香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Greedy Algorithms

Today

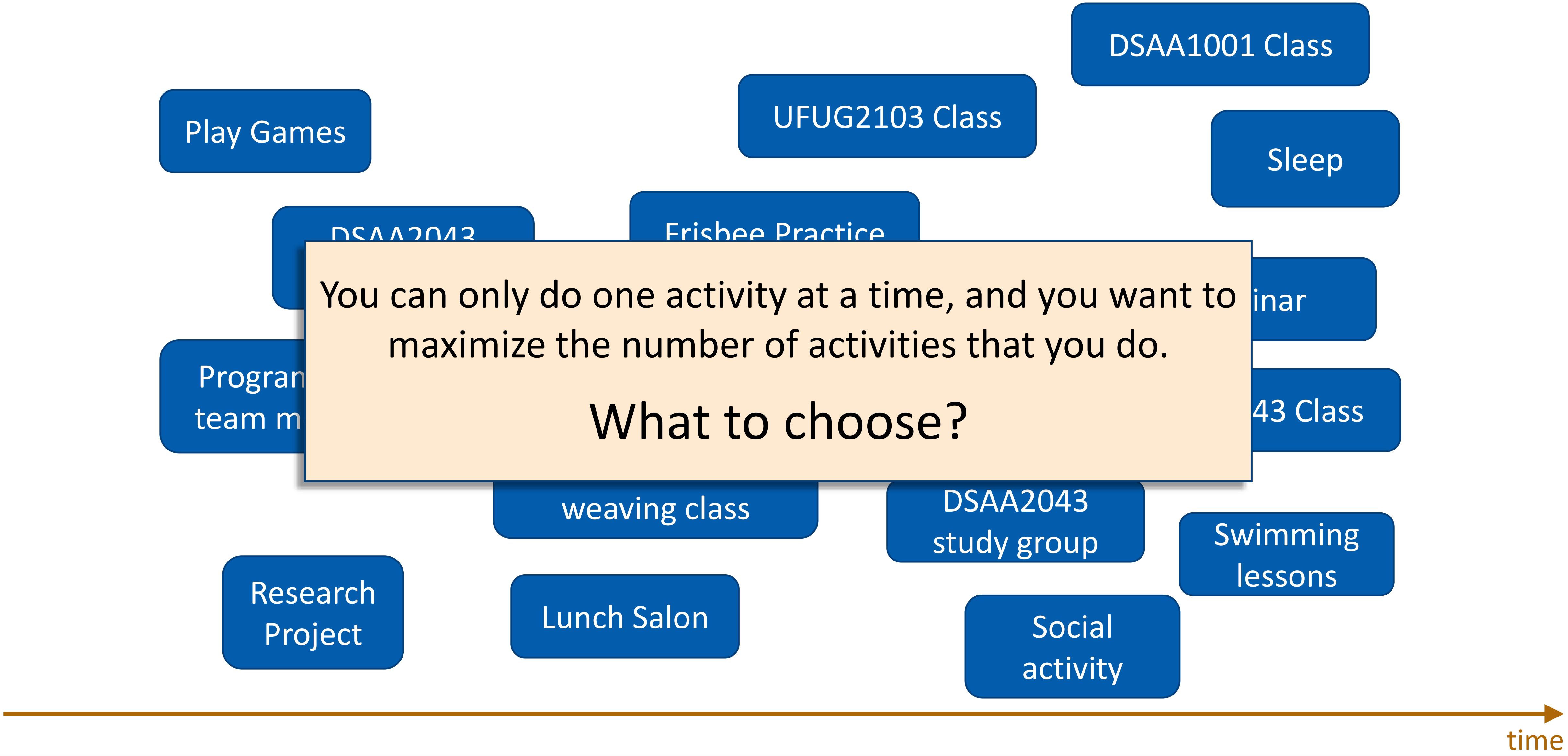
One example of a greedy algorithm that **does not work**:

- Knapsack

Three examples of greedy algorithms that **do work**:

- Activity Selection
- Job Scheduling
- Minimum Spanning Tree

Example where greedy works

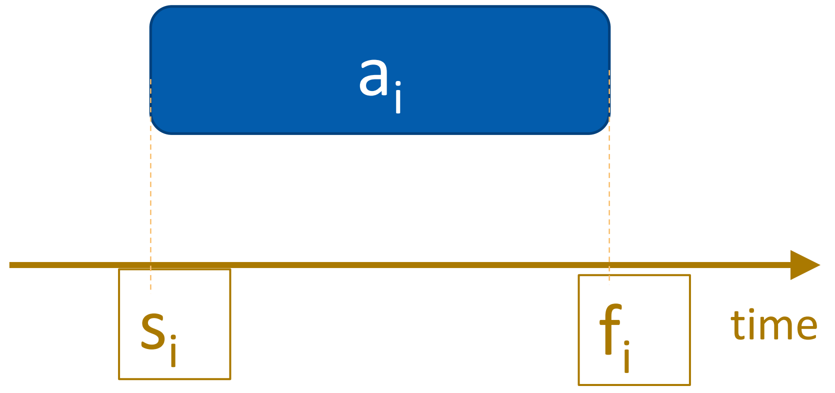


You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

Activity selection

- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$

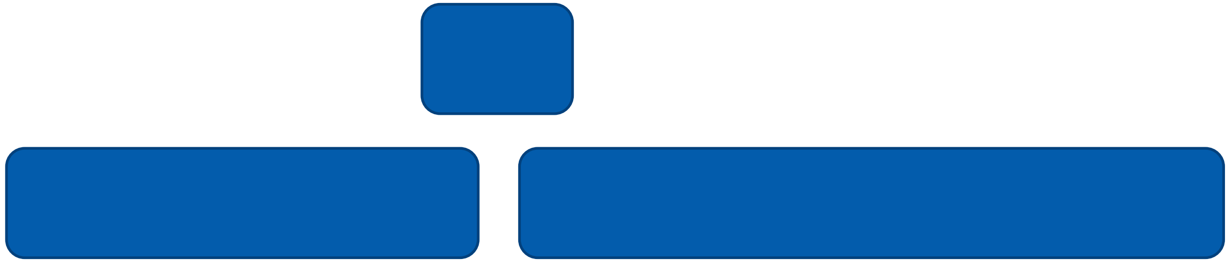


- Output:
  - A way to maximize the number of activities you can do today.

In what order should you greedily add activities?

In what order?

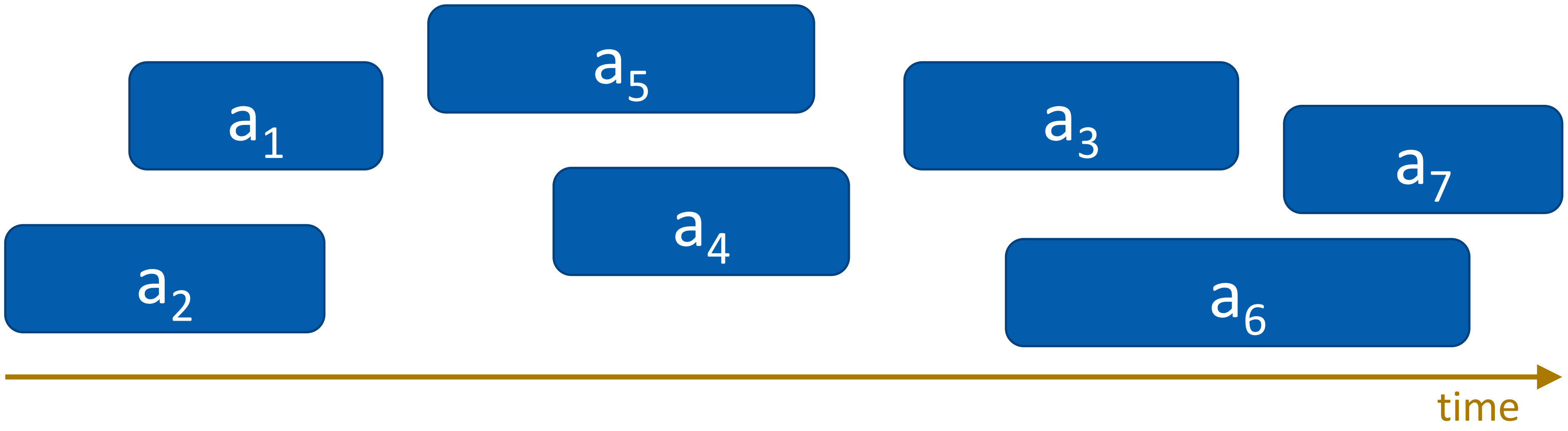
- Shortest job first?



- Earliest start time?

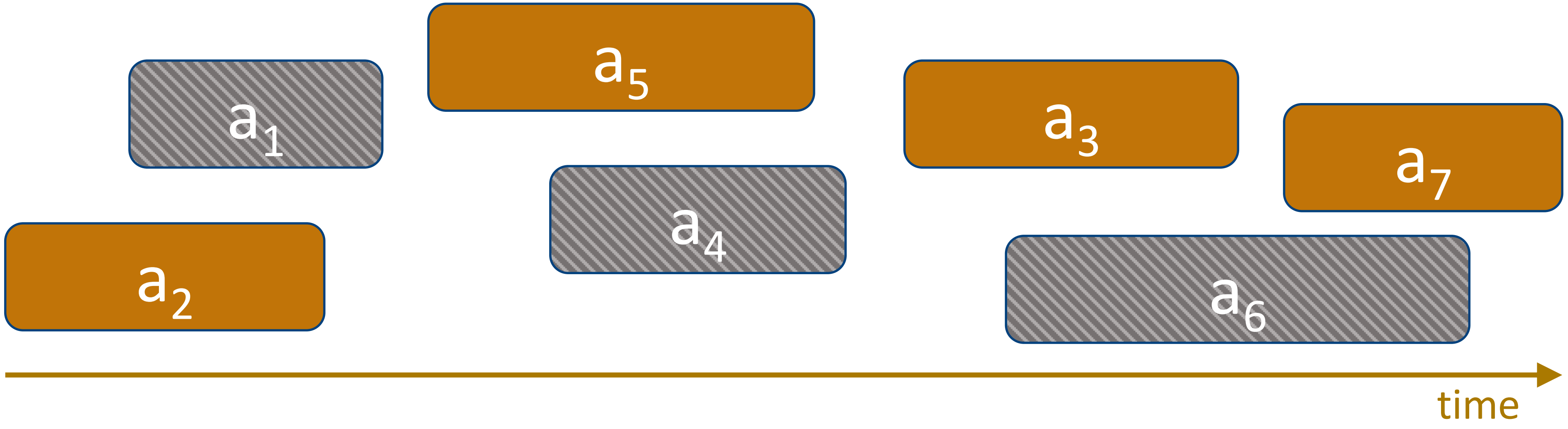
- Earliest finish time?

