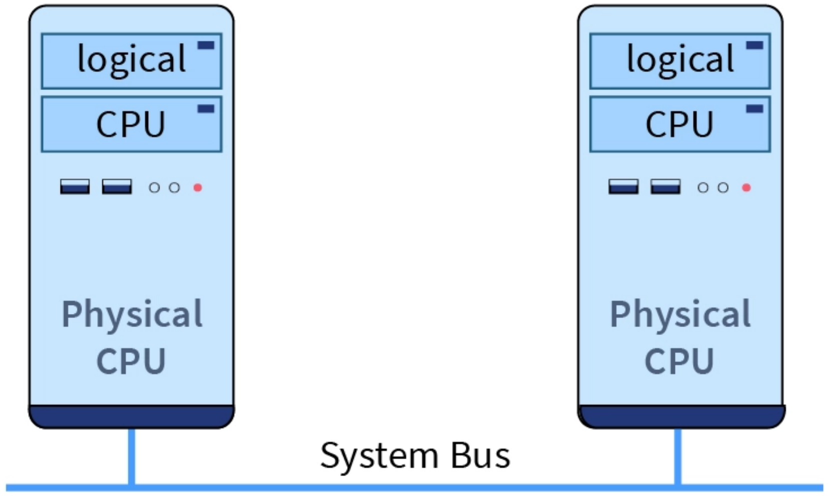
Multiprocessor Scheduling & HU's Algorithm



Abdullah Al Forkan
Matriculation Number: 2190314

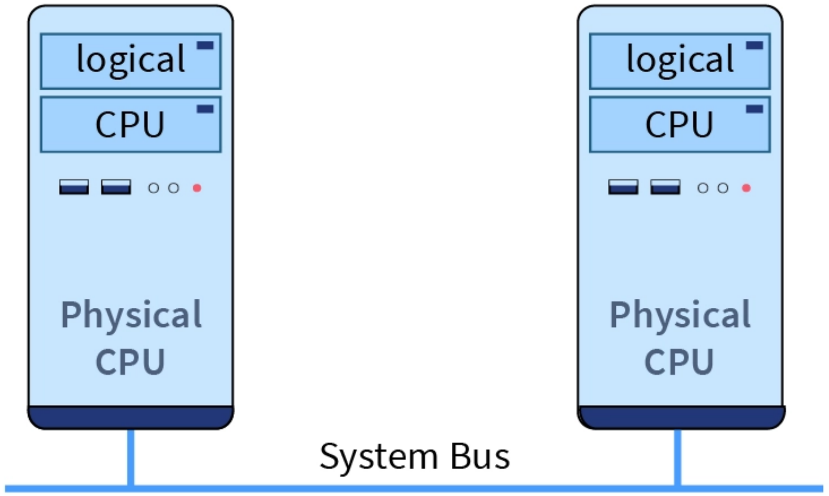
Motivation

Challenges:

