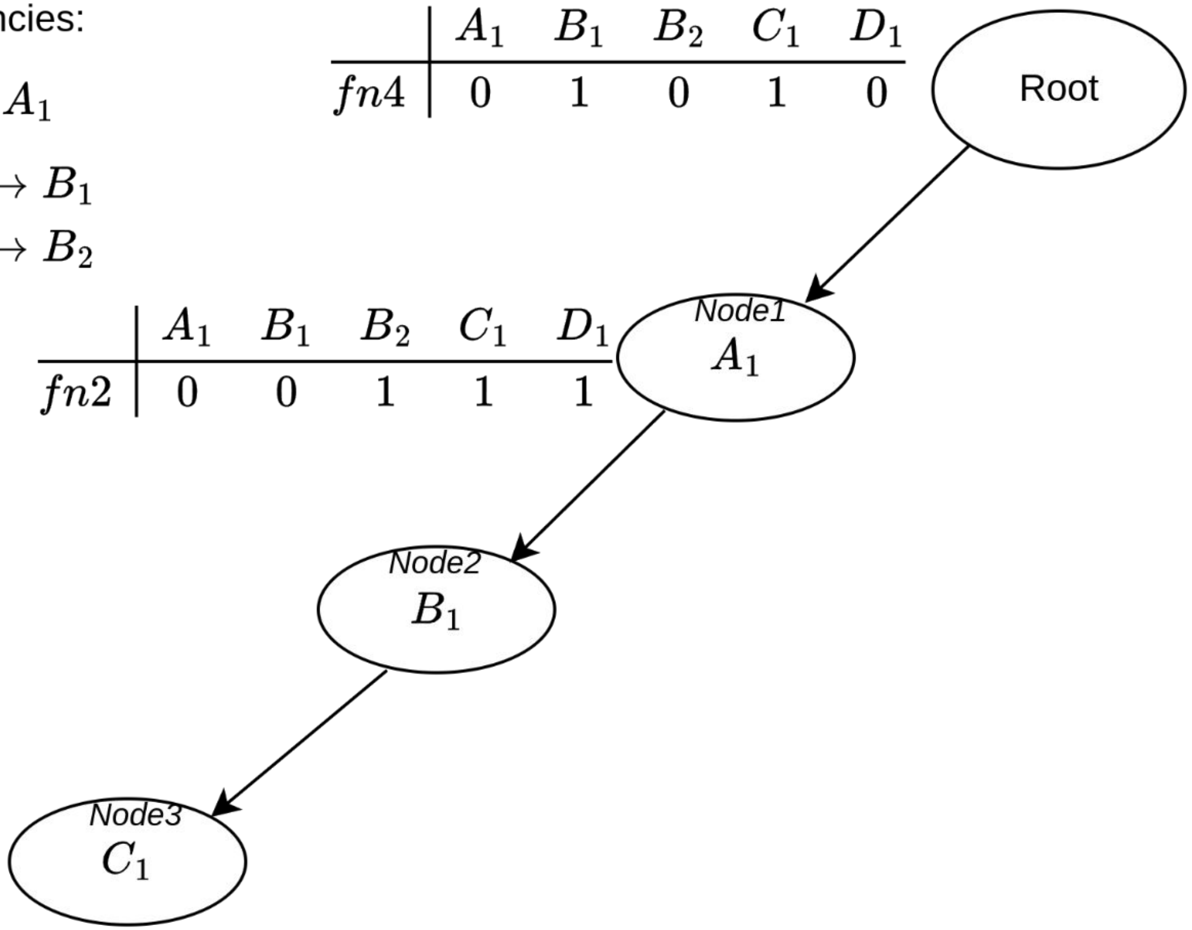
Dependencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn4$	0	1	0	1	0



	A_1	B_1	B_2	C_1	D_1
$fn2$	0	0	1	1	1

	A_1	B_1	B_2	C_1	D_1
$fn1$	0	0	0	$\cancel{1} \rightarrow 0$	0
$fn3$	0	0	0	$\cancel{1} \rightarrow 0$	1

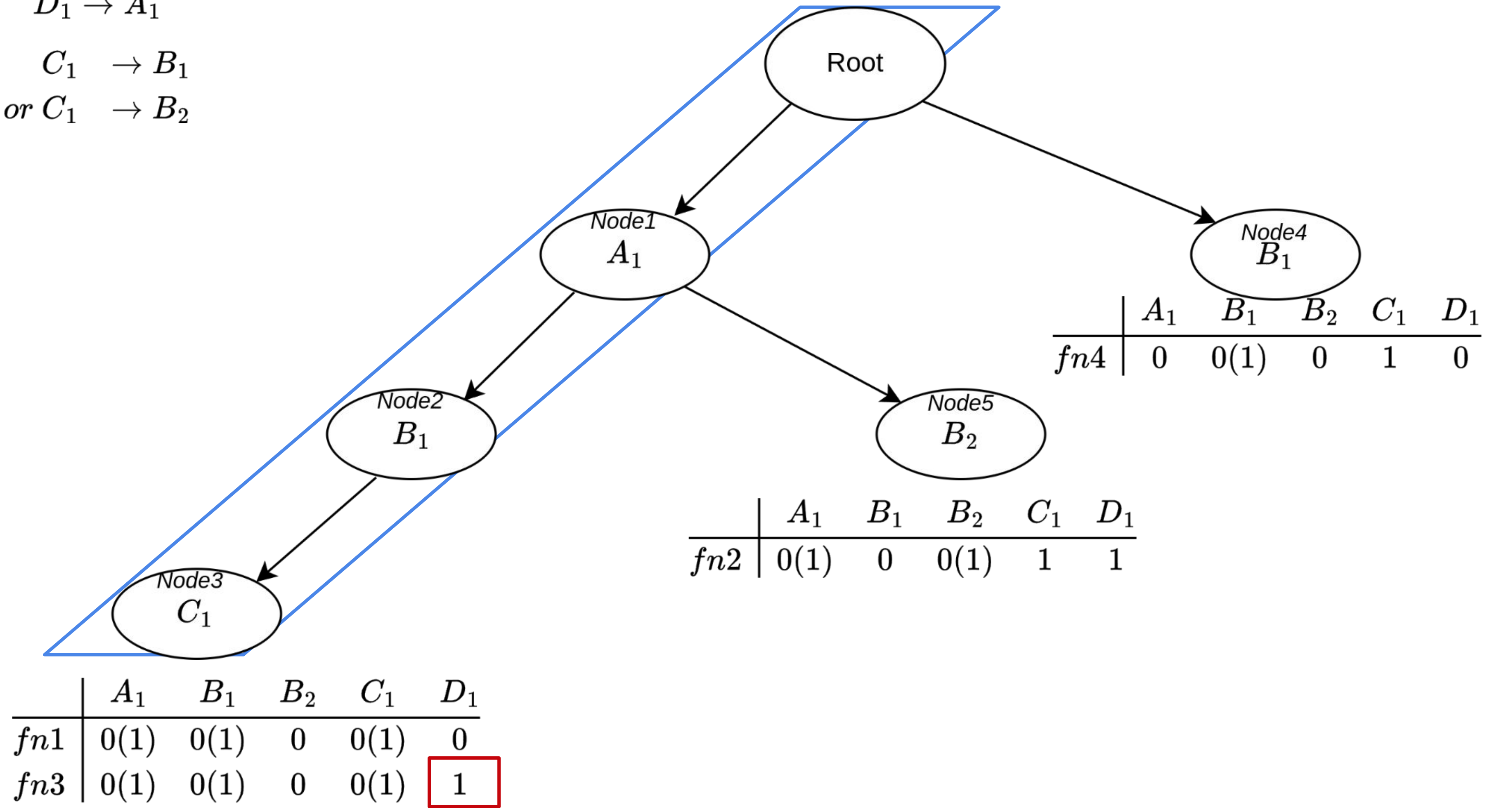
Eventually, the tree is ...