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Efficiency

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise, O(n log(n)) if you have to sort them first.

Back to Activity Selection

Why does it work?

- **We never rule out an optimal solution**
- At the end of the algorithm, we've got some solution.
- So it must be optimal.

A Common Strategy

A common strategy for proving the correctness of greedy algorithms:

- Make a **series of choices**.
- Show that, at each step, our choice **won't rule out an optimal solution** at the end of the day.
- After we've made all our choices, we haven't ruled out an optimal solution, **so we must have found one.**

A Common Strategy

- Inductive Hypothesis:
  - After greedy choice t, you haven't ruled out success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

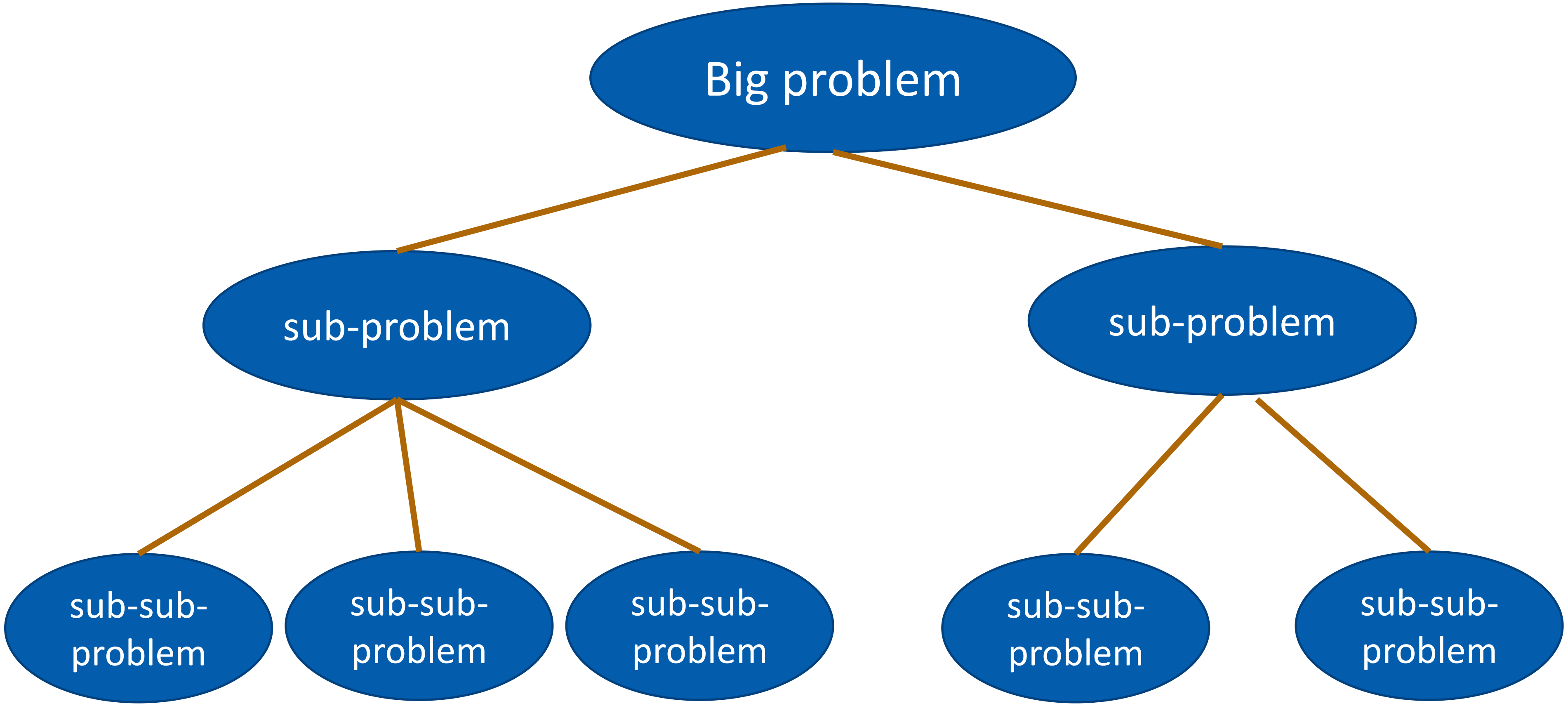
A Common Strategy

A common strategy for showing we don’t rule out the optimal solution:

- Suppose that you’re on track to make an optimal solution T*.
  - E.g., after you’ve picked activity i, you’re still on track.
- Suppose that T* *disagrees* with your next greedy choice.
  - E.g., it *doesn’t* involve activity k.
- Manipulate T* in order to make a solution T that’s not worse but that *agrees* with your greedy choice.
  - E.g., swap whatever activity T* did pick next with activity k.

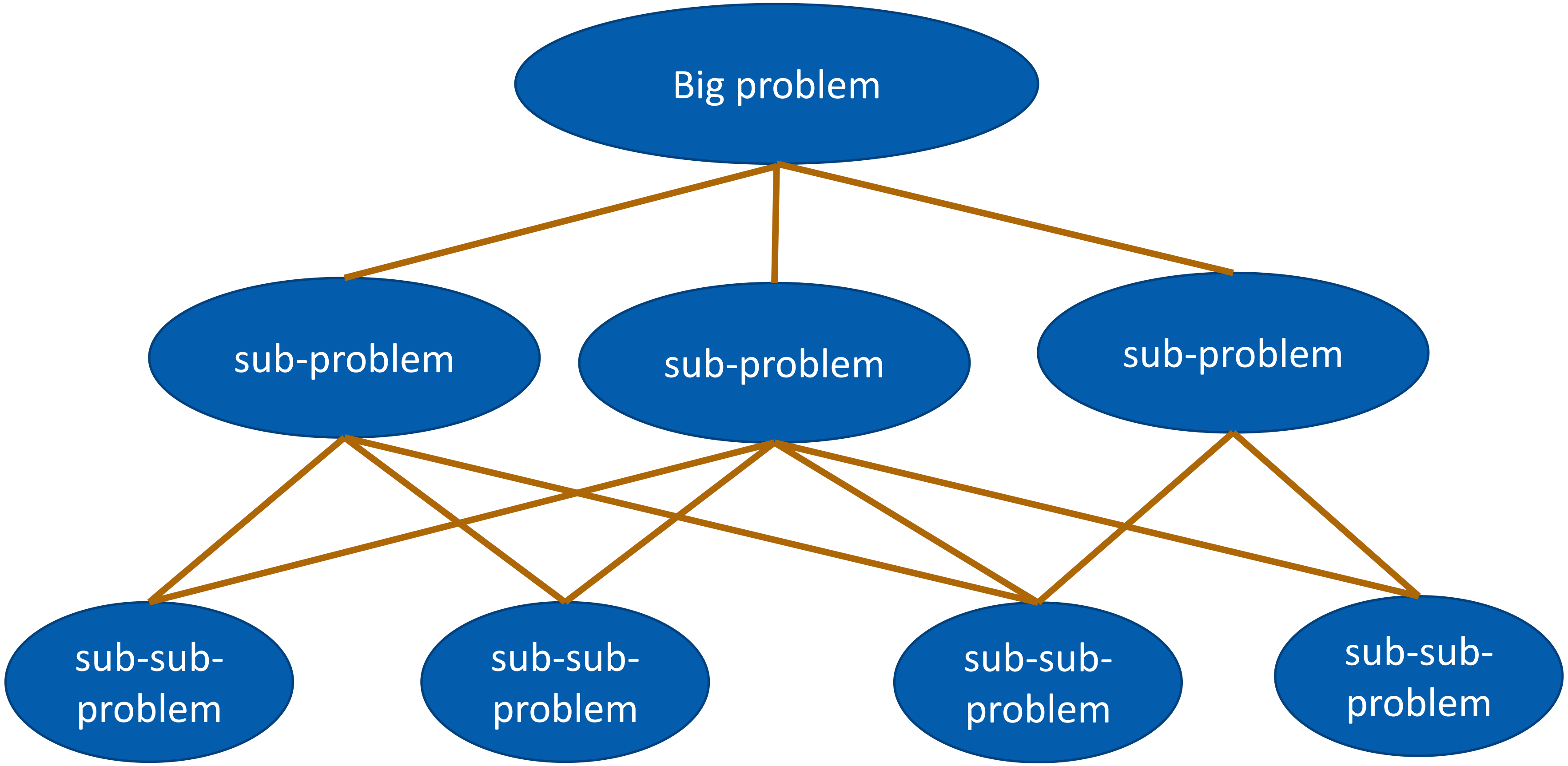
Sub-problem graph view

- Divide-and-conquer:



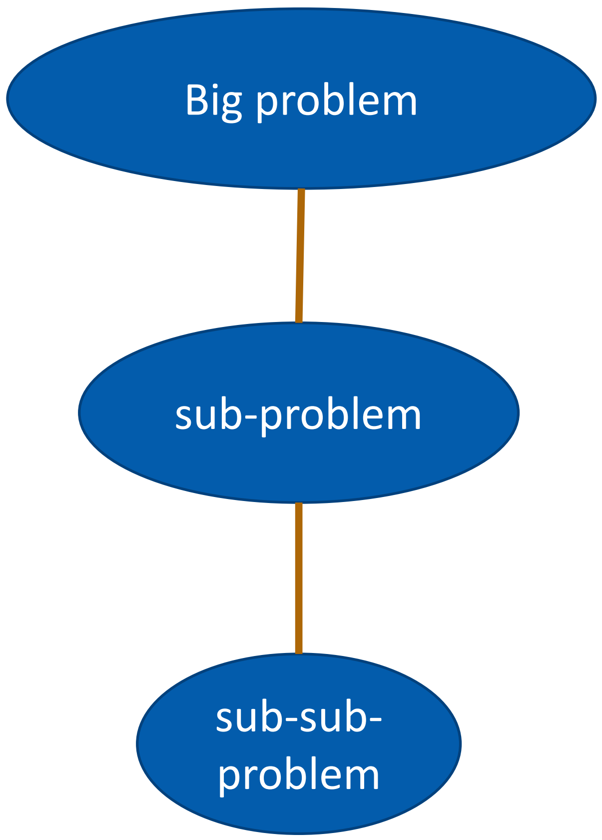
Sub-problem graph view

- Dynamic Programming:



Sub-problem graph view

- Greedy algorithms:



- Not only is there **optimal sub-structure:**
  - optimal solutions to a problem are made up from optimal solutions of sub-problems

- but each problem **depends on only one sub-problem**.

