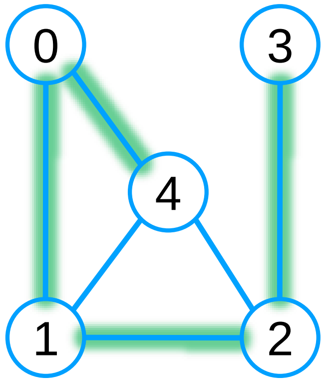
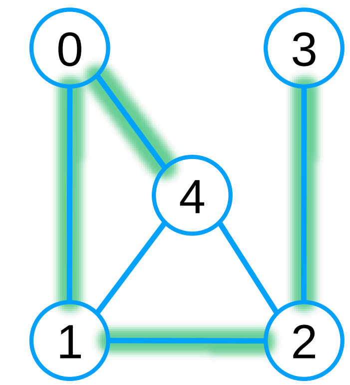
Union-find



Review

- Spanning trees
  - Edge-centric algorithm: O(ev)
  - Vertex-centric algorithm: O(v + e) Clear winner
- Minimum spanning trees
  - Kruskal's algorithm: O(ev)
  - Prim's algorithm: O(v + e log e) Clear winner



Review

Kruskal's Algorithm

Given a graph G, construct a **minimum spanning tree** T for it

0. Sort the edges of G by increasing weight — O(e log e)
1. Start T with the isolated vertices of G — O(v)
2. For each edge (u,v) in G — e times
   - *are u and v already connected in T?* — O(v)
     - **yes**: discard the edge
     - **no**: add it to T — O(1)
   - Stop once T has v-1 edges

O(ev)

- *Can we do better?*

Today's lecture

Towards Union-find

Opportunities for Improvement

Given a graph G, construct a minimum spanning tree T for it

0. Sort the edges of G by increasing weight O(e log e)
1. Start T with the isol ted vertices of G O(v)
2. For each edge (u,v n G e times
   - *are u and v alread nnected in T?* O(v)
     - **yes**: discard the e
     - **no**: add it to T O(1)
   - Stop once T has es

O(n log n) is the complexity of the **problem** of sorting n elements: **no** (sequential) **algorithm can do better**

Opportunities for Improvement

Given a graph G, construct a minimum spanning tree T for it

0. Sort the edges of G by increasing weight	O(e log e)
1. Start T with the isolated vertices of G	O(v)
2. For each edge (u,v) in G	e times
○ *are u and v already connected in T?*	O(v)
➢ **yes**: discard he edge	
➢ **no**: add it to	O(1)
○ Stop once T s v-1 edges	

In general, there is no way around examining every edge in G

Opportunities for Improvement

Given a graph G, construct a minimum spanning tree T for it

0. Sort the edges of G by increasing weight O(e log e)
1. Start T with the isolated vertices of G O(v)
2. For each edge (u,v) in G e times
   - *are u and v already connected in T?* O(v)
     - **yes**: discard the edge
     - **no**: add it to T O(1)
   - Stop once T has v-1 edges

Everything else is O(1)

- *Can we check that u and v are connected in **less than O(v) time**?*

Checking Connectivity

- *are u and v already connected in T?* O(v)

- We use BFS or DFS to check connectivity
  - O(v) is the complexity of the **problem** of checking connectivity on a tree
    - no algorithm can do better than O(v)

- BFS and DFS assume u and v are **vertices we know nothing about**
  - arbitrary vertices in an arbitrary tree

… but **we** put them in T in an earlier iteration

- we know a lot about them!

Checking Connectivity

- *are u and v already connected in T?* O(v)

Let's reframe the question as

Are u and v in the same connected component?

- If we have an efficient way to know
  - in what connected components u and v are, and
  - if these connected components are the same

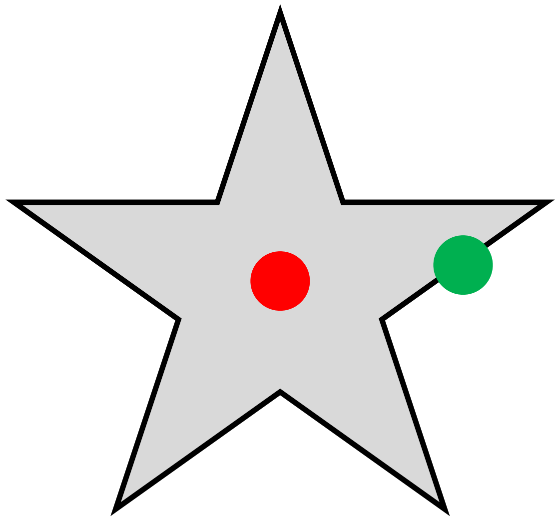
  we have an efficient way to check if u and v are connected

Identifying Connected Components

- We are looking for an efficient way to know
  - in what connected components u and v are, and
  - if these connected components are the same

Idea:

- Appoint a **canonical representative** for each component
  - some vertex that represents the whole connected component
- Arrange that we can easily find the canonical representative of (the connected component of) any vertex



Kruskal's Algorithm Revisited

Given a graph G, construct a minimum spanning tree T for it

0. Sort the edges of G by increasing weight
1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *are u and v already connected in T?*

     find their canonical representatives, and
     check if they are equal
     - **yes**: discard the edge
     - **no**: add it to T

       merge the two connected component by taking their union, and
       appoint a new canonical representative for the merged component
   - Stop once T has v-1 edges

Union-find

- ○ *are u and v already connected in T?*

  find *their canonical representatives and and check if they are equal*

  - **yes**: discard the edge
  - **no**: add it to T

    merge the two connected component by taking their **union**, and appoint a new canonical representative for the merged component

- This algorithm is called **union-find**

- *Let's implement it*

  … in better than O(v) complexity

Equivalences

Connectedness, Algebraically

- *"u and v are connected"* is a relation between vertices
  - let's write it u ### v
- As a relation, what properties does it have?
  - **reflexivity**: u ### u
  - **symmetry**: if u ### v, then v ### u
  - **transitivity**: if u ### v and v ### w, then u ### w

Every vertex is connected to itself
(by a path of length 0)

If u is connected to v,
then v is connected to u
(by the reverse path)

If u is connected to v
and v is connected to w,
then v is connected to v
(by the combined path)

- It is an **equivalence relation**
- A connected component is then an **equivalence class**

Checking Equivalence

- Given any equivalence relation, we can use union-find to check if two elements x and y are equivalent
  - find the canonical representatives of x and y and check if they are equal

- For this, we need to represent the equivalence relation in such a way we can use union-find
  - appoint a canonical representative for every equivalence class
  - provide an easy way to find the canonical representative of any element

How to do this?

Basic Union-find

Back to the Edge-centric Algorithm

- Recall the edge-centric algorithm for unweighted graphs
  - instrumented to use union-find

Given a graph G, construct a spanning tree T for it

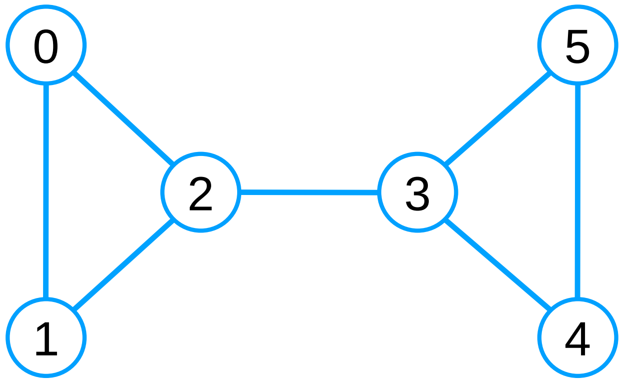
1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *are u and v already connected in T?*
     find their canonical representatives, and check if they are equal
     - **yes**: discard the edge
     - **no**: add it to T
       merge the two connected component by taking their union, and appoint a new canonical representative for the merged component
   - Stop once T has v-1 edges

This is Kruskal's algorithm without the preliminary edge-sorting step

Example

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges

- We will use it to compute a spanning tree for this graph

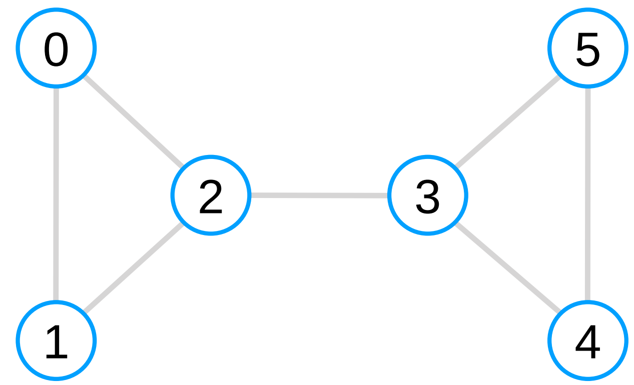


considering the edges in this order

Edges
(4, 5)
(3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

The Union-find Data Structure

- We start with a forest of isolated vertices



- We need a data structure to keep track of the canonical representative of every vertex
  - an *array* UF with a position for every vertex
    - UF[v] contains the canonical representative of v
      - or a way to get to it
  - this is the **union-find data structure**

UF:

0	1	2	3	4	5

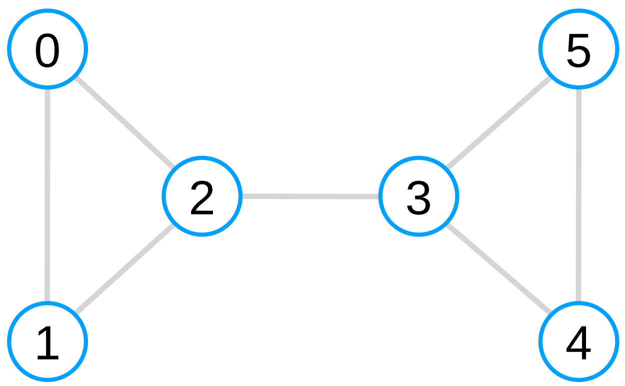
- Initially, every vertex is its own canonical representative

UF[v] = v

0	1	2	3	4	5
0	1	2	3	4	5

Initial Configuration

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



The spanning tree so far

Edges

(4, 5)
(3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

We will consider this edge next

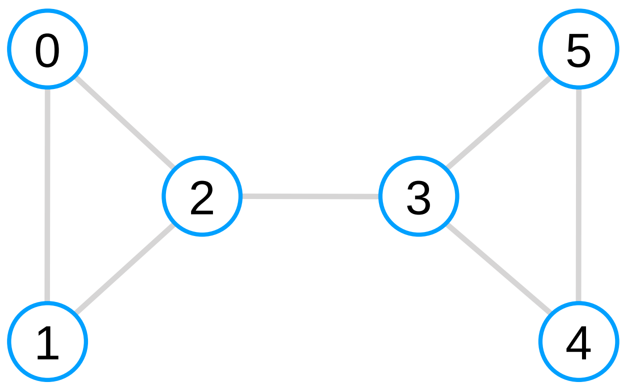
0	1	2	3	4	5
0	1	2	3	4	5

The union-find data structure at this point

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges

First Step

0	1	2	3	4	5
0	1	2	3	4	5



Edges

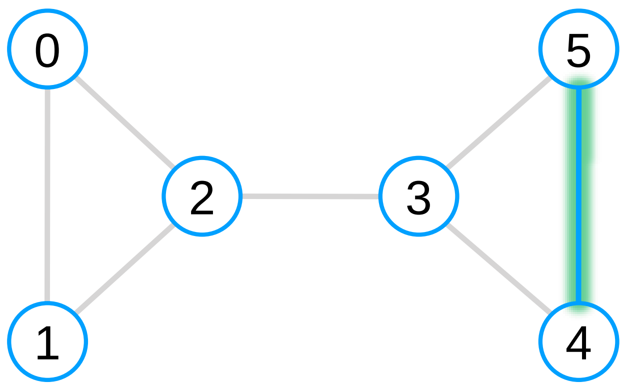
(4, 5)
(3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

We consider this edge

- the canonical representative of 4 is 4
- the canonical representative of 5 is 5
- 4 ≠ 5, so we add (4, 5) to the tree

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges

First Step



Edges

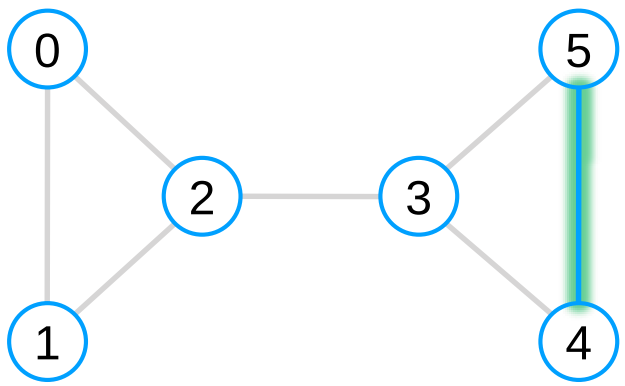
- ✓ (4, 5)
- (3, 5)
- (1, 2)
- (3, 4)
- (2, 3)
- (0, 2)
- (0, 1)

0	1	2	3	4	5
0	1	2	3	4	5

- 4 and 5 are now in the same connected component
  - which one should we appoint as the new canonical representative?
  - either of them will do
    - let's pick 4

First Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



Edges
✓ (4, 5)
(3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

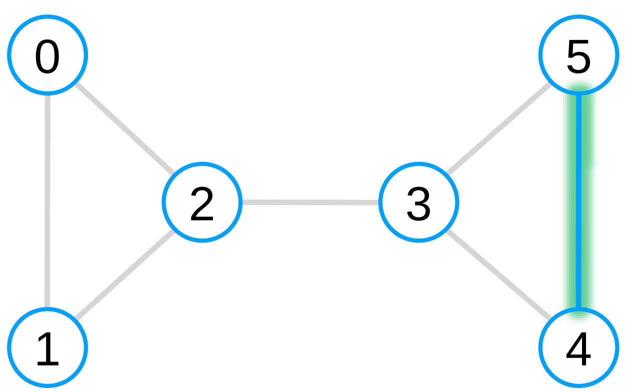
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**

Updated union-find data structure

- 4 and 5 are now in the same connected component
  - which one should we appoint as the new canonical representative?
  - either of them will do
    - let’s pick 4

Second Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



Edges

✓ (4, 5)
(3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

We consider this edge

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**

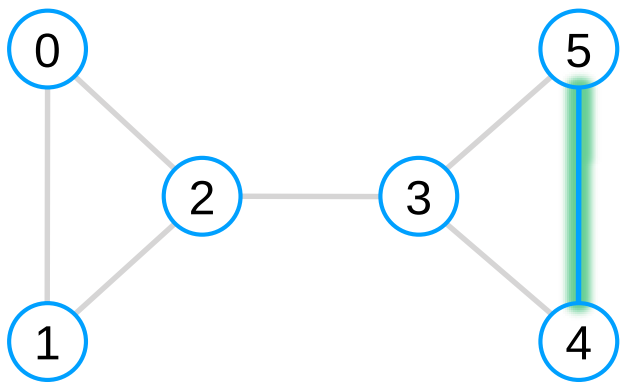
- the canonical representative of 3 is 3
- the canonical representative of 5 is 4
- 3 ≠ 4, so we add (3, 5) to the tree

Chasing canonical representatives in an array is fine for computers but it's hard for humans.

Let's visualize the union-find data structure in a more intuitive way

Second Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



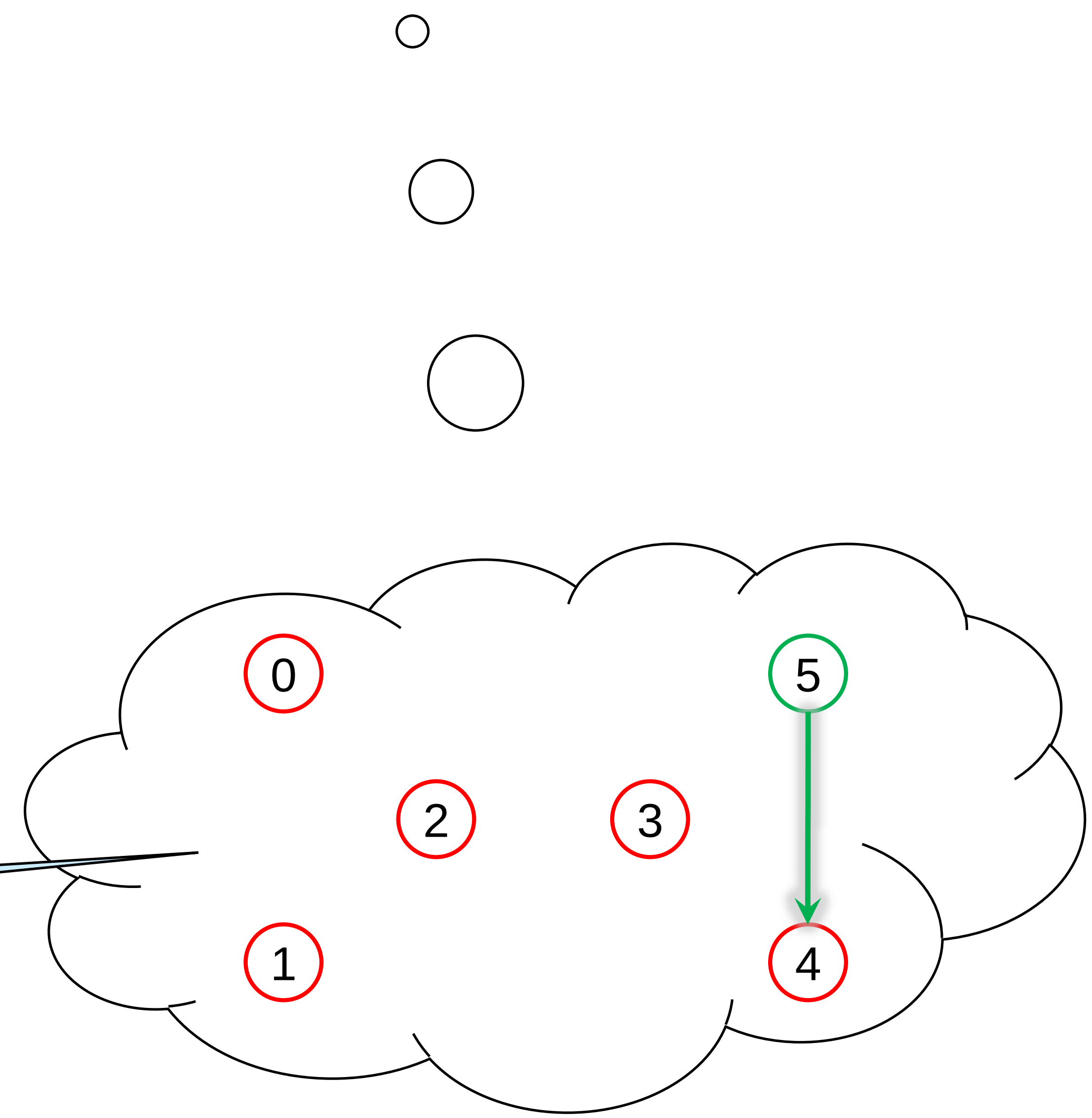
Edges

- ✓ (4, 5)
- **(3, 5)**
- (1, 2)
- (3, 4)
- (2, 3)
- (0, 2)
- (0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**

- This visualizes the union-find data structure in a more intuitive way
  - there is an edge from u to v if UF[u] = v

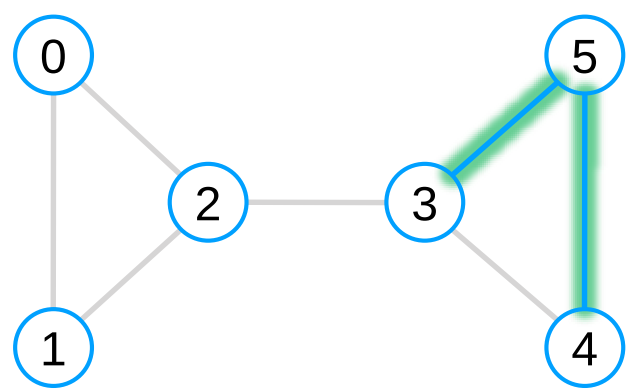
This is a **directed** graph



Second Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges

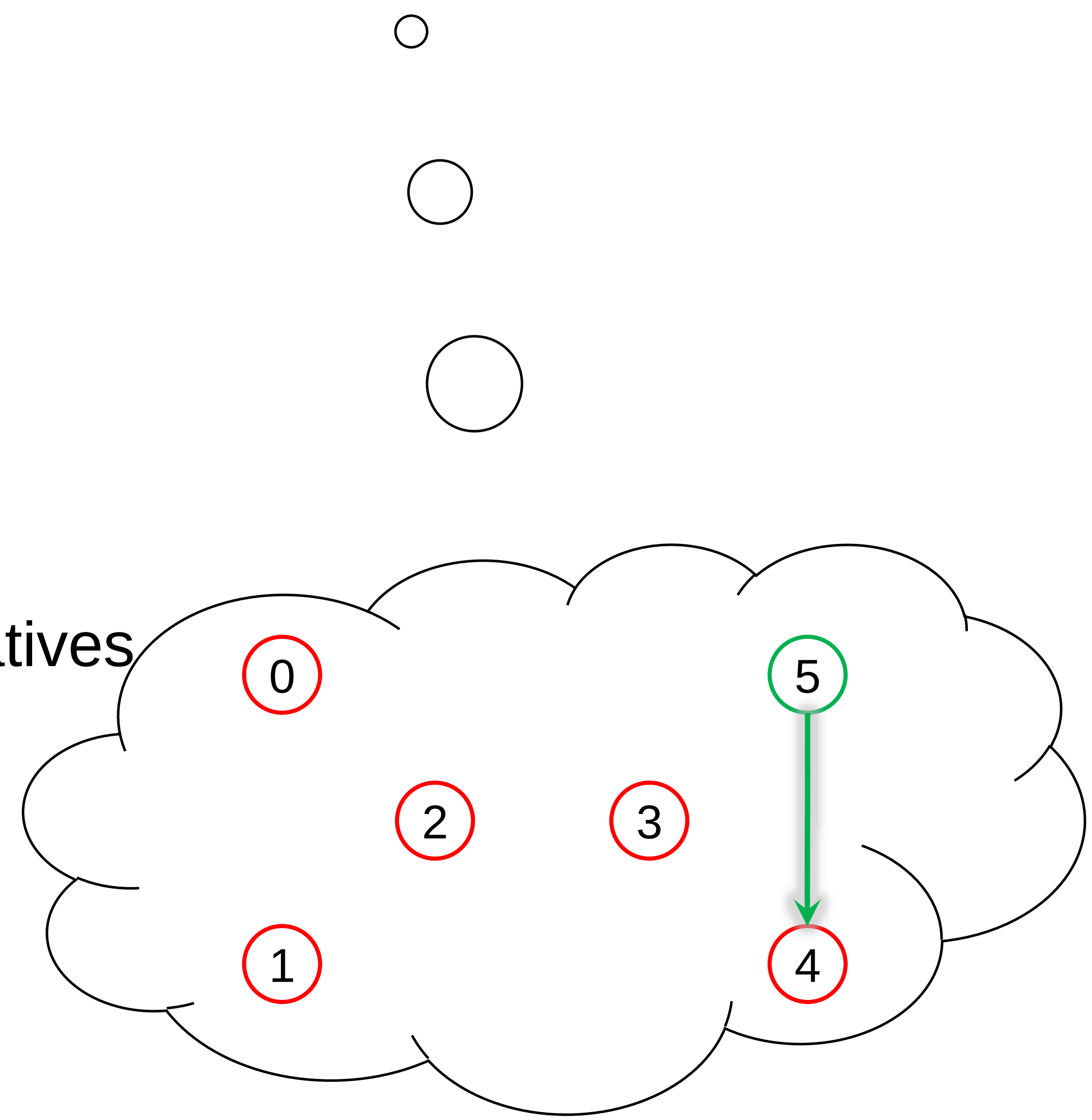
- Who should the new canonical representative be?