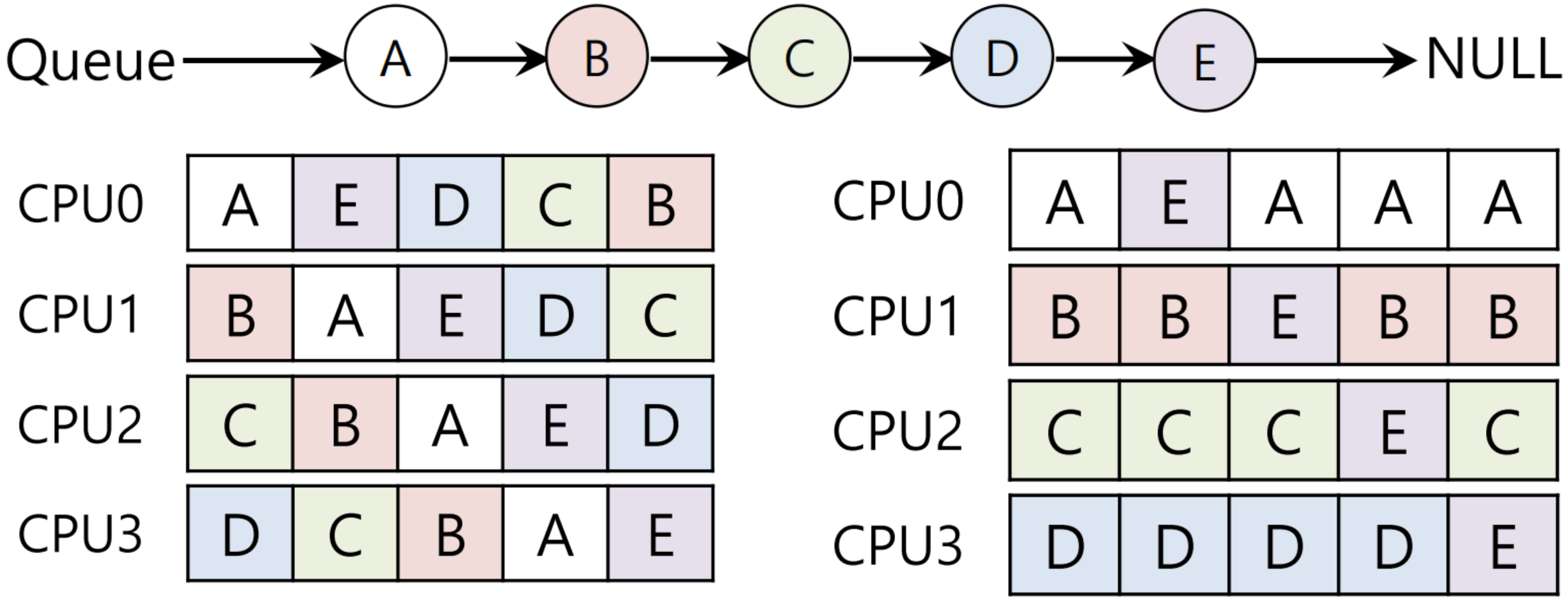
- Complex Problems
- Larger Dataset
- Slow Processors
- Parallel Programming

Factors:

- Powerful Processor
- System Performance
- Task Dependencies
- Optimal
- Short Time