Another Example: Scheduling

- DSAA2043 HW
- Personal hygiene
- Math HW
- Administrative stuff for student club
- Econ HW
- Do laundry
- Sports
- Practice musical instrument
- Read lecture notes
- Have a social life
- Sleep

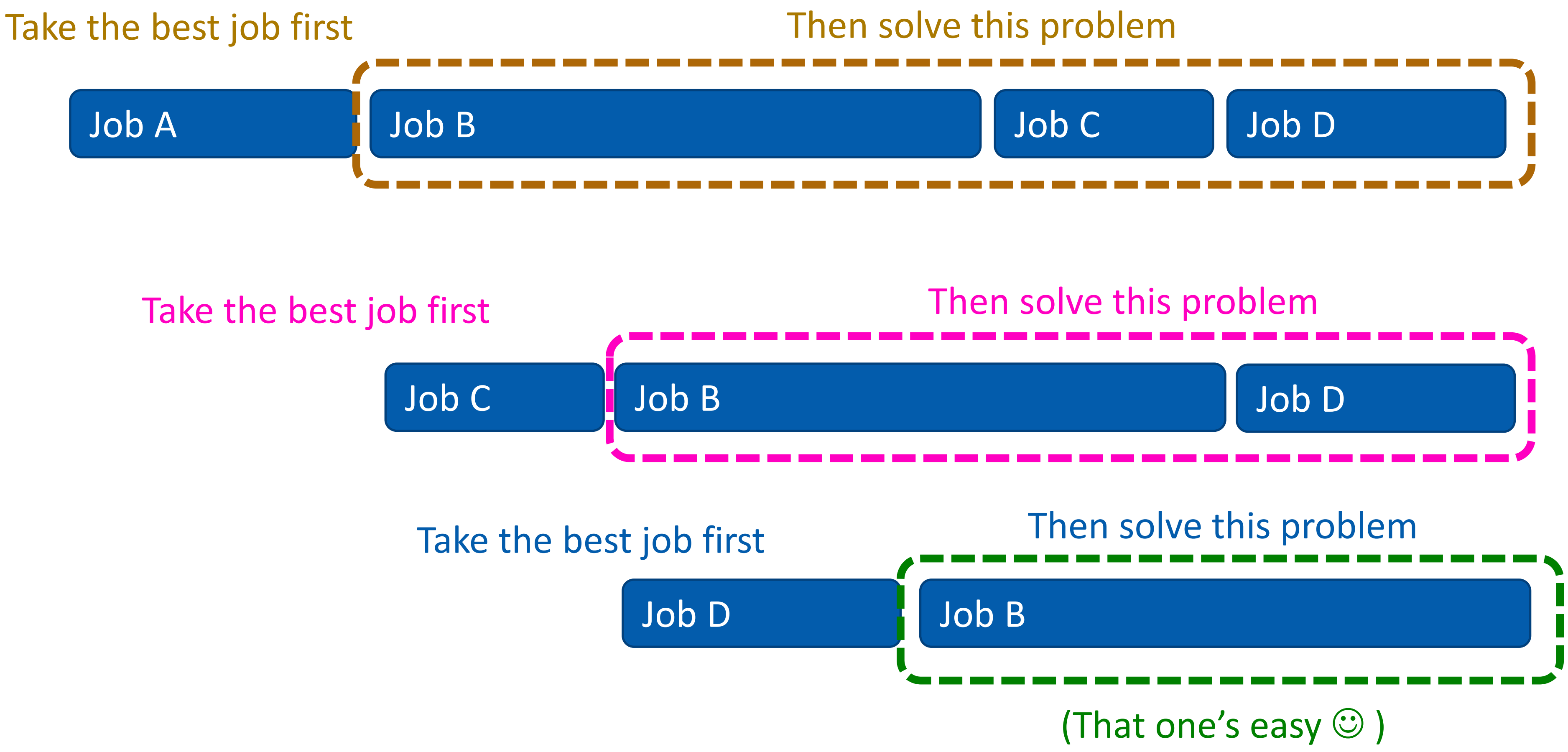
Scheduling

- n tasks
- Task i takes t_i hours
- For every hour that passes until task i is done, pay c_i

- DSAA2043 HW, then Sleep: costs $\mathbf{10 \cdot 2 + (10 + 8) \cdot 3 = 74}$ **units**
- Sleep, then DSAA2043 HW: costs $\mathbf{8 \cdot 3 + (10 + 8) \cdot 2 = 60}$ **units**

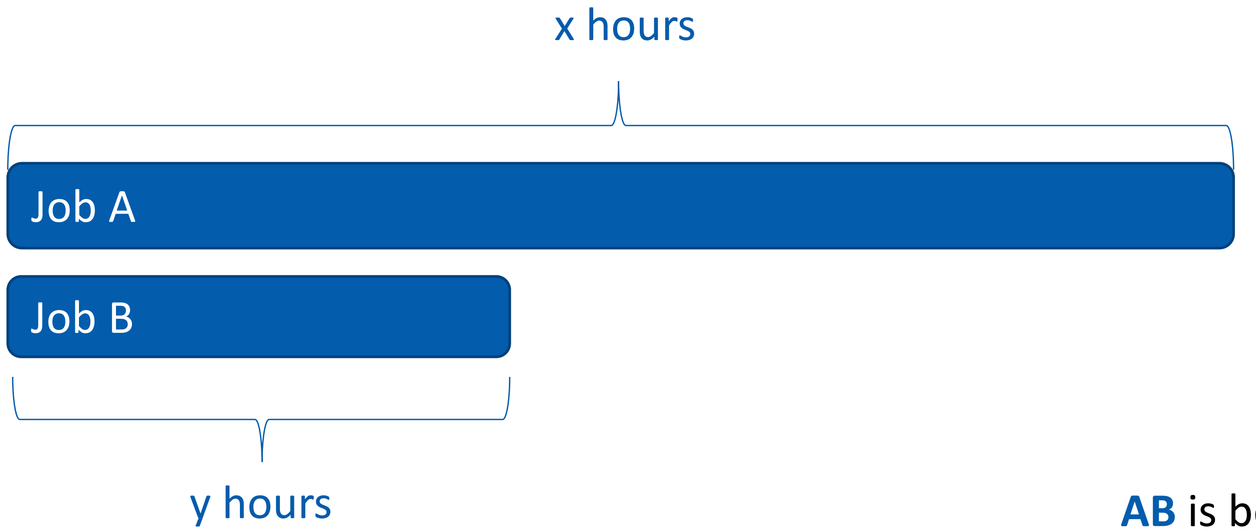
Scheduling

- Seems amenable to a greedy algorithm:



What does "best" mean?

- Of these two jobs, which should we do first?



Cost: z units per hour until it's done.

Cost: w units per hour until it's done.

- Cost(**A then B**) = x · z + (x + y) · w
- Cost(**B then A**) = y · w + (x + y) · z

AB is better than **BA** when:

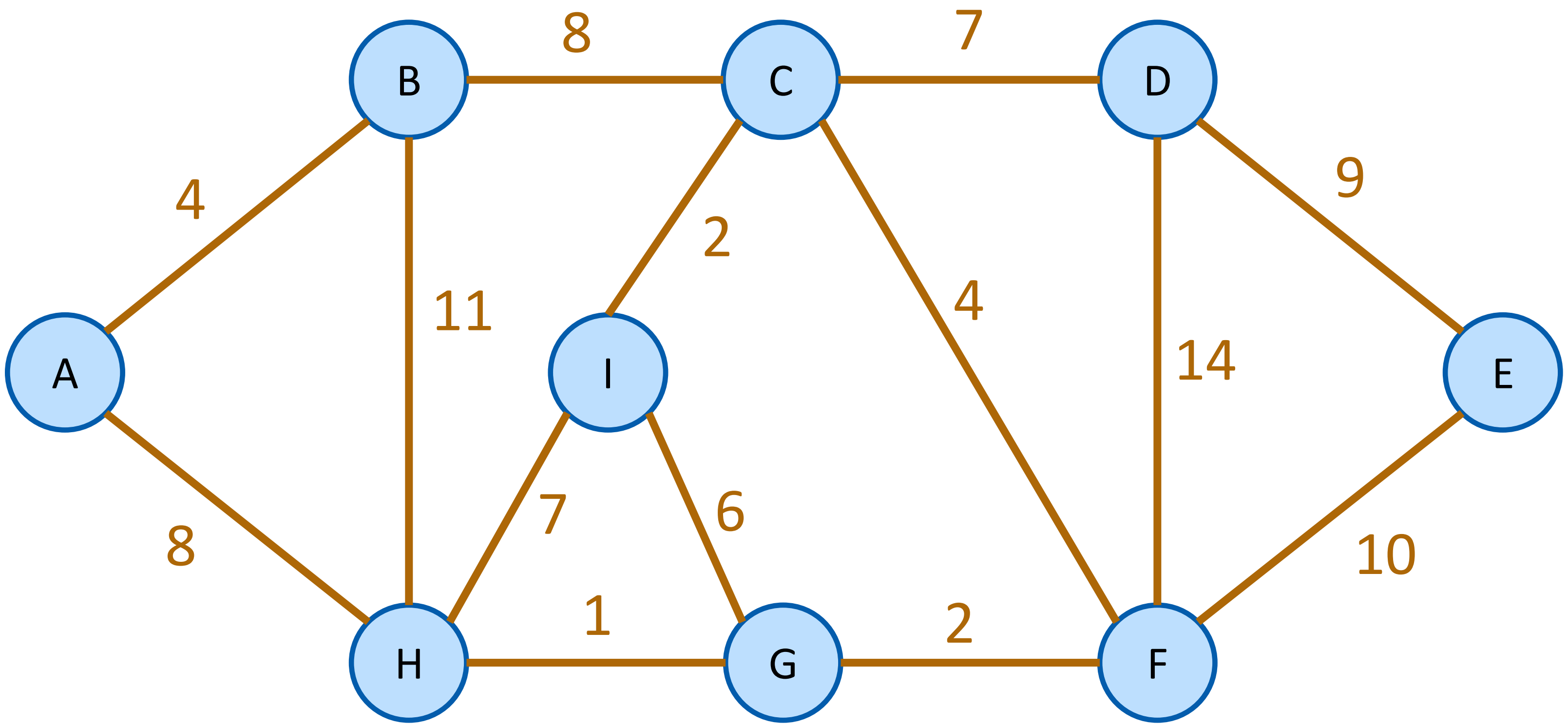
$$xz + (x+y)w \leq yw + (x+y)z$$
$$xz + xw + yw \leq yw + xz + yz$$
$$wx \leq yz$$
$$\frac{w}{y} \leq \frac{z}{x}$$

- Choose the job with the biggest $\dfrac{\text{cost of delay}}{\text{time it takes}}$ ratio.