Edges

✓ (4, 5)
✓ (3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

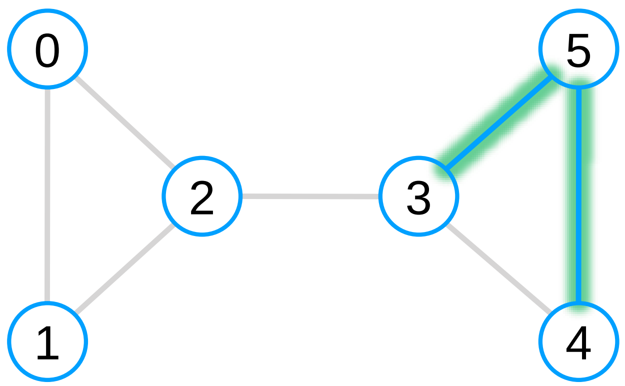
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**



- 5?
  - this forces us to change UF[4] and UF[5]
    - and possibly many more in a larger graph
  - We want to pick one of the old representatives
- 3?
  - This will do
- 4?
  - This would do too

Third Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



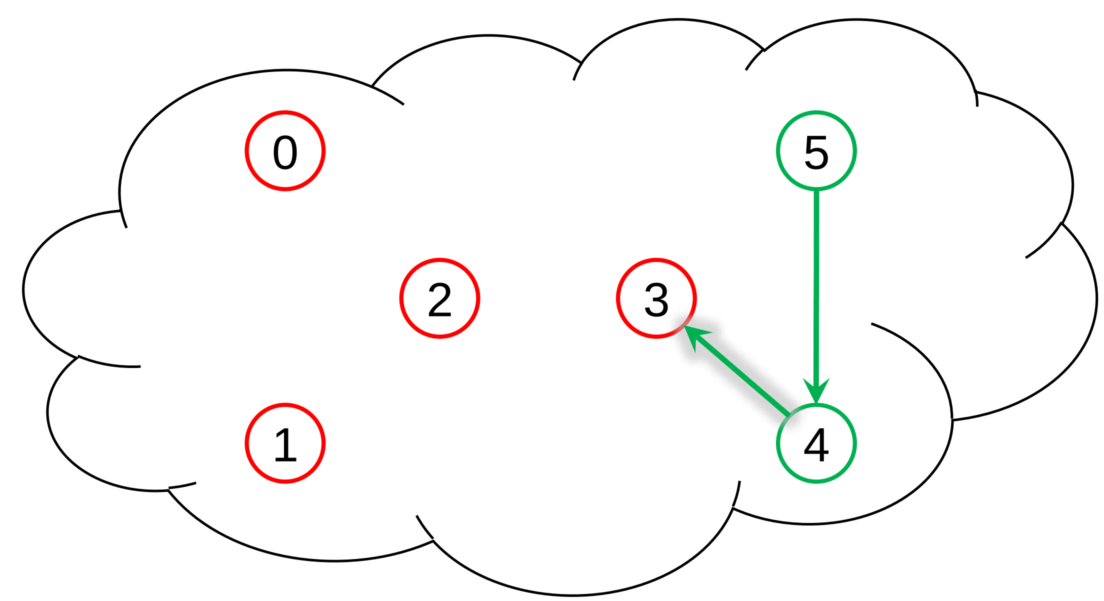
Edges

✓ (4, 5)
✓ (3, 5)
(1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4

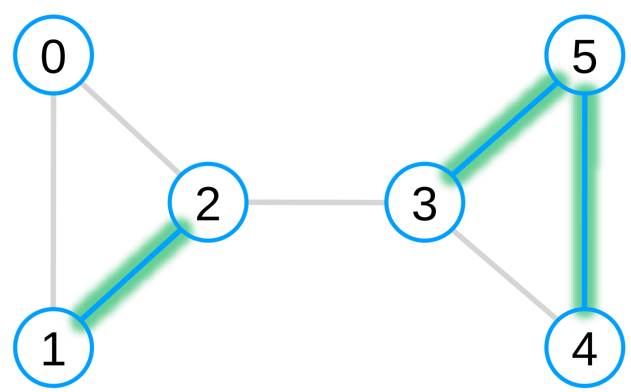
Note that 4 is **not** the canonical representative of 5: *but it's way to get to it*

- 1 and 2 are their own canonical representatives
  - we add the edge (1,2)
  - we appoint 1 as the new canonical representative



Fourth Step

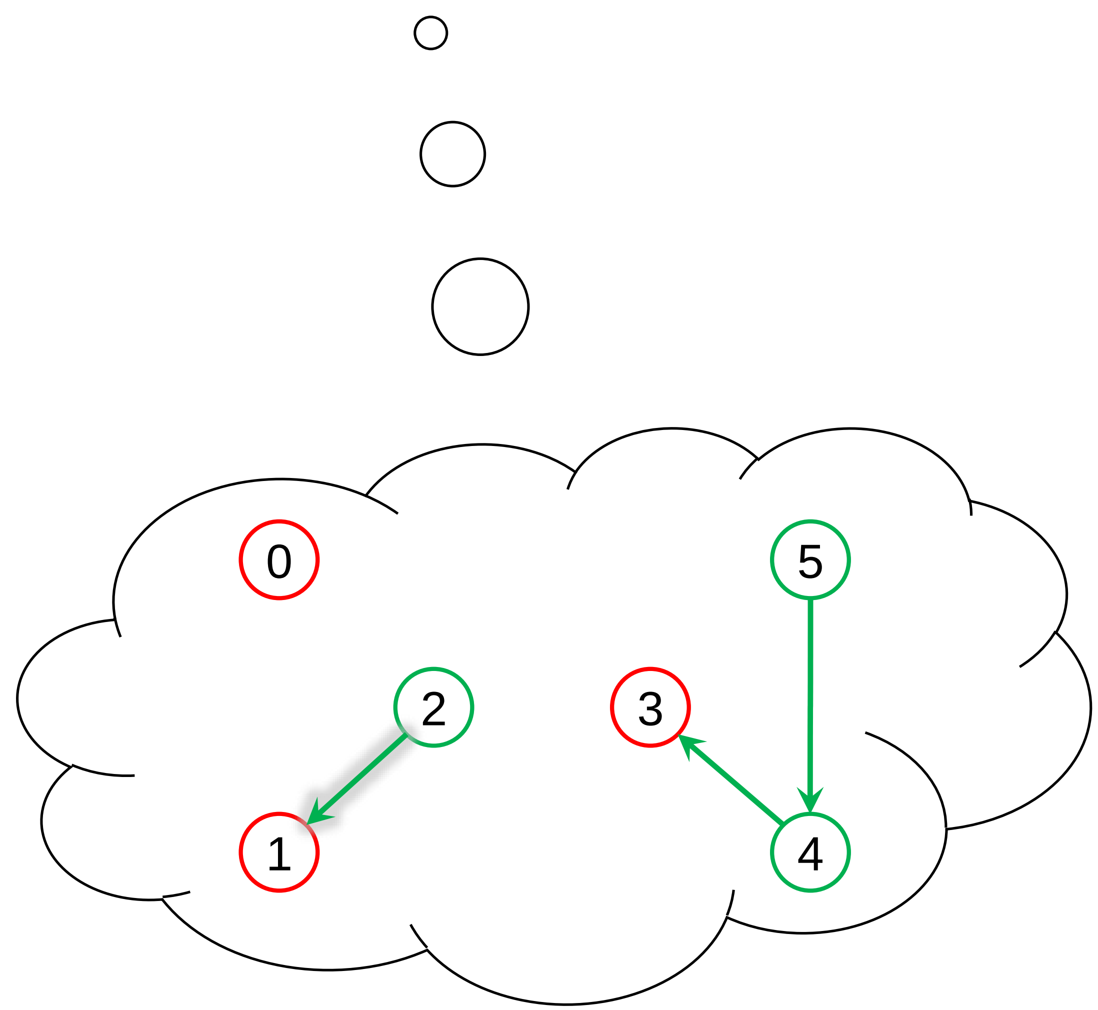
1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



Edges
✓ (4, 5)
✓ (3, 5)
✓ (1, 2)
(3, 4)
(2, 3)
(0, 2)
(0, 1)

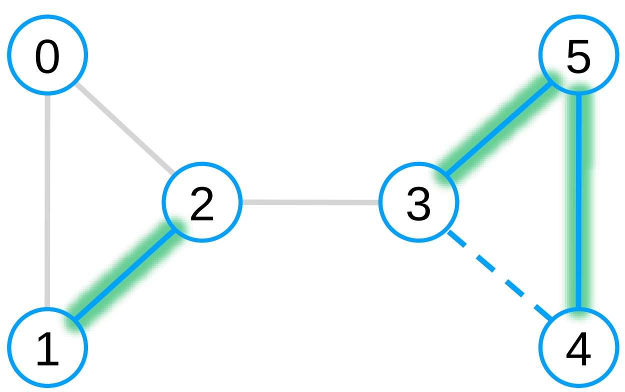
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4

- 3 and 4 have the **same** canonical representative
  - 3
  - we discard the edge (3,4)



Fifth Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



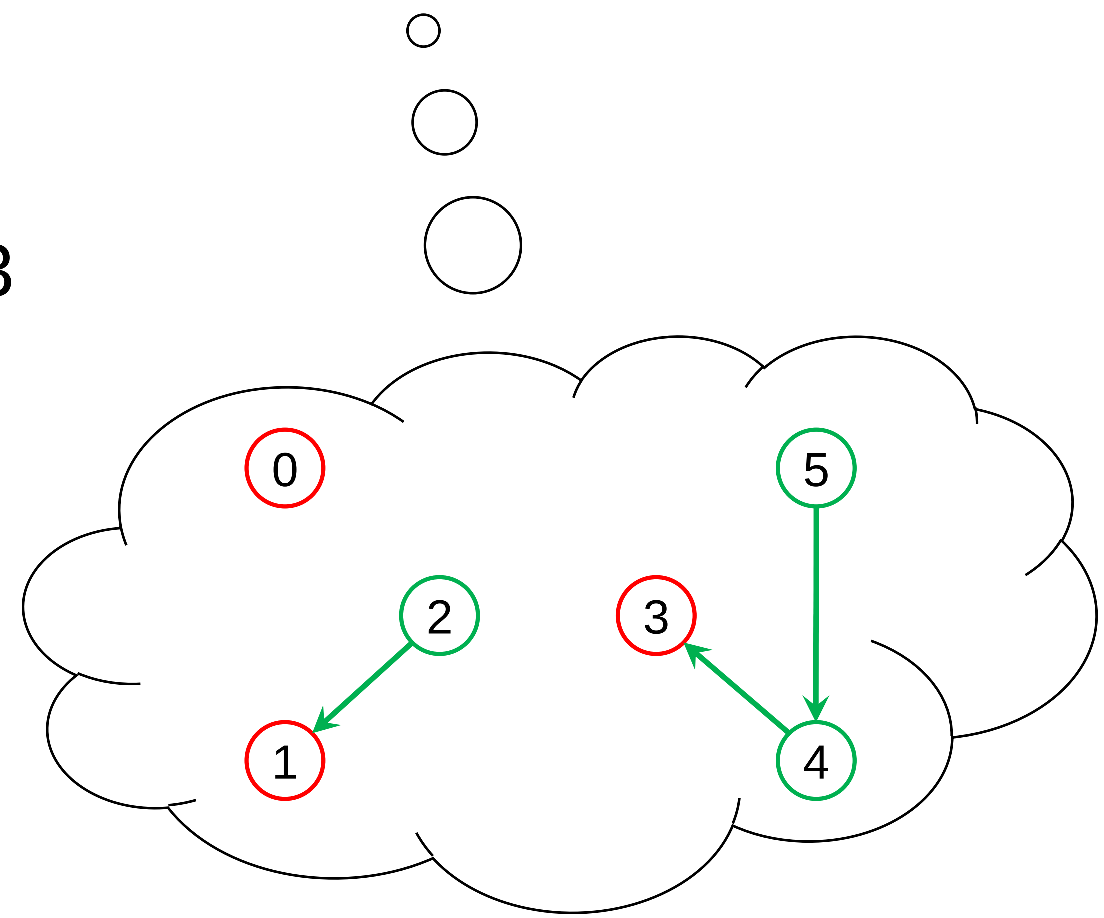
Edges

- ✓ (4, 5)
- ✓ (3, 5)
- ✓ (1, 2)
- × (3, 4)
- **(2, 3)**
- (0, 2)
- (0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4

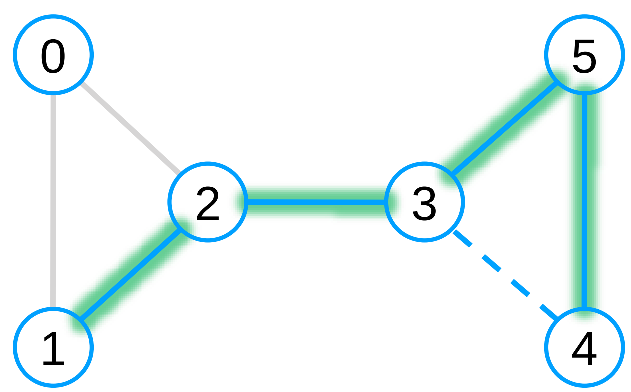
- the canonical representative of 2 is 1
- the canonical representative of 3 is 3
- so we add the edge (2,3)

- The new canonical representative is one among 1 and 3
  - let's pick 1



Sixth Step

1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges

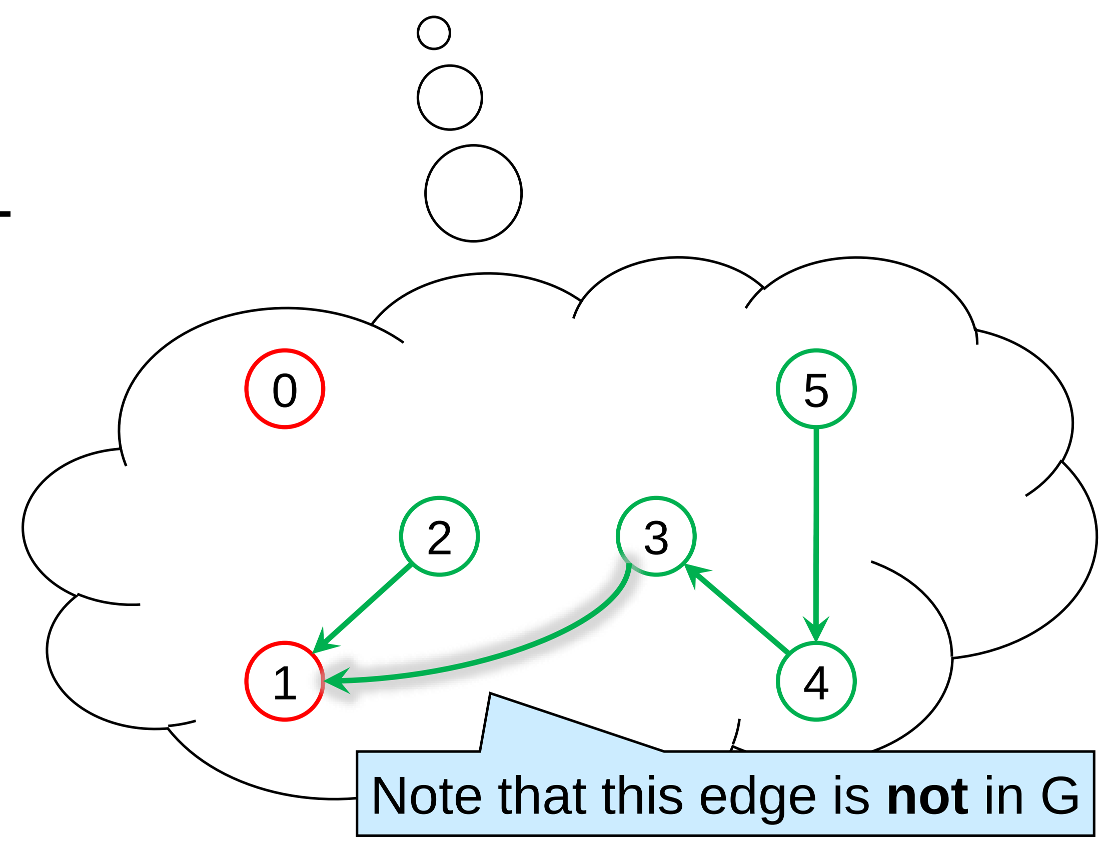


Edges

- ✓ (4, 5)
- ✓ (3, 5)
- ✓ (1, 2)
- × (3, 4)
- ✓ (2, 3)
- **(0, 2)**
- (0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4

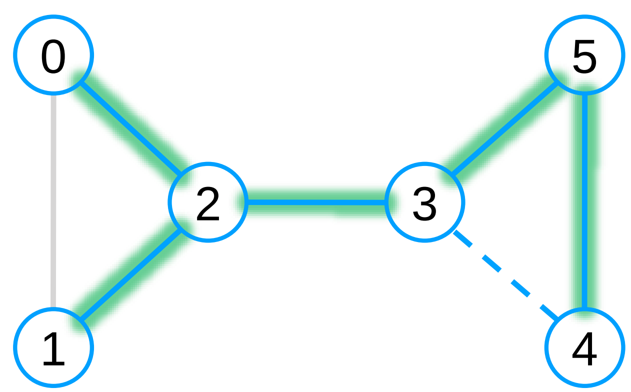
- 0 is its own canonical representative
- the canonical representative of 2 is 1
- so we add the edge (0,2)



- The new canonical representative is one among 0 and 1
  - let's pick 0

Last Step

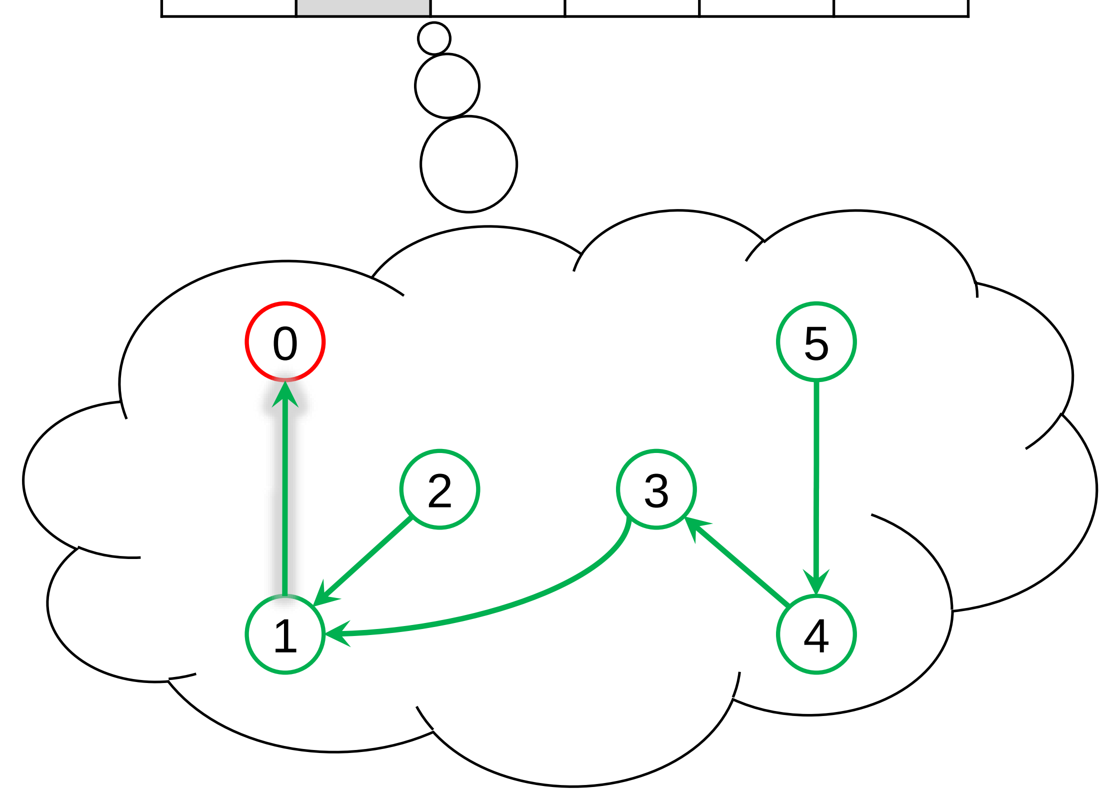
1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



Edges

✓ (4, 5)
✓ (3, 5)
✓ (1, 2)
× (3, 4)
✓ (2, 3)
✓ (0, 2)
(0, 1)

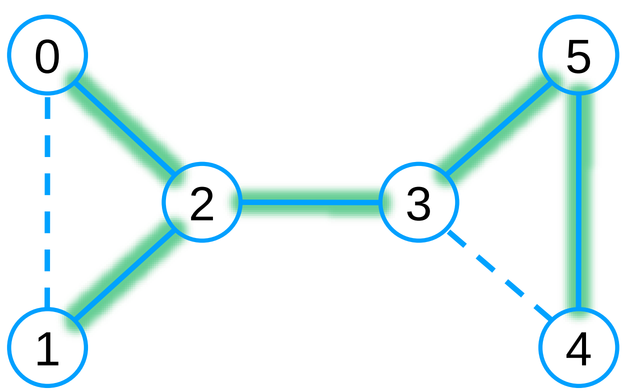
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4
0	**0**	1	1	3	4



- We don't need to consider (0,1)
  - T already has v-1 edges

Final Configuration

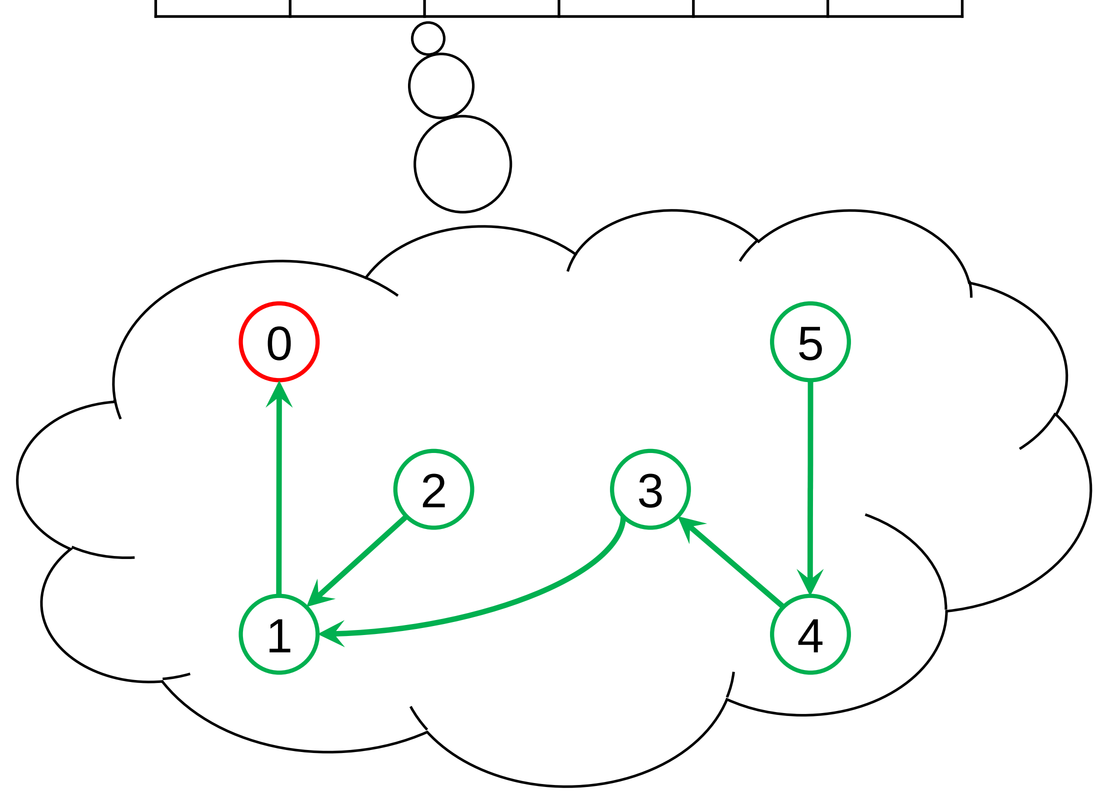
1. Start T with the isolated vertices of G
2. For each edge (u,v) in G
   - *find their canonical representatives and check if they are equal*
     - **yes**: discard the edge
     - **no**: merge the two connected component, and appoint a new canonical representative
   - Stop once T has v-1 edges



Edges

✓ (4, 5)
✓ (3, 5)
✓ (1, 2)
✗ (3, 4)
✓ (2, 3)
✓ (0, 2)
~~(0, 1)~~

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4
0	**0**	1	1	3	4



Complexity

Given a graph G, construct a **minimum spanning tree** T for it

0. Sort the edges of G by increasing weight   O(e log e)
1. Start T with the isolated vertices of G   O(v)
2. For each edge (u,v) in G   e times
   - *are u and v already connected in T?*
     find the canonical representative of u
     find the canonical representative of v
     check if they are equal

     This was O(v)
     - **yes**: discard the edge
     - **no**: add it to T
       merge the two connected component
       appoint a new canonical representative

       This was O(1)
   - Stop once T has v-1 edges

Complexity of Union-find

- Finding the canonical representative of a vertex
  - in the worst case, we have to go through all the vertices
  - O(v)

- Merging two connected components and appointing the new canonical representative
  - a single array write
  - O(1)

Complexity

Given a graph G, construct a **minimum spanning tree** T for it

0. Sort the edges of G by increasing weight O(e log e)
1. Start T with the isolated vertices of G O(v)
2. For each edge (u,v) in G e times
   - *are u and v already connected in T?* O(v)
     find the canonical representative of u
     find the canonical representative of v
     check if they are equal
     This was O(v)
     - **yes**: discard the edge
     - **no**: add it to T O(1)
       merge the two connected component
       appoint a new canonical representative
       This was O(1)
   - Stop once T has v-1 edges

O(ev)

Complexity

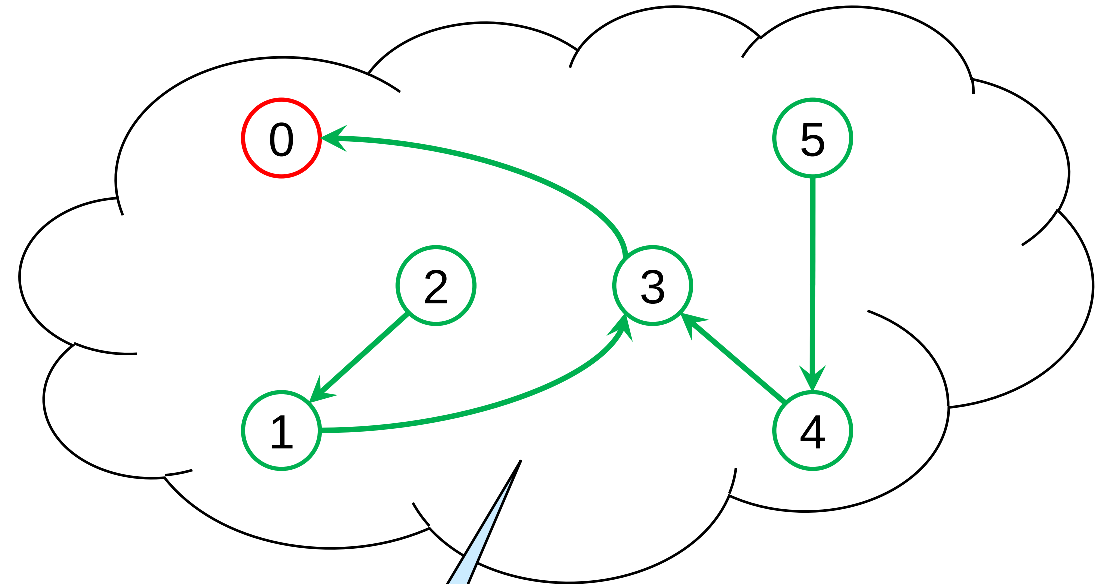
- By swapping BFS or DFS with union find,
  the complexity of Kruskal's algorithm remains O(ev)
  - no gain