Symmetric Multiprocessing



CPU0	A	E	D	C	B
CPU1	B	A	E	D	C
CPU2	C	B	A	E	D
CPU3	D	C	B	A	E

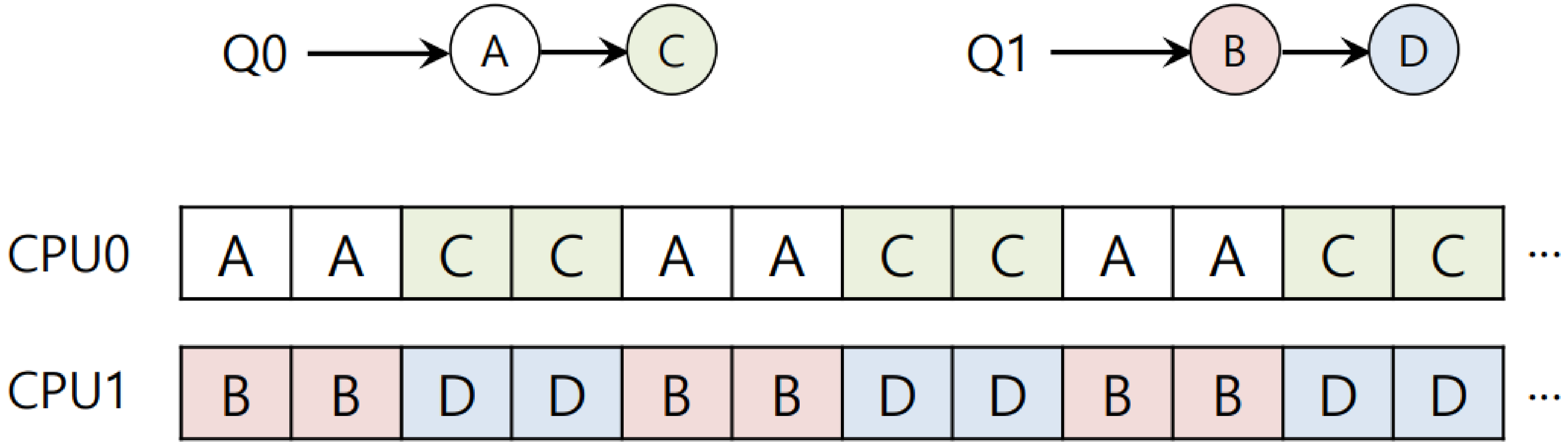
(a). SM with cache affinity

CPU0	A	E	A	A	A
CPU1	B	B	E	B	B
CPU2	C	C	C	E	C
CPU3	D	D	D	D	E

(b). Preserving affinity for most

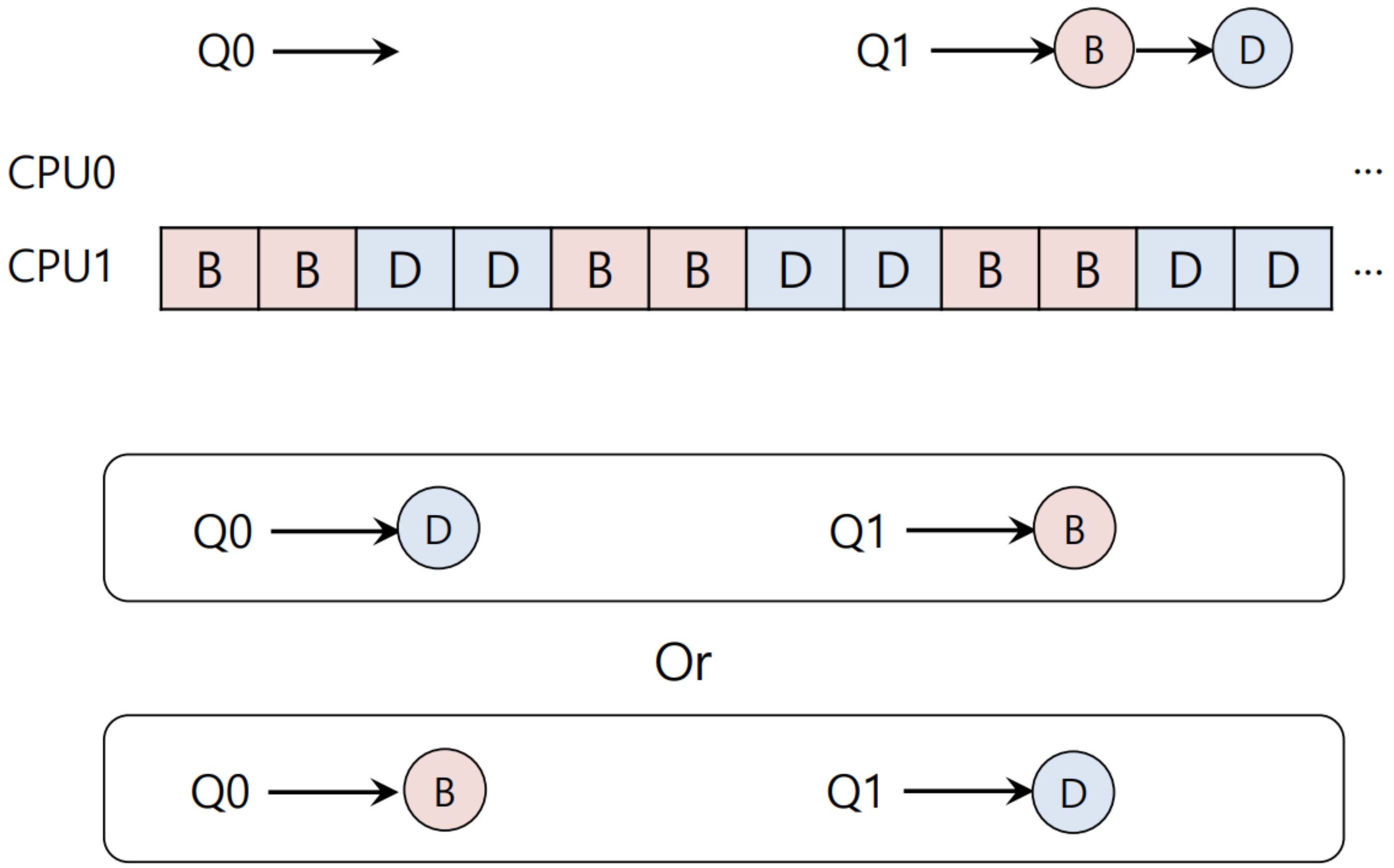
Multiprocessor Scheduling

Asymmetric Multiprocessing



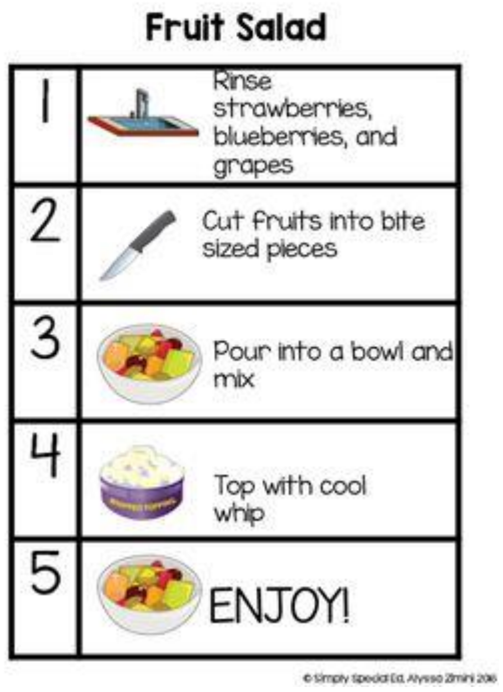
Multiprocessor Scheduling

Load Balancing



HU's Algorithm

- Task dependencies
- Divided into multiple sub tasks.
- Recipe that has to be cocked to maintain an order
- The optimal
- Reduce the total time needed



Fruit Salad

1	Rinse strawberries, blueberries, and grapes
2	Cut fruits into bite sized pieces
3	Pour into a bowl and mix
4	Top with cool whip
5	ENJOY!