Minimum Spanning Trees

Minimum Spanning Trees

- Say we have an undirected weighted graph



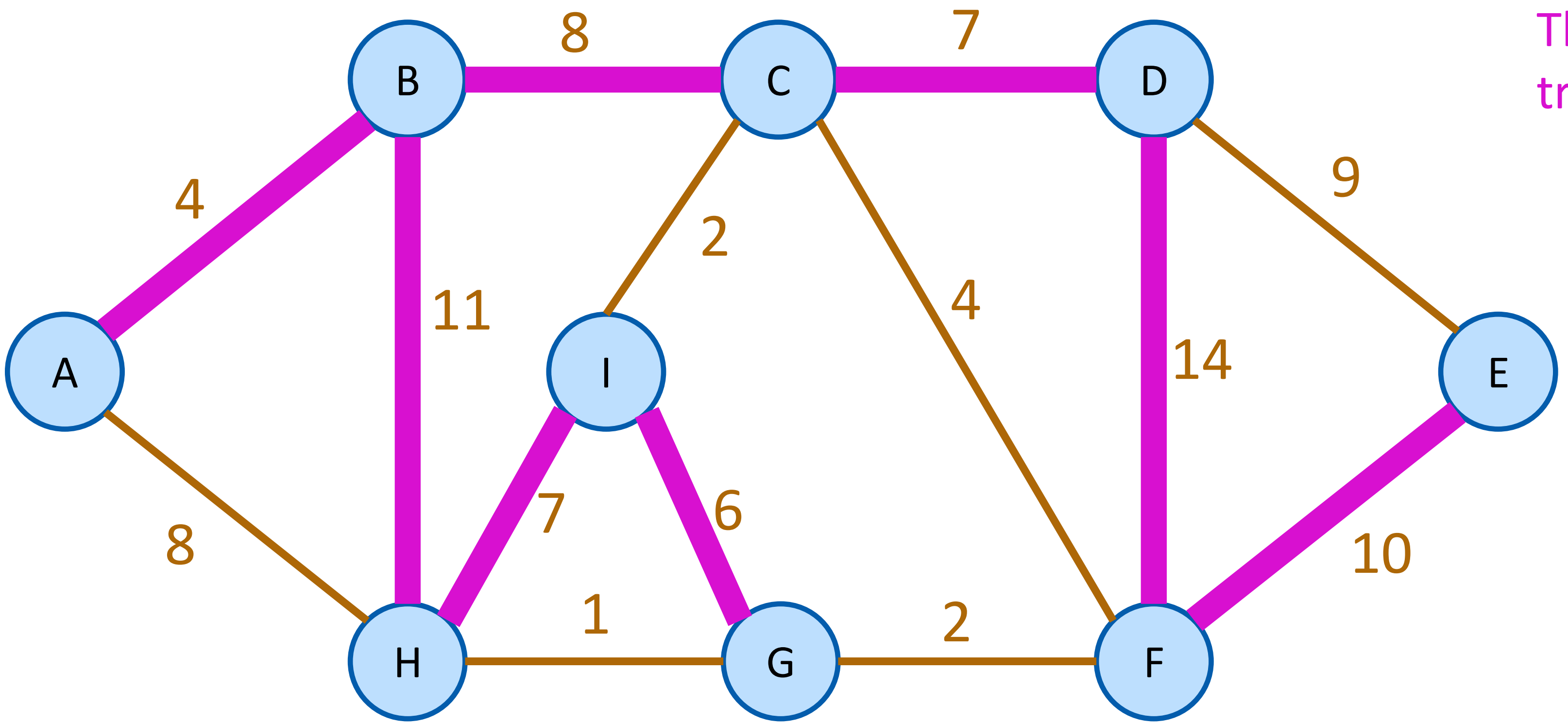
A **spanning tree** is a **tree** that connects all of the vertices.

A **tree** is a connected graph with no cycles!

Minimum Spanning Trees

- Say we have an undirected weighted graph

The **cost** of a spanning tree is the sum of the weights on the edges.

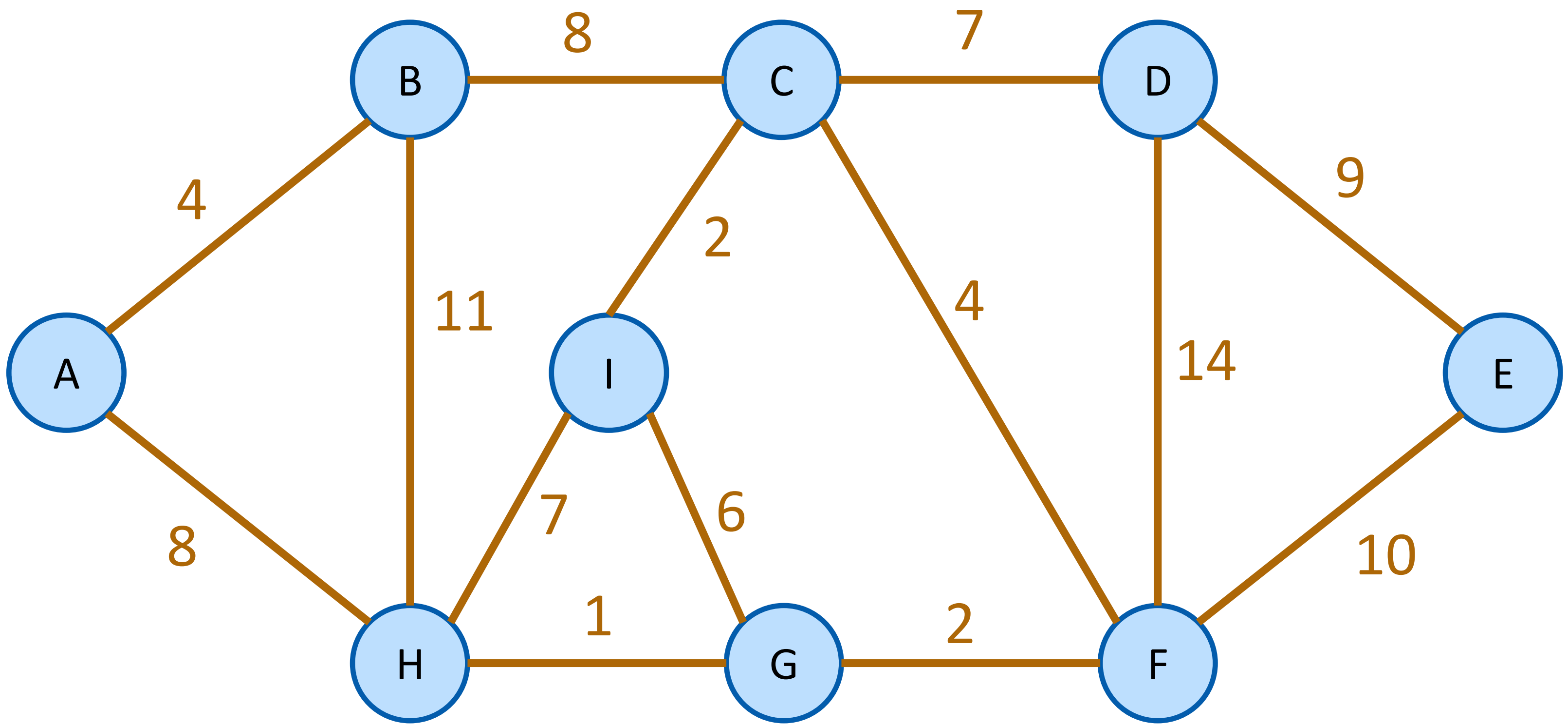


This is a spanning tree with cost 67.

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Trees

- Say we have an undirected weighted graph



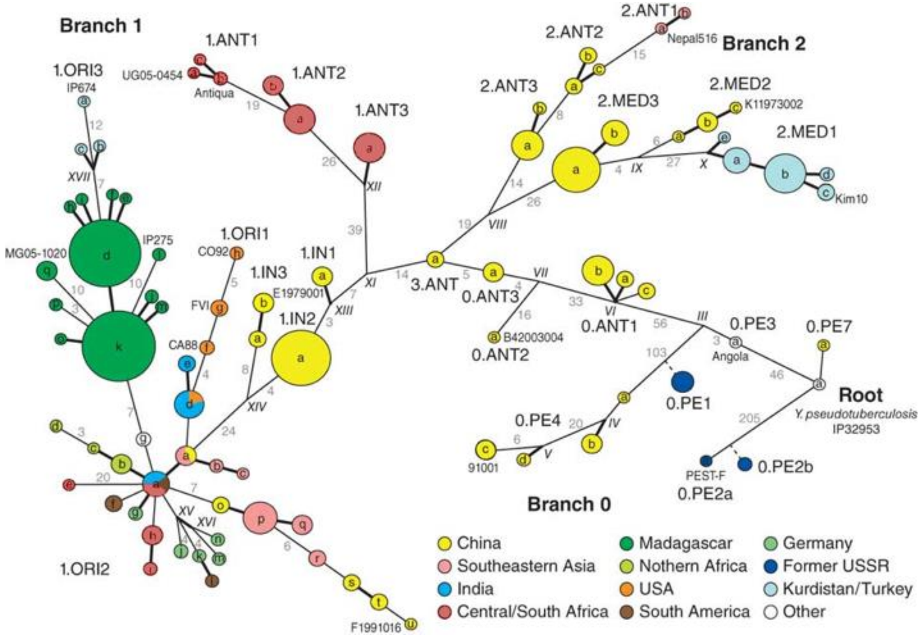
minimum

of minimum cost

A **spanning tree** is a **tree** that connects all of the vertices.

Why MSTs?