- *Can we do better?*

Height Tracking

About the Visualization Graph

- The graph visualization of the union-find data structure is a **directed tree**
  - ❑ not a binary tree in general
  - the edges point from child to parent
    - towards the root
  - the root is the canonical representative
  - We find a canonical representative by going from a vertex to the root of the tree it is in



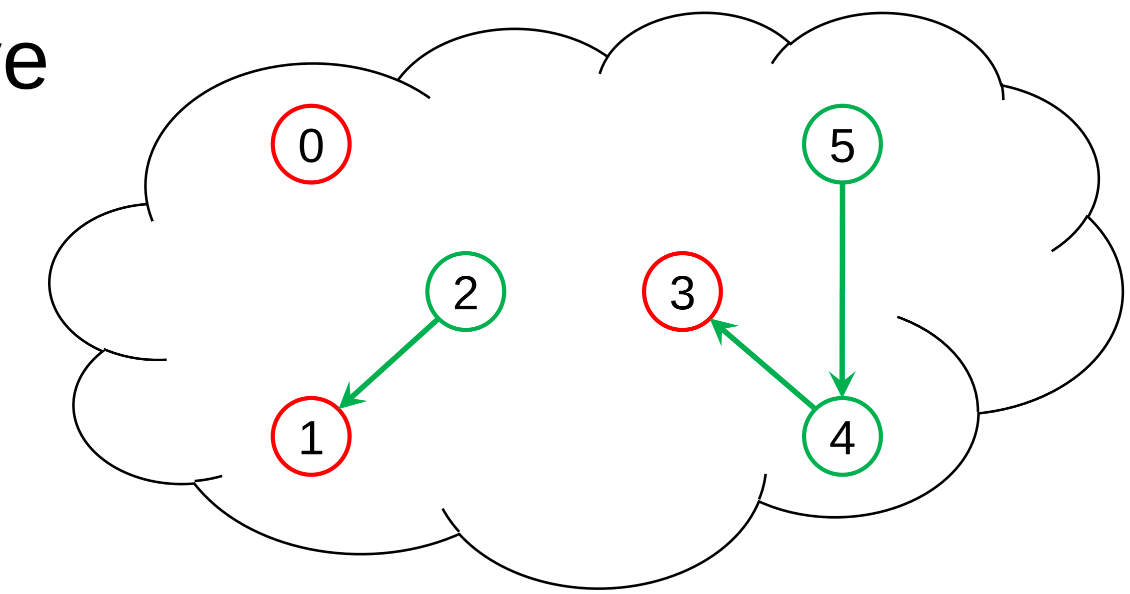
This tree has height 4

Half-way through, this is a **directed forest**

- The cost is the **height** of the tree
  - O(v) in general
  - but O(log v) if the tree is **balanced**

Merging Trees

- Finding a canonical representative costs O(log v) on a balanced visualization tree



- Can we arrange so that it grows balanced as we construct it?
  - when we merge trees by taking their union

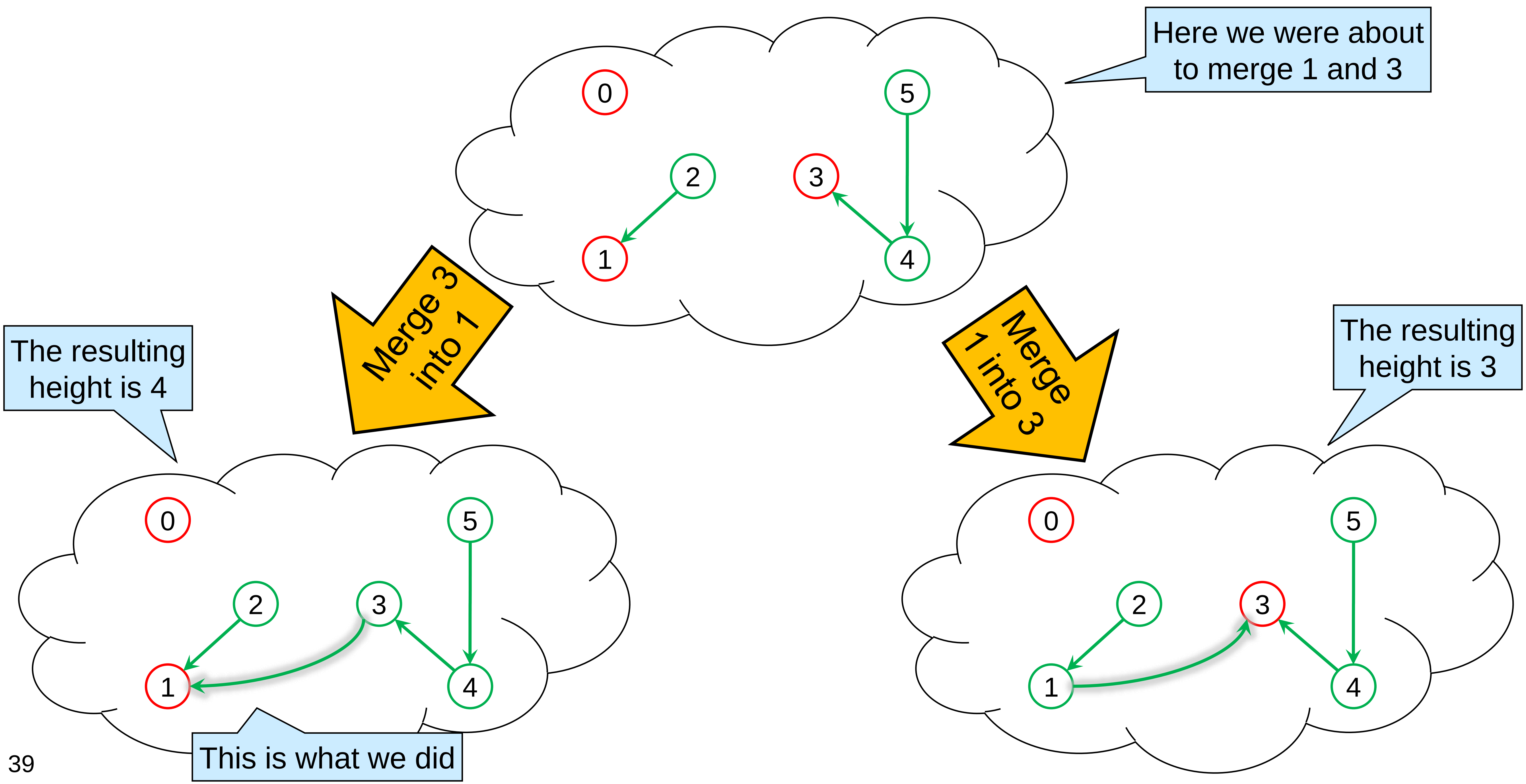
Each tree represents a connected component

- When picking the new canonical representative, we can arrange so that the merged tree remains shallow whenever possible

Will this be enough to ensure that is its balanced?

Merging Trees

- When picking the new canonical representative, arrange so that the merged tree remains shallow whenever possible



Height Tracking

- When picking the new canonical representative, arrange so that the merged tree remains shallow whenever possible

- We want to **merge shorter trees into taller trees**
  - then the height does not change

- If the trees have the same height, we can merge them either way
  - the height will grow by 1 no matter what

- This strategy is called **height tracking**

Tracking the Height

- We now need to track the height of each tree
  - How do we do that?

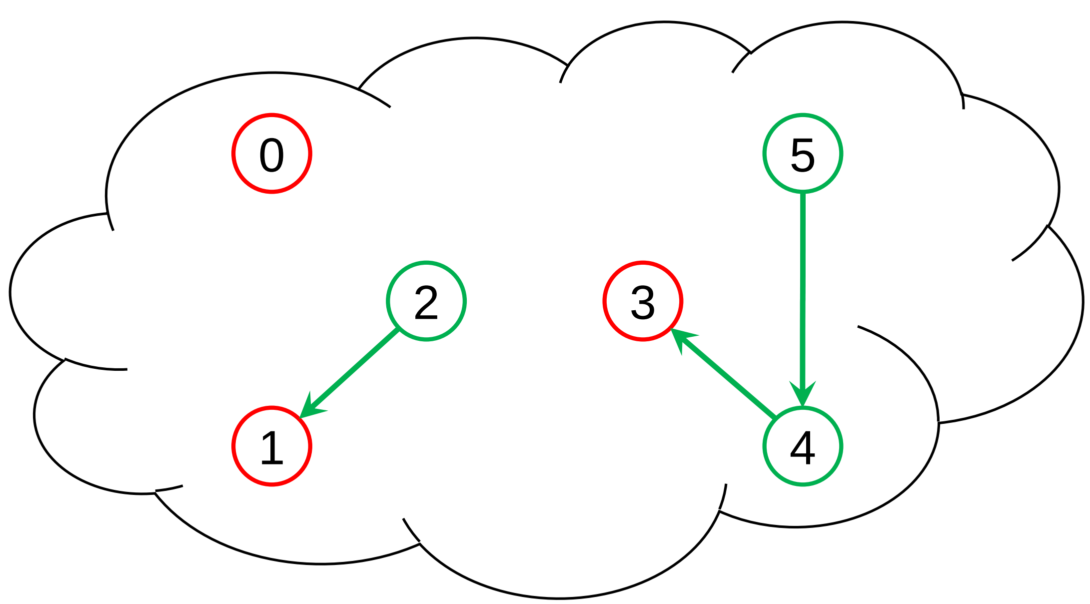
- Update the union-find data structure so that each position stores both the parent in the tree and the height
  - using a struct
  - or *two* arrays

- *Can we do better?*

Tracking the Height

- Observations
  - we need the height only when reaching the root
    - that's when we need to decide which way to merge the trees
  - the root has no parent
    - a canonical representative points to itself

- Idea: store the parent in a child node and the height in the roots



0	1	2	3	4	5
1	2	1	3	3	4

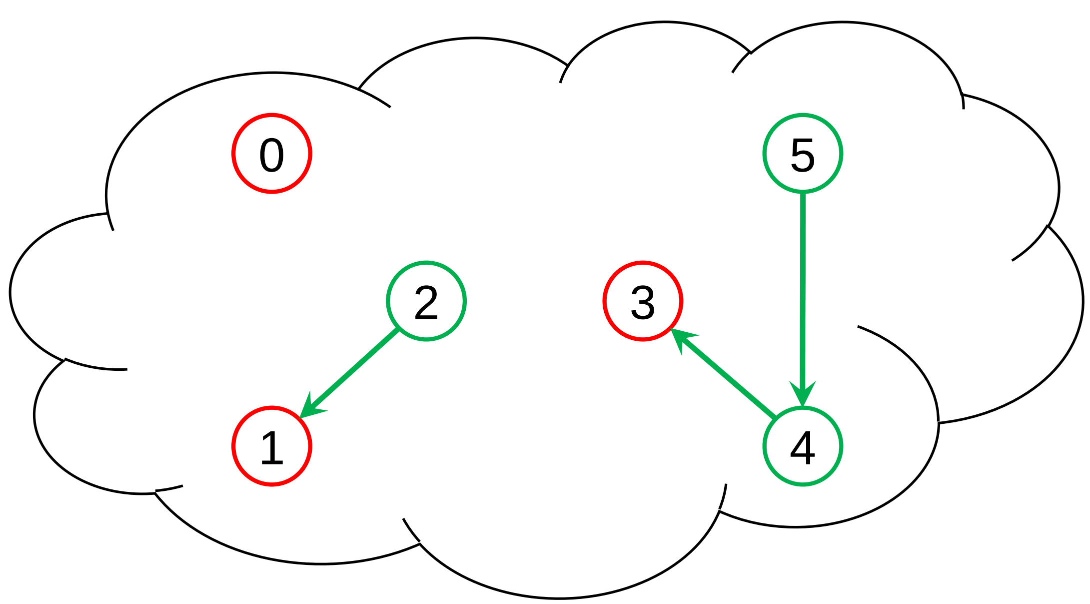
But how do we know if a position contains a parent or a height?

Tracking the Height

- Store the parent in a child node and the height in the roots
  - but how do we know if a position contains a parent or a height?

- We need to be able to recognize a root when we see one
  - add a flag
    - a single bit is enough
    - make the roots store the height as a **negative** numbers

That's the sign bit



0	1	2	3	4	5
-1	-2	1	-3	3	4

The parent of 2 is 1

The tree rooted at 3 has height 3

Example

- Let’s run Kruskal’s algorithm
  - using union-find with height tracking to check if two vertices are connected

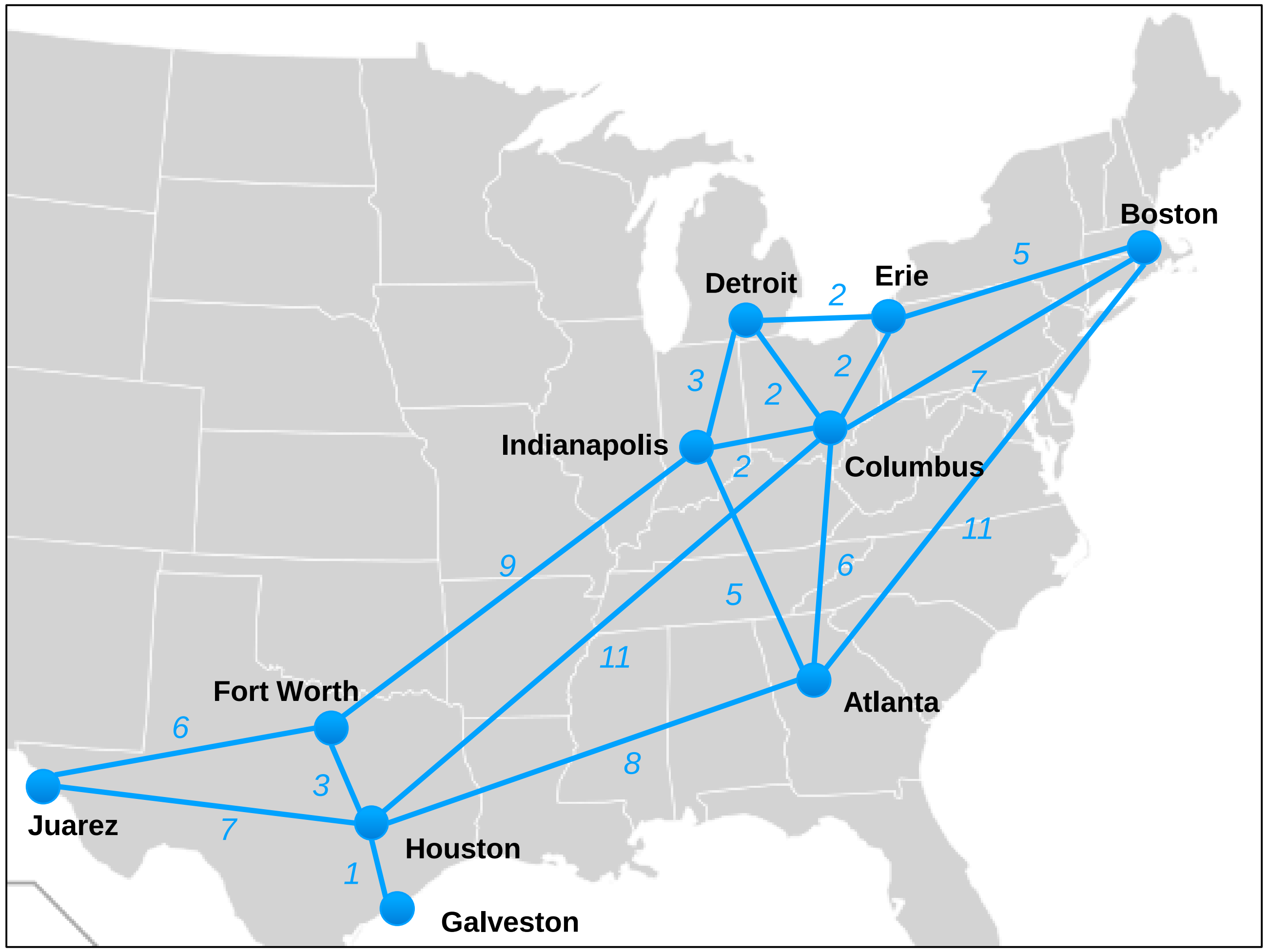
  on the road network example

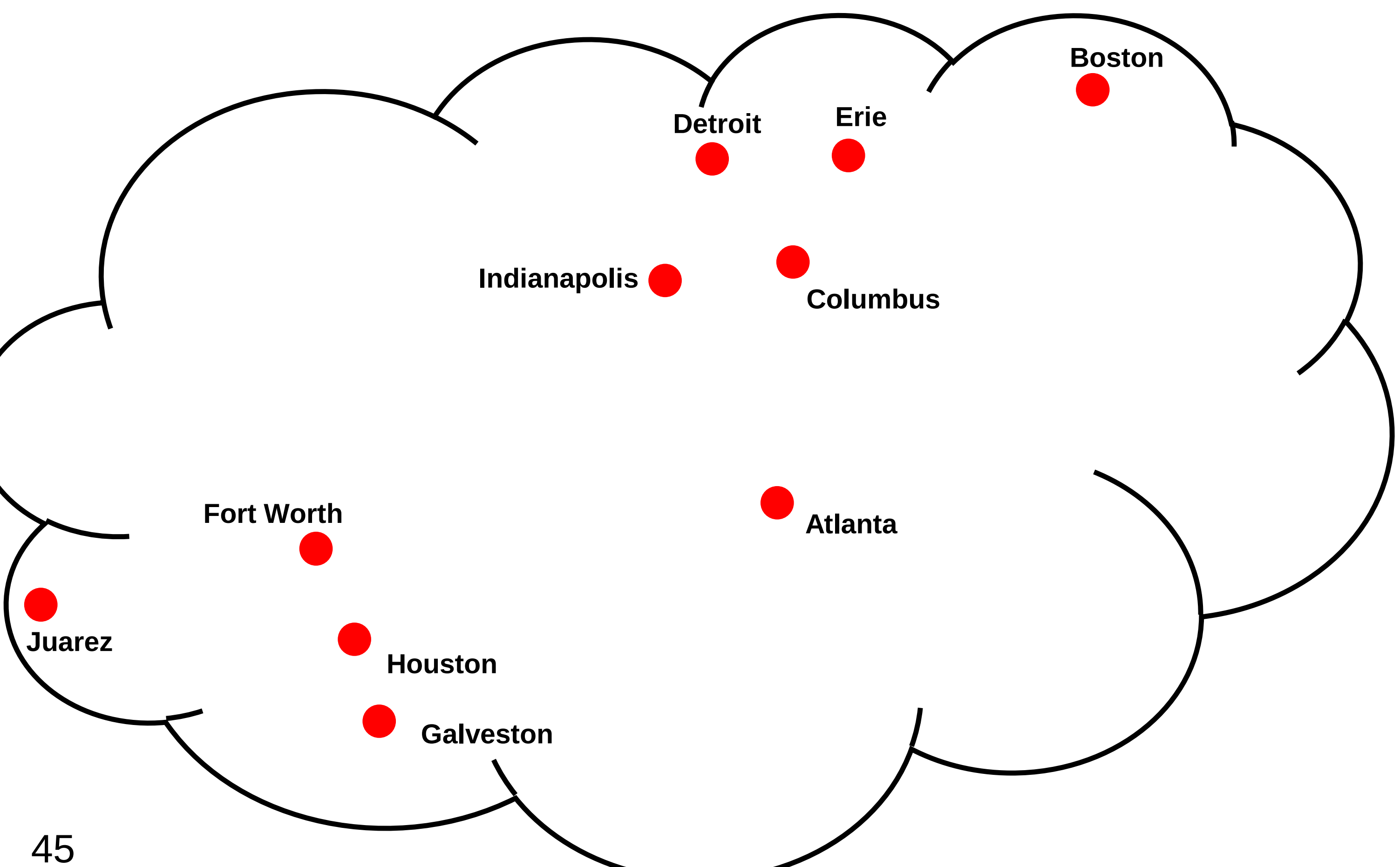
The edges are in the same order as in the last lecture

The resulting spanning tree will be the same

Sorted edges

G-H
C-E
C-I
D-E
C-D
D-I
F-H
B-E
A-I
F-J
A-C
B-C
H-J
A-H
F-I
C-H
A-B



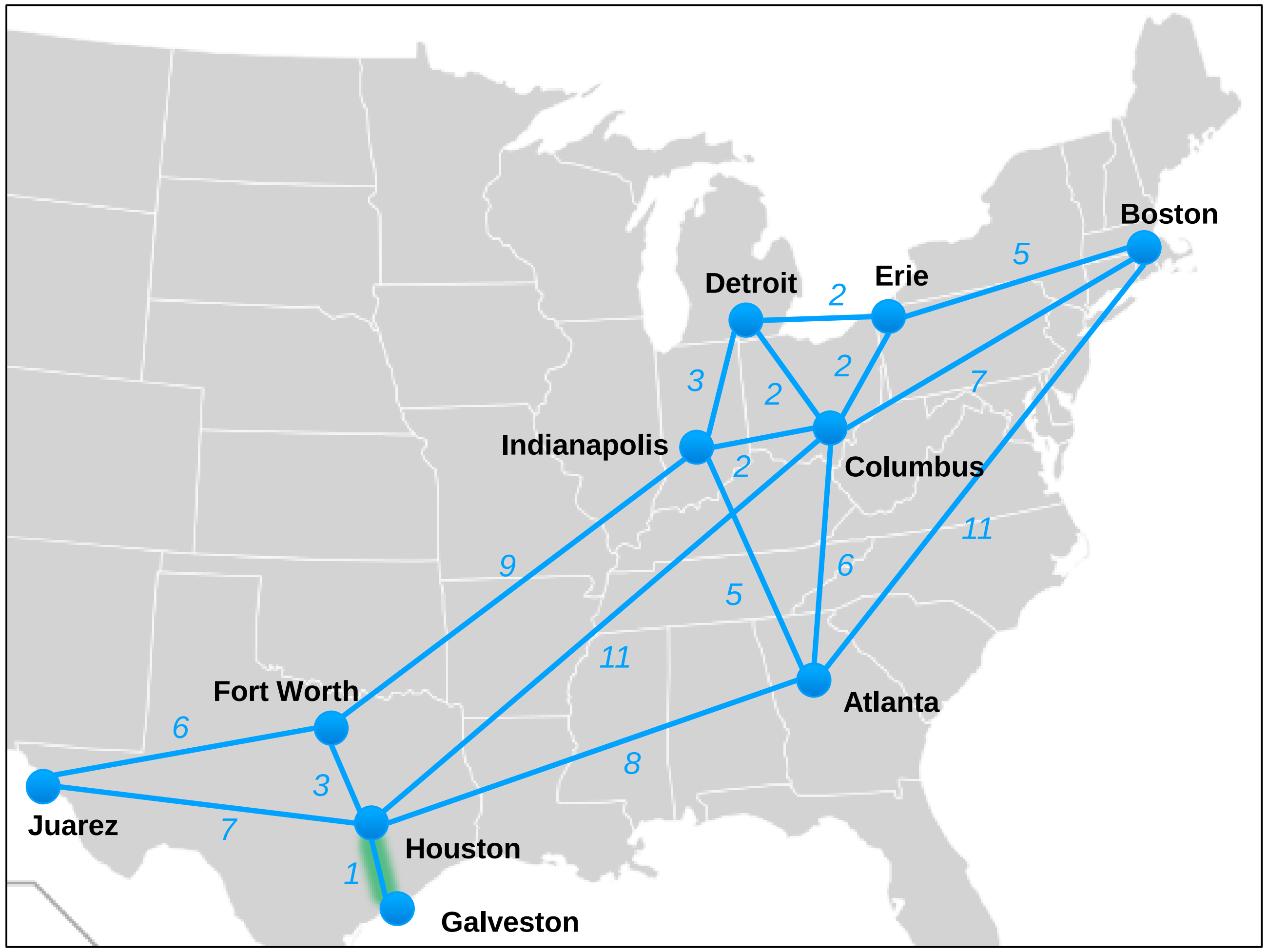


Sorted edges

G-H
C-E
C-I
D-E
C-D
D-I
F-H
B-E
A-I
F-J
A-C
B-C
H-J
A-H
F-I
C-H
A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