Depdencies:

$D_1 \rightarrow A_1$

$C_1 \quad \rightarrow B_1$

$or\ C_1 \quad \rightarrow B_2$



	A_1	B_1	B_2	C_1	D_1
$fn4$	0	0(1)	0	1	0

	A_1	B_1	B_2	C_1	D_1
$fn2$	0(1)	0	0(1)	1	1

	A_1	B_1	B_2	C_1	D_1
$fn1$	0(1)	0(1)	0	0(1)	0
$fn3$	0(1)	0(1)	0	0(1)	$\boxed{1}$

Optimizations

1. Replace 0/1 in the binary call matrix with weight values, e.g. import latency. → Time-based Weight

2. Python packages often have many dependencies
(e.g., pandas 2.2.3 requires 5 packages, Jupyter 1.0.0 requires 98 packages).
why not import **a package together with its dependencies** *(multiple packages)* in *one* node? → Multi-package (per node)

Multi-package Tree Example: bootstrap the construction

CandidateQ:

enqueue the highest utility packages at Root to candidateQ

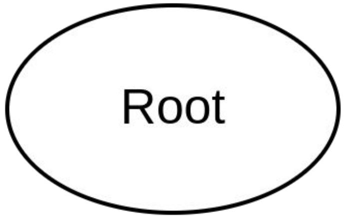
Depdencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

$$or\ C_1 \rightarrow B_2$$

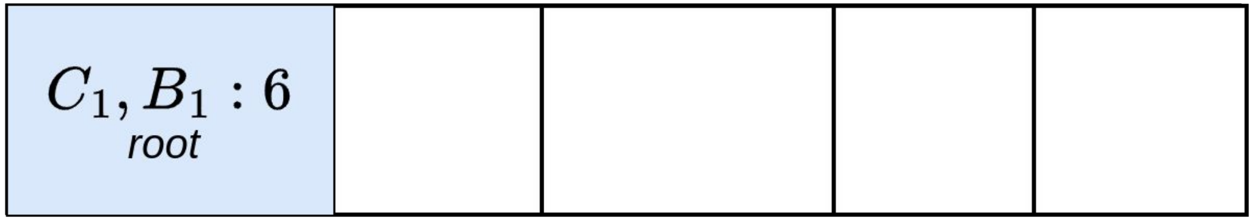
	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0



Root

$$\text{utility(package)} = \sum_{i \in \{\text{rows containing } \{\text{package+dependencies}\}\}} M[i, \{\text{package} + \text{dependencies}\}]$$

Multi-package Tree Example: bootstrap the construction



CandidateQ:

$C_1, B_1 : 6$
root

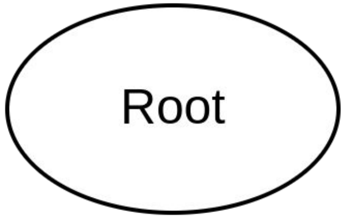
enqueue the highest utility packages at Root to candidateQ

Depdencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$
$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn1$	1	1	0	1	0
$fn2$	1	0	1	1	1
$fn3$	1	1	0	1	1
$fn4$	0	1	0	1	0



$$\text{utility}(\text{package}) = \sum_{i \in \{\text{rows containing } \{\text{package}+\text{dependencies}\}\}} M[i, \{\boxed{\text{package} + \text{dependencies}}\}]$$

$$\text{utility}(C_1) = \sum_{\boxed{i \in \{\text{rows containing } \{C_1+B_1\}\}}} M[i, \{\boxed{C_1 + B_1}\}]$$

$$\text{utility}(C_1) = \sum M[\boxed{\{fn1, fn3, fn4\}}, \{C_1 + B_1\}]$$

step 1: add a child by popping the CandidateQ

CandidateQ:

$C_1, B_1 : 6$ *root* **poped**				

pop the B_1, C_1 at *Root*, then add them to the child(*Node1*)

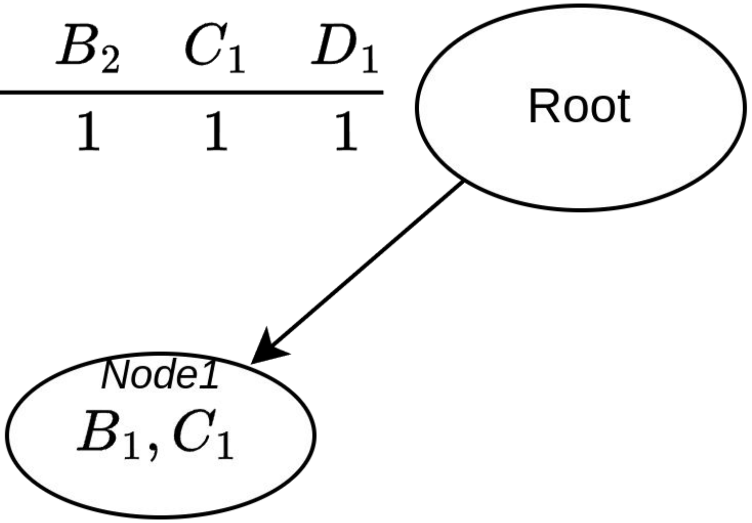
Depdencies:

$$D_1 \rightarrow A_1$$

$$C_1 \rightarrow B_1$$

$$or\ C_1 \rightarrow B_2$$

	A_1	B_1	B_2	C_1	D_1
$fn2$	1	0	1	1	1



	A_1	B_1	B_2	C_1	D_1
$fn1$	1	$\cancel{1} \rightarrow 0$	0	$\cancel{1} \rightarrow 0$	0
$fn3$	1	$\cancel{1} \rightarrow 0$	0	$\cancel{1} \rightarrow 0$	1
$fn4$	0	$\cancel{1} \rightarrow 0$	0	$\cancel{1} \rightarrow 0$	0

step 2: Enqueue for next branching

CandidateQ:

$C_1, B_2 : 2$ *root*	$A_1 : 2$ *Node1*			

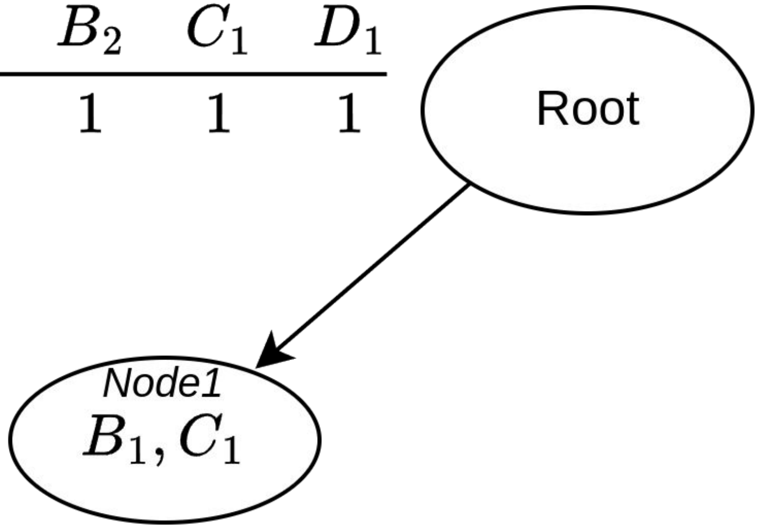
enqueue the highest utility packages at Root and *Node1* seperately to candidateQ

Depdencies:

$$D_1 \rightarrow A_1$$

$$\begin{aligned} C_1 &\rightarrow B_1 \\ or\ C_1 &\rightarrow B_2 \end{aligned}$$