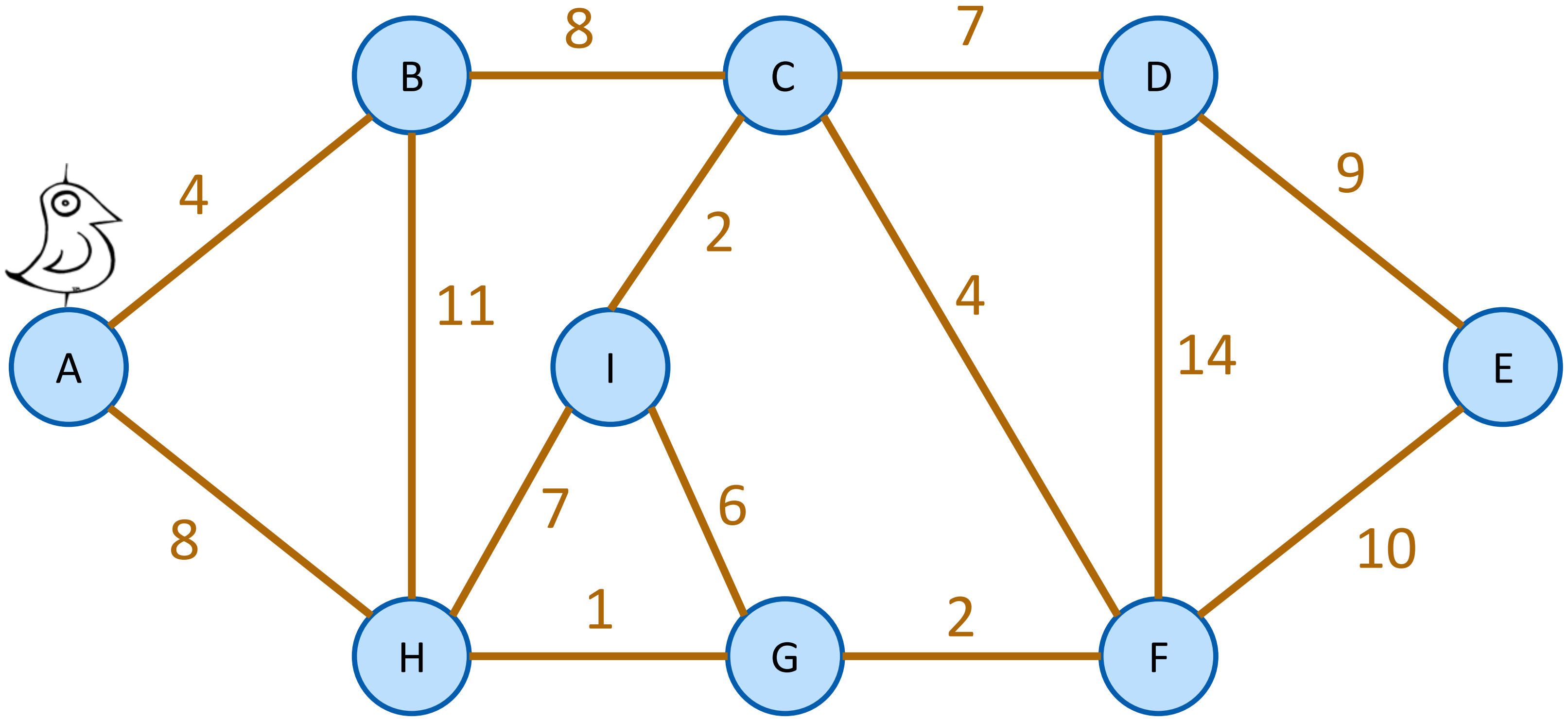
- Network design
  - Connecting cities with roads/electricity/telephone/…
- Cluster analysis
  - E.g., genetic distance
- Image processing
  - E.g., image segmentation
- Useful primitive
  - For other graph algs



How to find an MST

Idea:

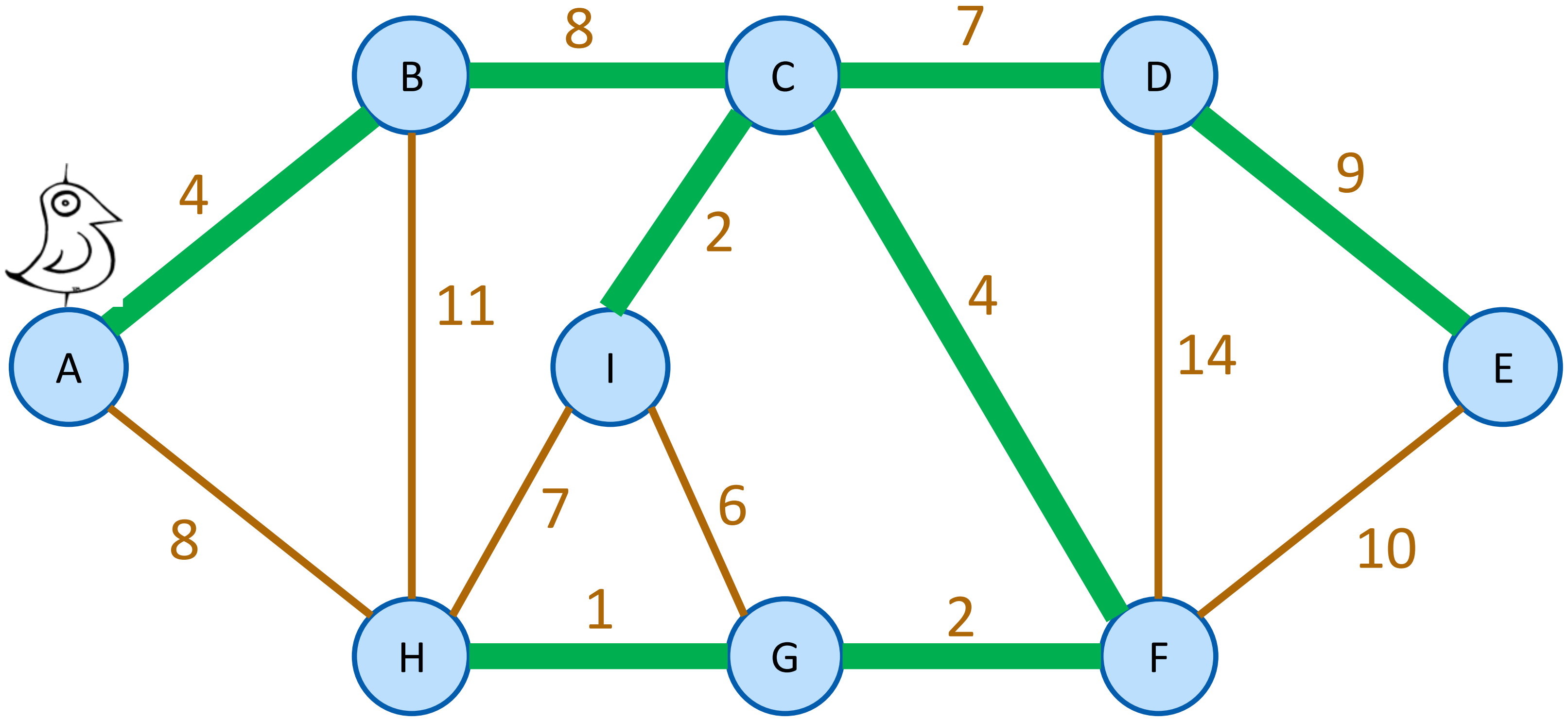
Start growing a tree, greedily add the shortest edge we can to grow the tree.



How to find an MST

Idea:

Start growing a tree, greedily add the shortest edge we can to grow the tree.



We've discovered Prim's algorithm!

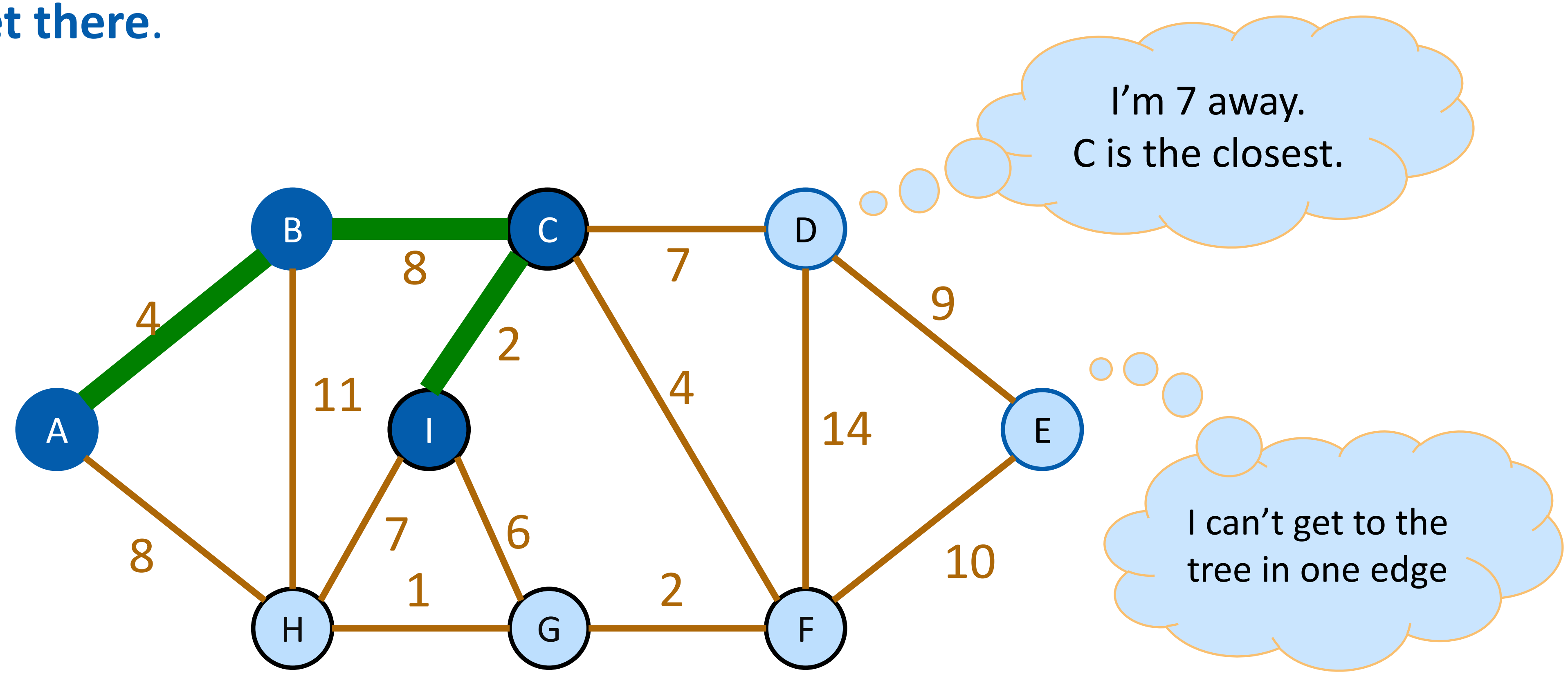
- slowPrim(G = (V,E), starting vertex s):
  - MST = {}
  - verticesVisited = { s }
  - **while** |verticesVisited| < |V|:
    - find the lightest edge {x,v} in E so that:
      - x is in verticesVisited
      - v is not in verticesVisited
    - add {x,v} to MST
    - add v to verticesVisited
  - **return** MST

Naively, the running time is O(nm):

- For each of ≤n-1 iterations of the while loop:
  - Go through all the edges.