We have a choice

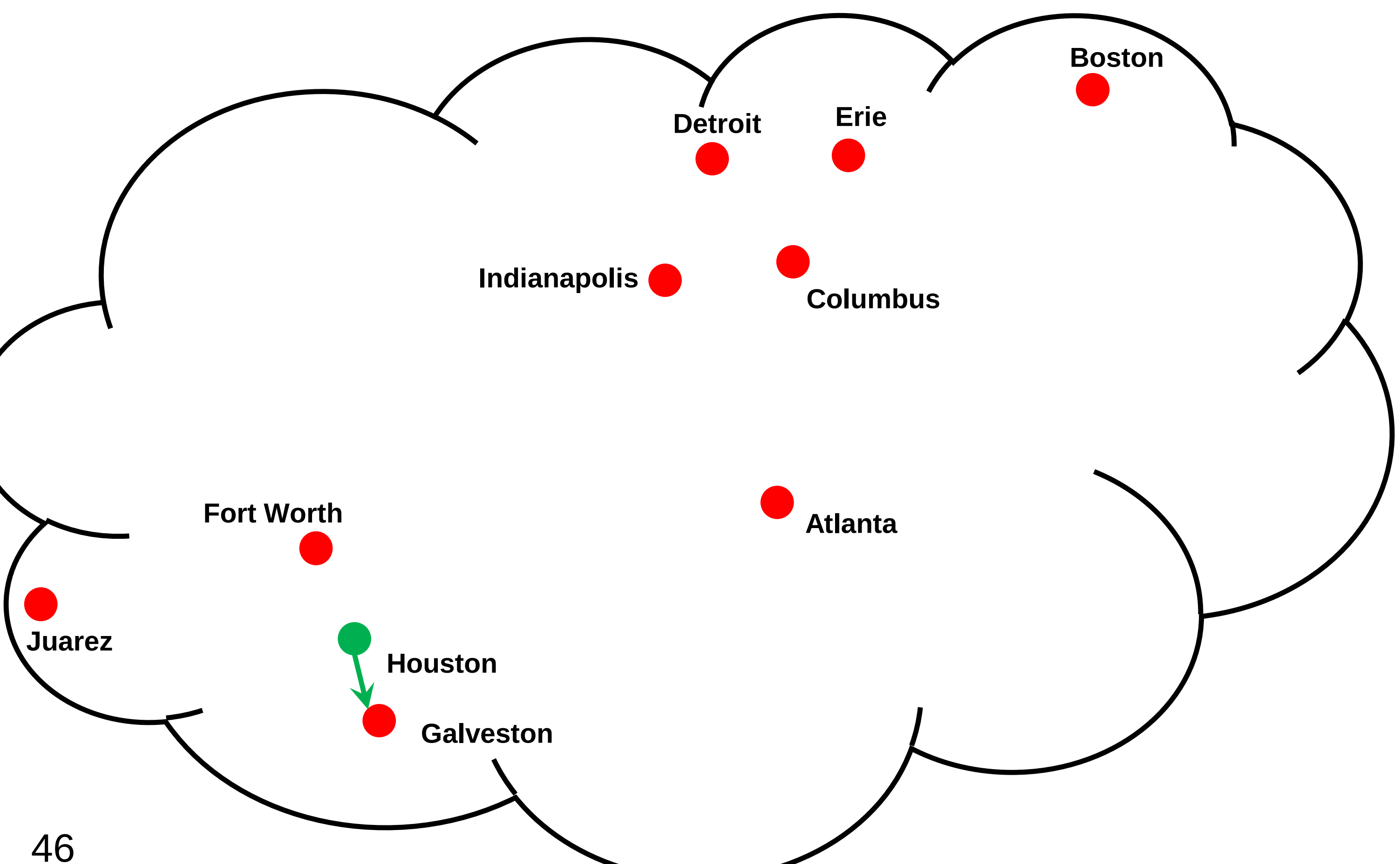


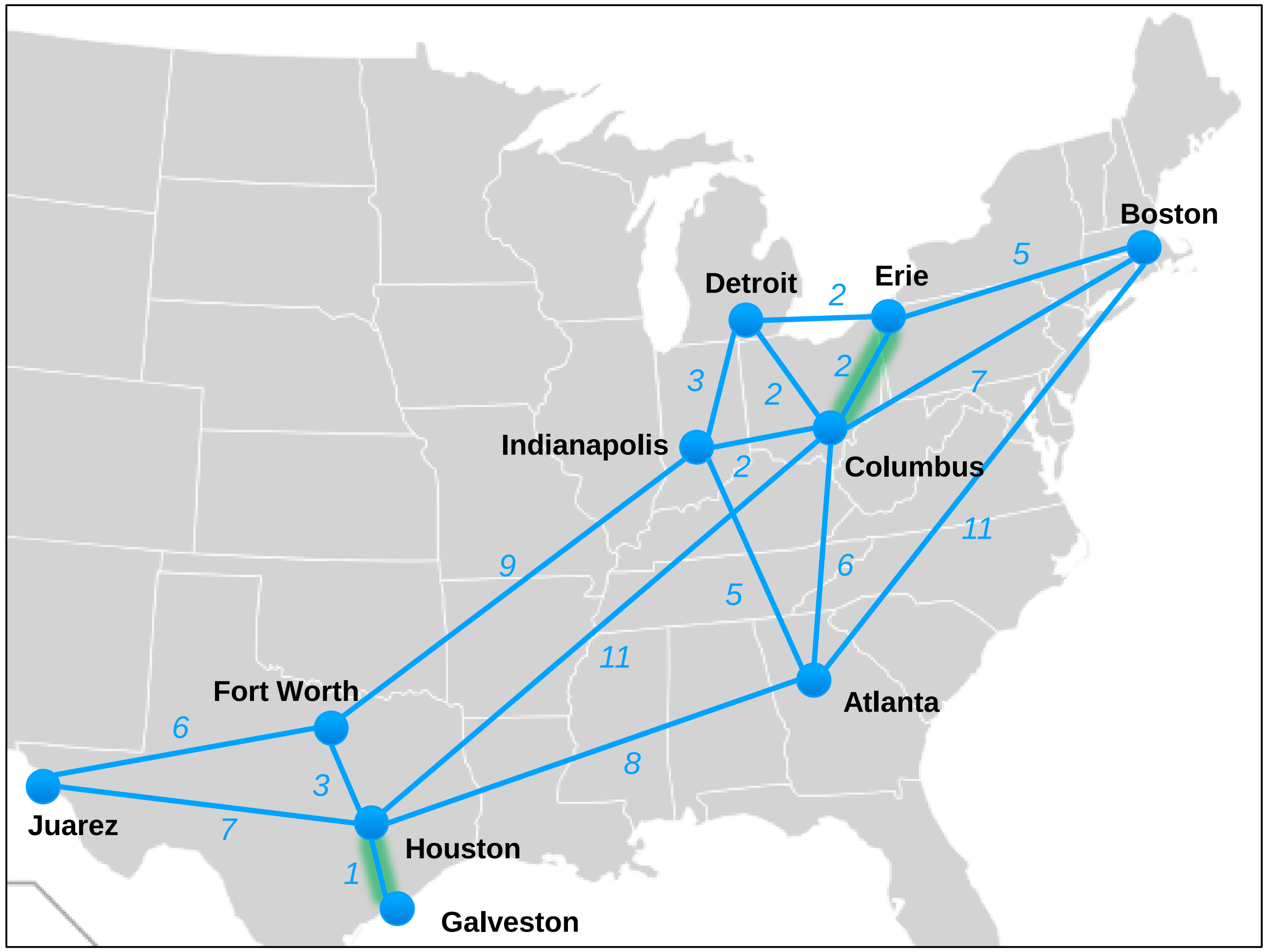
Sorted edges

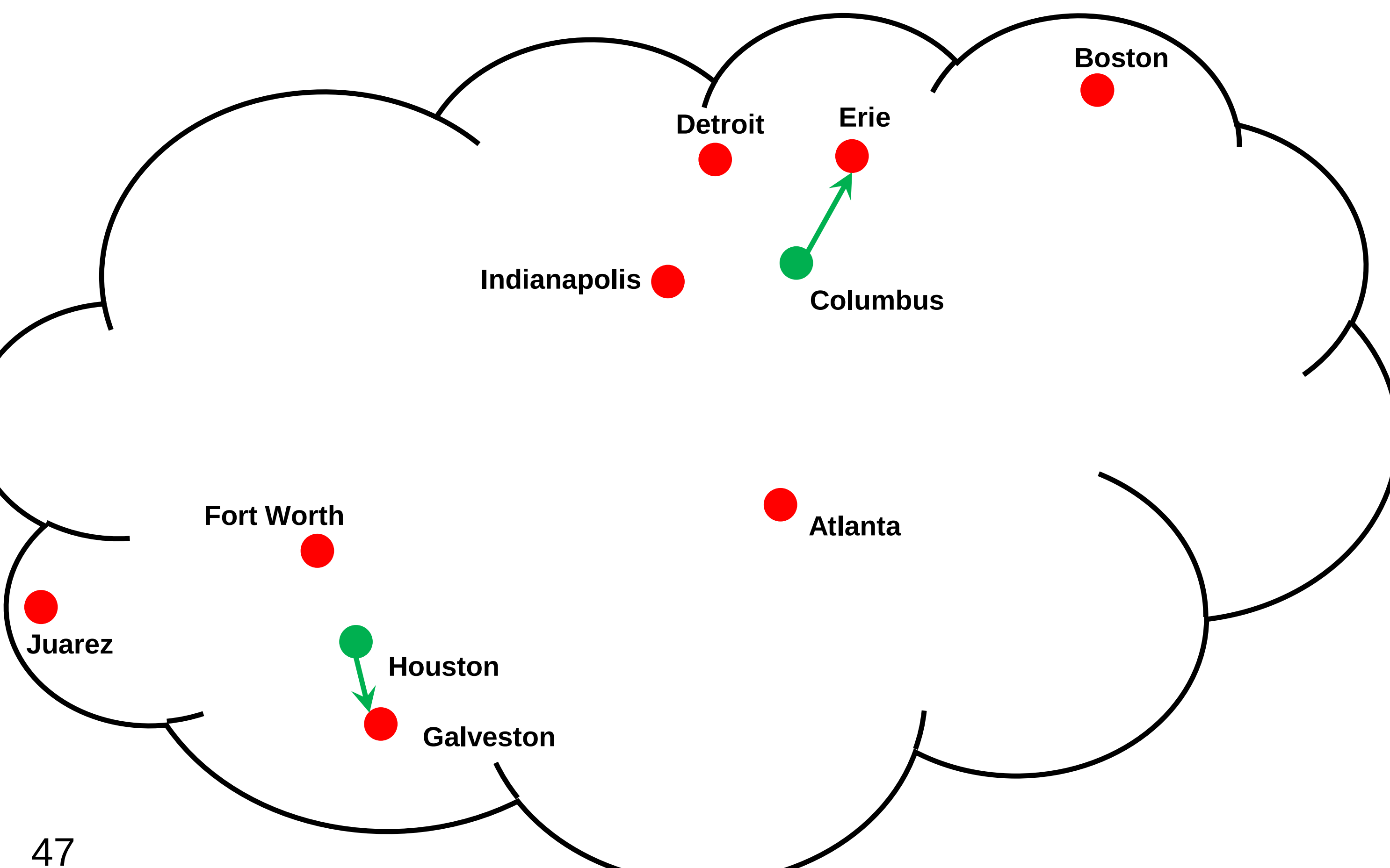
- ✓ G-H
- **C-E**
- C-I
- D-E
- C-D
- D-I
- F-H
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1

We have a choice



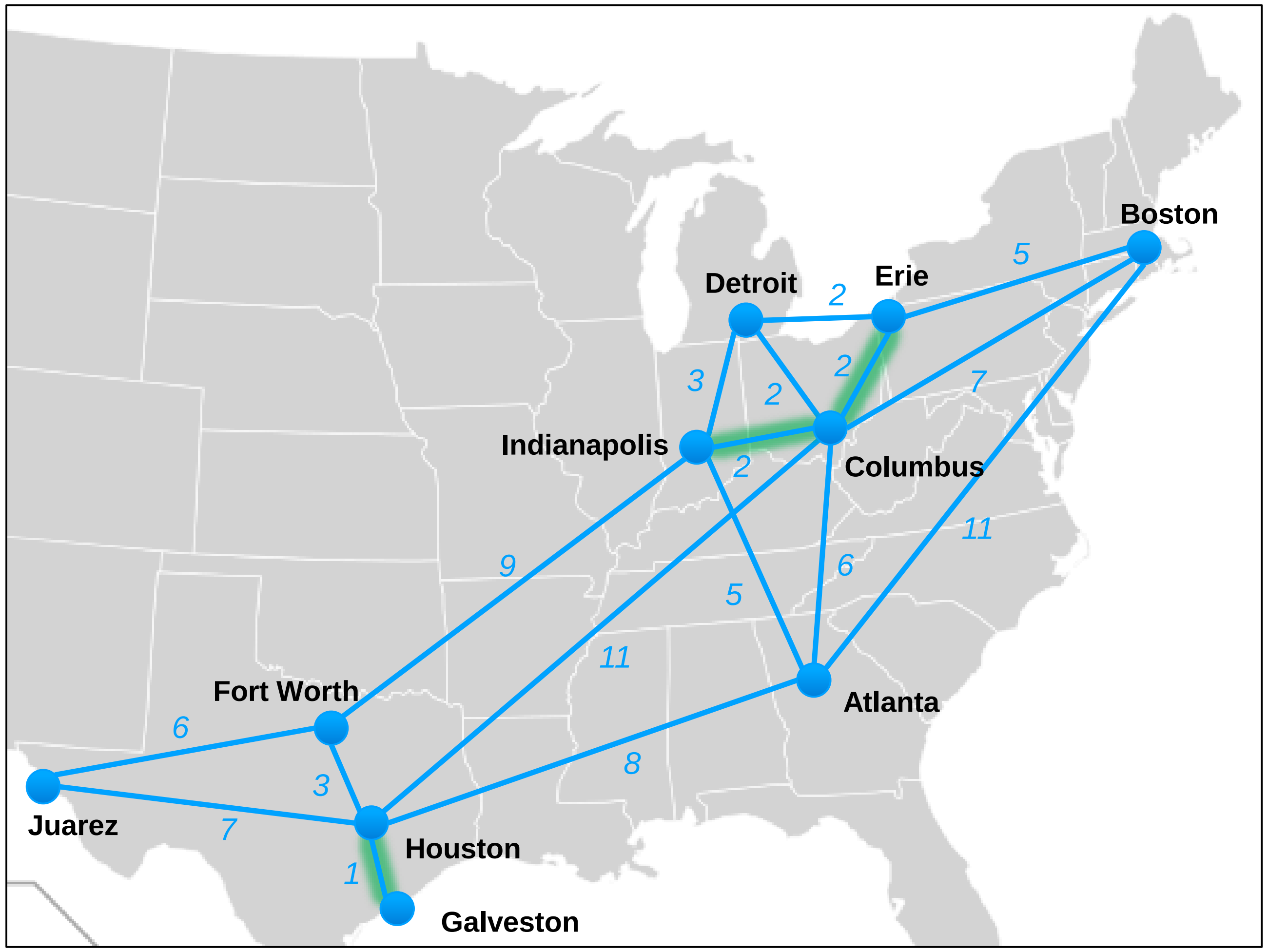


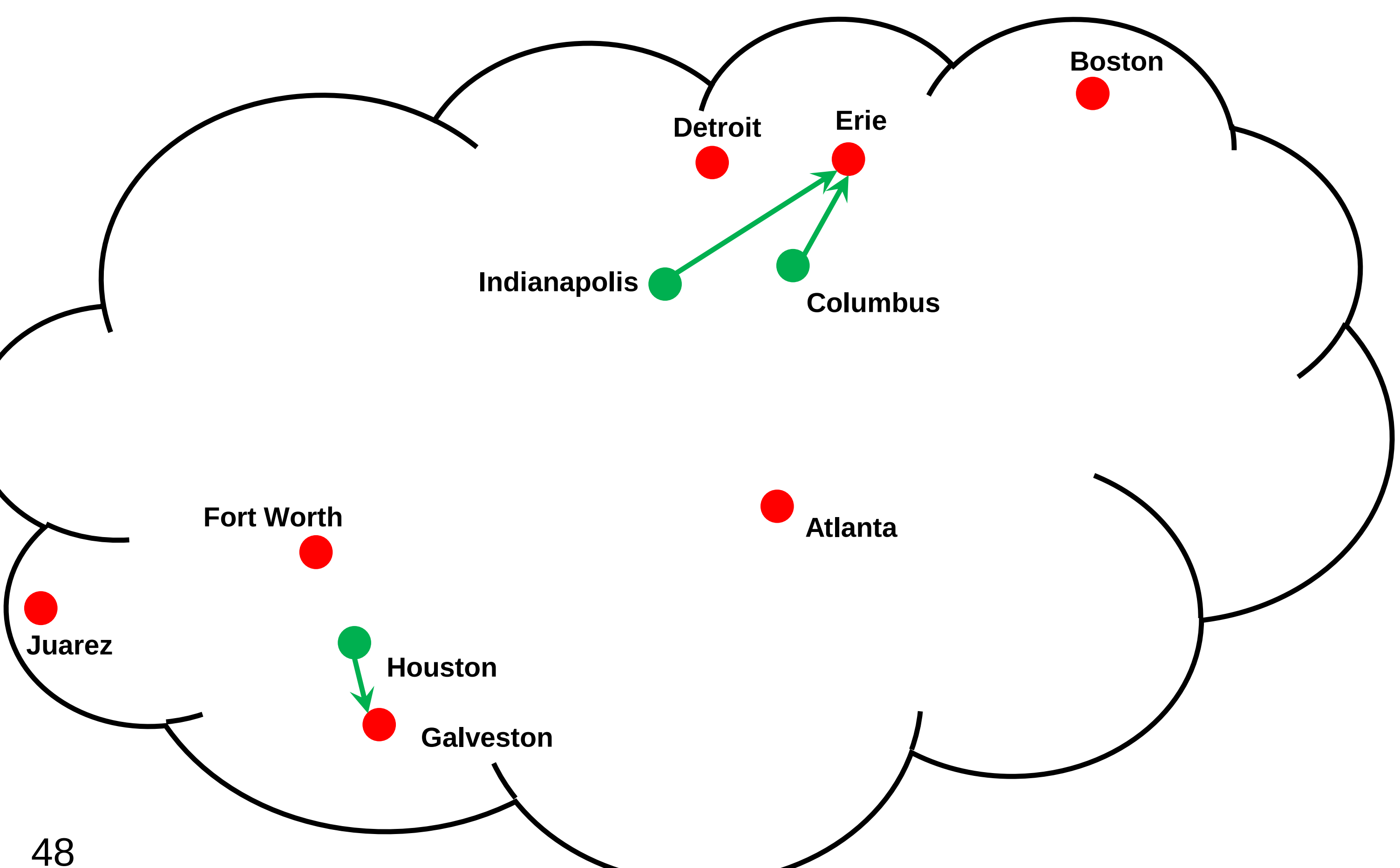


Sorted edges

- ✓ G-H
- ✓ C-E
- **C-I**
- D-E
- C-D
- D-I
- F-H
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1

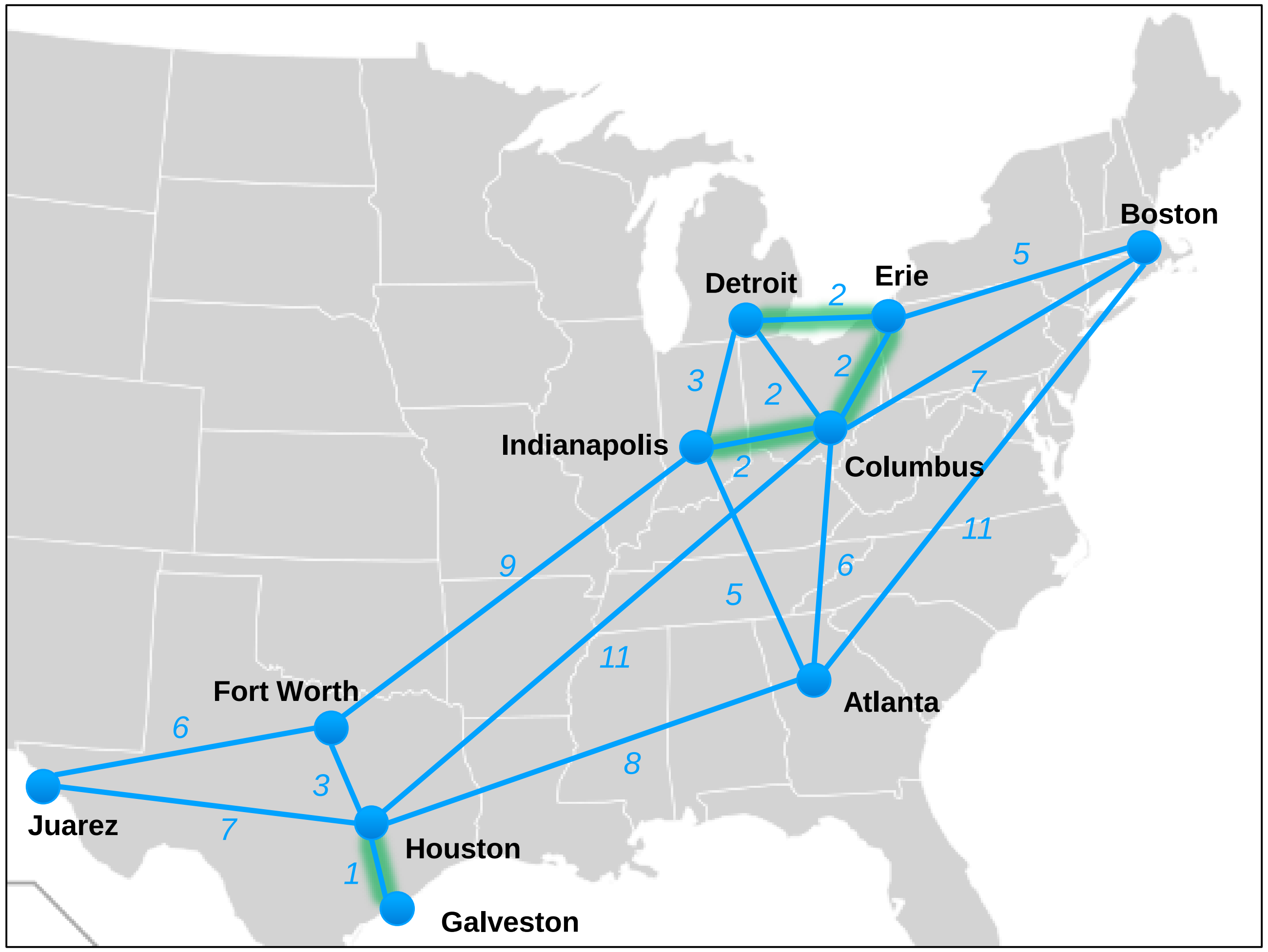


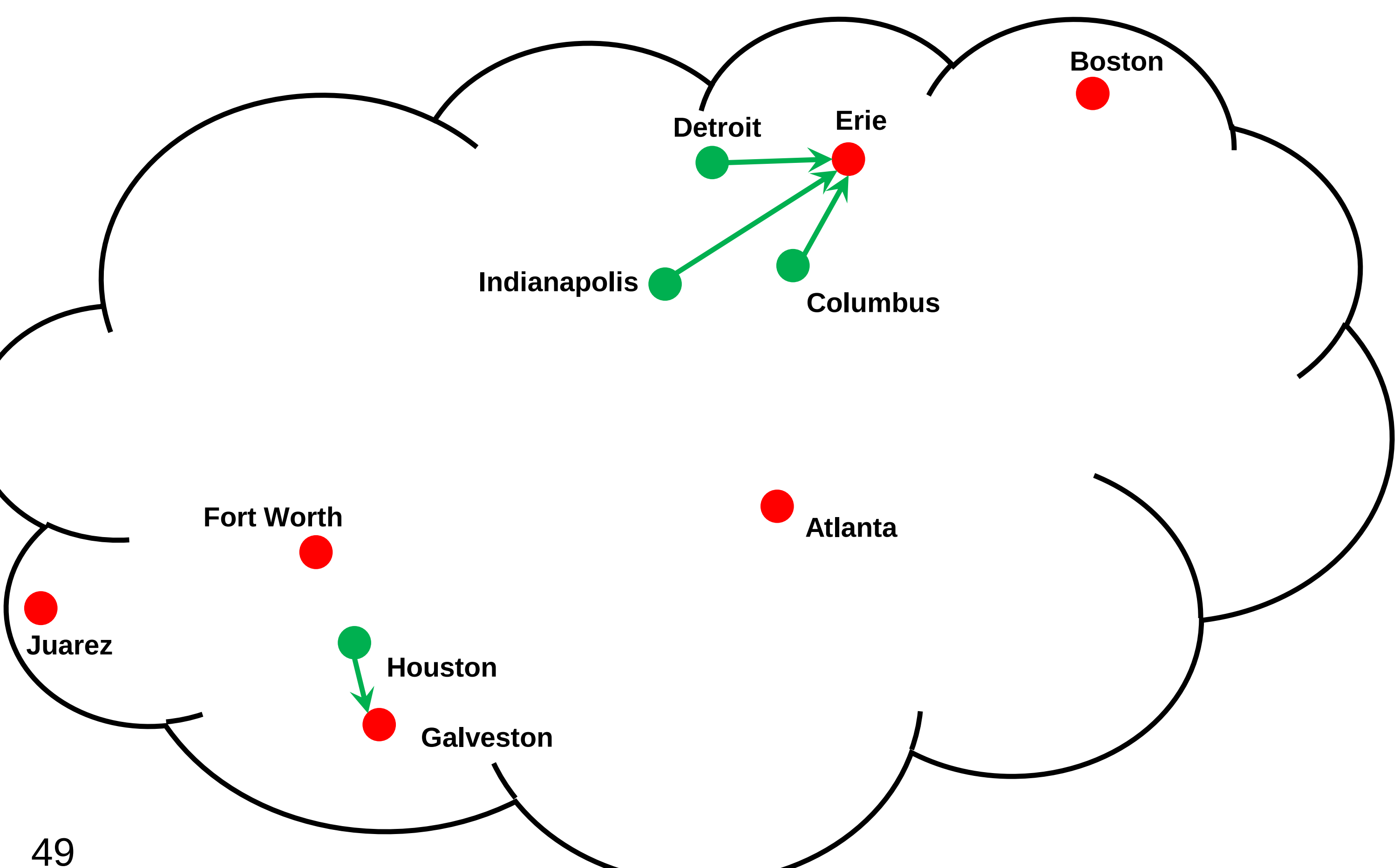


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- **D-E**
- C-D
- D-I
- F-H
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1