	A_1	B_1	B_2	C_1	D_1
$fn2$	1	0	1	1	1



	A_1	B_1	B_2	C_1	D_1
$fn1$	1	0	0	0	0
$fn3$	1	0	0	0	1
$fn4$	0	0	0	0	0

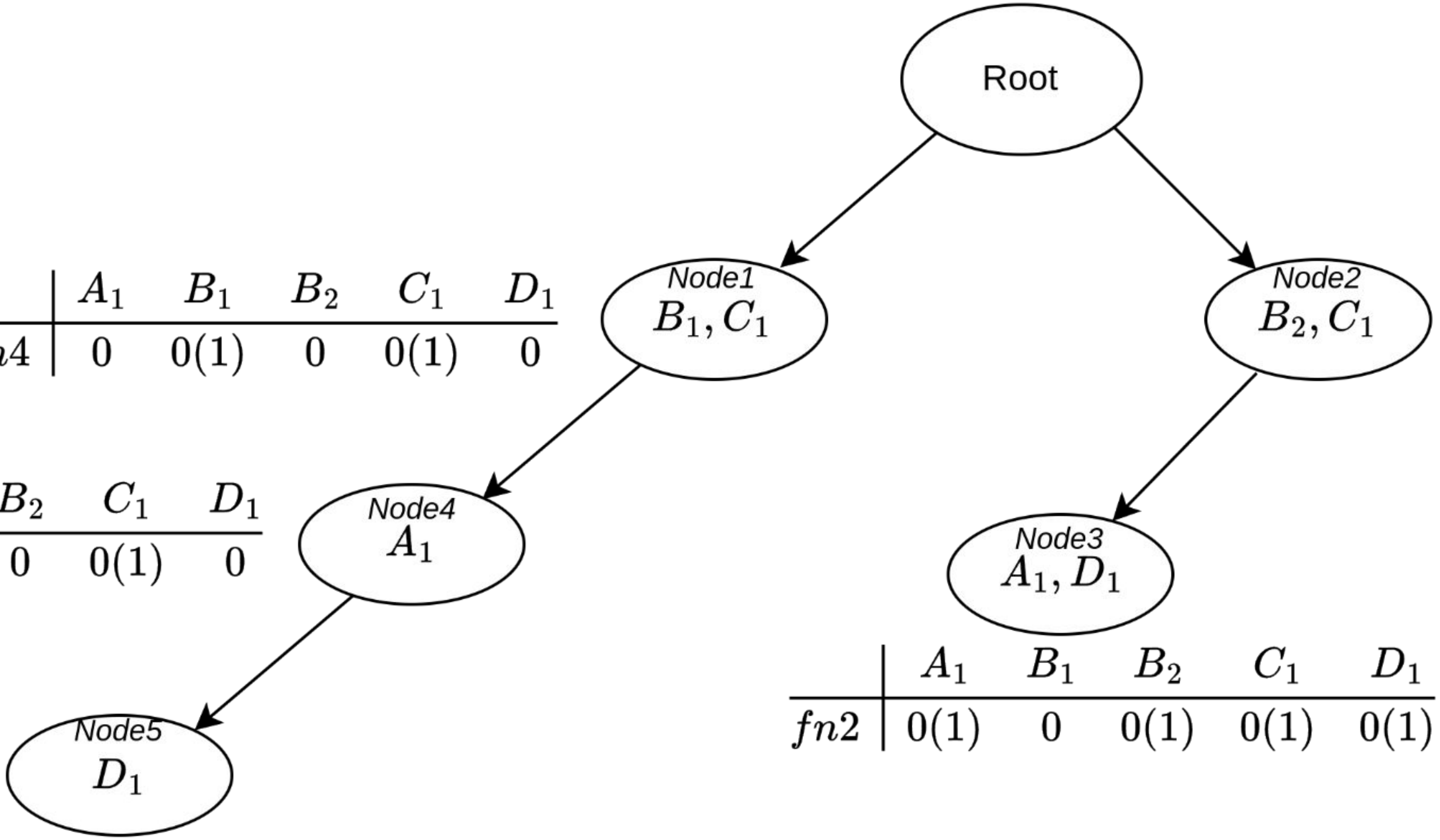
Eventually, the tree is …

Packages required by each function are satisfied.

Depdencies:

$$D_1 \rightarrow A_1$$

$$\begin{aligned} & C_1 && \rightarrow B_1 \\ or\ & C_1 && \rightarrow B_2 \end{aligned}$$



	A_1	B_1	B_2	C_1	D_1
$fn4$	0	$0(1)$	0	$0(1)$	0

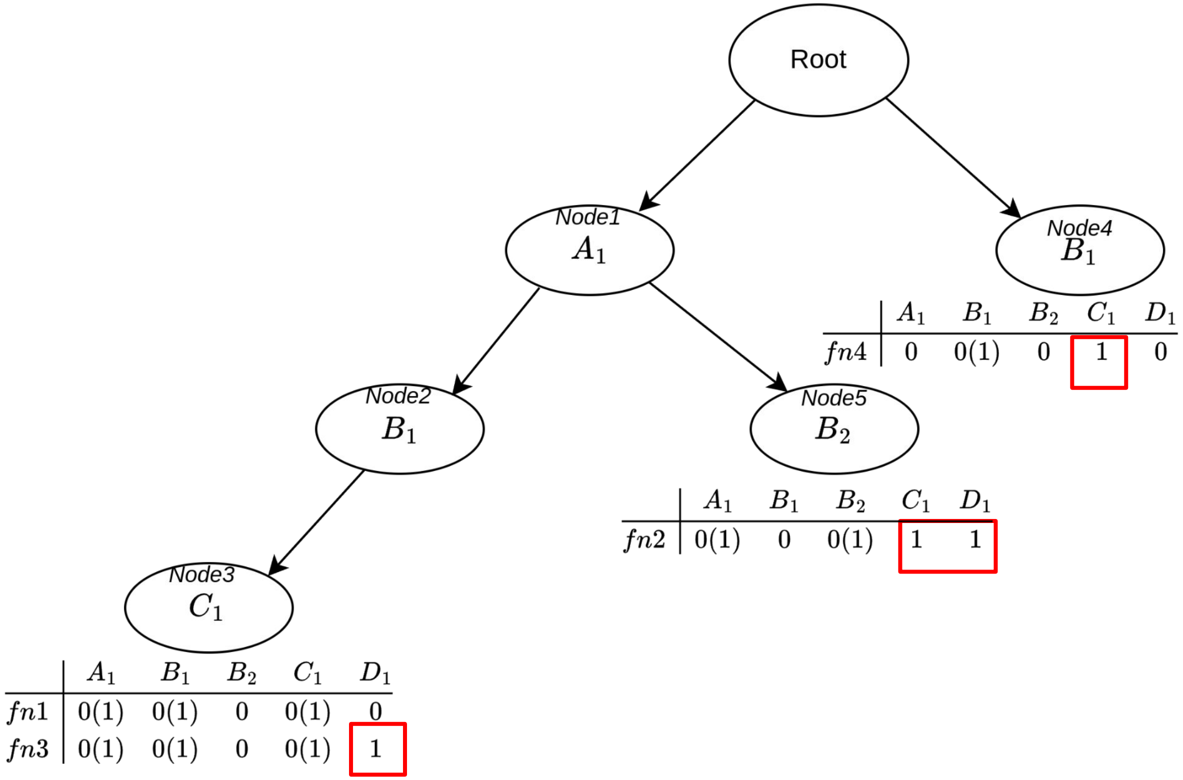
	A_1	B_1	B_2	C_1	D_1
$fn1$	$0(1)$	$0(1)$	0	$0(1)$	0

	A_1	B_1	B_2	C_1	D_1
$fn3$	$0(1)$	$0(1)$	0	$0(1)$	$0(1)$

	A_1	B_1	B_2	C_1	D_1
$fn2$	$0(1)$	0	$0(1)$	$0(1)$	$0(1)$

Comparison

single-package



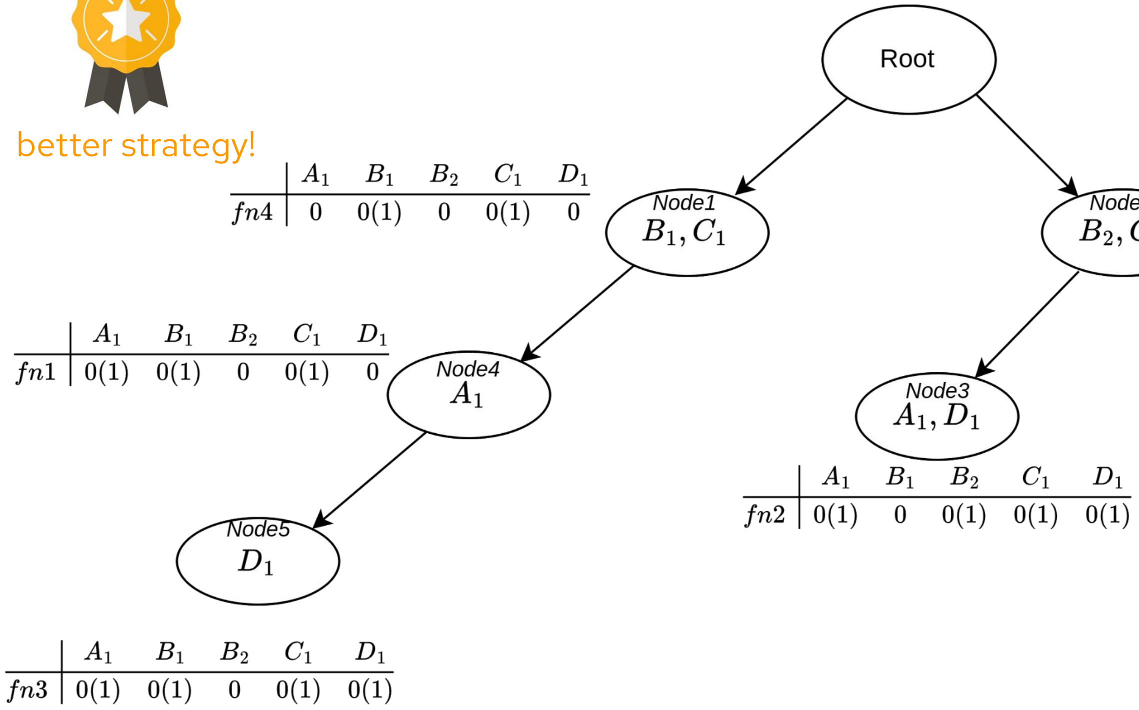
	A_1	B_1	B_2	C_1	D_1
$fn4$	0	0(1)	0	$\boxed{1}$	0

	A_1	B_1	B_2	C_1	D_1
$fn2$	0(1)	0	0(1)	$\boxed{1}$	$\boxed{1}$

	A_1	B_1	B_2	C_1	D_1
$fn1$	0(1)	0(1)	0	0(1)	0
$fn3$	0(1)	0(1)	0	0(1)	$\boxed{1}$

multi-package

better strategy!