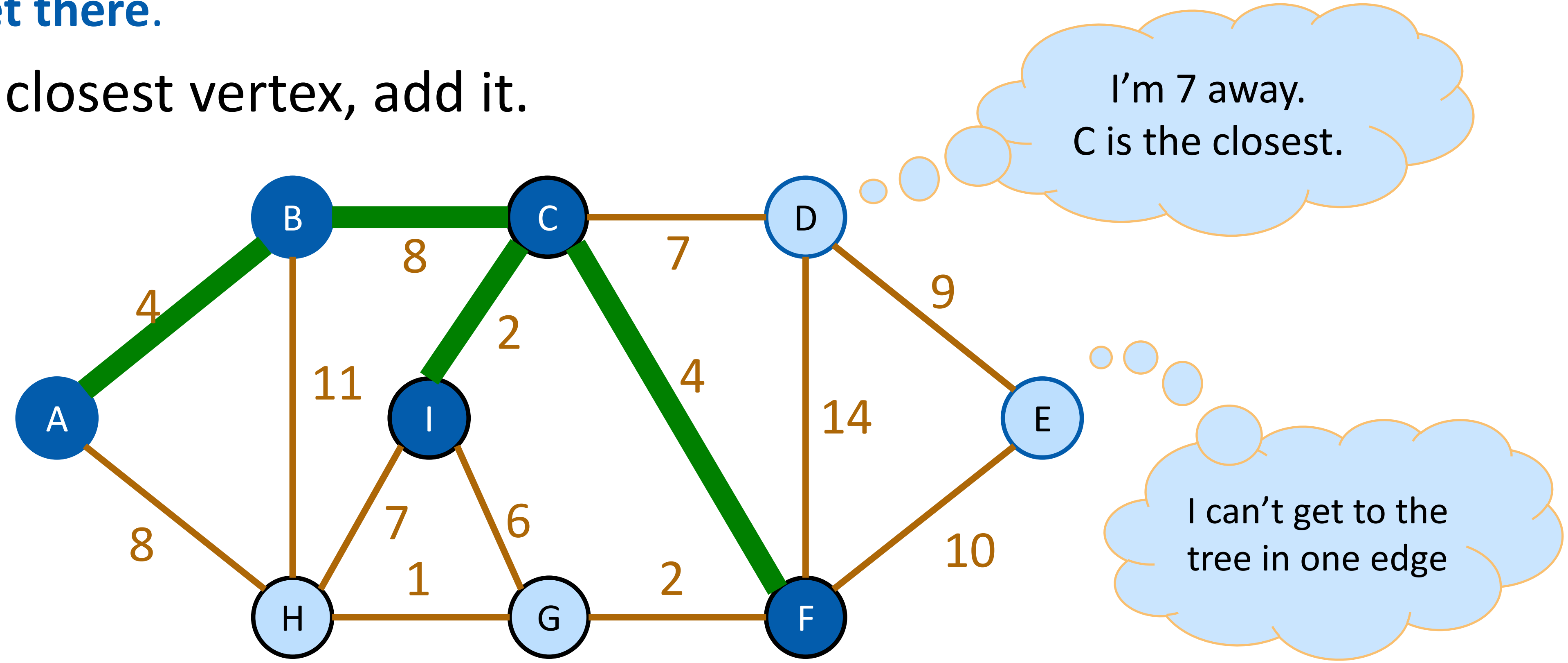
Efficient Implementation

- Each vertex keeps:
  - the **(single-edge) distance** from itself to the **growing spanning tree**
  - **how to get there**.



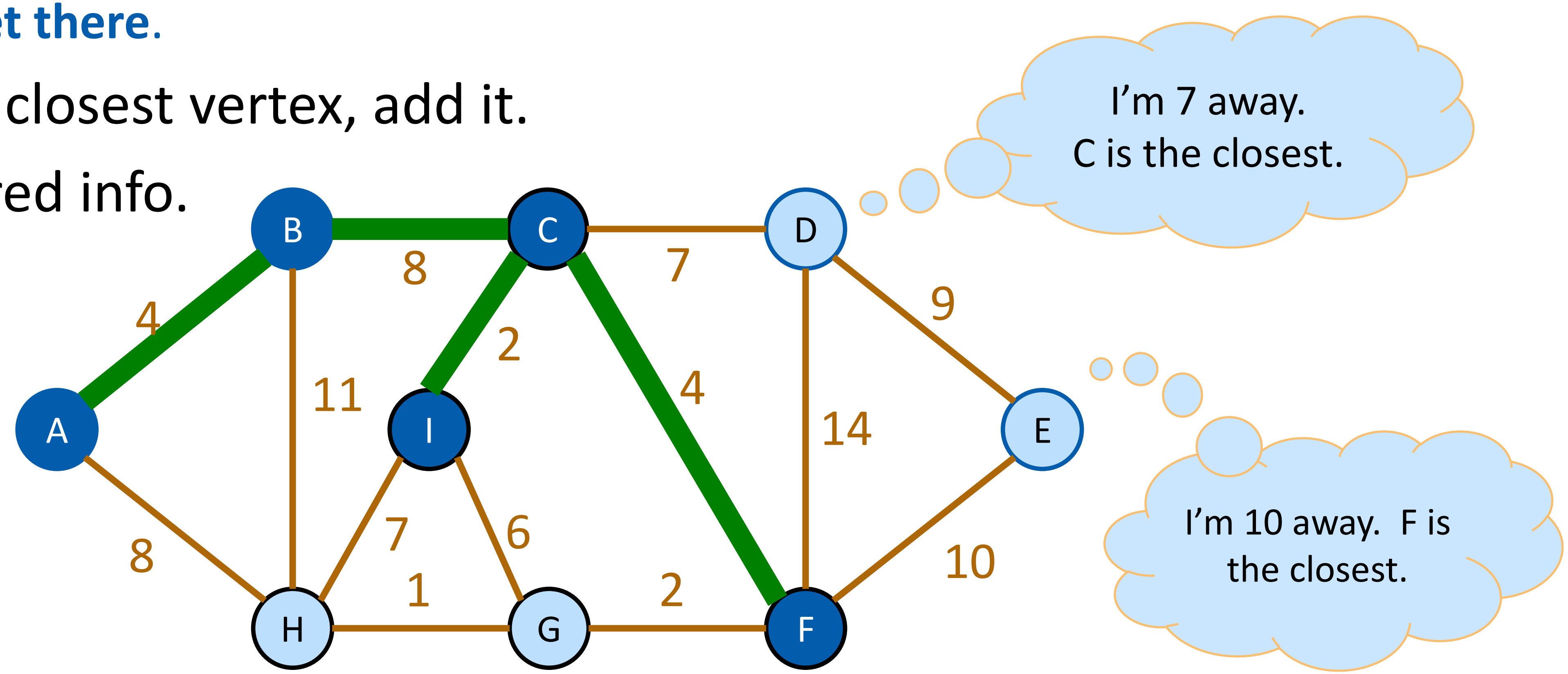
Efficient Implementation

- Each vertex keeps:
  - the **(single-edge) distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.