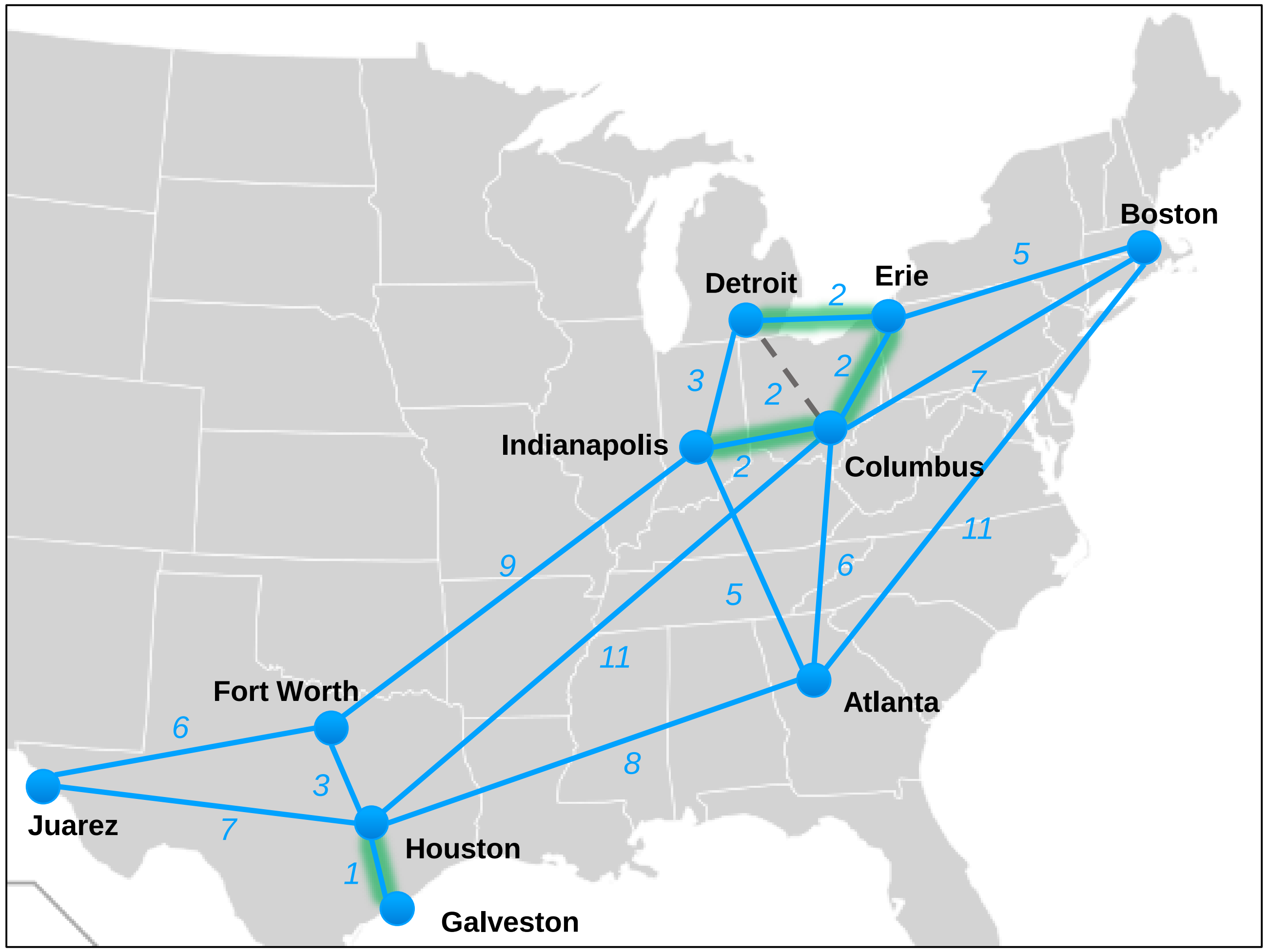


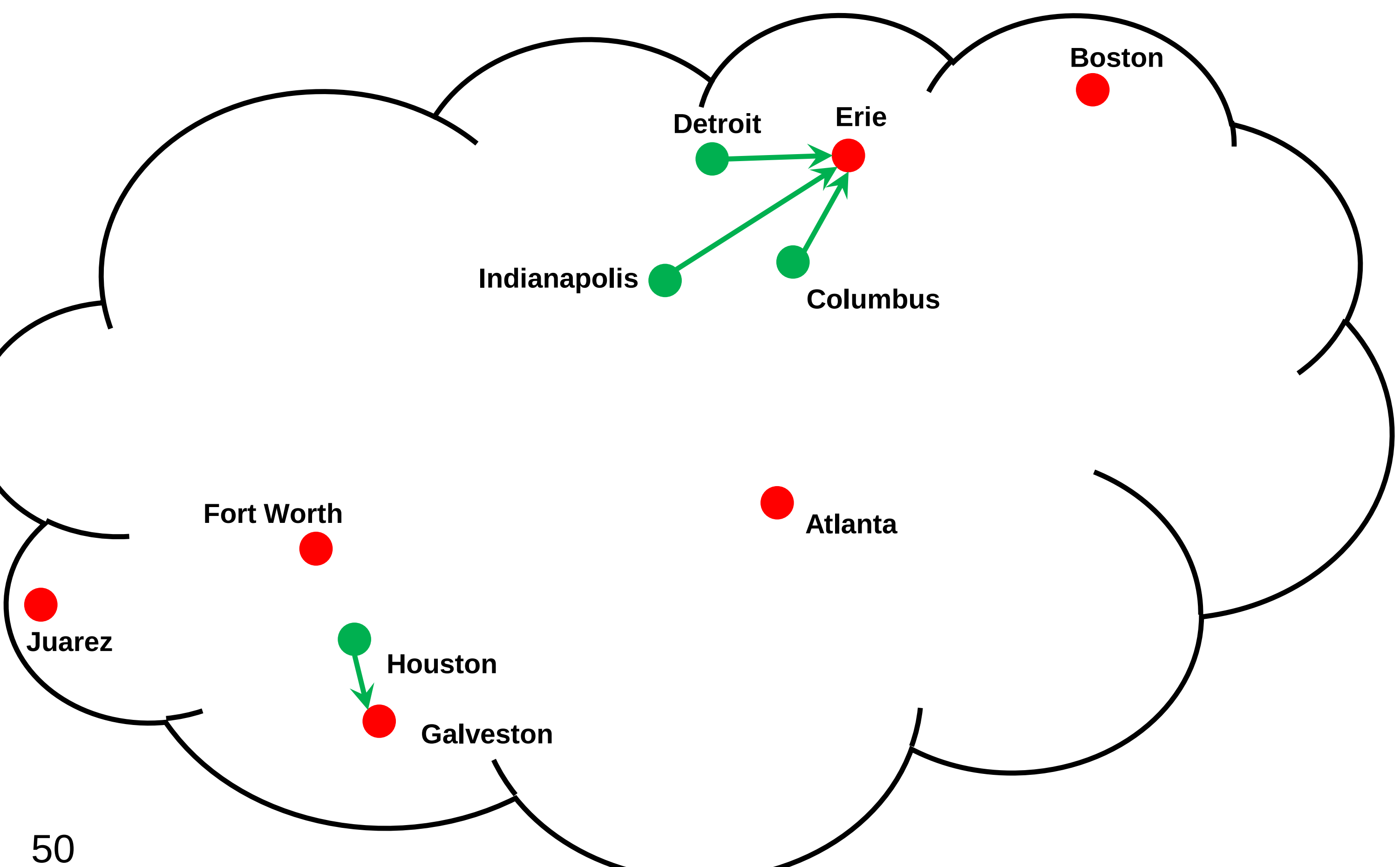


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- **C-D**
- D-I
- F-H
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1
-1	-1	4	**4**	-2	-1	-2	6	4	-1

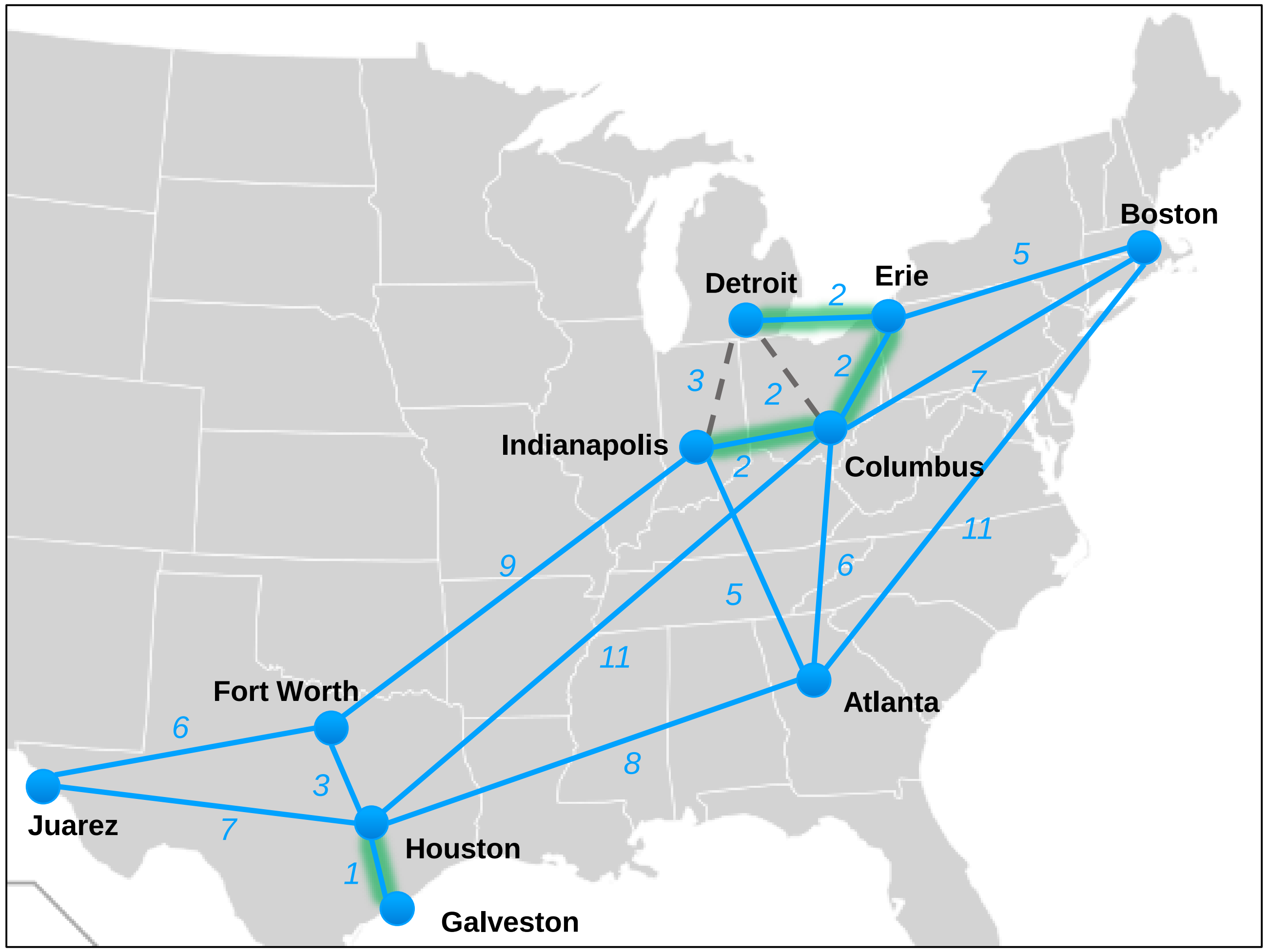


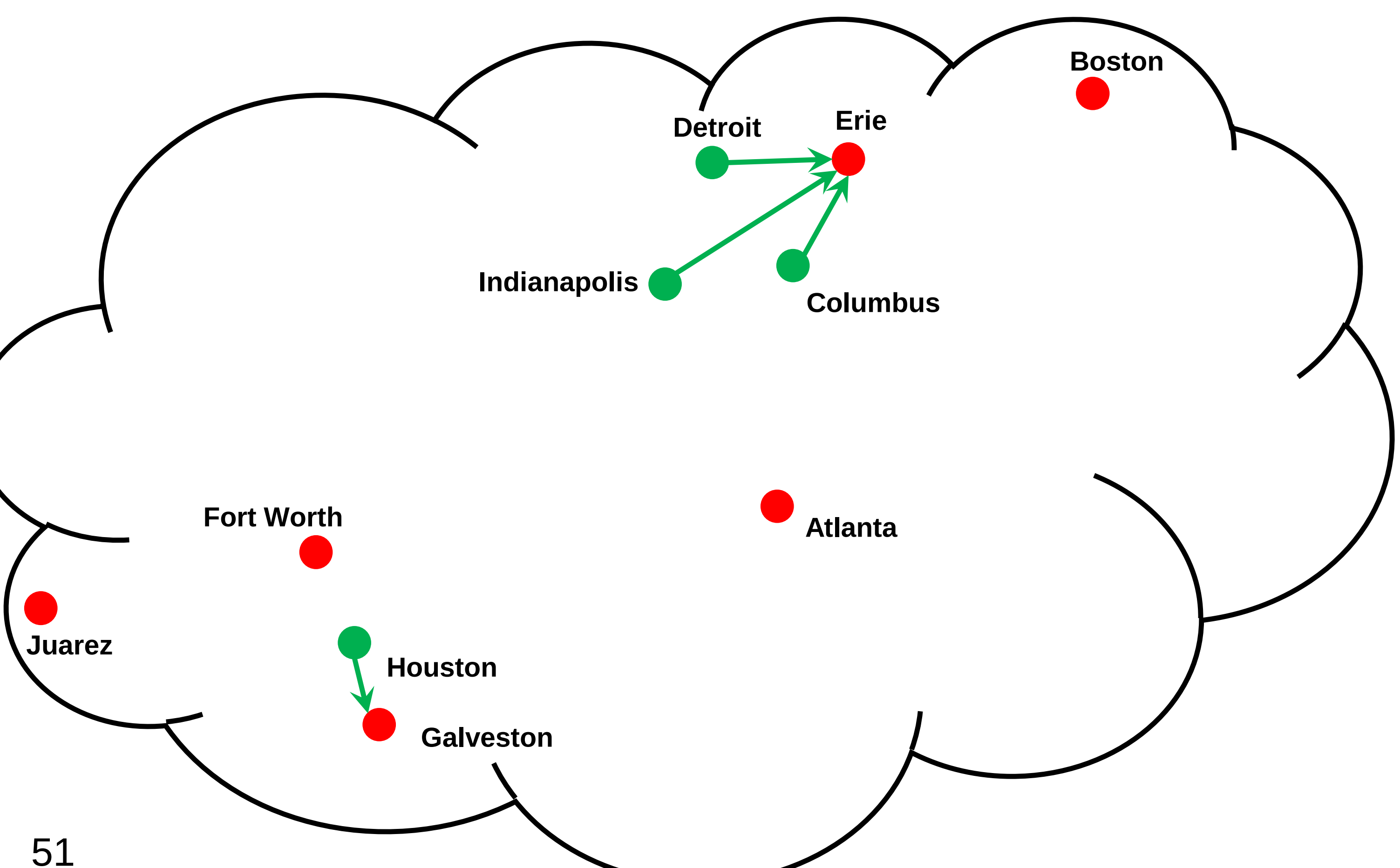


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- ✗ C-D
- **D-I**
- F-H
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1
-1	-1	4	**4**	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1

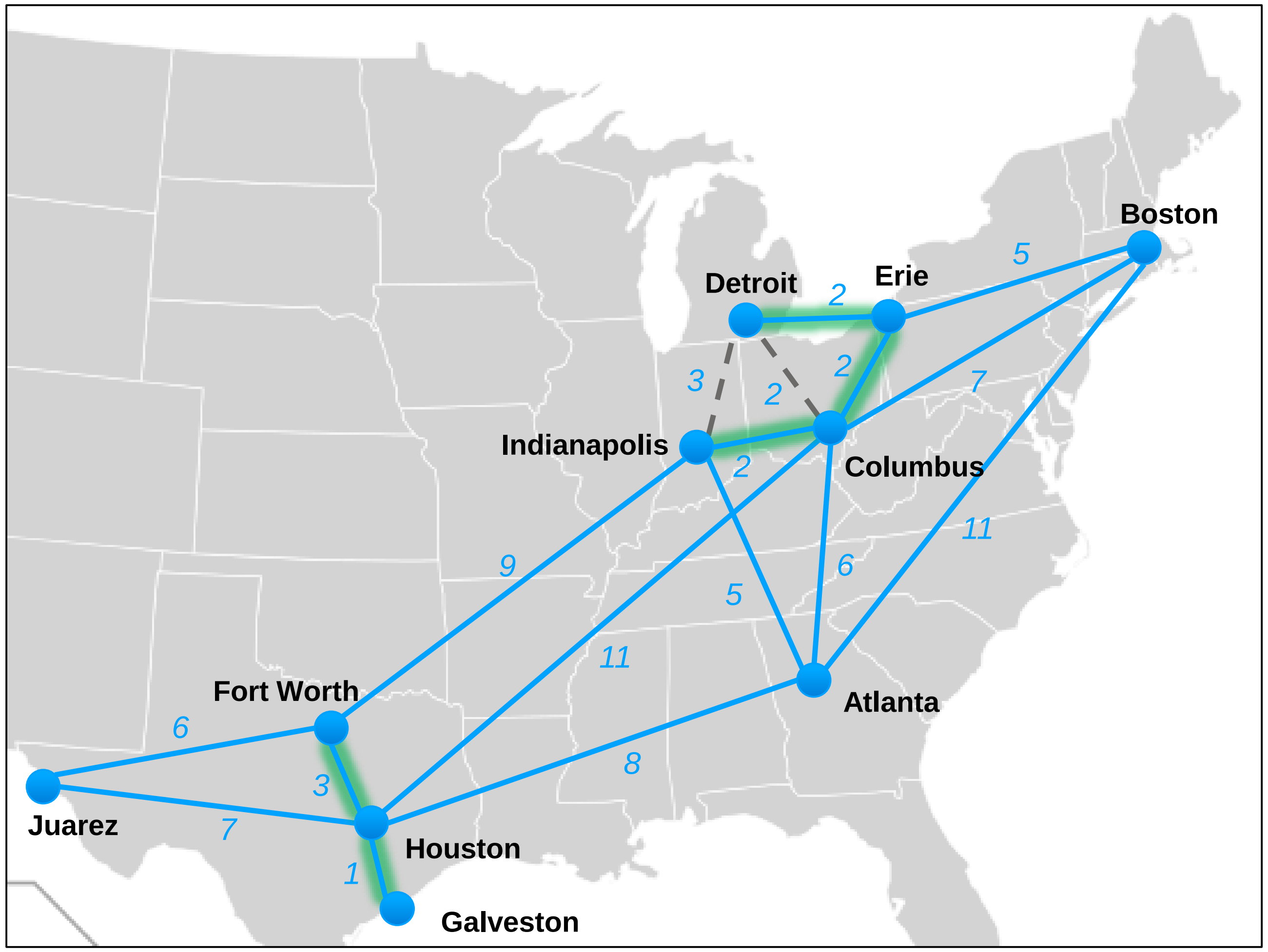




Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- **F-H**
- B-E
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

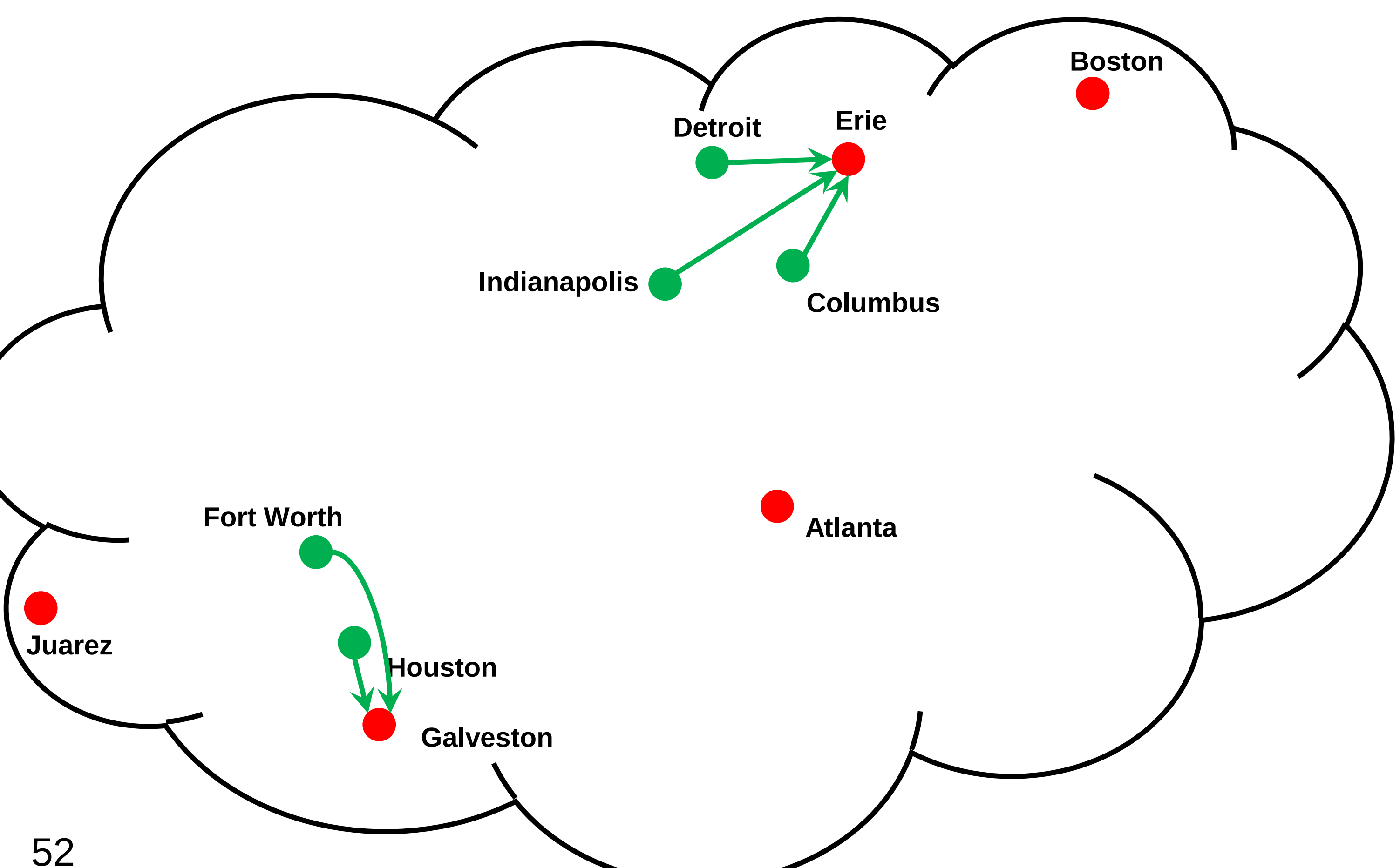
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1

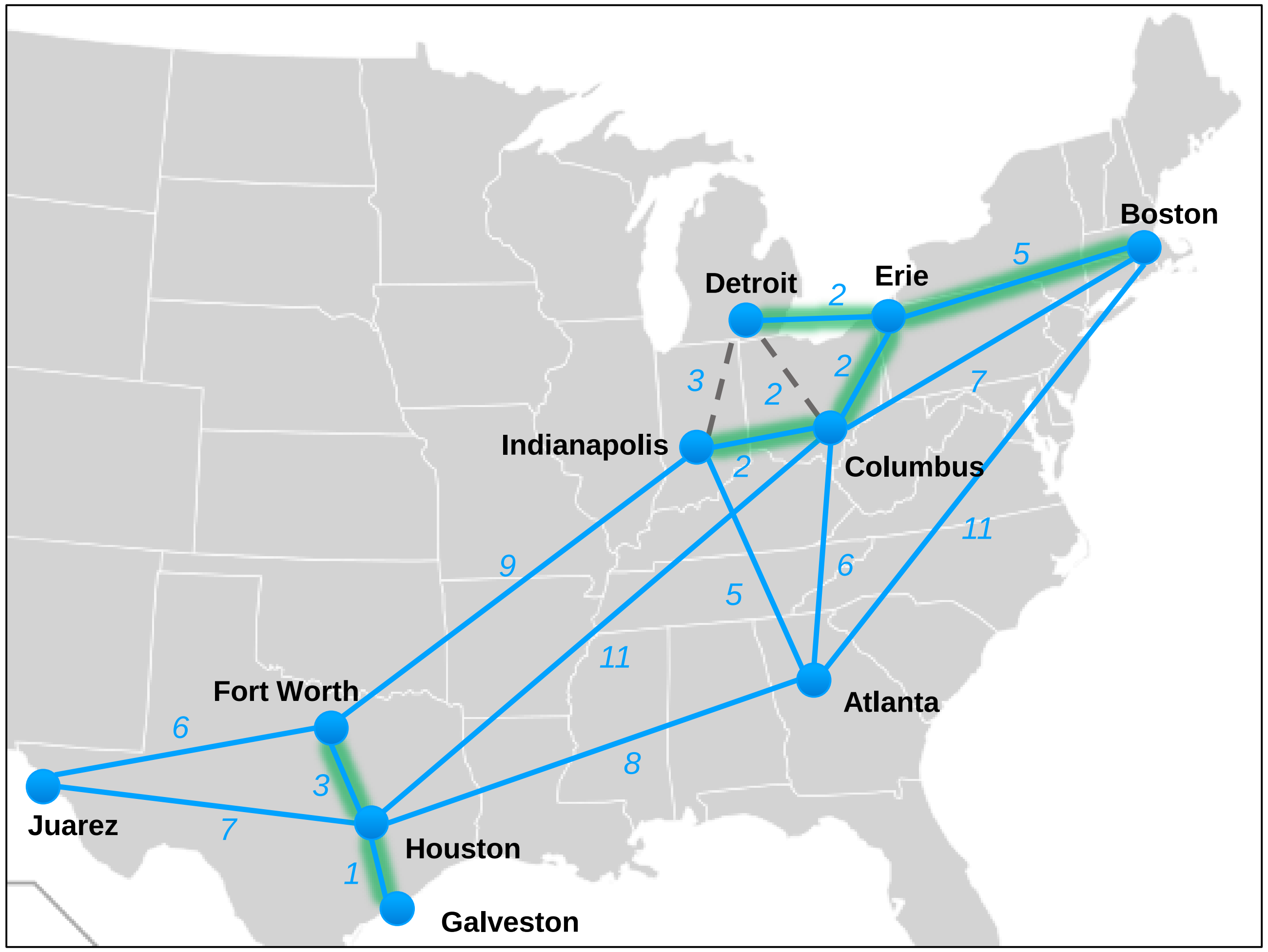


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- ✓ F-H
- **B-E**
- A-I
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	**6**	-2	6	4	-1