	A_1	B_1	B_2	C_1	D_1
$fn4$	0	0(1)	0	0(1)	0

	A_1	B_1	B_2	C_1	D_1
$fn1$	0(1)	0(1)	0	0(1)	0

	A_1	B_1	B_2	C_1	D_1
$fn2$	0(1)	0	0(1)	0(1)	0(1)

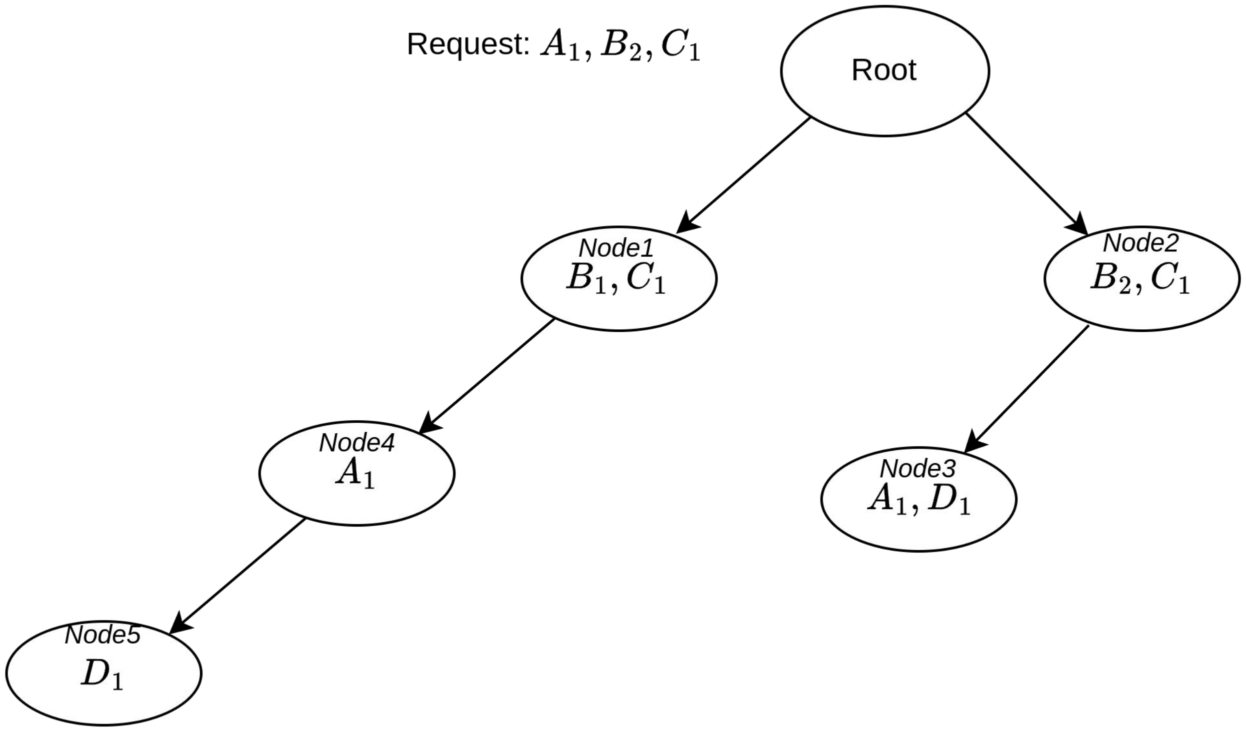
	A_1	B_1	B_2	C_1	D_1
$fn3$	0(1)	0(1)	0	0(1)	0(1)

Packages required by each function are satisfied.

Deployment in OpenLambda

We choose to deploy in OpenLambda as it is based on SOCK container that supports ***sandbox-level fork***.

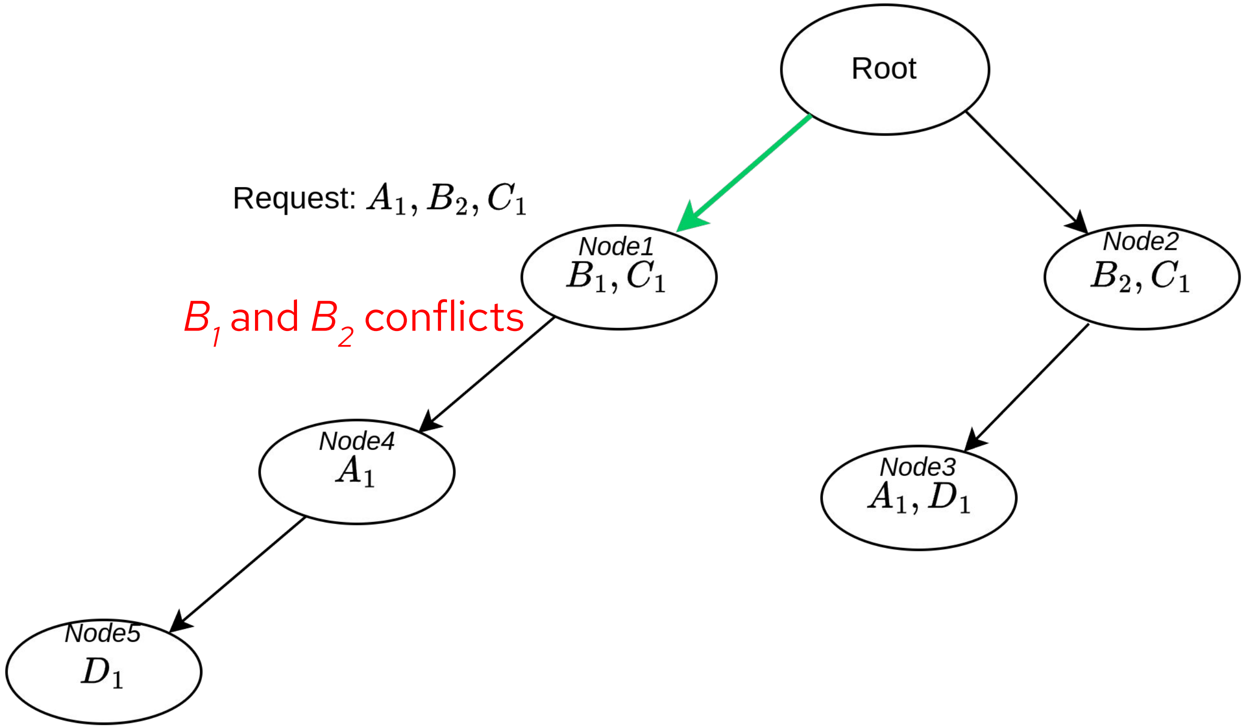
When a request arrives, do a **DFS** search in the tree and first non-root node is selected to serve the requests.



Deployment in OpenLambda

We choose to deploy in OpenLambda as it is based on SOCK container that supports ***sandbox-level fork***.

When a requests comes in, do a **DFS** search in the tree and first non-root node is selected to serve the requests.