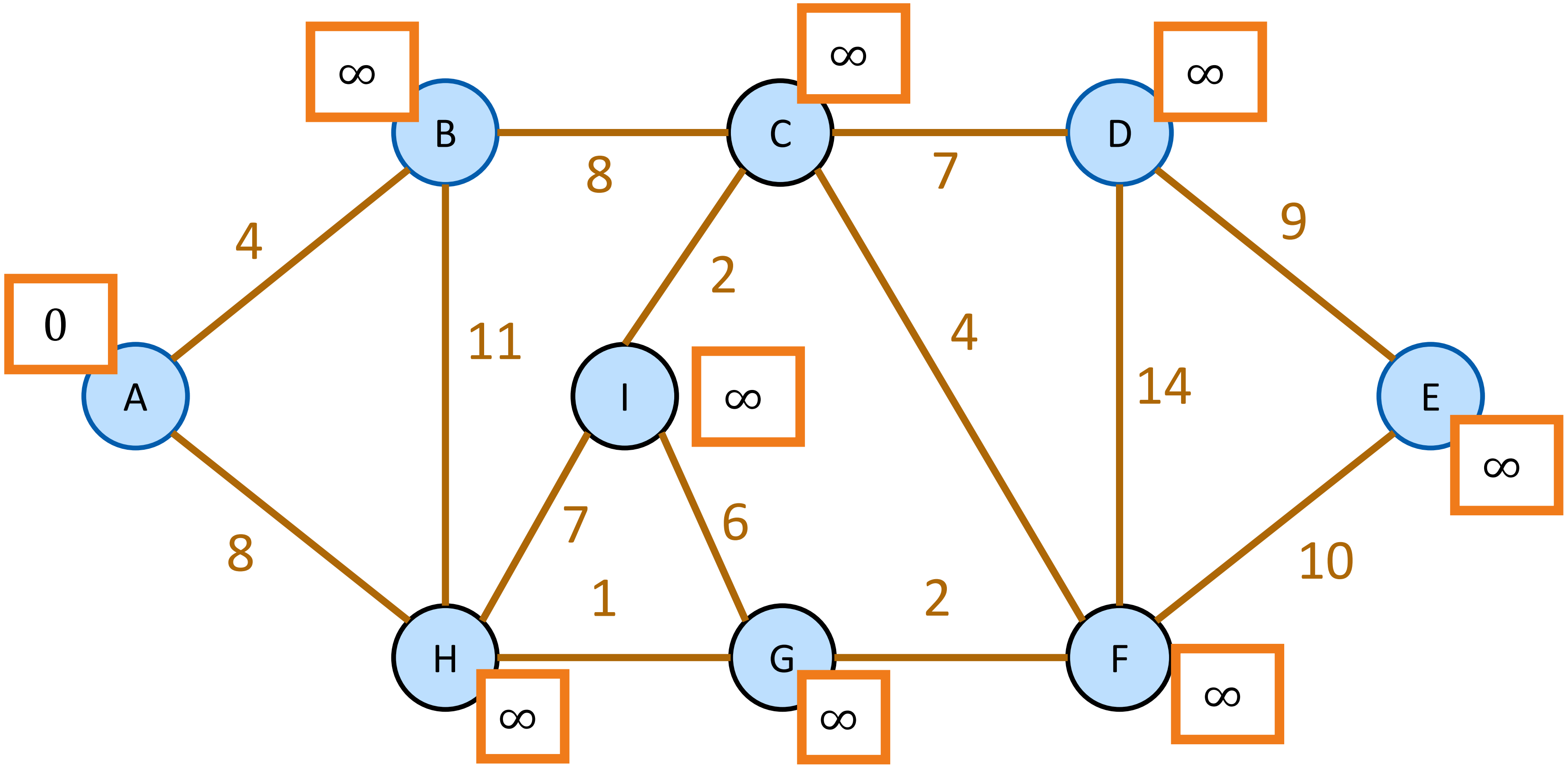
Efficient Implementation

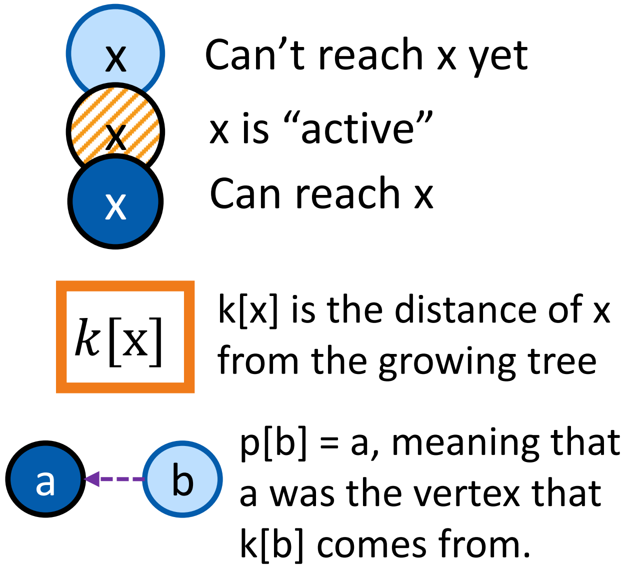
- Each vertex keeps:
  - the **(single-edge) distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.
- Update stored info.



Efficient Implementation

Every vertex has a key and a parent





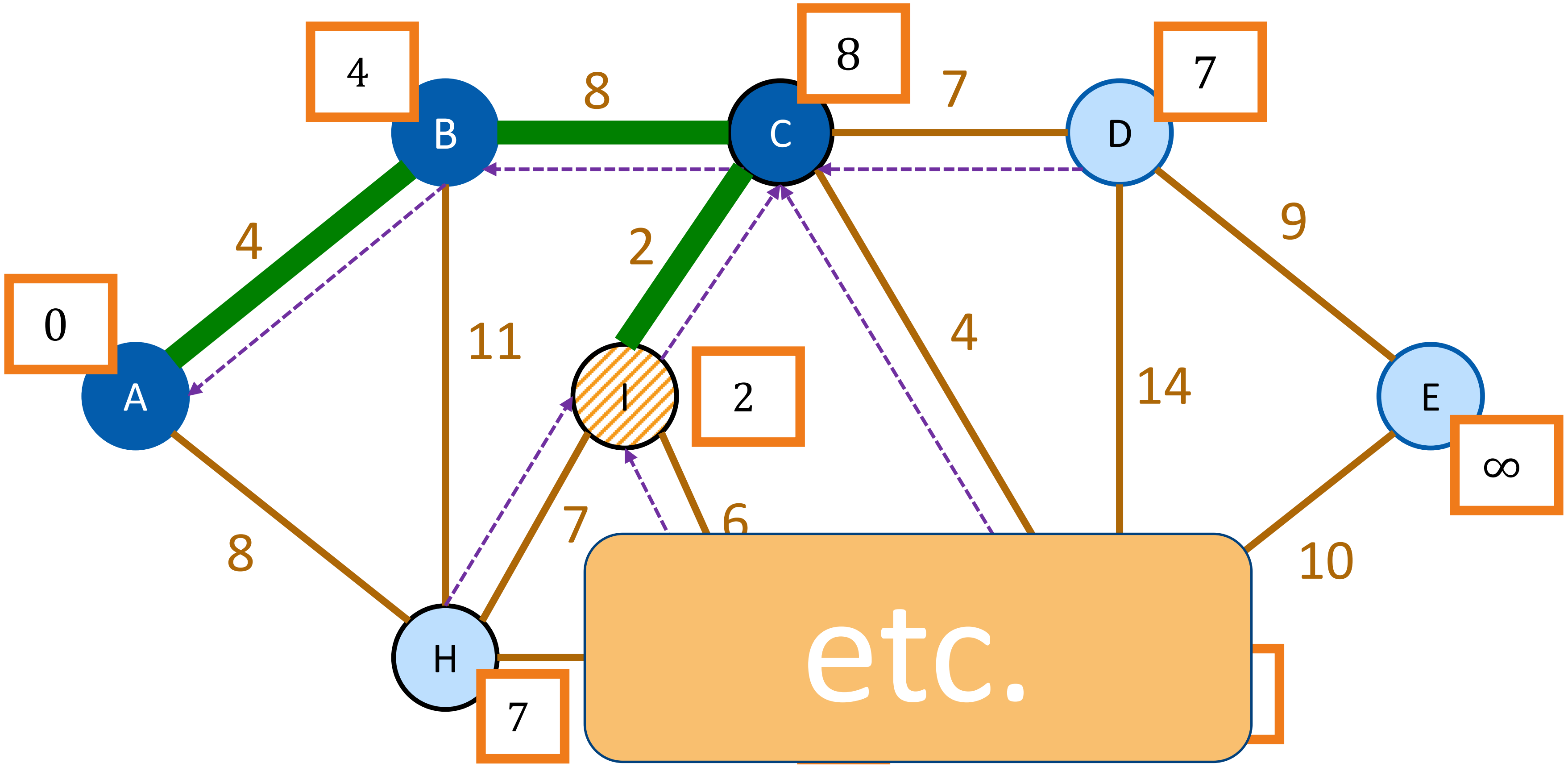
Until all the vertices are **reached:**

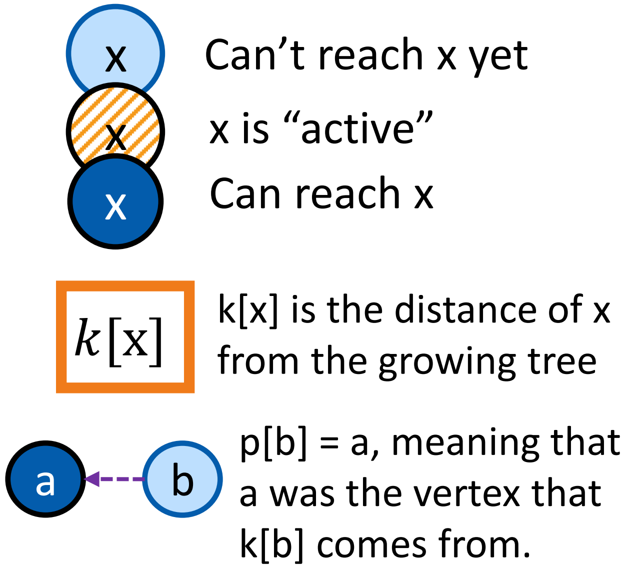
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u’s unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Efficient Implementation

Every vertex has a key and a parent





Until all the vertices are **reached:**

- Activate the unreached vertex u with the smallest key.
- for each of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u

Prim's Algorithm

- Very similar to Dijkstra's algorithm!
- **Differences:**
  1. Keep track of p[v] in order to return a tree at the end
     - But Dijkstra's can do that too, that's not a big difference.
  2. Instead of d[v] which we update by
     - d[v] = min(d[v], d[u] + w(u,v))

     we keep k[v] which we update by
     - k[v] = min(k[v], w(u,v))

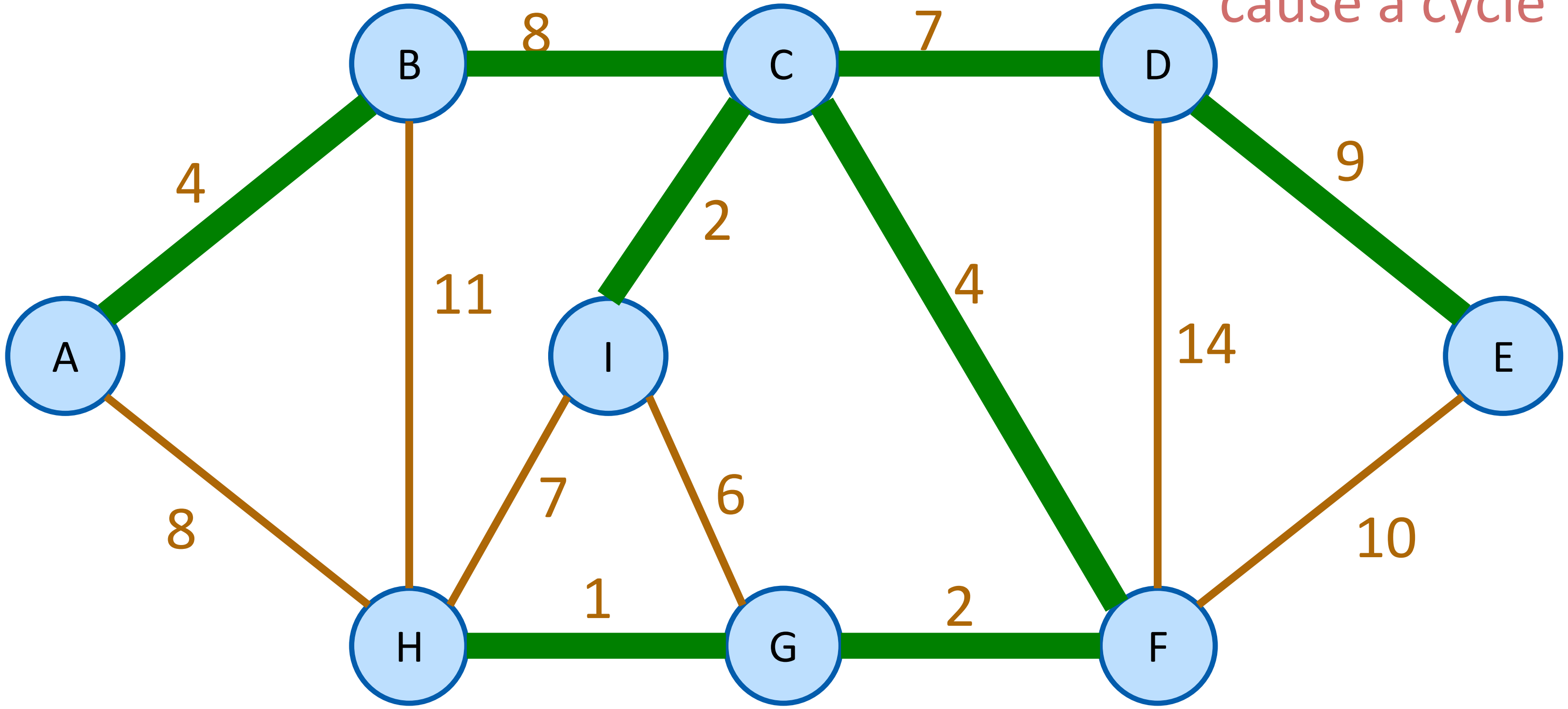
Thing 2 is the main difference.

Kruskal's Algorithm

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

That won't cause a cycle



Kruskal’s Algorithm

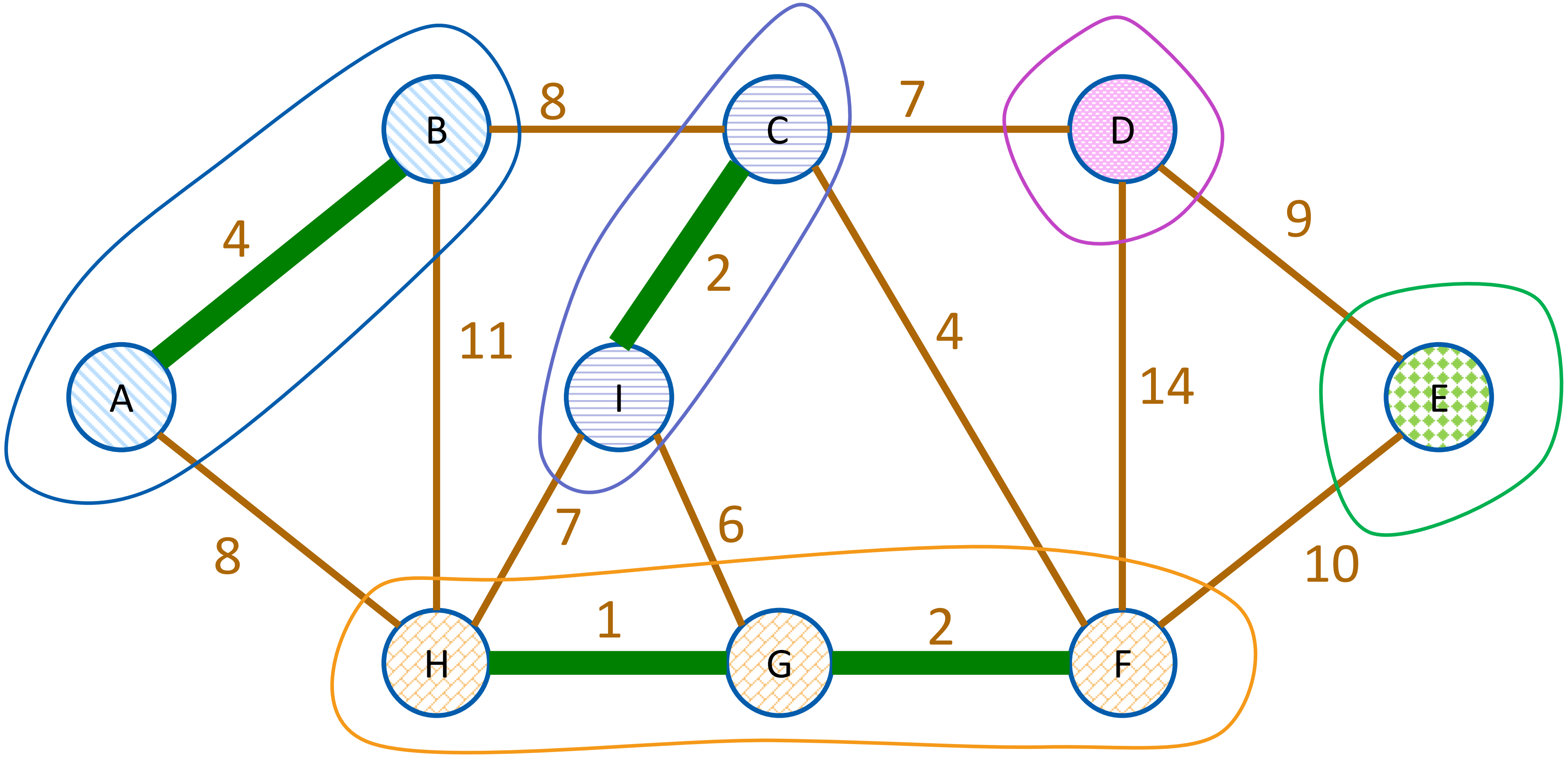
- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won’t cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?

Kruskal's Algorithm

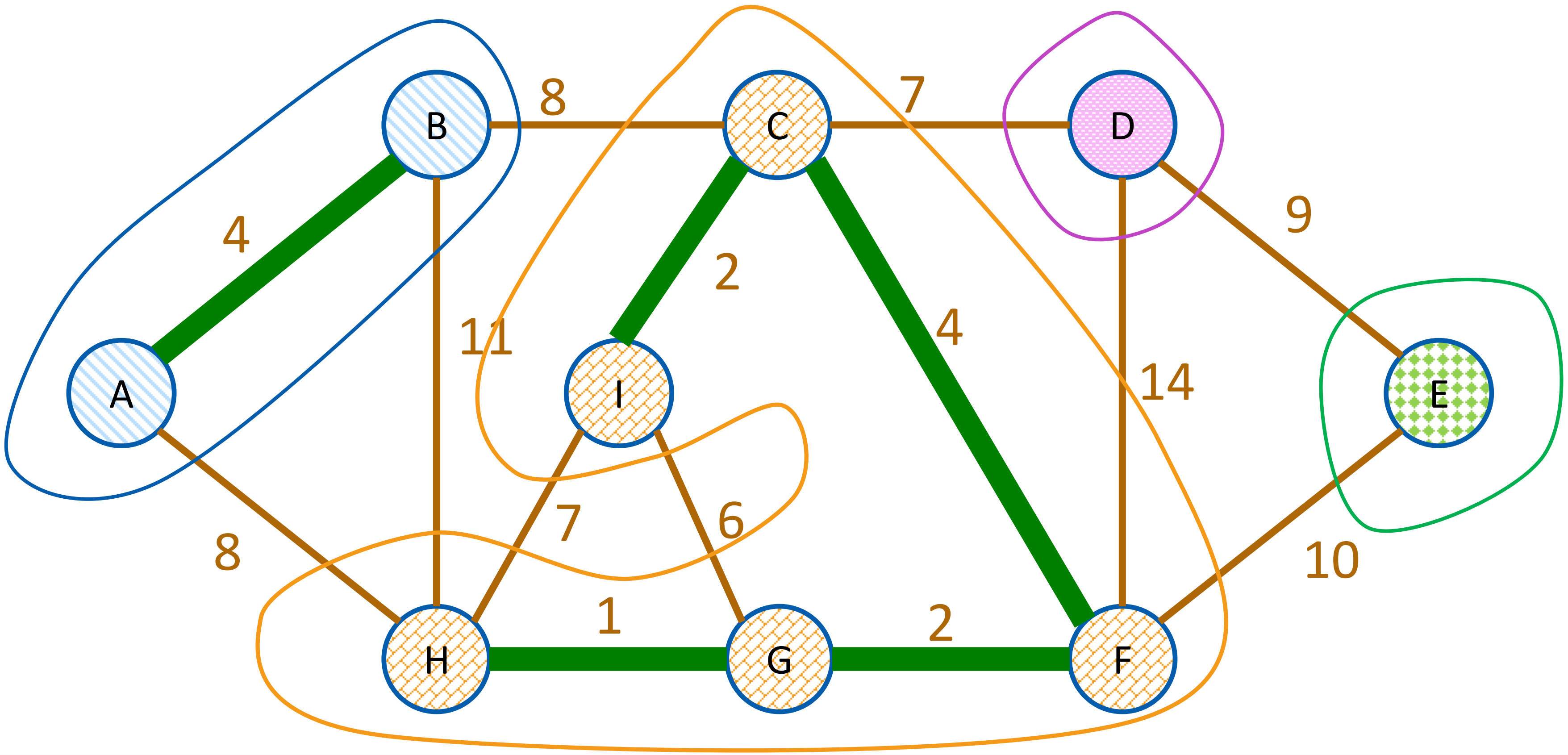
At each step of Kruskal's, we are maintaining a forest.



Kruskal's Algorithm

At each step of Kruskal's, we are maintaining a forest.

When we add an edge, we merge two trees:



Kruskal's Algorithm

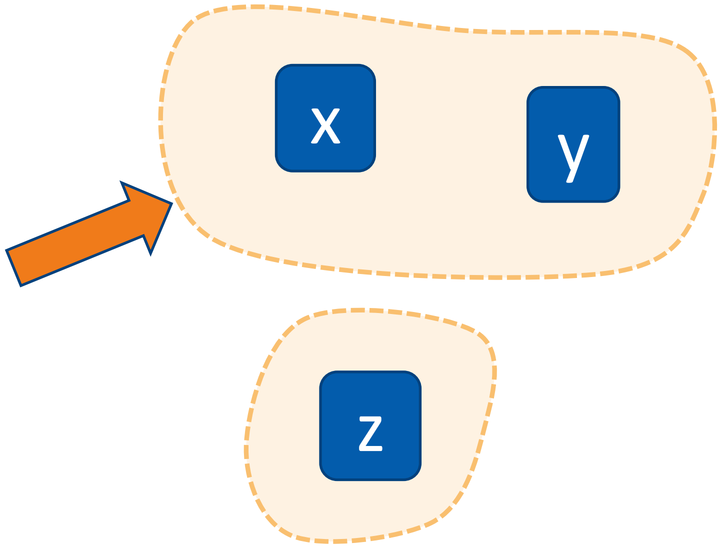
Union-find data structure

Implementation – Lab 13

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
find(x)
```



Kruskal's Algorithm

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

Kruskal's Algorithm

Running time

- Sorting the edges takes O(m log(n))
  - In practice, if the weights are small integers we can use radixSort and take time O(m)
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - 2m calls to **find**
    - for each edge, **find** its endpoints
  - n-1 calls to **union**
    - we will never add more than n-1 edges to the tree,
    - so we will never call **union** more than n-1 times.
- Total running time: **O(mlog(n))**

Complexity Classes

P and NP

- Definition: The class P consists of all decision problems that are solvable in polynomial time

- Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate that can be verified in polynomial time.
  – NP stands for "Nondeterministic Polynomial time".

- P = NP?

The End