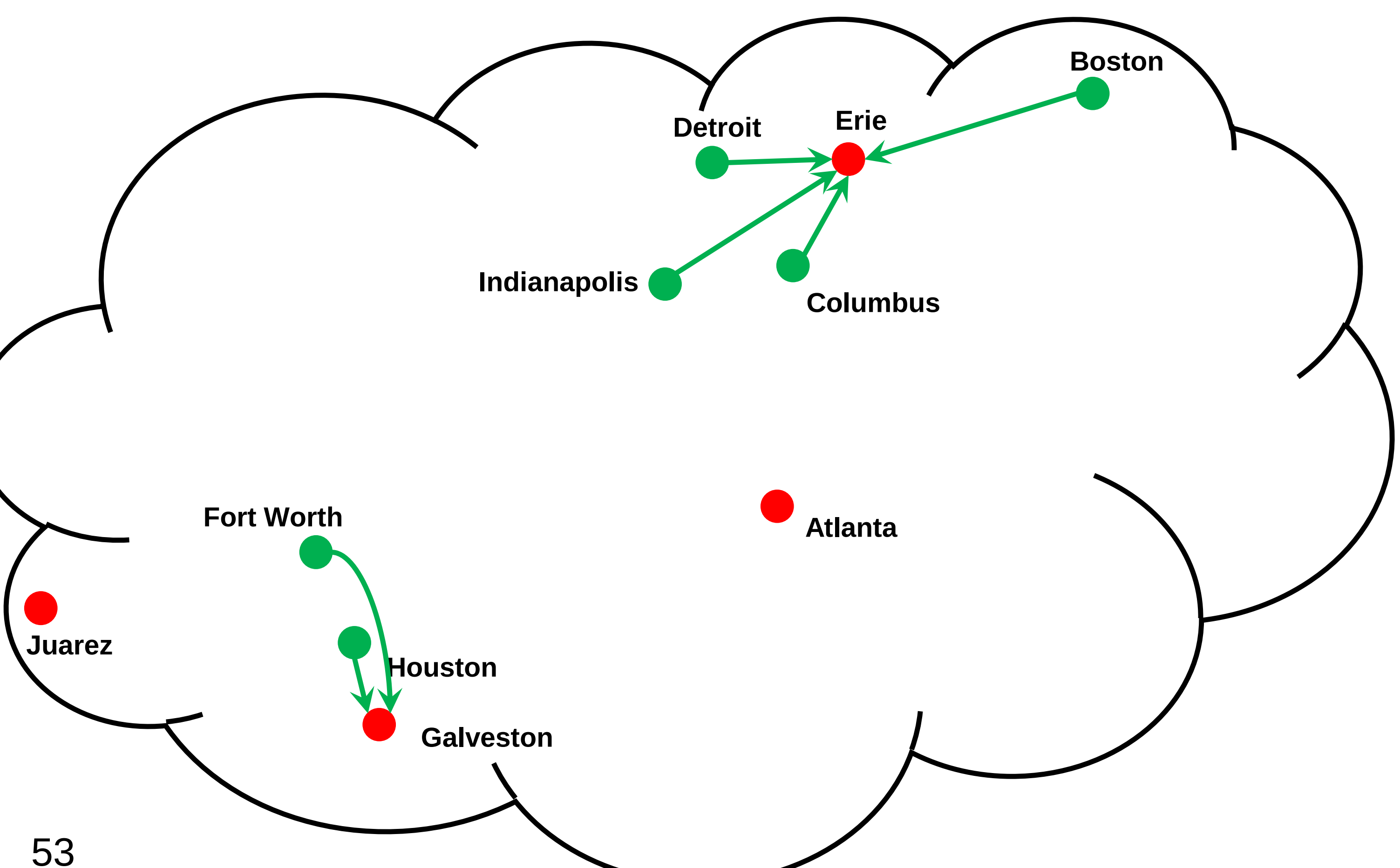


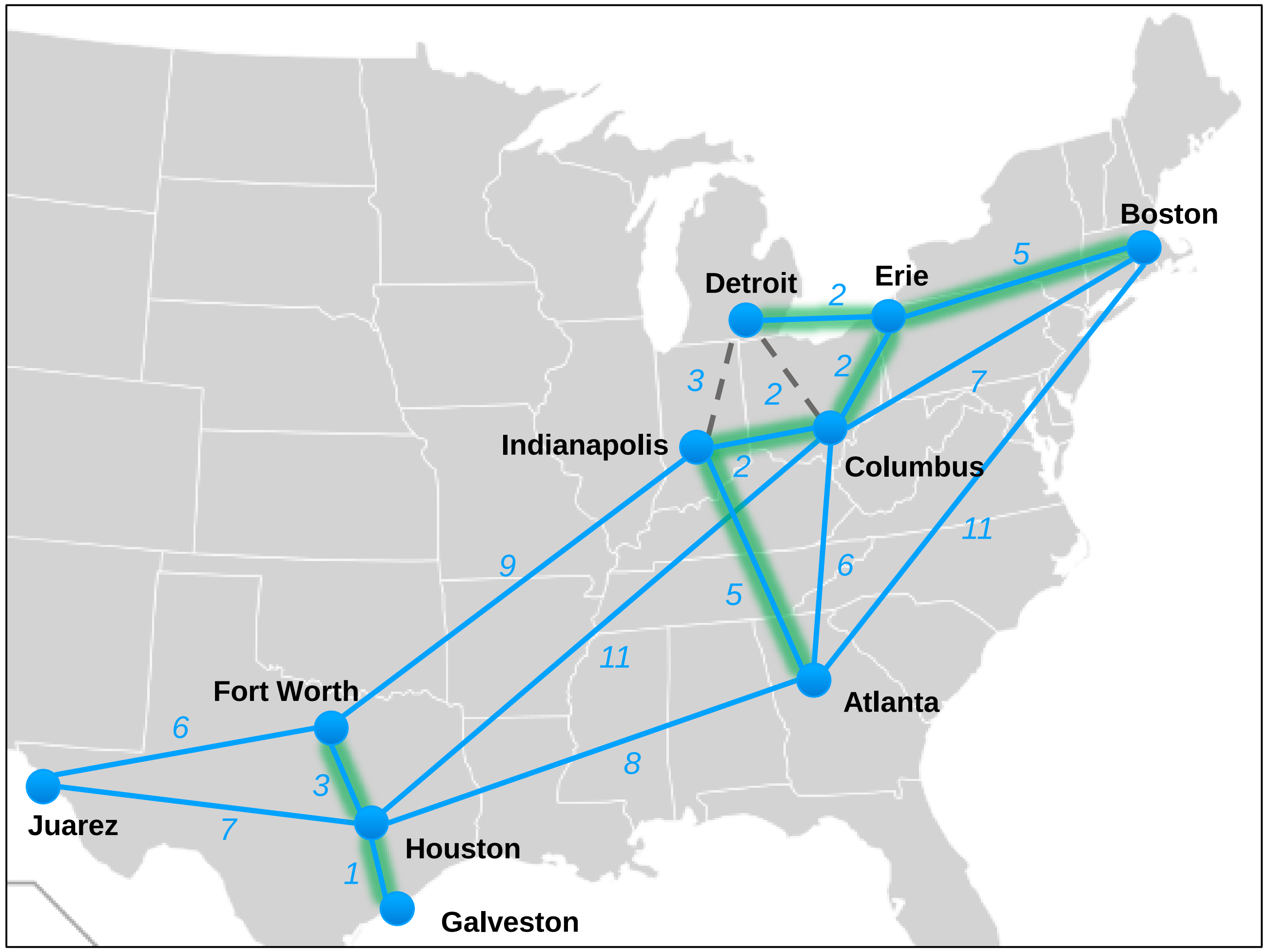


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- ✓ F-H
- ✓ B-E
- **A-I**
- F-J
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	6	-2	6	4	-1
-1	4	4	4	-2	6	-2	6	4	-1

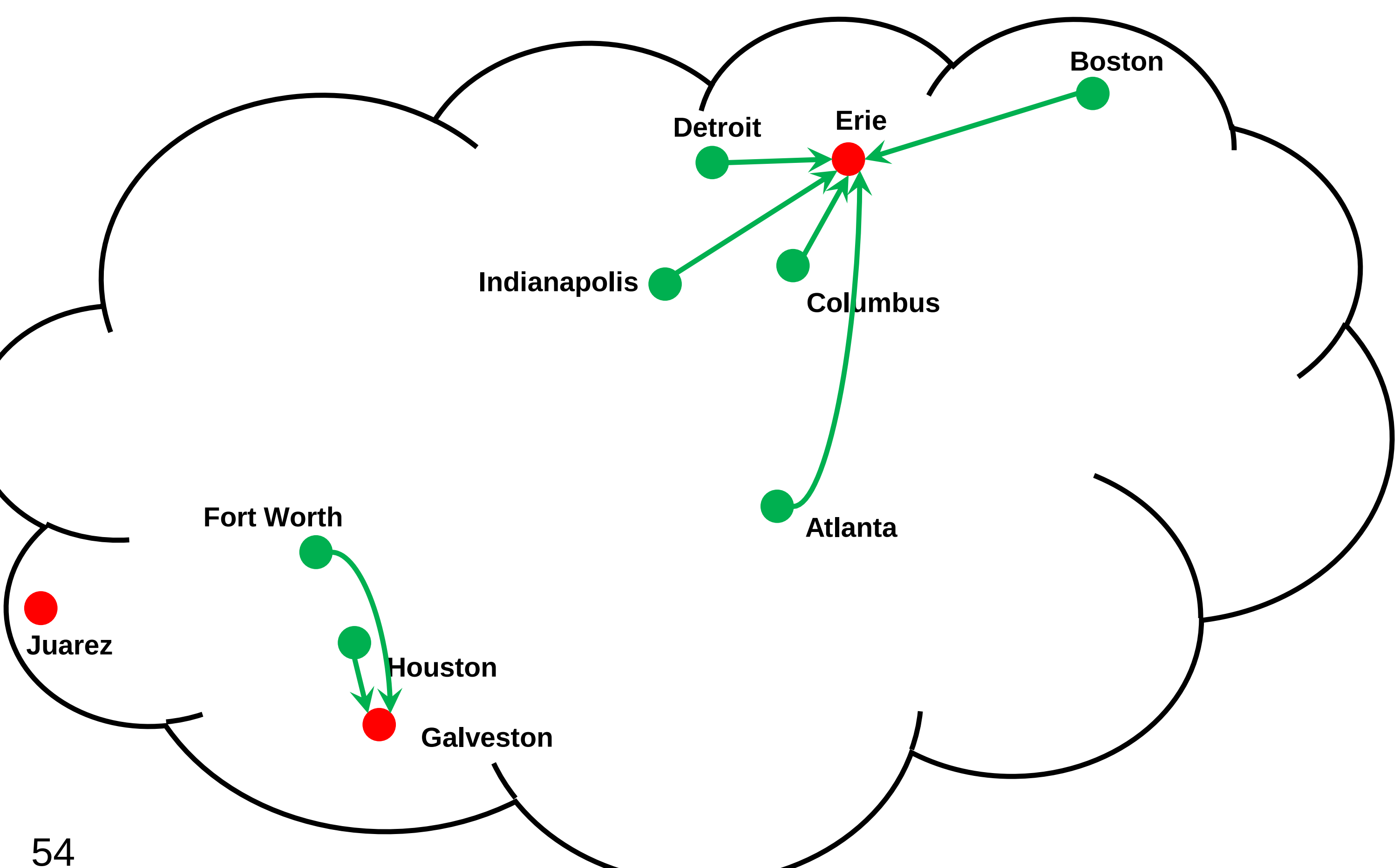


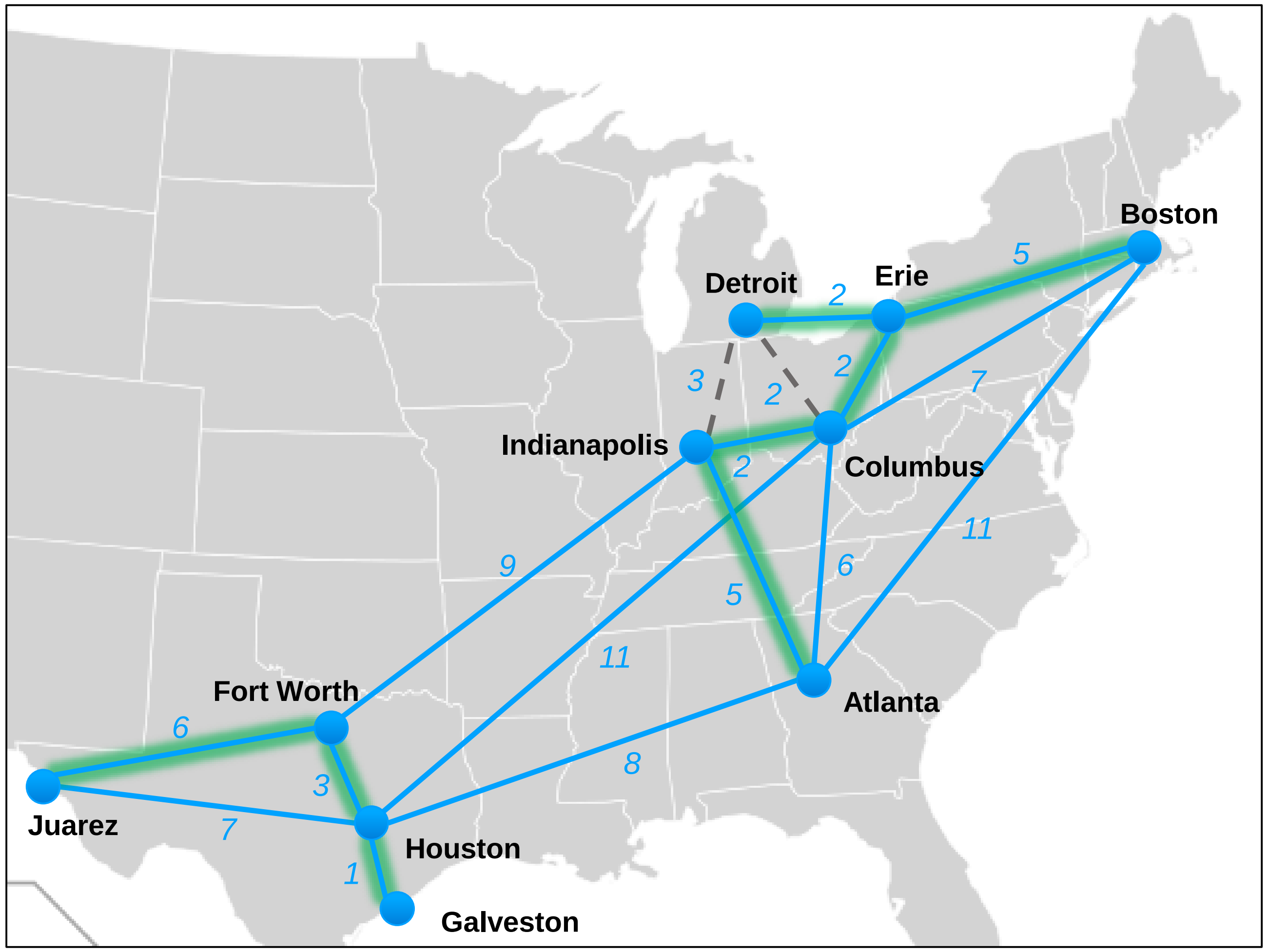


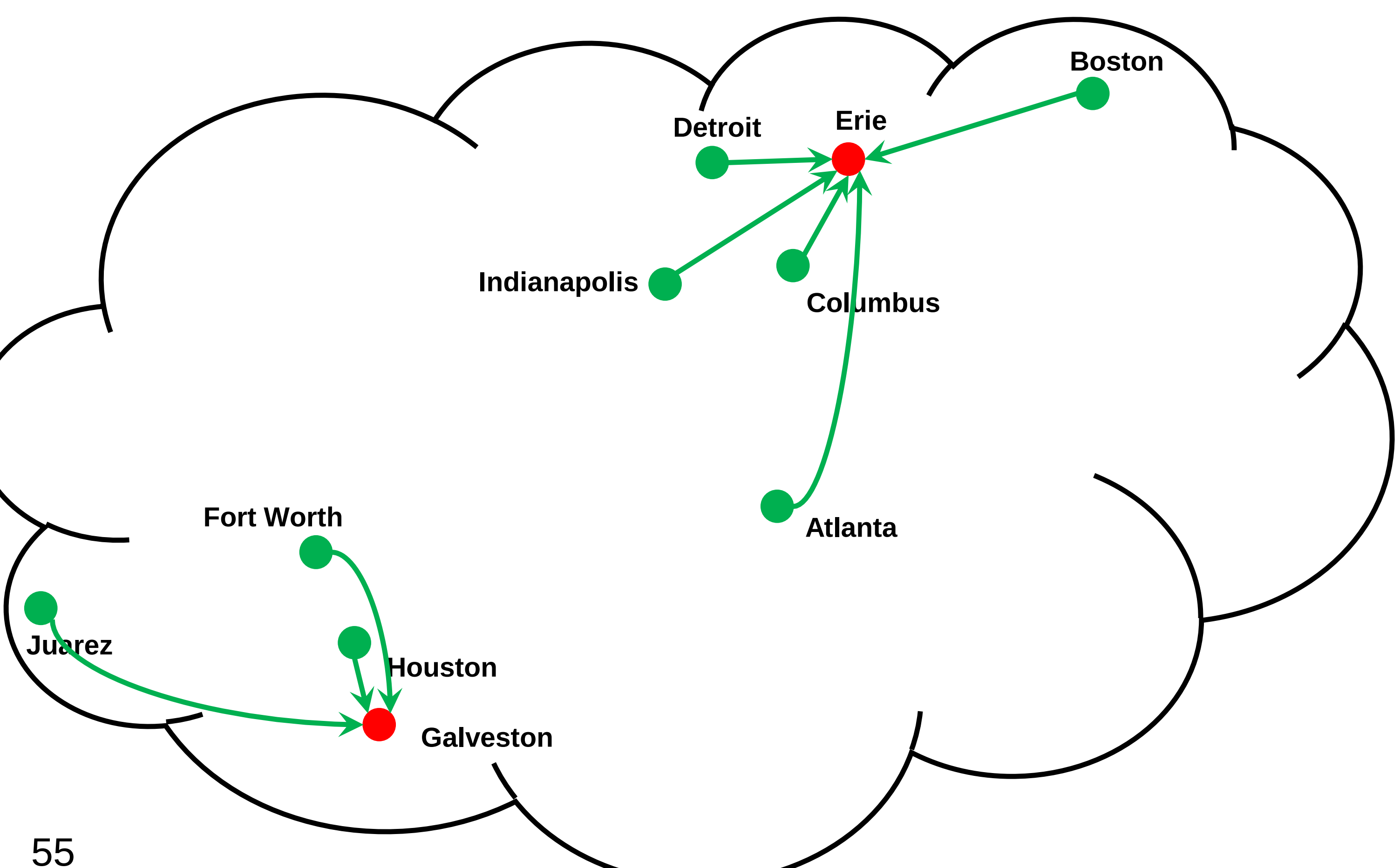
Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- ✗ C-D
- ✗ D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- **F-J**
- A-C
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A 0	*B* 1	*C* 2	*D* 3	*E* 4	*F* 5	*G* 6	*H* 7	*I* 8	*J* 9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	6	-2	6	4	-1
-1	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1



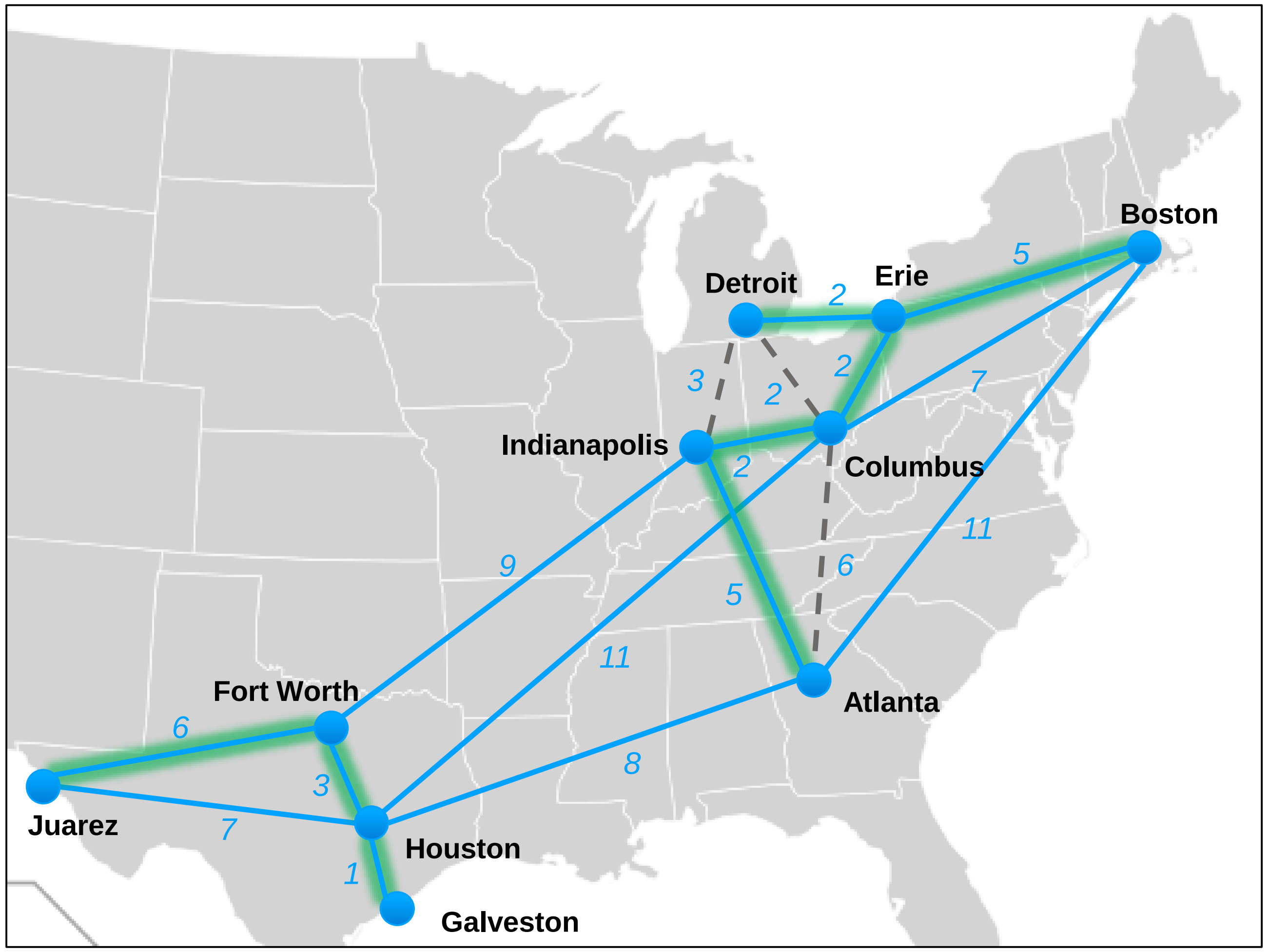


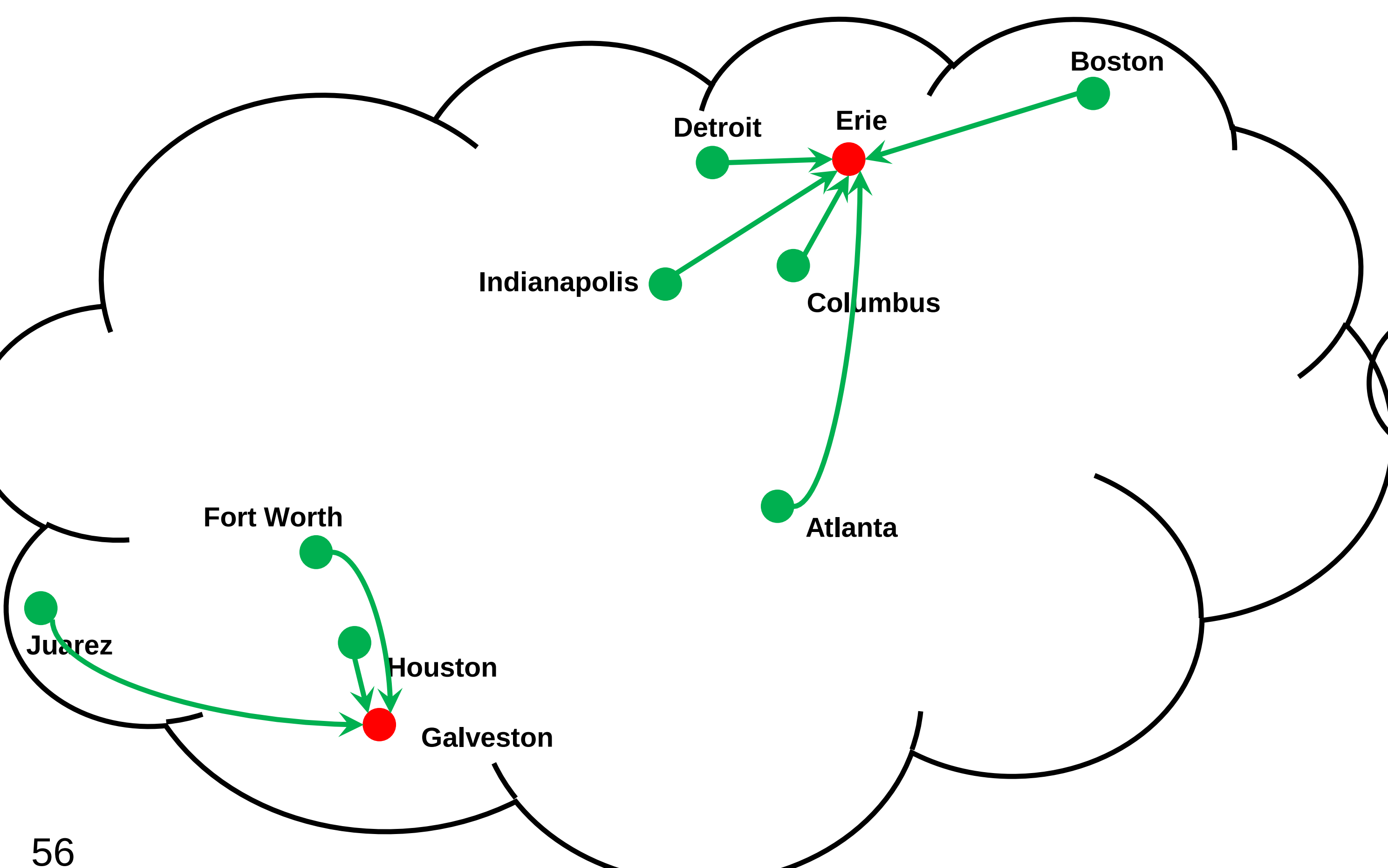


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- **A-C**
- B-C
- H-J
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1
-1	-1	4	**4**	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	**6**	-2	6	4	-1
-1	**4**	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	**6**