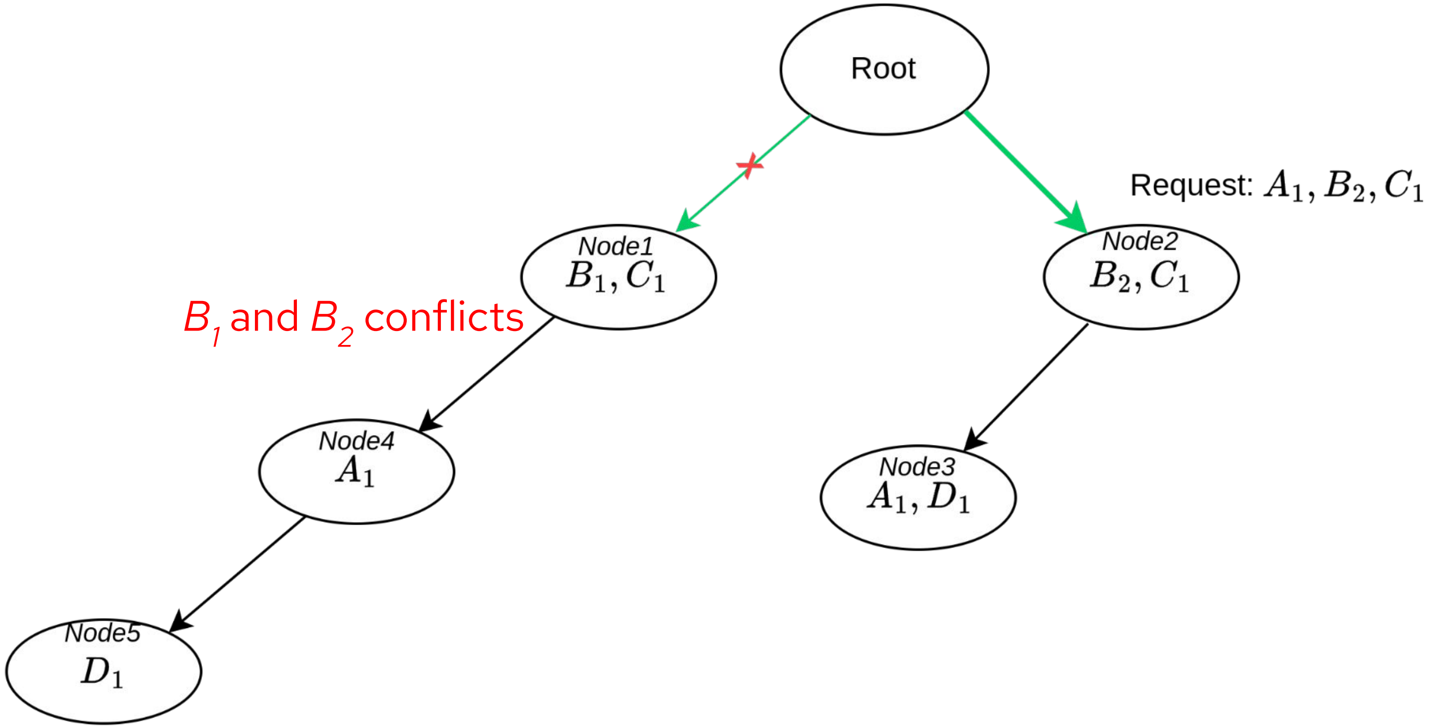
Request: A_1, B_2, C_1

B_1 and B_2 conflicts

Deployment in OpenLambda

We choose to deploy in OpenLambda as it is based on SOCK container that supports ***sandbox-level fork***.

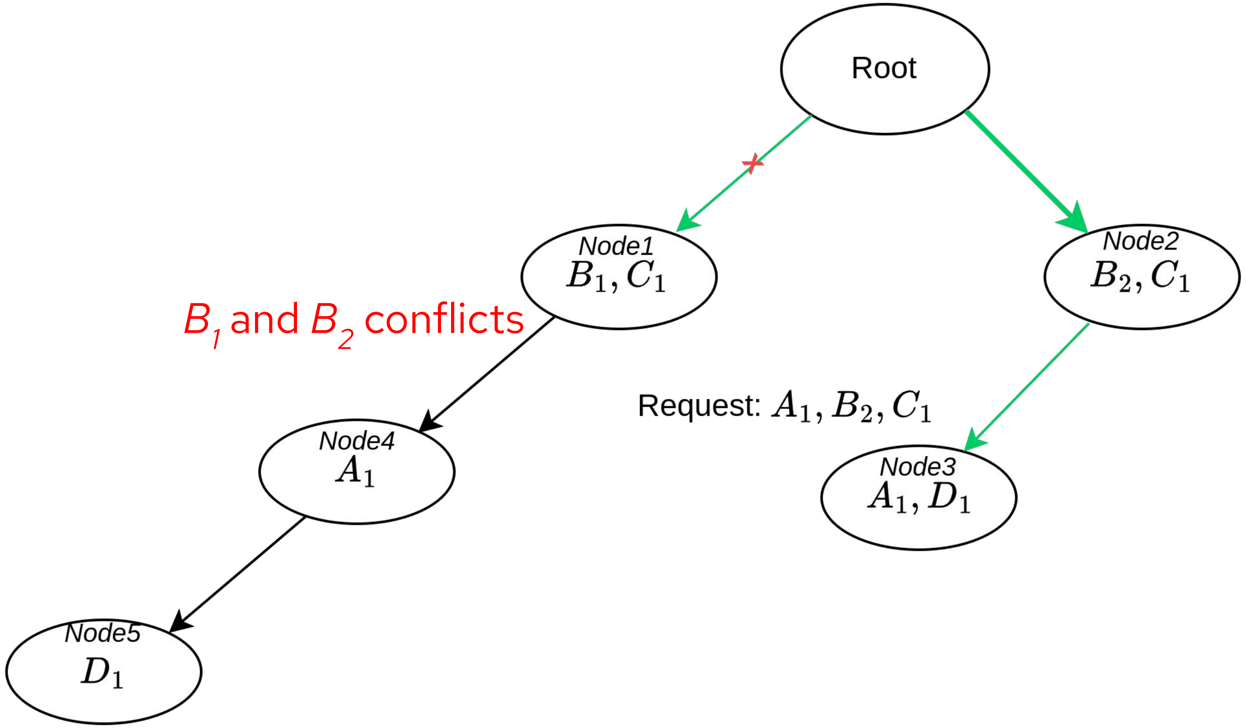
When a requests comes in, do a **DFS** search in the tree and first non-root node is selected to serve the requests.



Deployment in OpenLambda

We choose to deploy in OpenLambda as it is based on SOCK container that supports ***sandbox-level fork***.

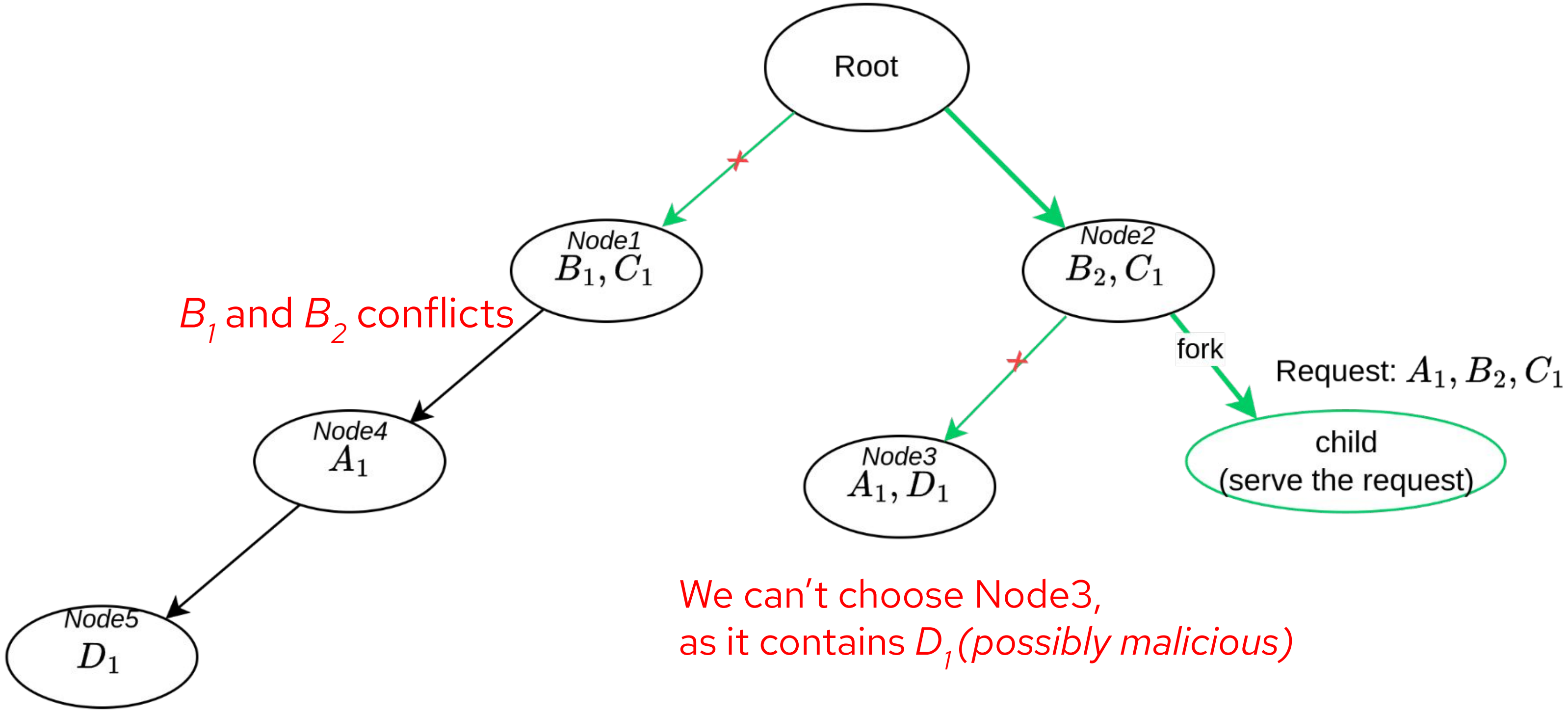
When a requests comes in, do a **DFS** search in the tree and first non-root node is selected to serve the requests.