© Simply Special Ed, Alyssa Zmini 2016

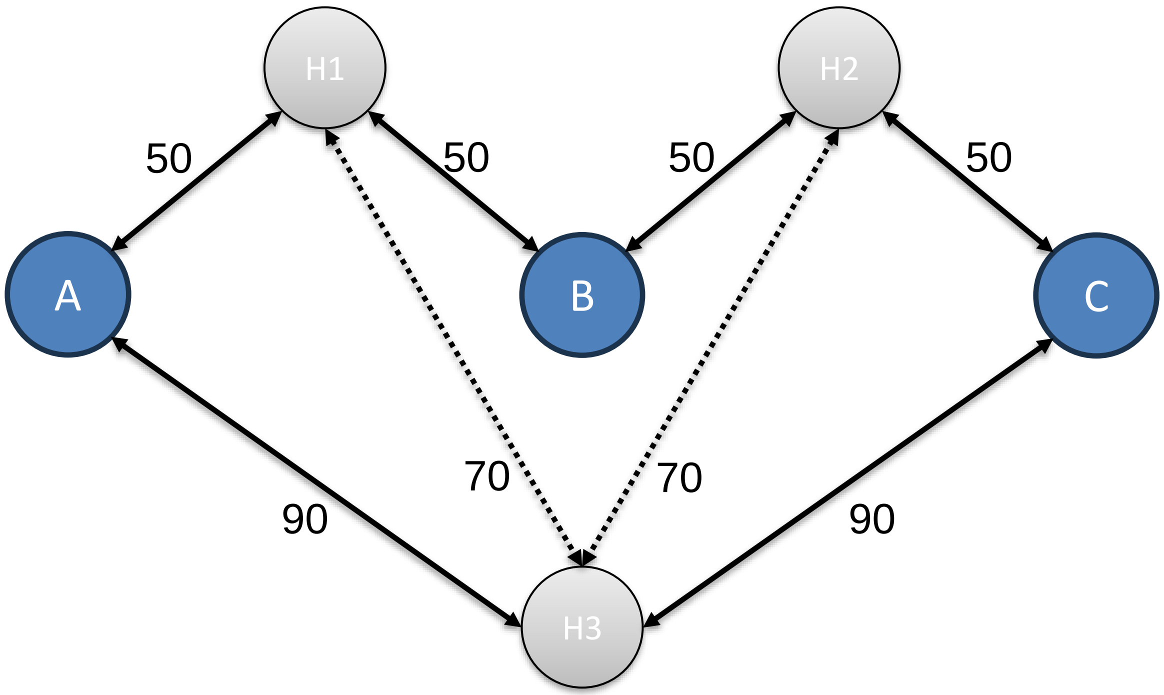
HU’s Algorithmic Steps

Physical Internet Scenario

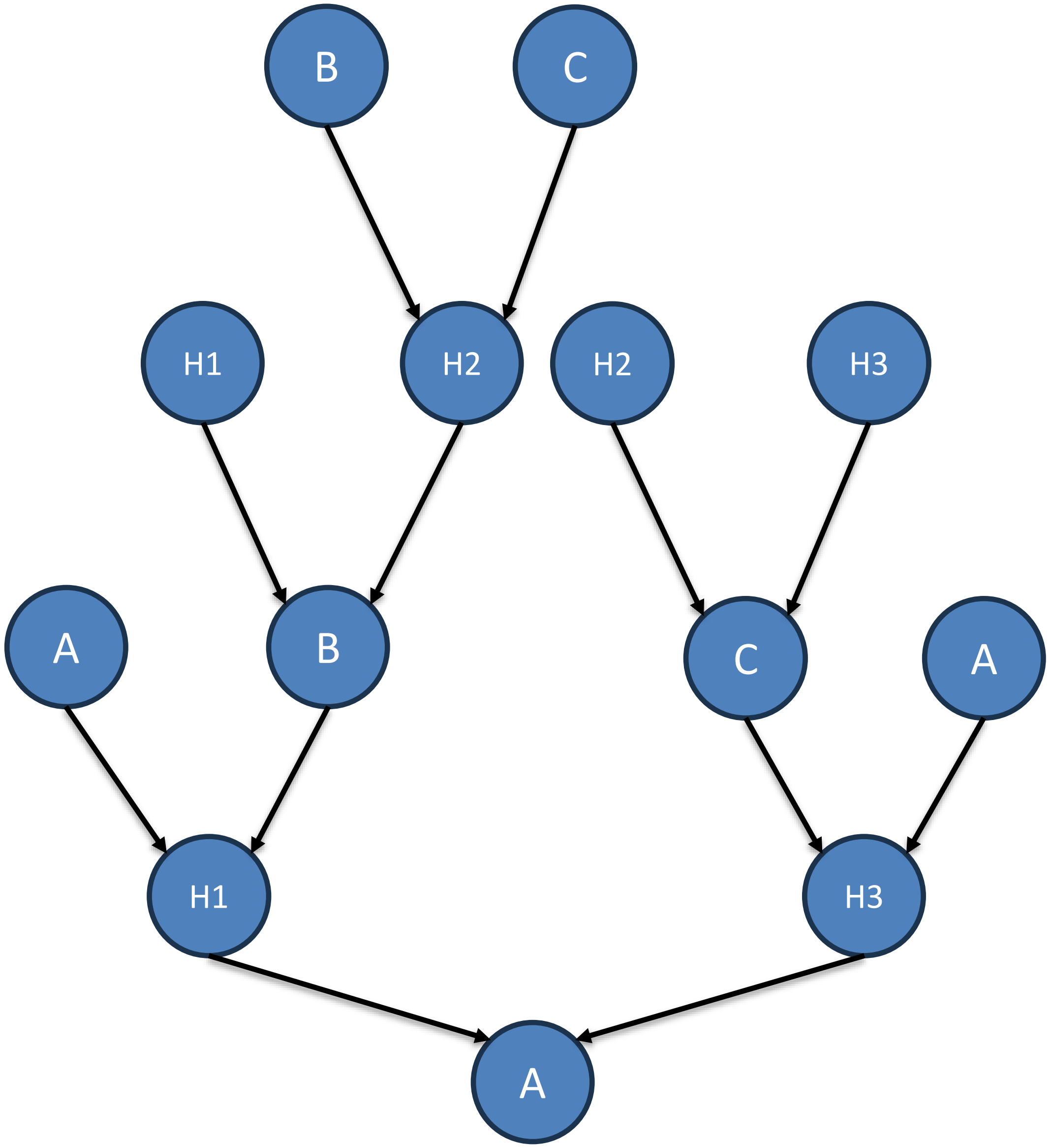
City	Hub	Hub dependency
A	H1	A, B
B	H2	B, C
C	H3	A, C

HU’s Algorithmic Steps

Task Graph & Assign Priorities



HU's Algorithmic Steps

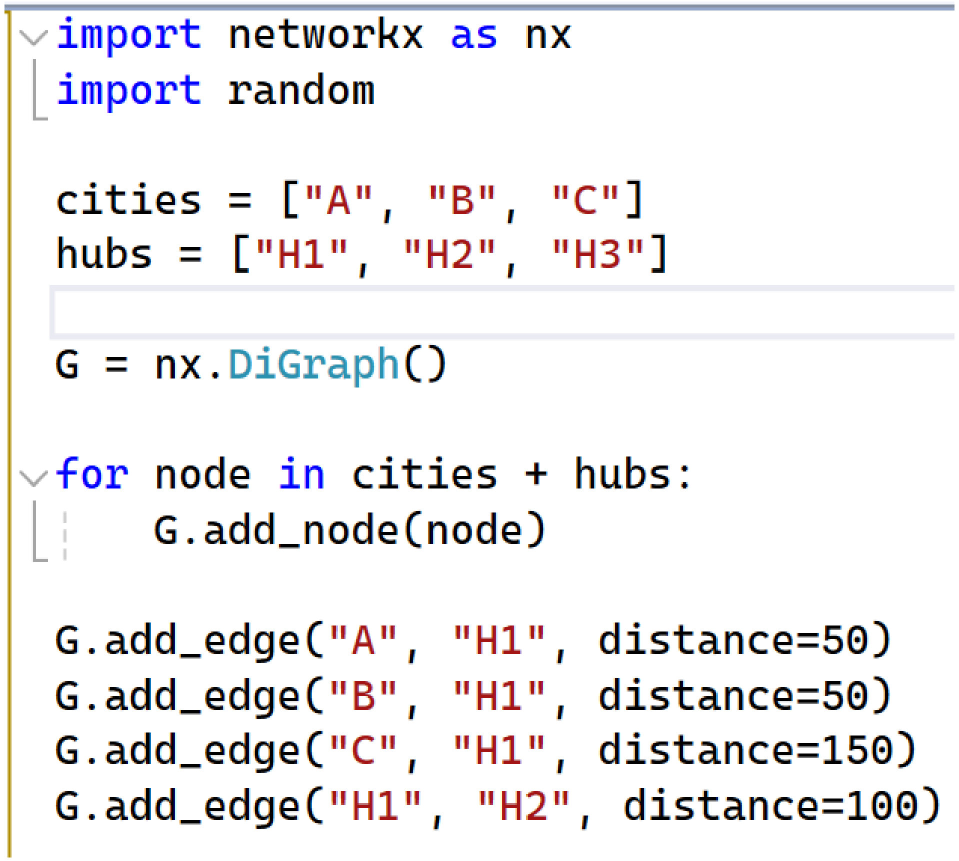


- Priority
- Execution Time

Python Implementation

HOCHSCHULE
HAMM-LIPPSTADT

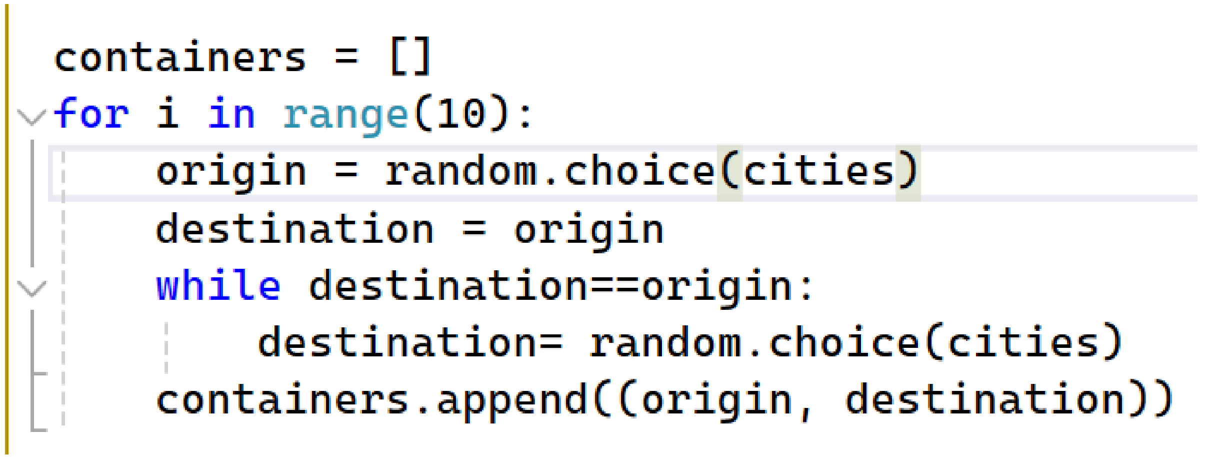
1. First Step



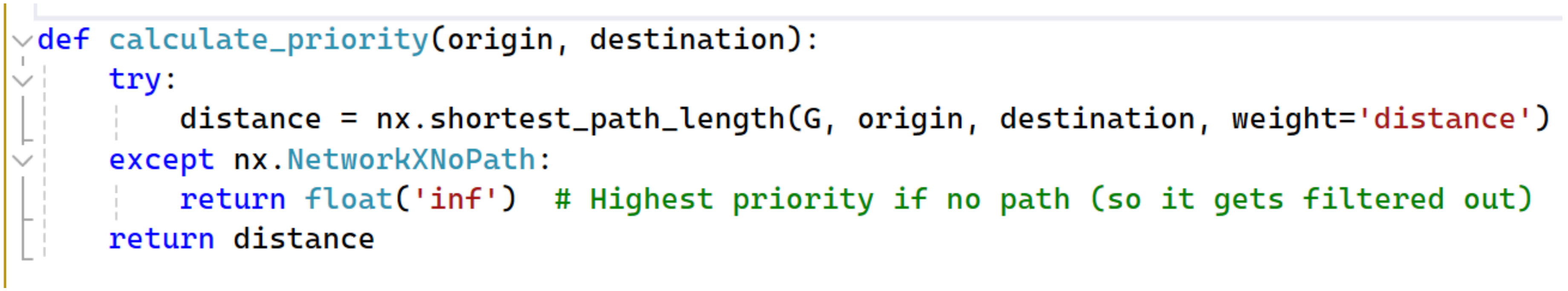
```python
import networkx as nx
import random

cities = ["A", "B", "C"]
hubs = ["H1", "H2", "H3"]

G = nx.DiGraph()

for node in cities + hubs:
    G.add_node(node)

G.add_edge("A", "H1", distance=50)
G.add_edge("B", "H1", distance=50)
G.add_edge("C", "H1", distance=150)
G.add_edge("H1", "H2", distance=100)
```

Python Implementation

2. Second Step



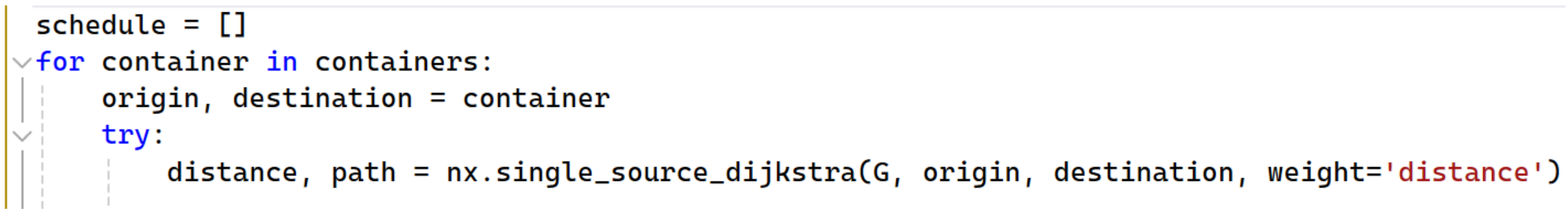
```python
containers = []
for i in range(10):
    origin = random.choice(cities)
    destination = origin
    while destination==origin:
        destination= random.choice(cities)
    containers.append((origin, destination))
```



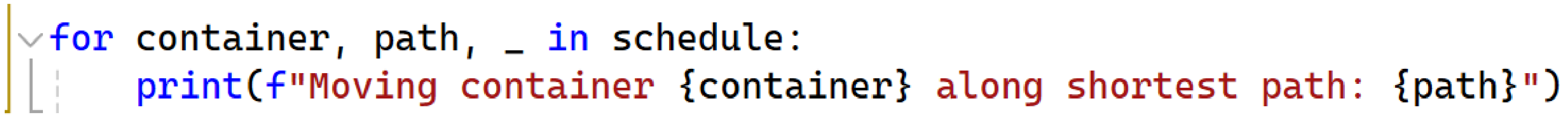
```python
def calculate_priority(origin, destination):
    try:
        distance = nx.shortest_path_length(G, origin, destination, weight='distance')
    except nx.NetworkXNoPath:
        return float('inf')  # Highest priority if no path (so it gets filtered out)
    return distance
```

Python Implementation

3. Third Step



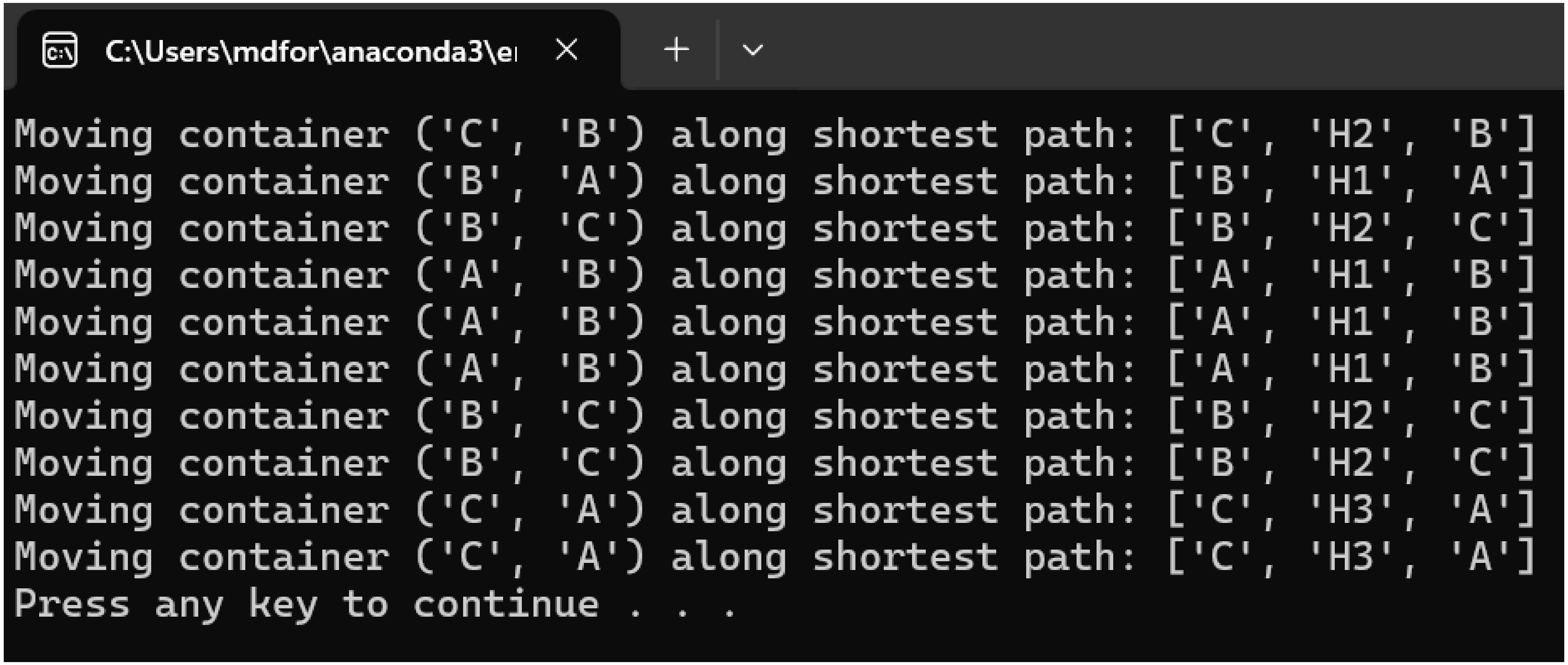
```python
schedule = []
for container in containers:
    origin, destination = container
    try:
        distance, path = nx.single_source_dijkstra(G, origin, destination, weight='distance')
```



```python
for container, path, _ in schedule:
    print(f"Moving container {container} along shortest path: {path}")
```

Python Implementation

Result



```
Moving container ('C', 'B') along shortest path: ['C', 'H2', 'B']
Moving container ('B', 'A') along shortest path: ['B', 'H1', 'A']
Moving container ('B', 'C') along shortest path: ['B', 'H2', 'C']
Moving container ('A', 'B') along shortest path: ['A', 'H1', 'B']
Moving container ('A', 'B') along shortest path: ['A', 'H1', 'B']
Moving container ('A', 'B') along shortest path: ['A', 'H1', 'B']
Moving container ('B', 'C') along shortest path: ['B', 'H2', 'C']
Moving container ('B', 'C') along shortest path: ['B', 'H2', 'C']
Moving container ('C', 'A') along shortest path: ['C', 'H3', 'A']
Moving container ('C', 'A') along shortest path: ['C', 'H3', 'A']
Press any key to continue . . .
```