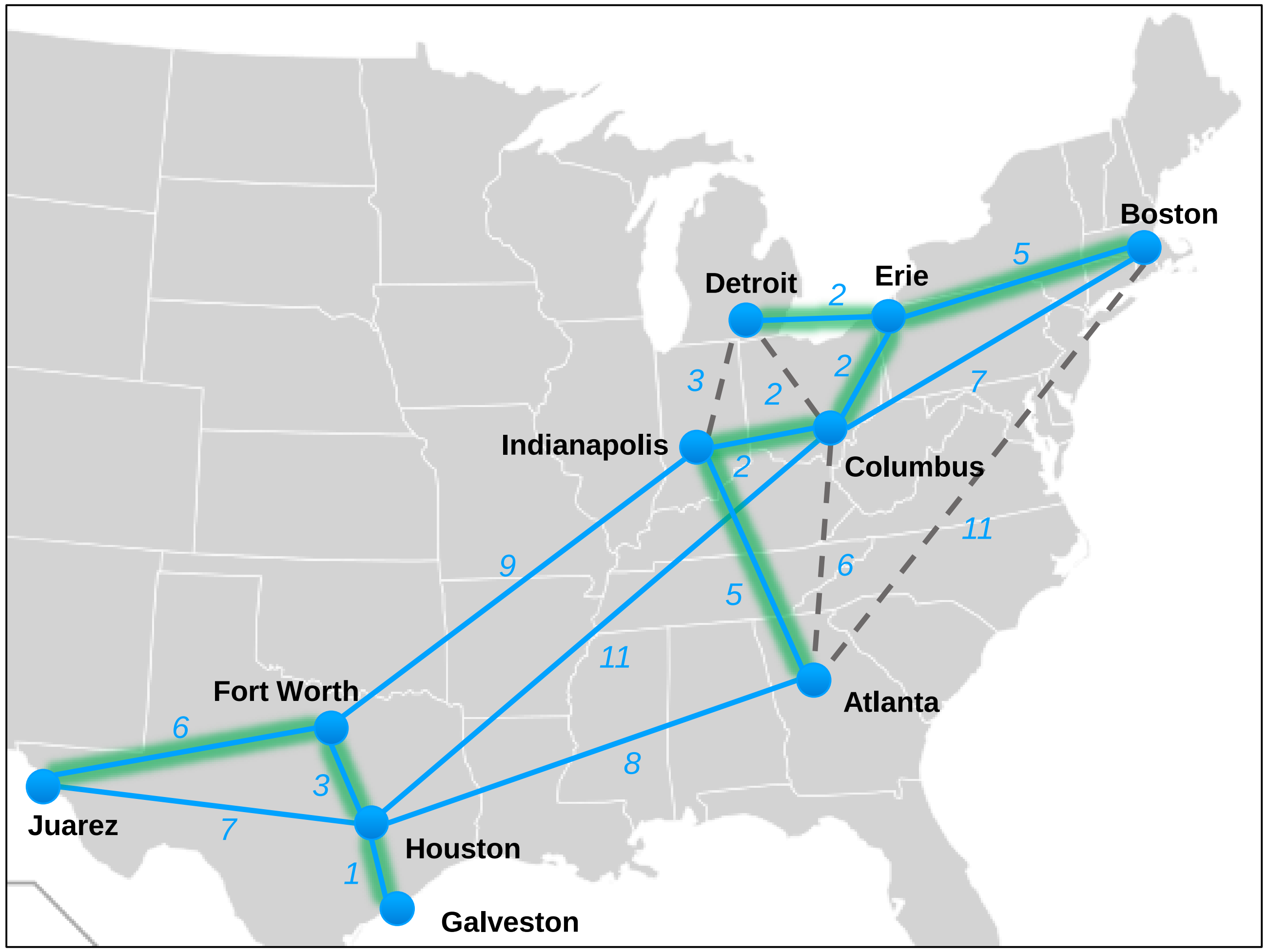


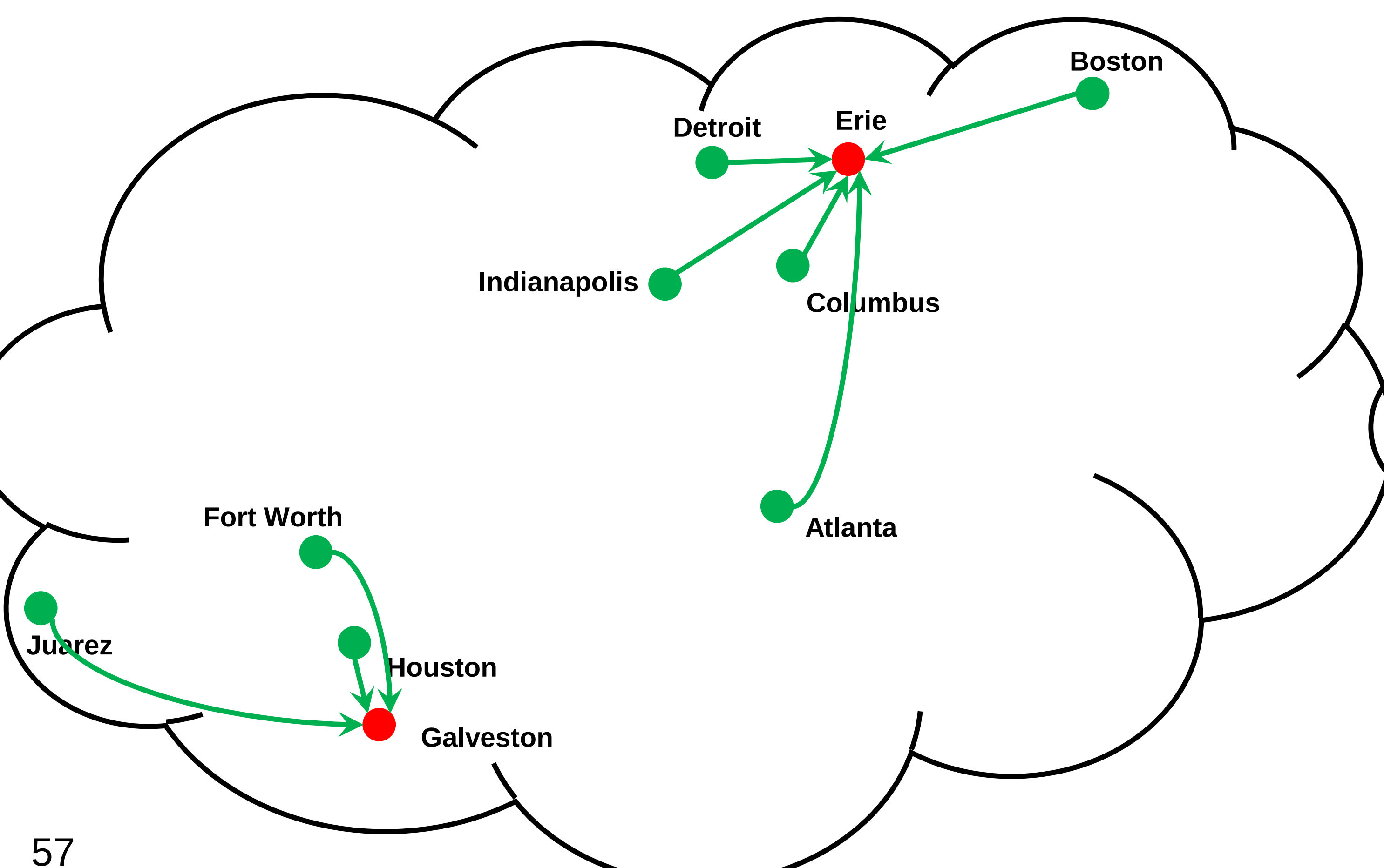


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- × A-C
- **B-C**
- H-J
- A-H
- F-I
- C-H
- A-B

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	6	-2	6	4	-1
-1	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6

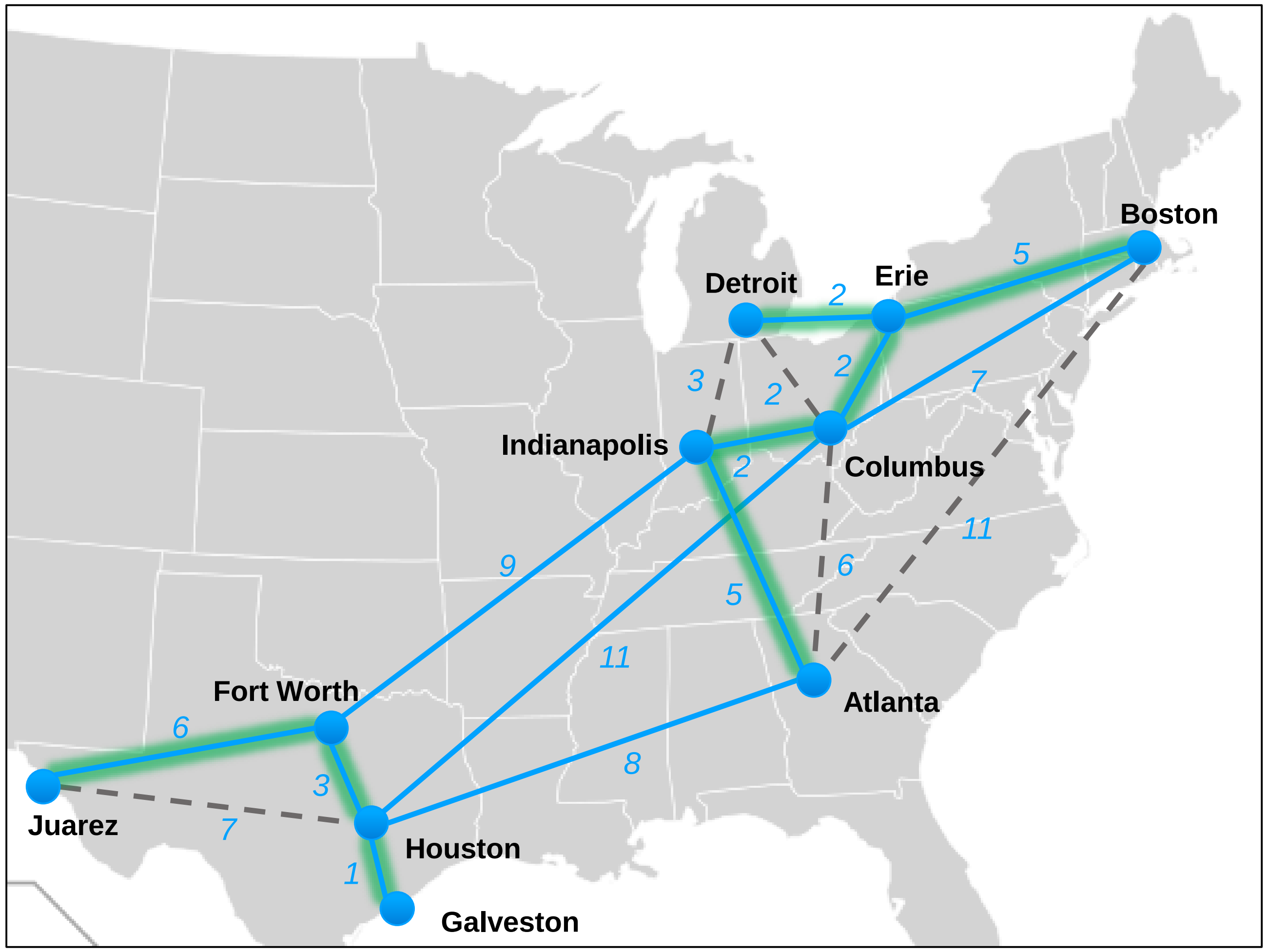


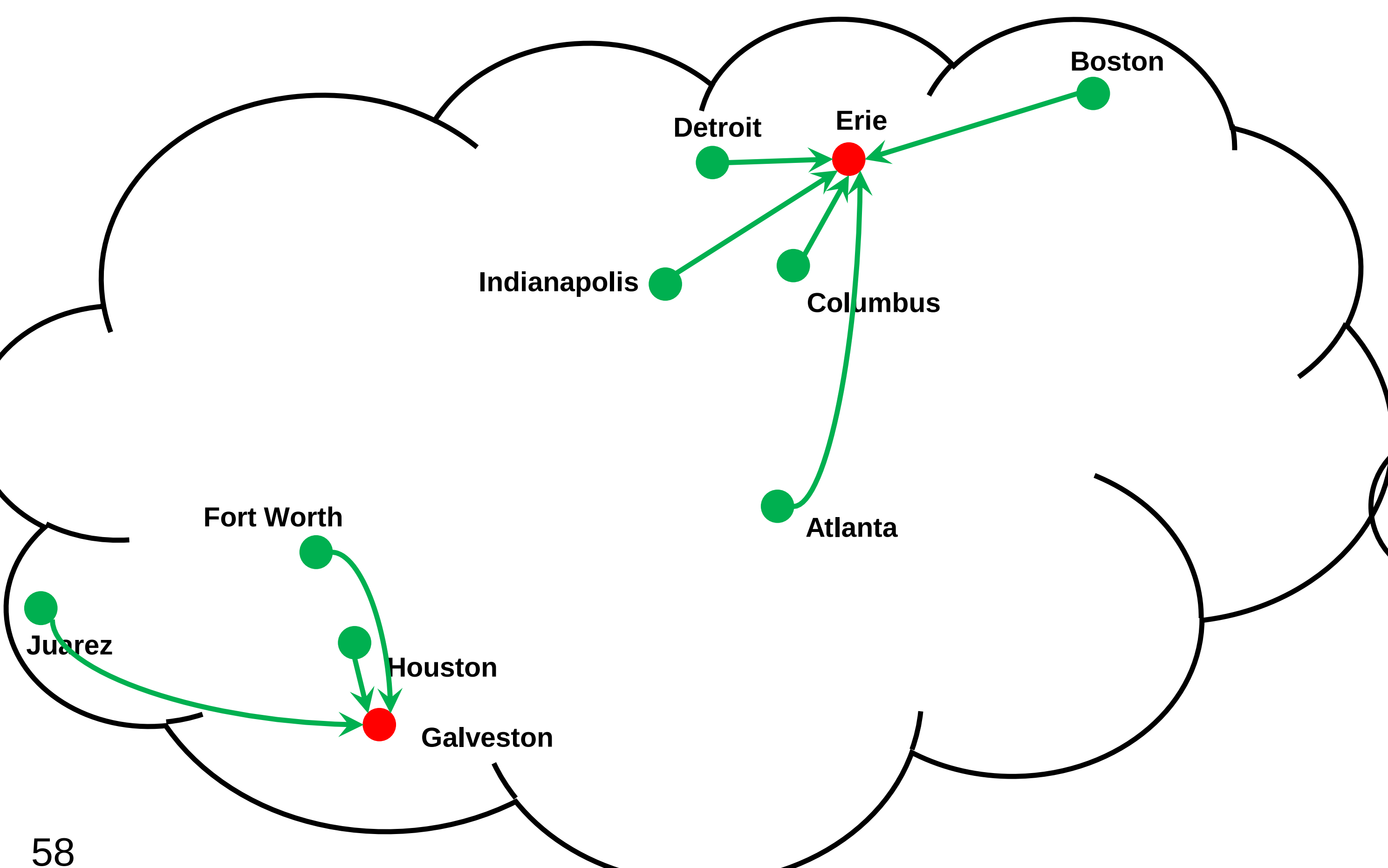


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- ✗ C-D
- ✗ D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- ✗ A-C
- ✗ B-C
- **H-J**
- A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	6	-2	6	4	-1
-1	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6

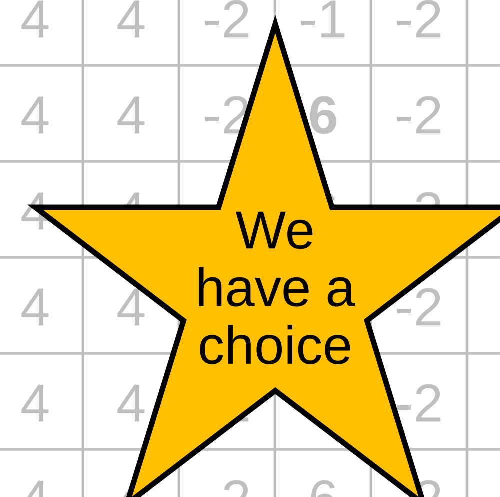


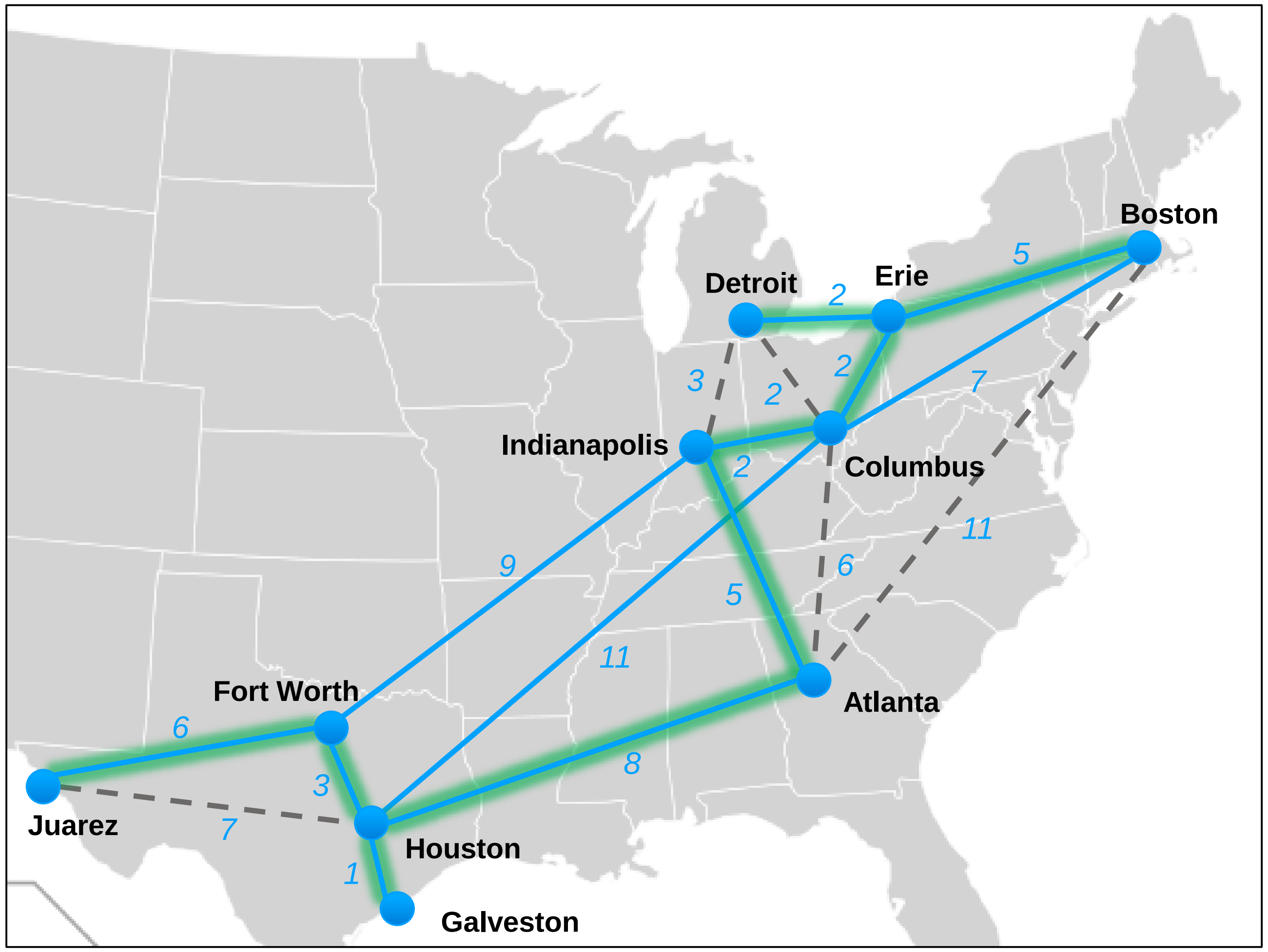


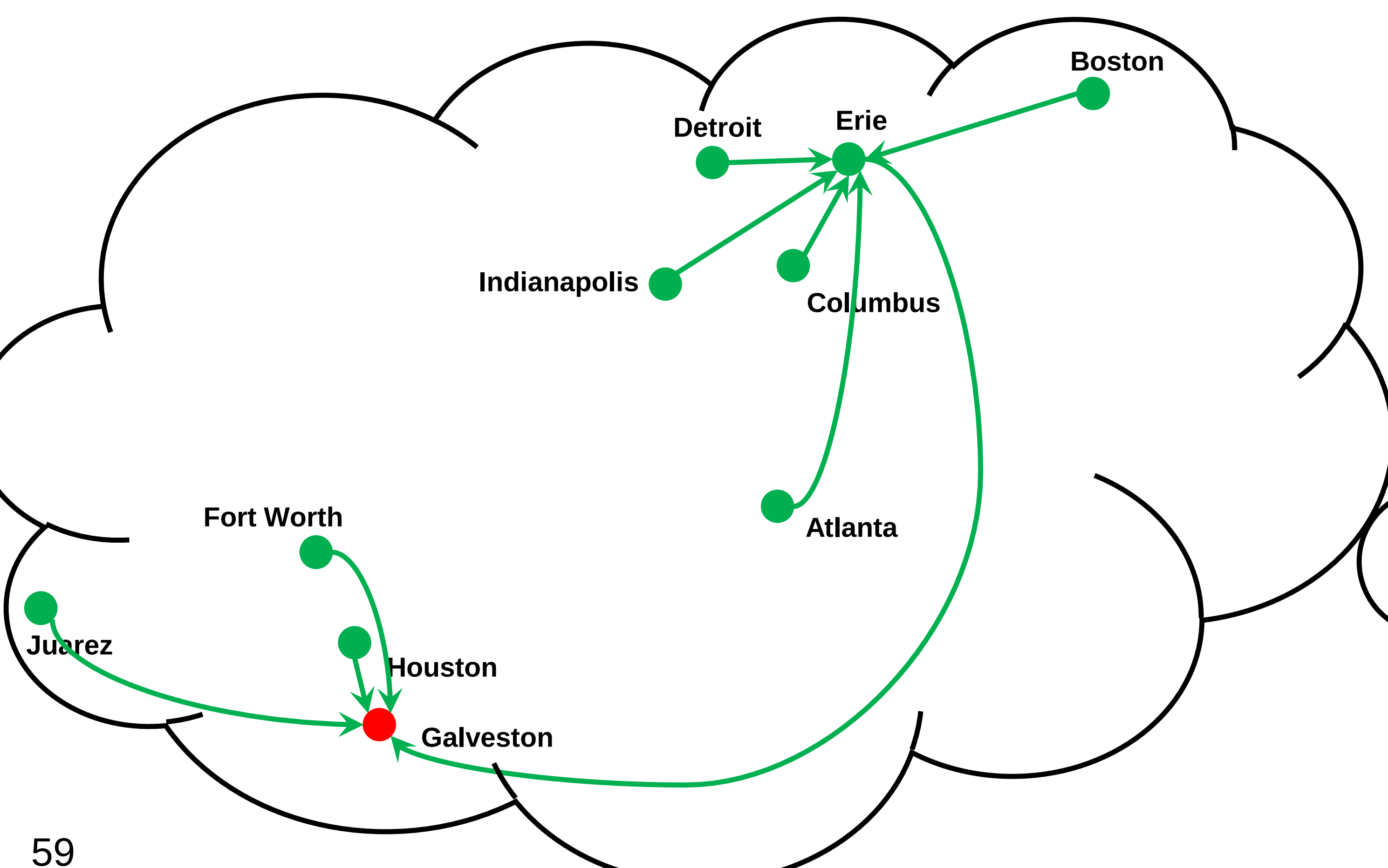
Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- ✗ C-D
- ✗ D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- ✗ A-C
- ✗ B-C
- ✗ H-J
- **A-H**
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1
-1	-1	4	**4**	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	**6**	-2	6	4	-1
-1	**4**	4	[illegible]	[illegible]	[illegible]	-2	6	4	-1
4	4	4	4	[illegible]	[illegible]	-2	6	4	-1
4	4	4	4	[illegible]	[illegible]	-2	6	4	**6**
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6



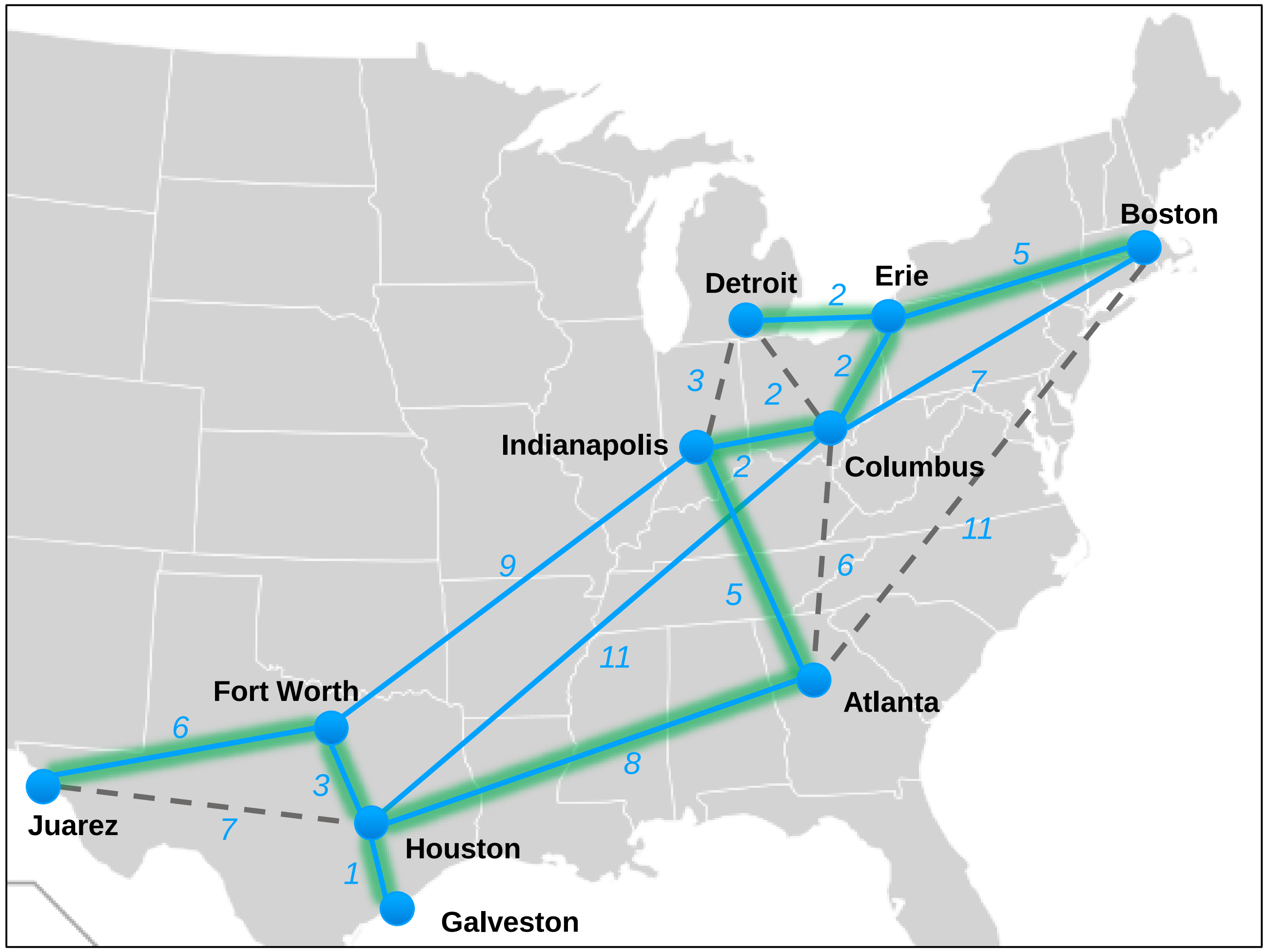


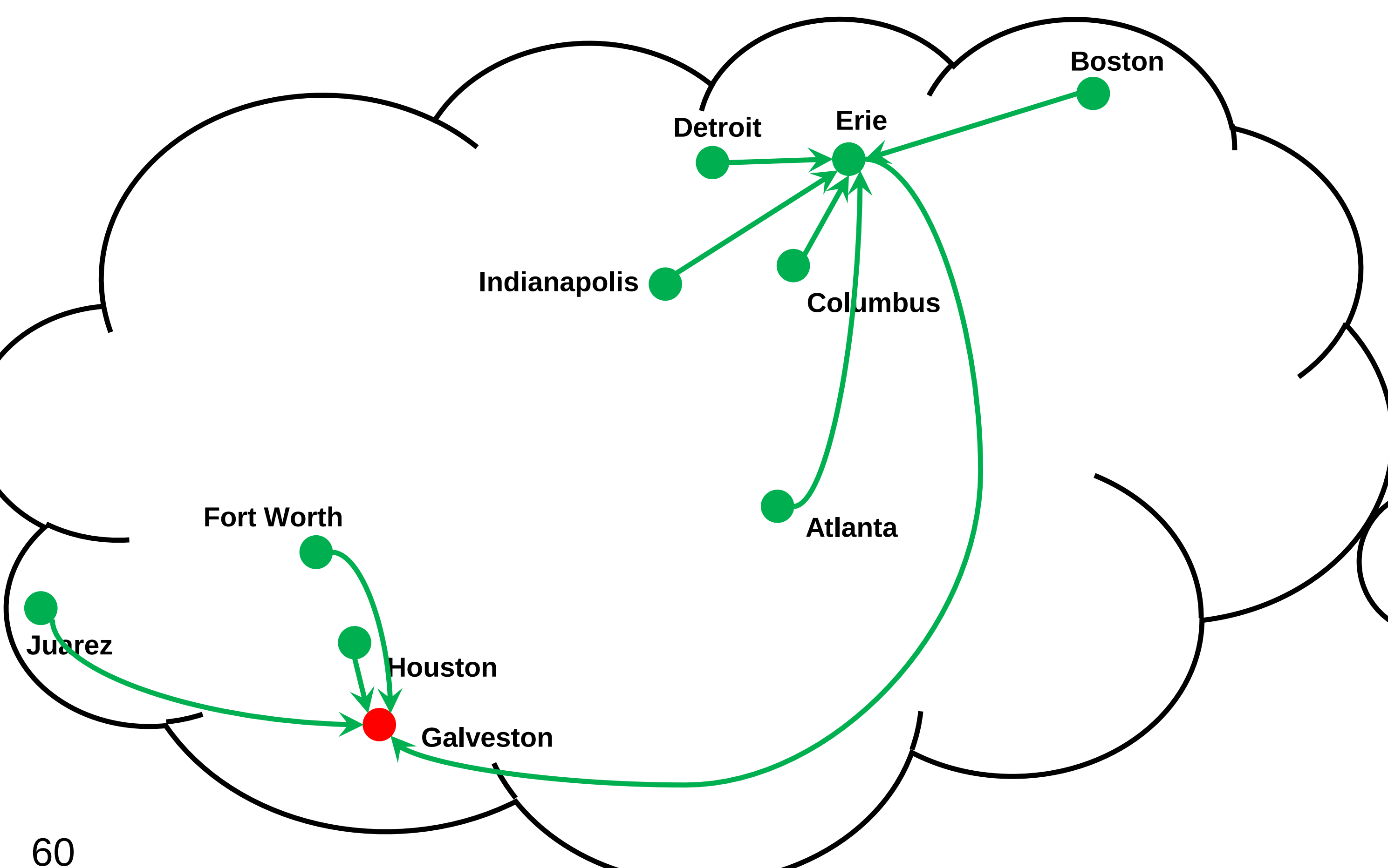


Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- ✗ C-D
- ✗ D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- ✗ A-C
- ✗ B-C
- ✗ H-J
- ✓ A-H
- F-I
- C-H
- A-B

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	6	-2	6	4	-1
-1	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	6	6	-3	6	4	6





Sorted edges

- ✓ G-H
- ✓ C-E
- ✓ C-I
- ✓ D-E
- × C-D
- × D-I
- ✓ F-H
- ✓ B-E
- ✓ A-I
- ✓ F-J
- × A-C
- × B-C
- × H-J
- ✓ A-H
- ~~F-I~~
- ~~C-H~~
- ~~A-B~~

A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	**-2**	**6**	-1	-1
-1	-1	**4**	-1	**-2**	-1	-2	6	-1	-1
-1	-1	4	-1	-2	-1	-2	6	**4**	-1
-1	-1	4	**4**	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	-1	-2	6	4	-1
-1	-1	4	4	-2	**6**	-2	6	4	-1
-1	**4**	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	-1
4	4	4	4	-2	6	-2	6	4	**6**
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	-2	6	-2	6	4	6
4	4	4	4	**6**	6	**-3**	6	4	6

Complexity

- Does union-find with height tracking produce a balanced tree?

- It fees like it does
  - We always merge smaller trees into bigger trees
    - the tree becomes bushier but the height doesn't change
  - The height grows only when merging trees of the same height
    - kind of like balanced binary trees

- Let's turn this into a mathematical property

The Height Property

Property

A tree T of height h has at least 2^{h-1} vertices

Proof

By induction on h

- Base case: $h = 1$
  - Then, T consists of a single vertex
  - and indeed $2^{1-1} = 2^0 = 1$

The Height Property

Proof

By induction on h

- Inductive case: h > 1
  - Then, T was obtained by merging two trees T1 and T2 of height h1 and h2
  - By inductive hypothesis,
    - T1 has at least 2^{h1-1} vertices, and
    - T2 has at least 2^{h2-1} vertices
  - We need to consider 3 subcases
    - Subcase h1 > h2:
      - Then we merged T2 into T1 and h = h1
      - T has at least $2^{h1-1} + 2^{h2-1}$ vertices, which is more than 2^{h1-1} vertices
    - Subcase h2 > h1: (similar)
    - Subcase h1 = h2:
      - Then we either merge T1 into T2 or T2 into T1 to obtain T and h = h1+1
      - T has at least $2^{h1-1} + 2^{h2-1} = 2^{h1-1} + 2^{h1-1} = 2^{h1} = 2^{(h1+1)-1}$ vertices
      - Thus T has at least 2^{h-1} vertices

Complexity

- A tree T of height h has **at least** 2^{h-1} vertices

Then,

- A tree T with v vertices has height **at most** log v + 1

Thus,

- The longest path to the root has length O(log v)
  - T is balanced

This is if we have a single tree

But what if we have a forest?
(like half-way through union-find)

Complexity

- During union-find with height tracking
  - we have a forest of trees
  - each tree T_i with v_i vertices has height **at most** $\log v_i + 1$
  - so, each tree has height **at most** $\log v + 1$

The total number of vertices

- Finding the canonical representative of a vertex costs $O(\log v)$

Complexity

Given a graph G, construct a **minimum spanning tree** T for it

0. Sort the edges of G by increasing weight $O(e \log e)$
1. Start T with the isolated vertices of G $O(v)$
2. For each edge (u,v) in G e times
   - *are u and v already connected in T?* $O(\log v)$
     find the canonical representative of u
     find the canonical representative of v
     check if they are equal
     - **yes**: discard the edge
     - **no**: add it to T $O(1)$
       merge the two connected component
       appoint a new canonical representative
   - Stop once T has v-1 edges

This was O(v)

This was O(1)

$O(v + e \log ev)$

Comparing Spanning Tree Algorithms

- Spanning trees
  - Edge-centric algorithm: $O(v + e \log v)$
  - Vertex-centric algorithm: $O(v + e)$

    Clear winner

- Minimum spanning trees
  - Kruskal's algorithm: $O(v + e \log ev)$
  - Prim's algorithm: $O(v + e \log e)$

    Same in common graphs

- Union-find does not buy us anything
  - but it is useful for checking equivalence
    - independently of spanning trees

Path Compression

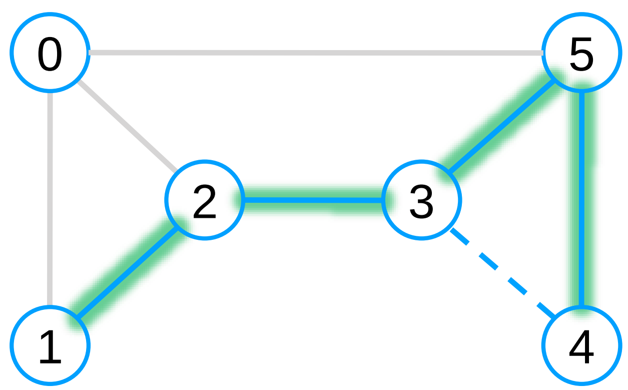
Complexity of Union-find

- Finding a canonical representative costs O(log v)

- *Can we do better?*
  - As we follow a path to the root,
    point all the intermediate nodes to the root

  - This is called **path compression**

Example

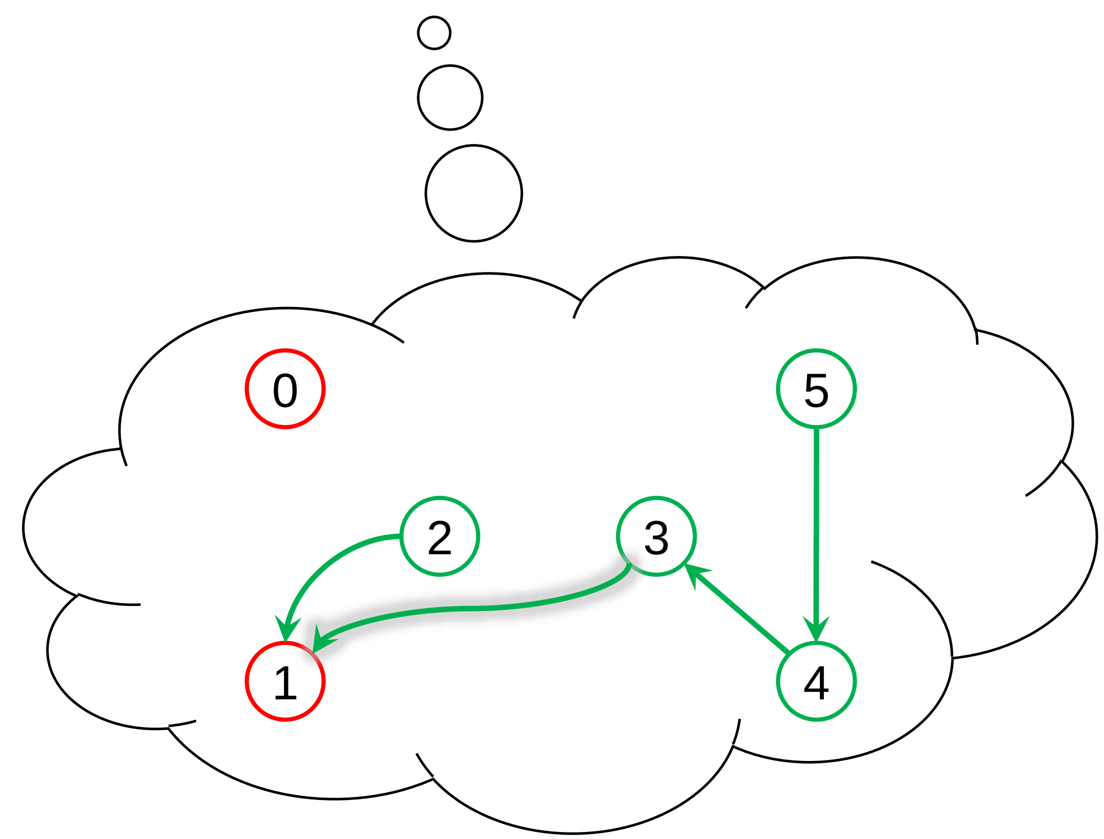


- Earlier example
  - with edge (0,5) added
- This is where we were after adding (2,3)
- We are adding (0,5) next

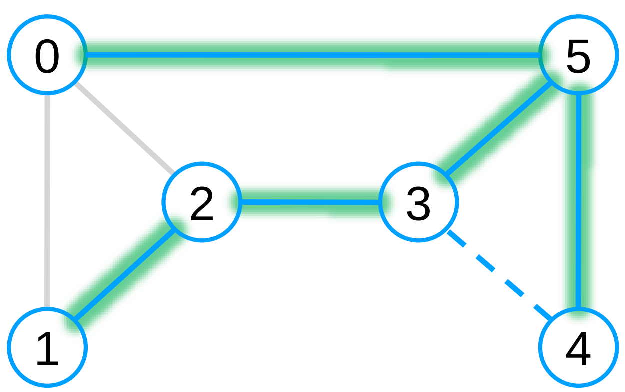
Edges

✓ (4, 5)
✓ (3, 5)
✓ (1, 2)
× (3, 4)
✓ (2, 3)
(0, 5)
(0, 2)
(0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4



Example



Edges

✓ (4, 5)

✓ (3, 5)

✓ (1, 2)

× (3, 4)

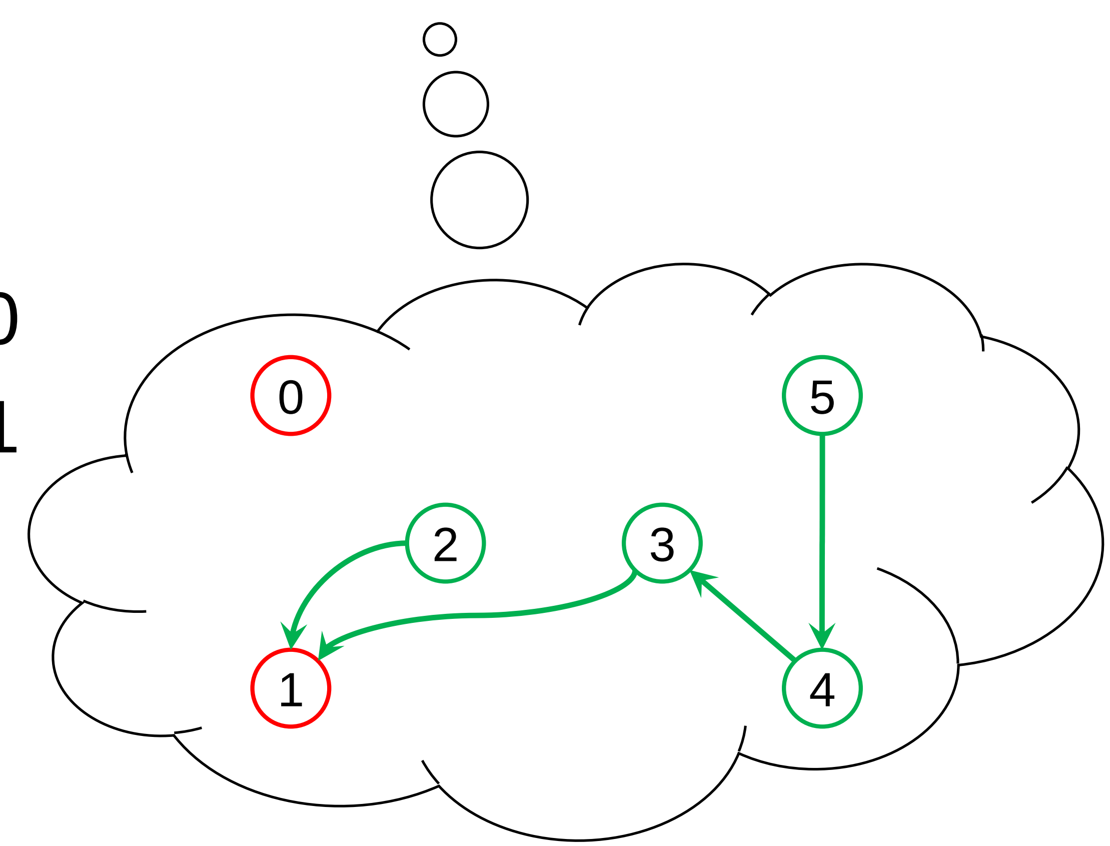
✓ (2, 3)

(0, 5)

(0, 2)

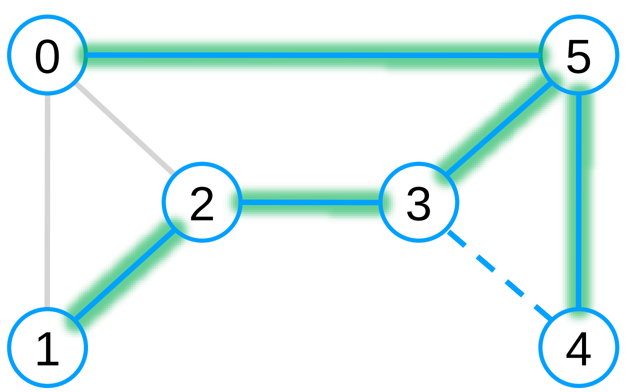
(0, 1)

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4



- We are adding (0,5)
  - the canonical representative of 0 is 0
  - the canonical representative of 5 is 1
    - to find it we go through 5, 4 and 3
    - repoint 5 and 4 them to 1

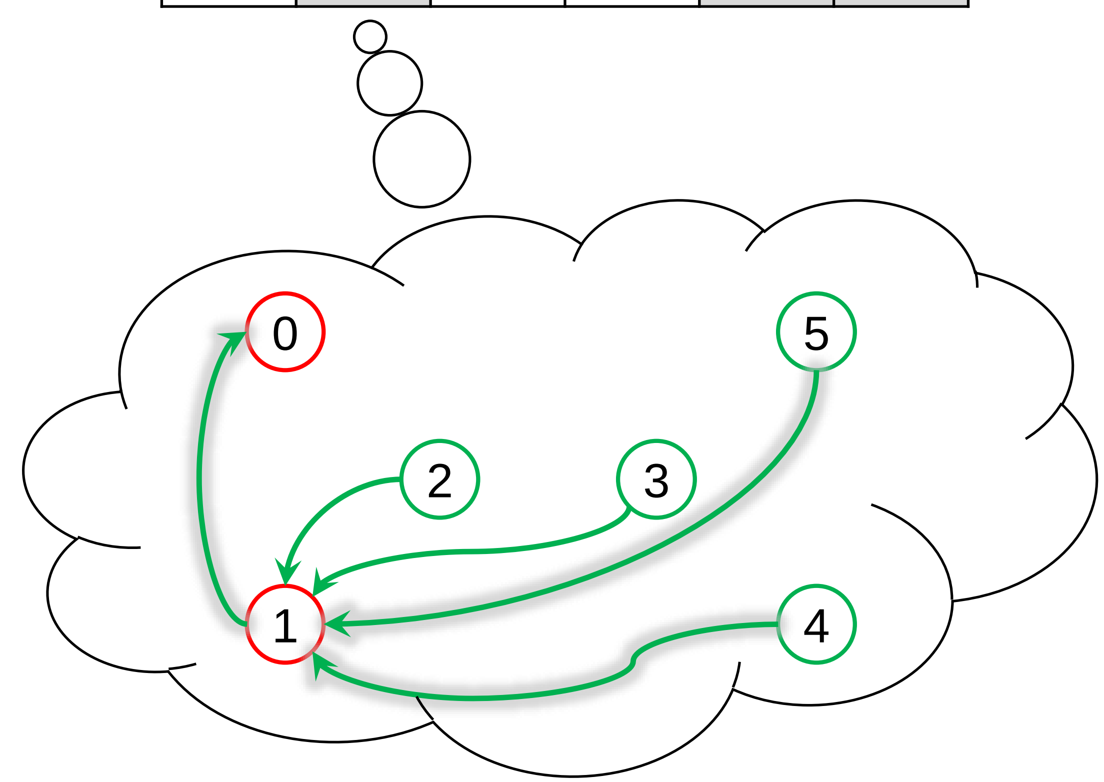
Example



Edges

- ✓ (4, 5)
- ✓ (3, 5)
- ✓ (1, 2)
- × (3, 4)
- ✓ (2, 3)
- ✓ (0, 5)
- ~~(0, 2)~~
- ~~(0, 1)~~

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	**4**
0	1	2	3	**3**	4
0	1	**1**	3	3	4
0	1	1	3	3	4
0	1	1	**1**	3	4
0	0	1	**1**	1	1



- We added (0,5)
  - we already have 5 edges
  - we ignore the remaining edges

The Ackermann Function

Wilhelm Ackermann

$$\begin{cases} Ack(0, n) = n+1 & \\ Ack(m, 0) = Ack(m-1, 1) & \text{if } m > 0 \\ Ack(m, n) = Ack(m-1, Ack(m, n-1)) & \text{if } m, n > 0 \end{cases}$$

$$A(n) = Ack(n, n)$$

- The Ackermann function grows very very fast
  - A(0) = 1
  - A(1) = 3
  - A(2) = 7
  - A(3) = 61
  - A(4) > number of atoms in the universe
- The inverse of the Ackermann function, $A^{-1}(n)$, grows **very very slowly**

That's the function such that $A^{-1}(A(n)) = n$

Complexity of Path Compression

- The cost of finding the canonical representative of a vertex using union-find with path compression is

  $O(A^{-1}(v))$ amortized

  - That a hair above O(1)

That's All, Folks