Deployment in OpenLambda

We choose to deploy in OpenLambda as it is based on SOCK container that supports ***sandbox-level fork***.

When a requests comes in, do a **DFS** search in the tree and first non-root node is selected to serve the requests.



B_1 and B_2 conflicts

We can't choose Node3,
as it contains D_1 *(possibly malicious)*

4. Evaluation

- Memory usage vs throughput, latency CDF
- Warmup time, package hit rate

Evaluation Overview

The call trace includes 1793 unique invocations.

Train trace:Test trace=50:50

We construct & test trees of varying sizes using **four variants** of the Forklift algorithm, they are:

Forklift is run on the train trace,

play the test trace on OpenLambda with 5 threads.

$$\underbrace{\begin{Bmatrix}\text{Uniform Weight}\\ \text{Time-based Weight}\end{Bmatrix}}_{\textit{weight policies}} \times \underbrace{\begin{Bmatrix}\text{Single-package}\\ \text{Multi-package}\end{Bmatrix}}_{\textit{single-package per node?}}$$

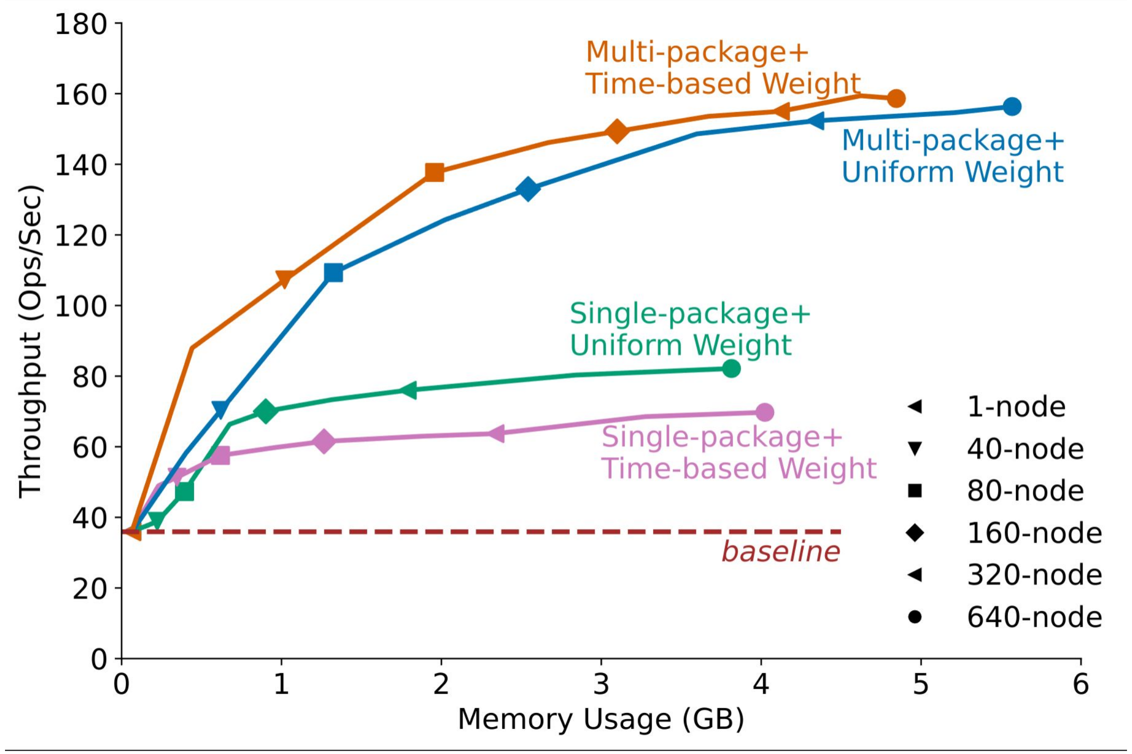
Memory Usage vs Throughput

Finding 1:

Multi-package optimization is crucial.

Finding 2:

Weighting packages by import latency benefits smaller trees significantly, but not for larger trees.



Latency CDF

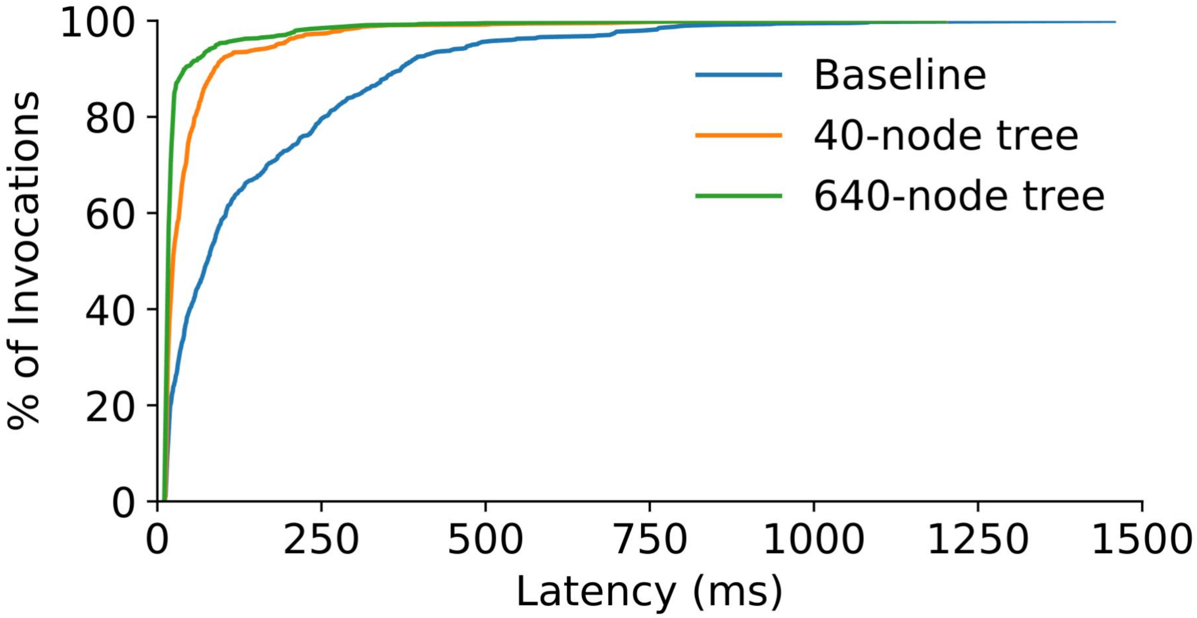
latency of different size trees under multi-package time-based weight strategy:

Median Speedup:

- 40-node(small) trees: **3.2× faster**
- 640-node(large) trees: **4.8× faster**

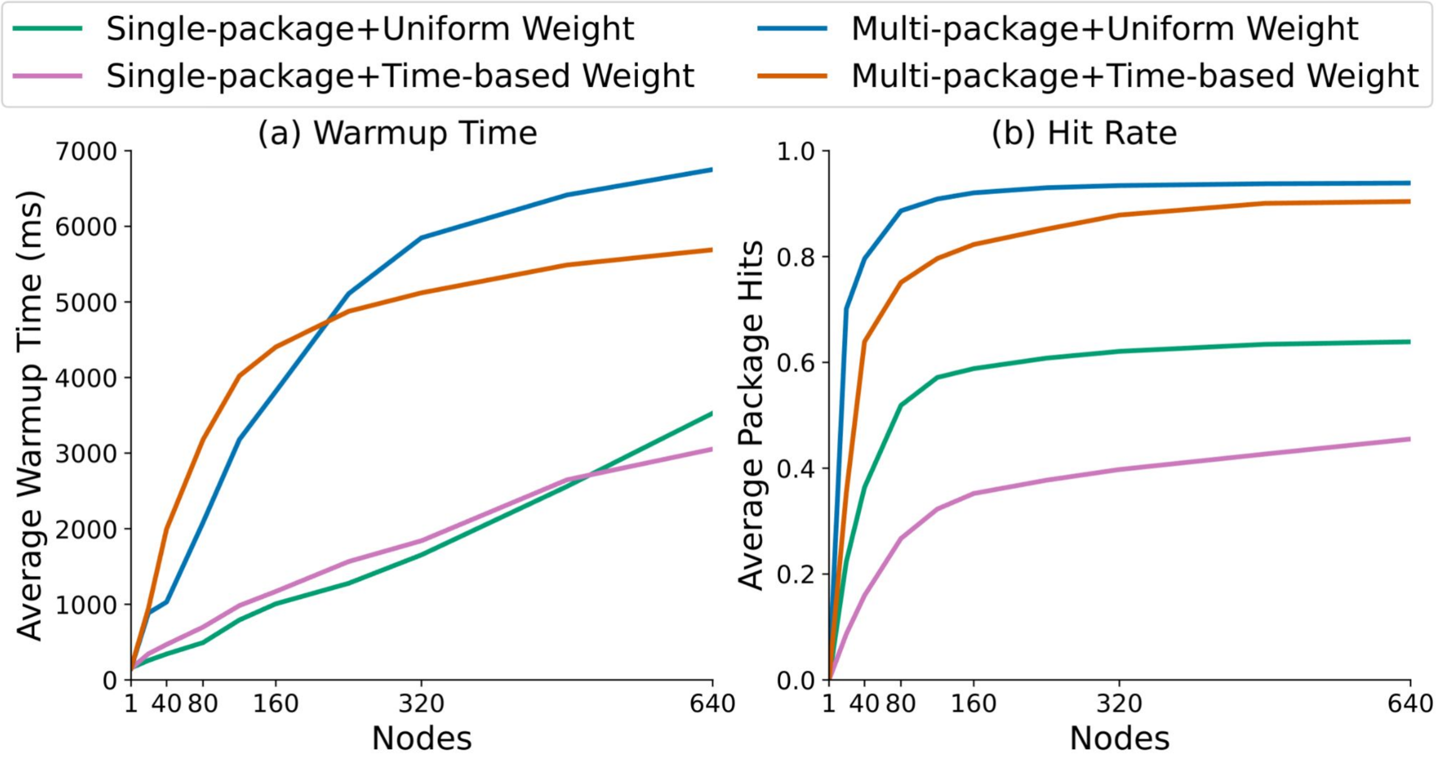
95th Percentile Speedup:

- 40-node trees: **2.7× faster**
- 640-node trees: **5.3× faster**



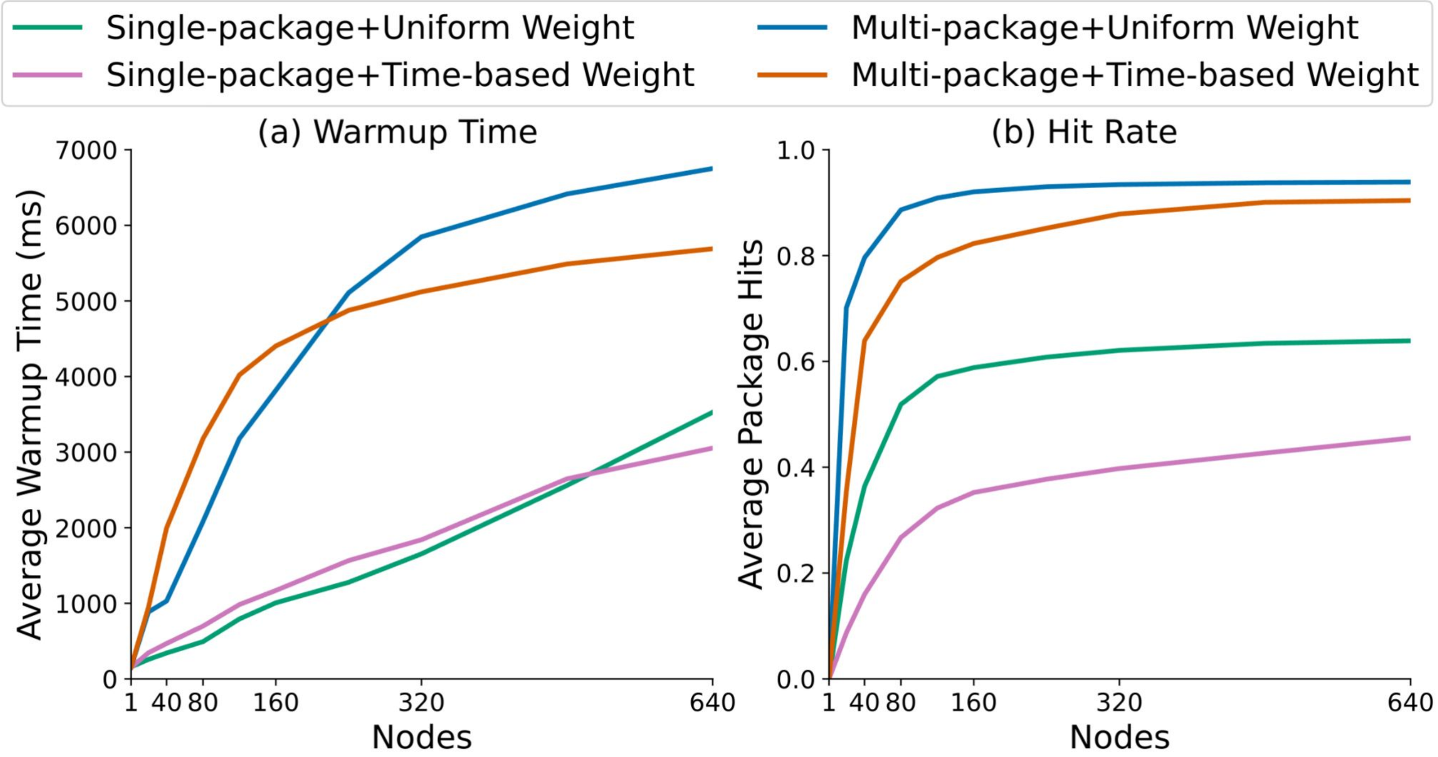
Warmup Time and Hit Rate

Concurrently create the zygote processes with six threads during warmup.
Package hit: packages required by functions provided by zygotes are hits



Warmup Time and Hit Rate

All zygotes can be created in less than 7 seconds, even for large trees
The multi-package, uniform-weighted tree has the best hit rates (over 90%)



Conclusion

> Forklift, a new algorithm for constructing hierarchical zygote trees

> Achieves ~5× faster invocation latency on OpenLambda

open-source at:
https://github.com/open-lambda/forklift
https://github.com/open-lambda/ReqBench

Contact

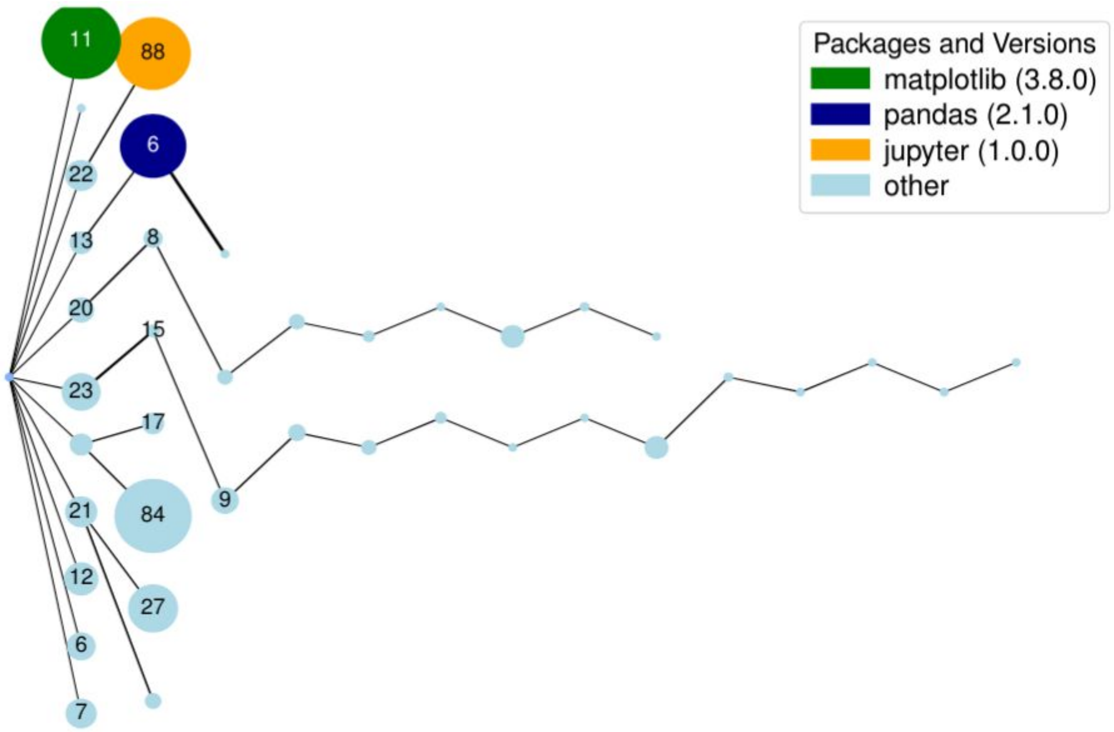
Yuanzhuo Yang

yyang682@wisc.edu
yuanzhuoyang@gmail.com

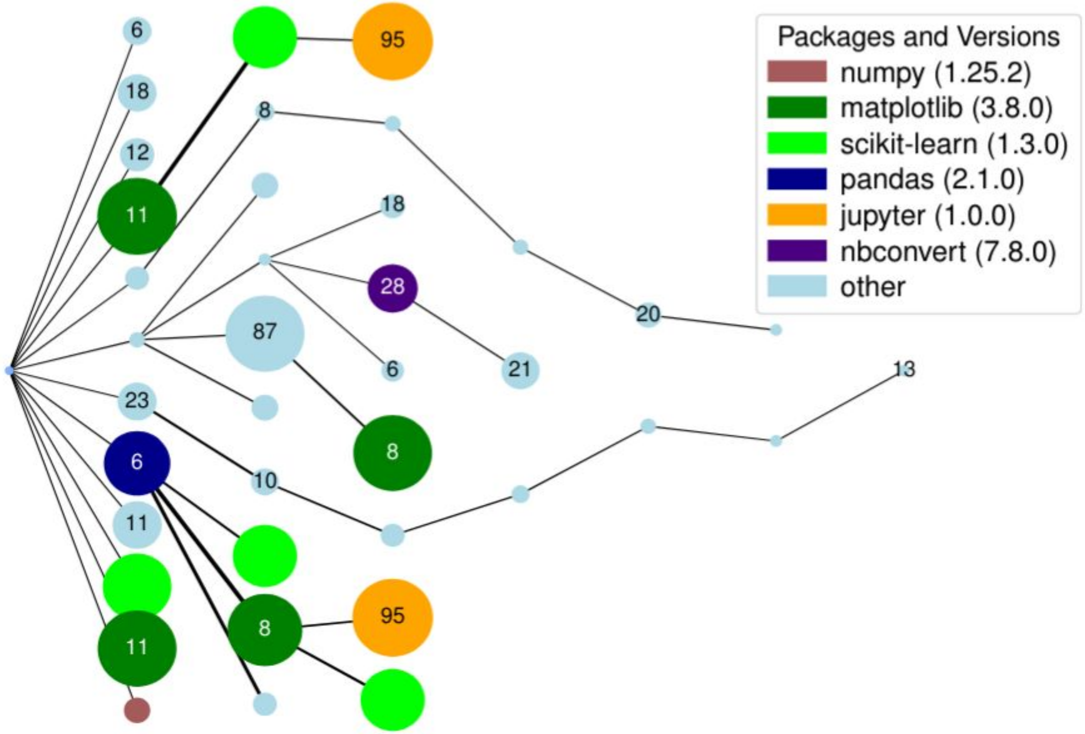
Feel free to drop an email if you have questions!

I am seeking for Ph.D. or funded M.S. positions worldwide starting 2025 Fall.

appendix: Multi-package trees visualization



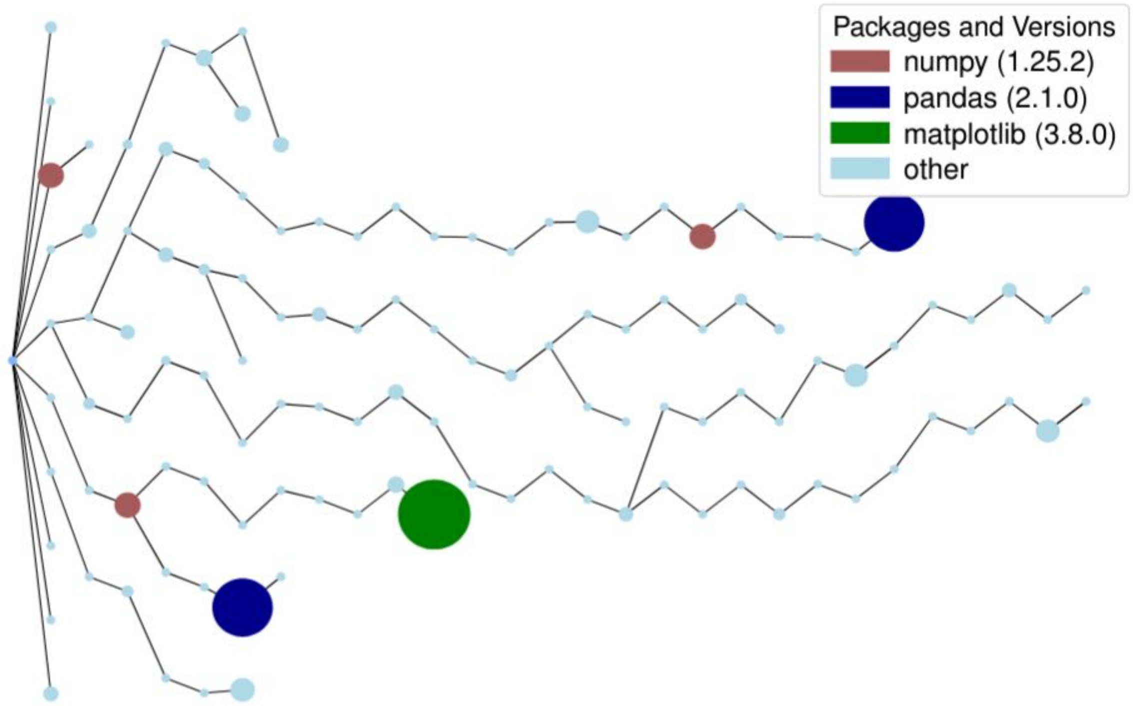
Multi-package, Uniform Weight 40-node tree



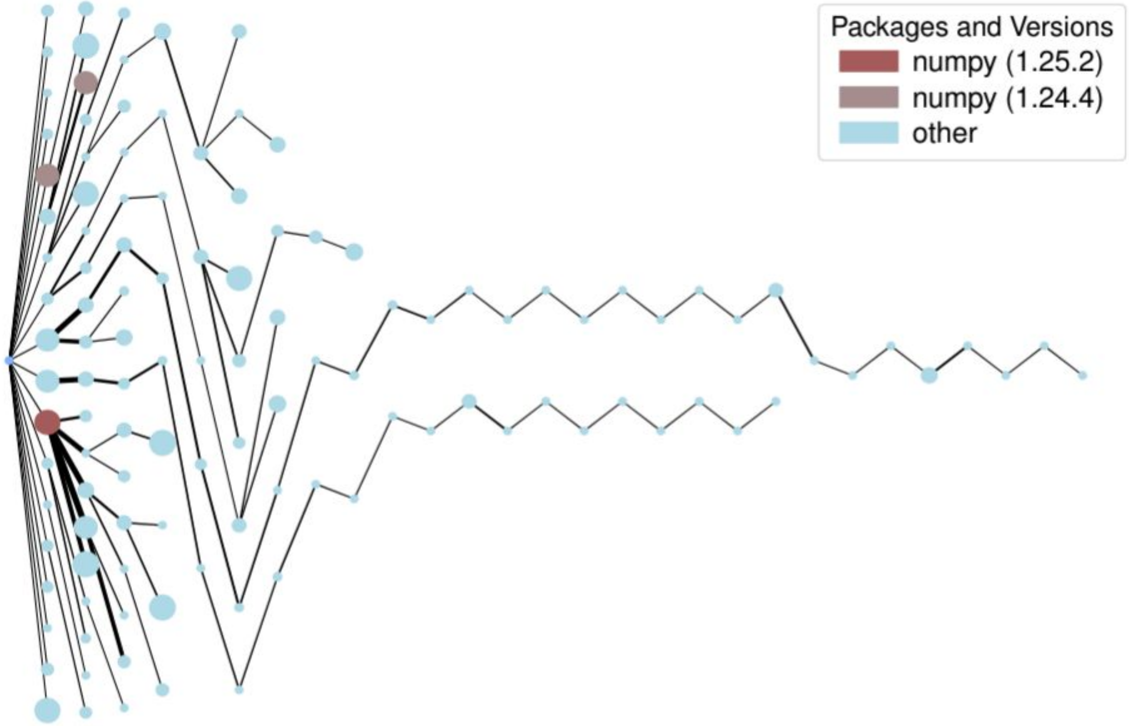
Multi-package, Time-based Weight 40-node tree

Note: Node numbers in nodes represent module imports, with only values above 5 shown.

appendix: Single-package trees visualization



Single-package, Uniform Weight 120-node tree



Single-package, Time-based Weight 120-node tree

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON