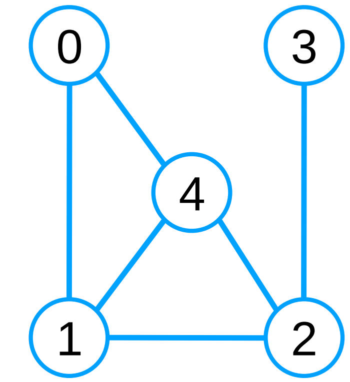
Spanning Trees

Review

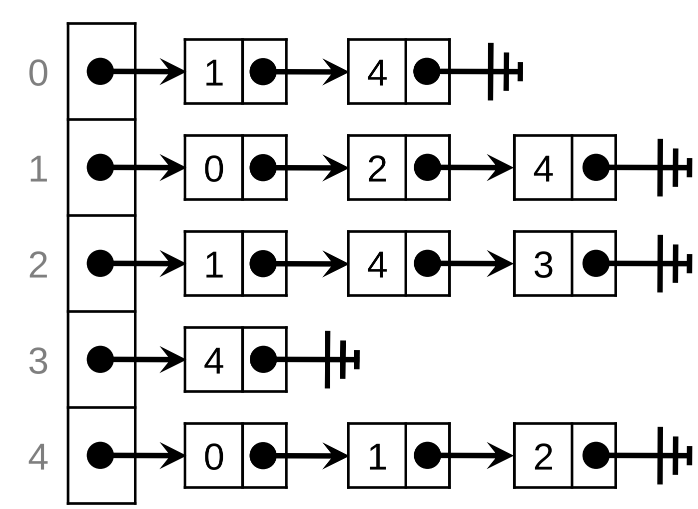


- Graphs
  - Vertices, edges, neighbors, paths, …
  - Dense, sparse

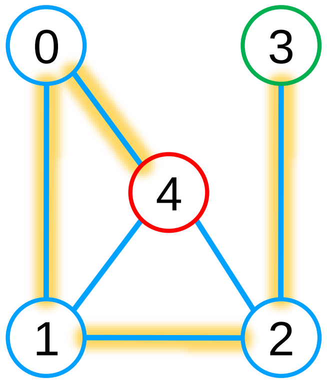
- Adjacency matrix implementation

	0	1	2	3	4
0		✓			✓
1	✓		✓		✓
2		✓		✓	✓
3			✓		
4	✓	✓	✓		

- Adjacency list implementation



Review



- Graph search
  - Determine whether two vertices are connected
    - and possibly report a path that connects them
- Explore the graph by expanding the frontier

Remember the vertices to visit next a work list

  - Depth-first search

in a stack

    - Charge ahead until we find the target vertex or hit a dead-end
    - then backtrack
  - Breadth-first search

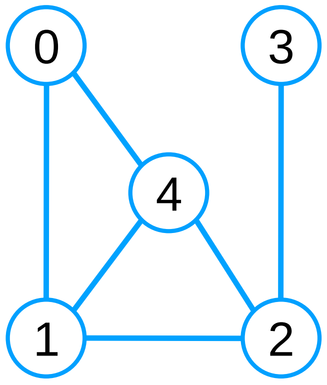
in a queue

    - Explore the graph level-by level
- Complexity

	DFS	BFS
Adjacency list	O(v + e)	O(v + e)
Adjacency matrix	$O(v^2)$	$O(v^2)$

Trees

Cycles

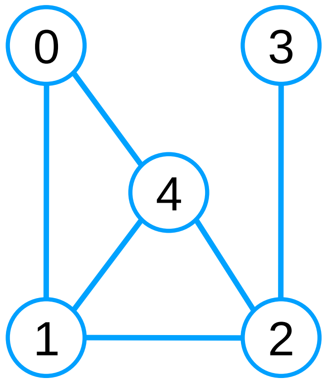


- A **cycle** is a path from a vertex to itself
  - 0–1–4–0 is a cycle
  - 0–1–0 is a cycle
  - 0 is a cycle too

- A **simple cycle** is a cycle
  with at least one edge and
  without repeated edges
  - 0–1–4–0 is a simple cycle
  - 0–1–0 is **not** a simple cycle
  - 0 is **not** a simple cycle either

these are **trivial** cycles

Simple Cycles



A cycle without repeated edges

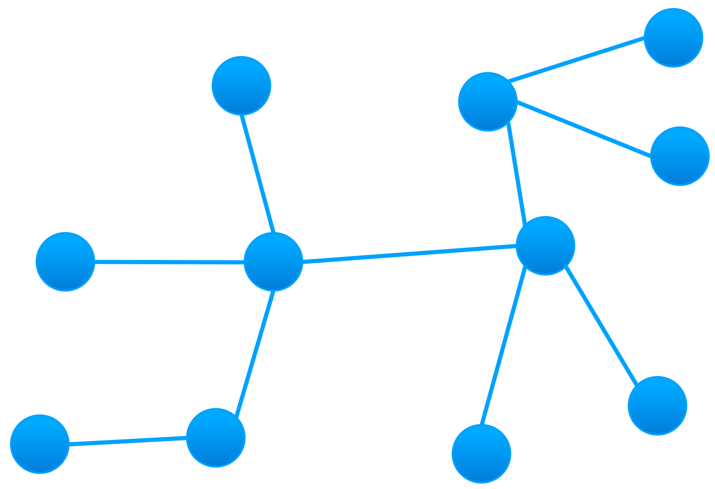
- ➢ and at least one edge

- Simple cycles are what forces us to use a mark array in DFS and BFS
  - After following edge (0,1) to go from 0 to 1,
    it is easy to avoid using (0,1) to go back to 0
    - ➢ remembering where we come from is trivial
  - After following (0,1) and (1,4) to go from 0 to 4,
    it is hard to know we shouldn't use (0,4)
    - ➢ unless we mark visited vertices

- Graphs without simple cycles are convenient to work with
  - no need for mark arrays

Trees

- A connected graph without simple cycles is called a **tree**

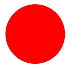


- The are many ways to define a tree

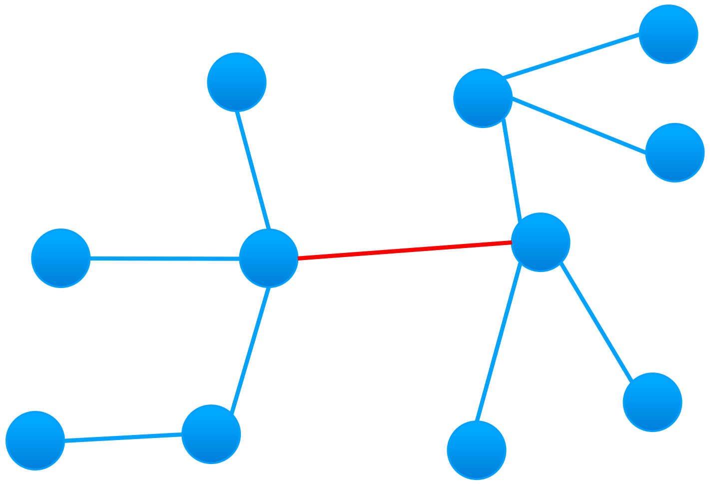
A Recursive Definition

We can also define trees recursively

A **tree** is

- a vertex by itself 

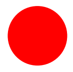
- two trees connected by an edge



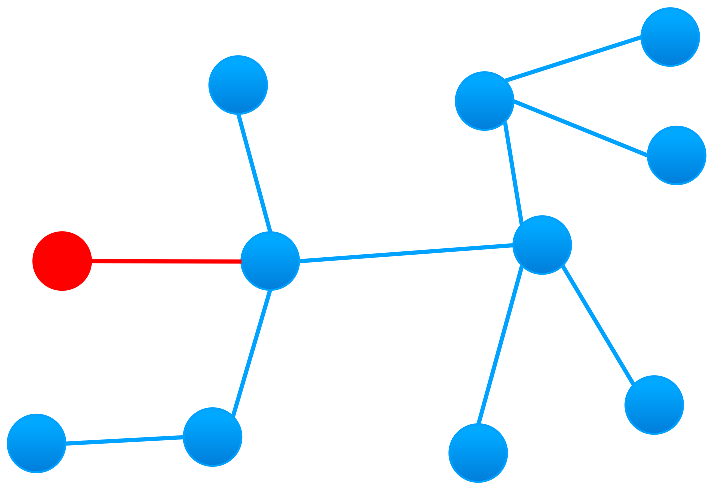
Another Recursive Definition

We can define trees recursively in several ways

A **tree** is

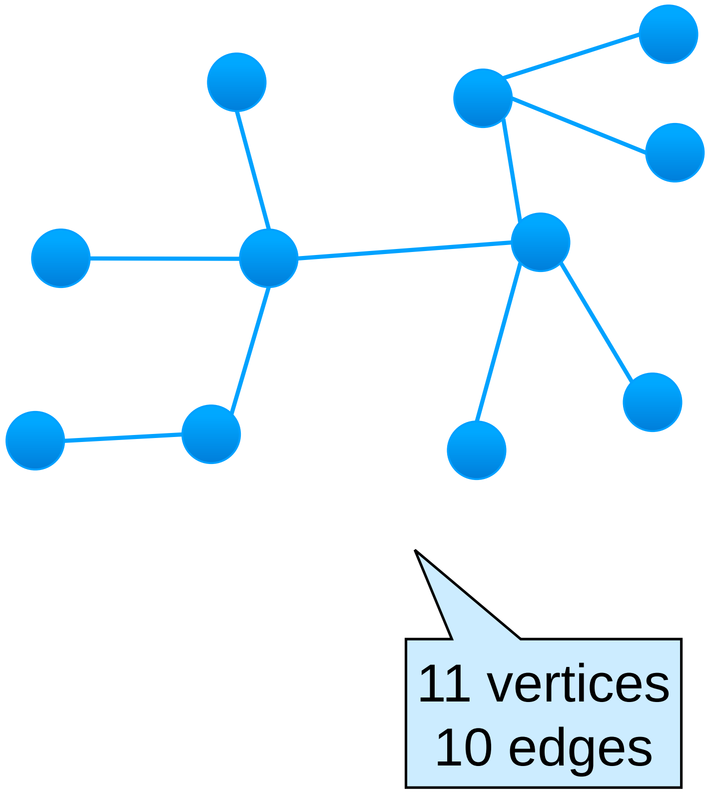
- a vertex by itself 

- a tree connected to a vertex by an edge



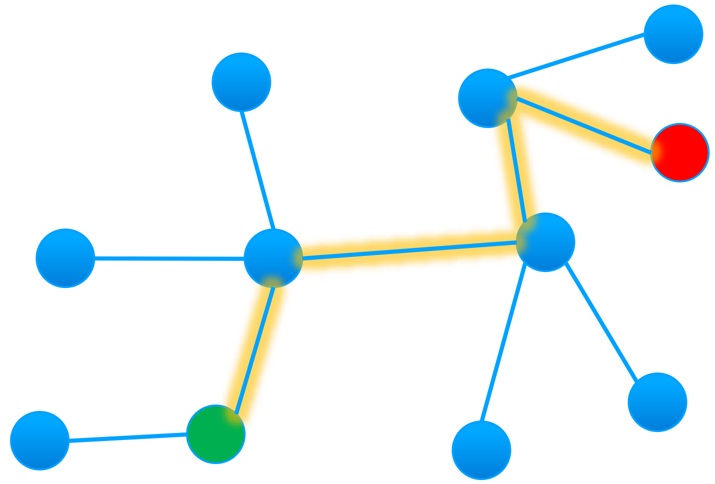
The Edges of a Tree

- A **tree** is a connected graph with **v** vertices and **v-1** edges



The Paths of a Tree

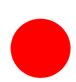
- A tree is a connected graph with exactly one path between any two vertices



The Edges of a Tree

- We can prove that these definitions are equivalent
  - *For example,*
    if we define a tree as a vertex by itself or a tree connected to a vertex by an edge,
    then if such a graph has v vertices it has v-1 edges

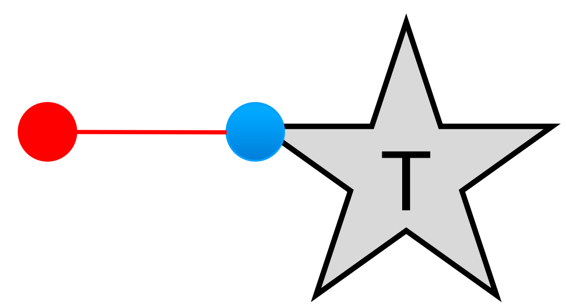
A vertex by itself



This graph has
- 1 vertex
- 0 = 1-1 edges

base case

A tree connected to a vertex by an edge



Assume by induction hypothesis that T has v vertices and v-1 edge.

Then, this graph has
- v+1 vertices
- v-1+1 = (v+1) - 1 edges

recursive case

In Summary, a Tree is …

A. a connected graph with no simple cycles

B. (*recursive definition #1*)
  - a vertex
  - two trees connected by an edge

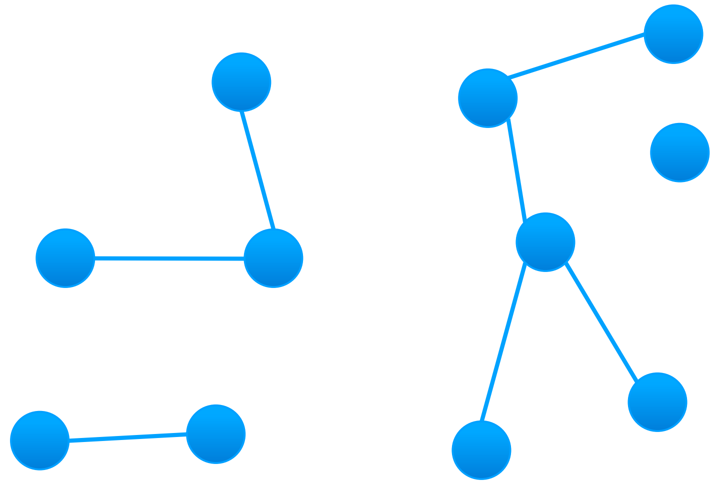
C. (*recursive definition #2*)
  - a vertex
  - a tree connected to a vertex by an edge

D. a connected graph with v vertices and v-1 edges

E. a connected graph with exactly 1 path between any two vertices

Forest

- A **forest** is a bunch of trees
  - a graph where each connected component is a tree
- Other definitions
  - that was the definition of a tree
  - a forest is a ~~*connected*~~ graph with no simple cycles
  - a graph with ***at most*** one path between any two vertices
- A forest with v vertices has ***at most*** v-1 edges



Reachability Problem on a Tree

- What is the cost of DFS or BSF on a tree?
  - assuming an adjacency list implementation

O(v) — always

- DFS and BFS cost O(v + e) in general
- in a tree, e = v-1
  - definition D
- so, the cost reduces to O(v)

A. a connected graph with no simple cycles
B. (*recursive definition #1*)
   - a vertex
   - two trees connected by an edge
C. (*recursive definition #2*)
   - a vertex
   - ~~a tree connected to a vertex by an edge~~
D. a connected graph with v vertices and v-1 edges
E. a connected graph with exactly 1 path between any two vertices

Are BSTs Trees?

- A binary search tree is a **tree** where every vertex has **at most 3 edges**
  - two children
  - one parent

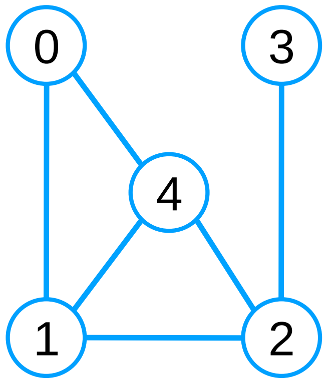
  (plus there is the ordering invariant)

- Which node is the root?
  - any vertex with at most 2 edges
    - the root does not have a parent

Simply hoist the graph by that node

Spanning Trees

Reaching Nodes Over and Over

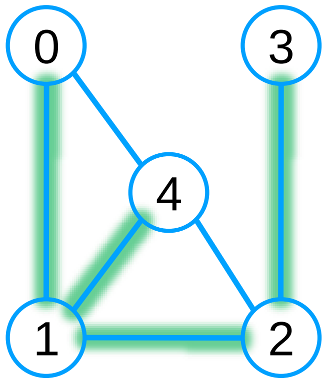


- Some applications need to frequently reach a connected vertex in a graph
  - ➢ diagnosis in communication networks
  - ➢ billing in power networks, ..

- We can use DFS or BFS
  - ○ but this is expensive: $O(v + e)$ each query
  - ○ it may go through a different path for the same query each time

- We can remember the paths
  - ○ but this requires a lot of space
    - ➢ $O(v^2)$ in each vertex
      - ❑ each vertex needs to remember v-1 paths
      - ❑ each of these paths can contain up to v-1 vertices
    - ➢ $O(v^3)$ for the whole graph

Reaching Nodes Over and Over



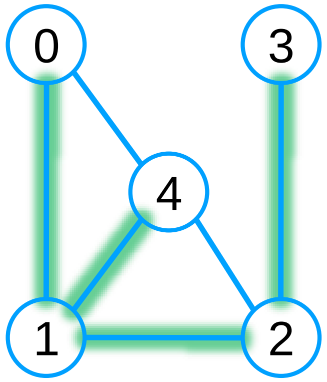
- Some applications need to frequently reach a connected vertex in a graph
  - using DFS or BFS is too expensive
  - remembering paths to all vertices is $O(v^2)$ in *each* vertex

Idea

- **Factor out the common subpaths** by superimposing a **tree** on the graph
  - provides a path from every vertex to every other vertex
  - requires $O(v)$ space in each vertex
    - $O(v)$ total if we have a "path server" vertex
- This is a **spanning tree**

If the graph has more than one connected component, we superimpose one spanning tree on each connected component — this is a spanning forest

Spanning Tree



- *Factor out the common subpaths by superimposing a tree (or forest) on the graph*

Formally,

- A **subgraph** of a graph G is a graph with the same vertices and a subset of its edges
- A **spanning tree** for G is a subgraph that
  - has the same connectivity as G
  - and is a tree

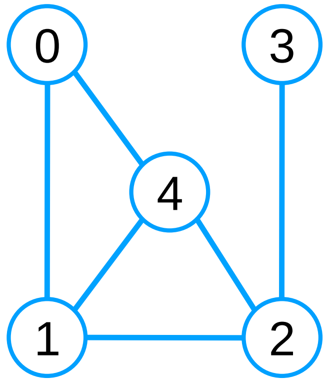
A graph has a spanning trees only if it consists of a single connected component

- A **spanning forest** for G is a subgraph that
  - has the same connectivity as G
  - and is a forest

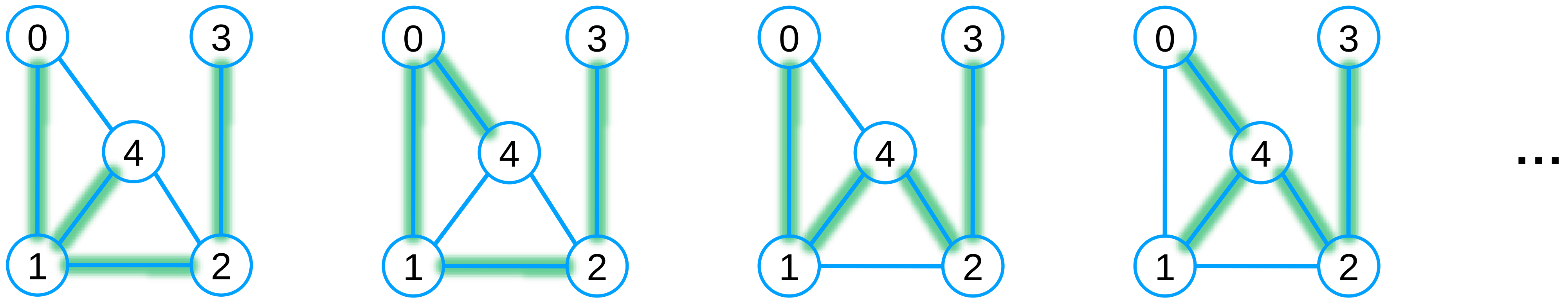
a bunch of spanning trees

The Spanning Trees of a Graph

- Most graphs have multiple spanning trees



- Here are some



- In general, any spanning tree will do

How to Compute a Spanning Tree?

Two classic algorithms

- The **edge-centric algorithm**

  Start with a spanning forest of singleton trees and add edges from the graph as long as they don't form a cycle

This leverages definition B
A tree is
- a vertex, or
- two trees connected by an edge

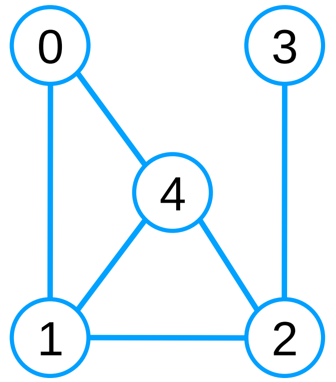
- The **vertex-centric algorithm**

  Start with a single vertex in the tree and add edges to vertices not in the tree

This leverages definition C
A tree is
- a vertex, or
- a trees connected to a vertex by an edge

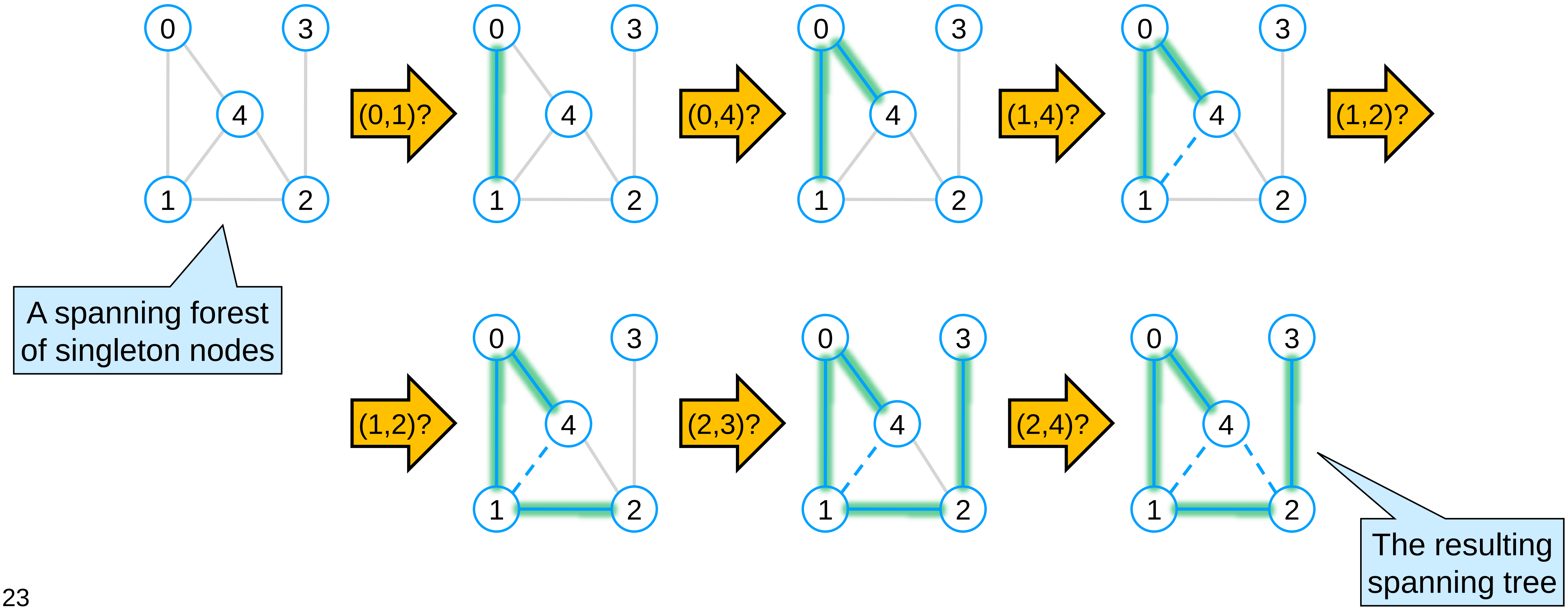
Edge-centric Algorithm

The Edge-centric Algorithm



Start with a spanning forest of singleton trees and add edges from the graph as long as they don't form a cycle

- Let's run it on the example graph



Towards an Actual Algorithm

Start with a spanning forest of singleton trees and add edges from the graph as long as they don't form a cycle

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

If G has more than 1 connected component, this will produce a spanning forest

Towards an Actual Algorithm

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

- Is there room for improvement?
  - Stop as soon as we added v-1 edges in T

By definition D
A tree is a connected graph
v vertices and v-1 edges

The Edge-centric Algorithm

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?*
     - **yes**: discard the edge
     - **no**: add it to T
   - Stop once T has v-1 edges

This won't apply if G has more than 1 connected component

- What is its complexity?

Complexity

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

1. Start T with the isolated vertices of G — O(v) — This is just graph_new
2. For each edge (u,v) in G — e times
   - *are u and v already connected in T?* — O(v) — Use DFS or BFS on T for this
     - **yes**: discard the edge
     - **no**: add it to T — O(1) — This is graph_addedge
   - Stop once T has v-1 edges

Complexity

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

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

Use DFS or BFS on T for this

- We run DFS/BFS on T
  - at most v-1 edges
  - the cost is O(v)
    - not O(e)

Complexity

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

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

- Even if we end up adding at most v-1 edges, we may need to go through all the edges in e

Complexity

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

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

- The edge-centric algorithm has complexity **O(ev)**

Greedy Algorithms

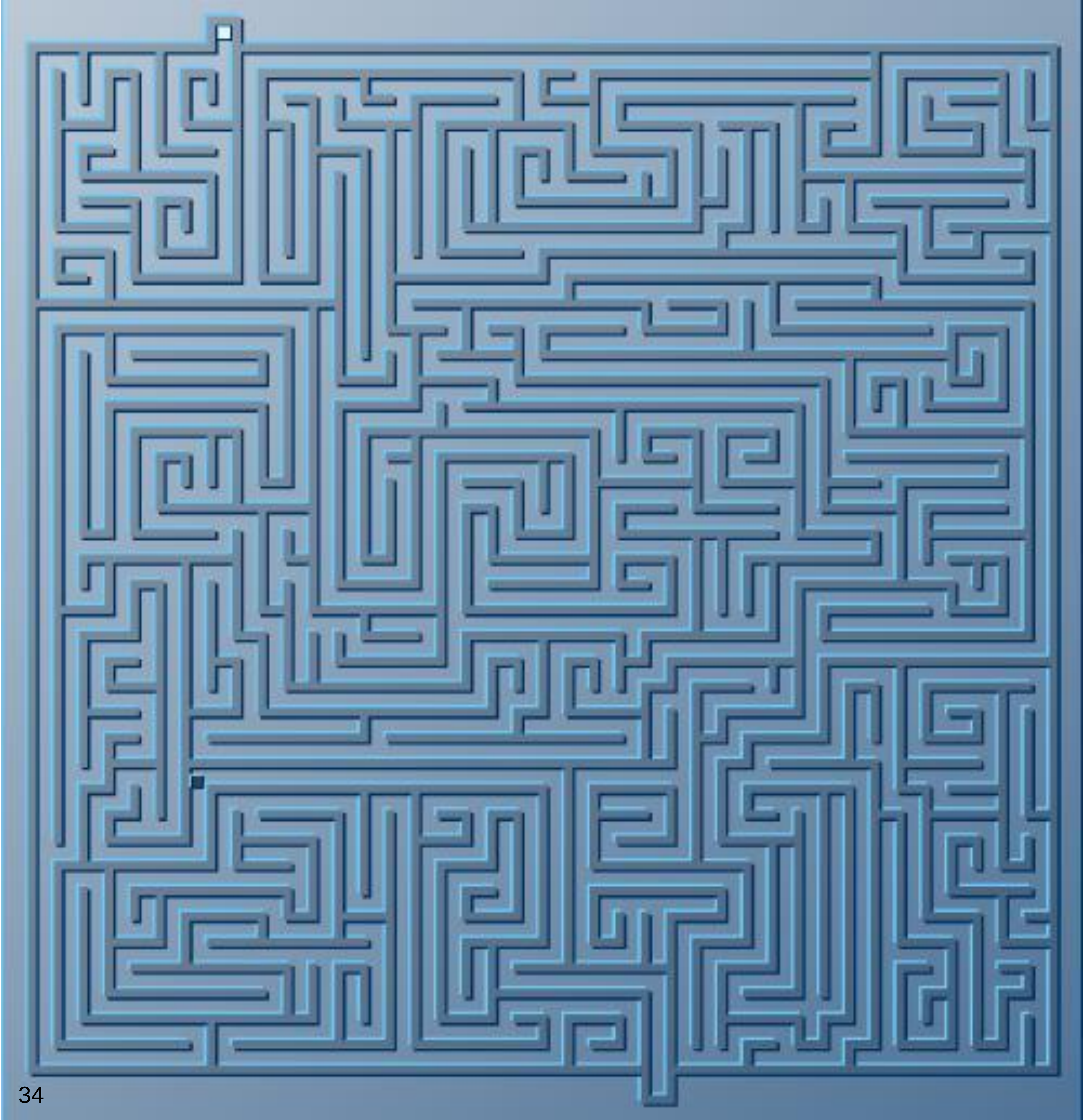
- At each step, we choose a candidate edge to add to the tree
- Which edge **does not matter**
  - we will get a spanning tree in the end
    - possibly a different one for each choice
- Algorithms where we have to make a choice but the actual choice does not matter are called **greedy**

Greedy Algorithms

- Algorithms where we have to make a choice but the actual choice does not matter are called **greedy**
- DFS and BFS also involve making a choice
    - which vertex to examine next
  - but if we don't pick the right one we may not compute the correct answer
  - we need to **remember** the alternative choices
    - in a work list

  DFS and BFS are not greedy
- Greedy algorithms are great
  - no need to remember alternatives
  - but few problems have greedy algorithms that solve them

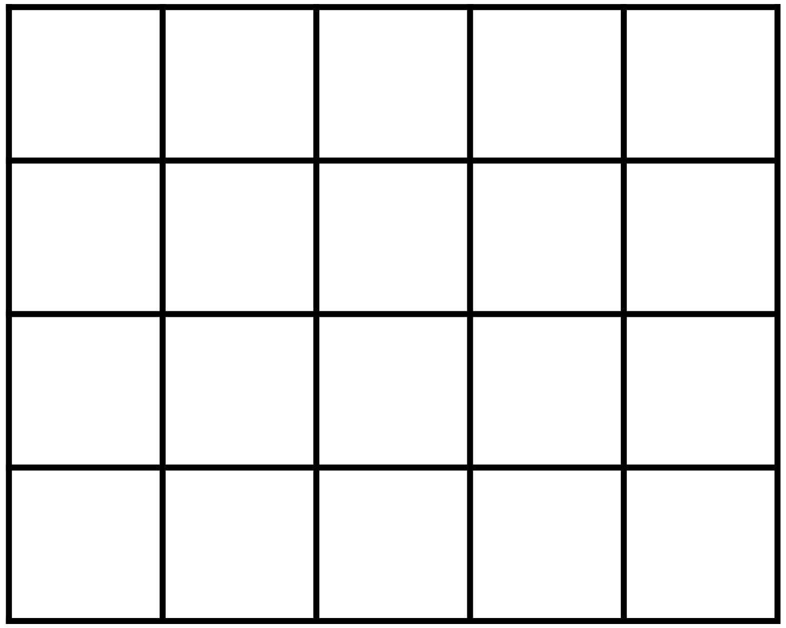
Intermission



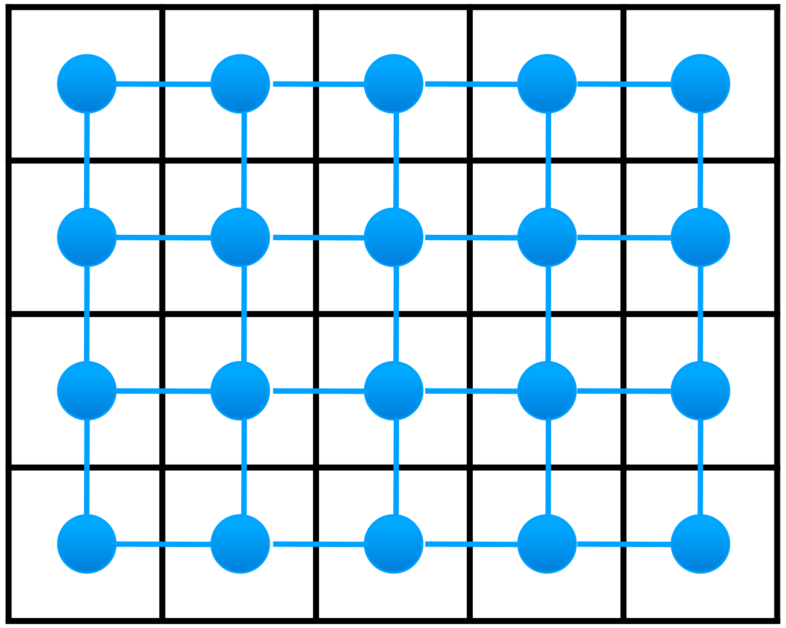
How are maze screen-savers generated?

How to Create a Screensaver Maze?

Start with an n * m grid of cells

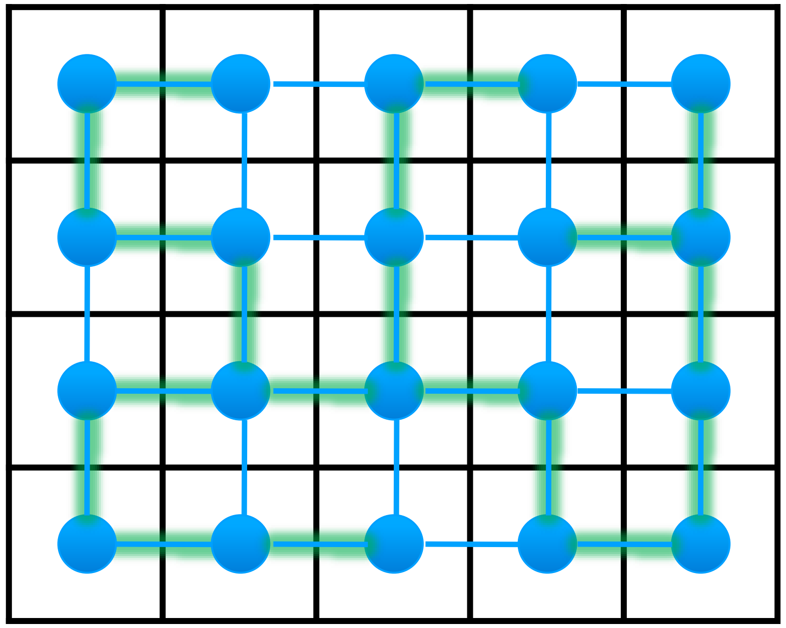


Place a node in every cell and an edge between adjacent cells

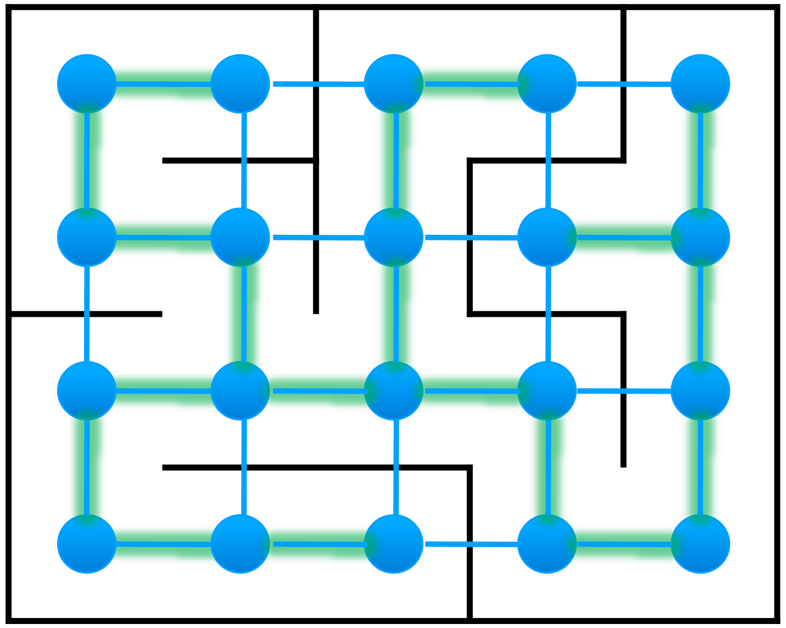


How to Create a Screensaver Maze?

Build a spanning tree for this graph

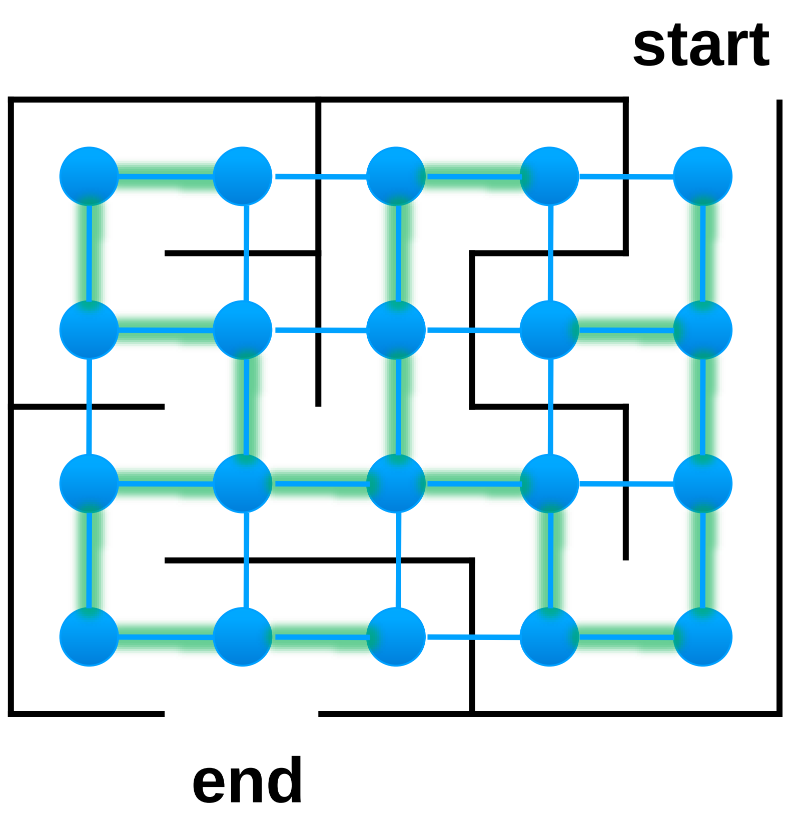


Dissolve the cell walls where its edges cross

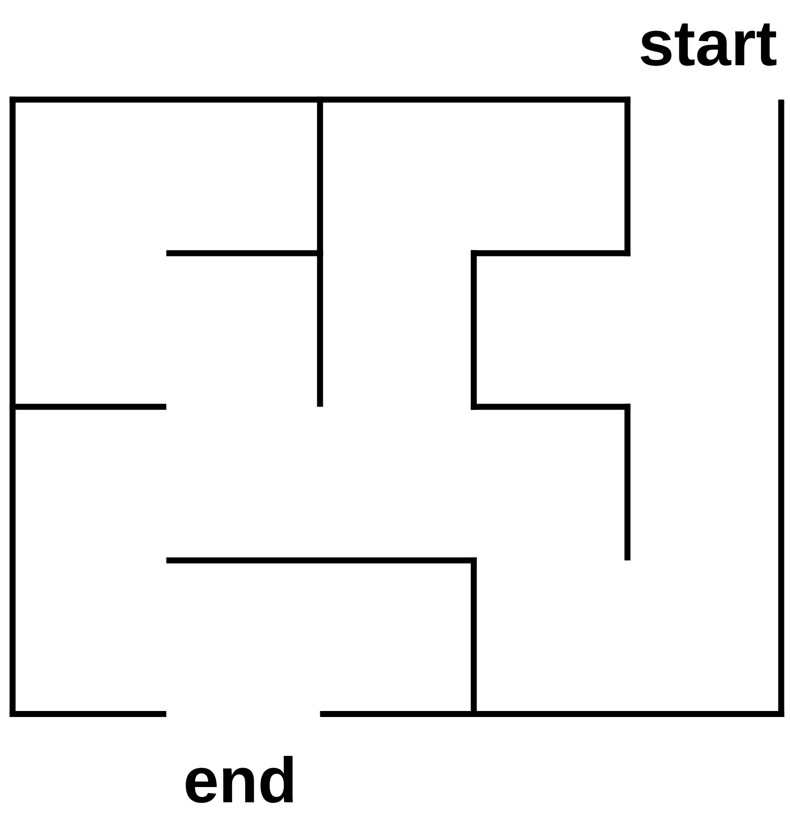


How to Create a Screensaver Maze?

Pick a start and an end along the perimeter

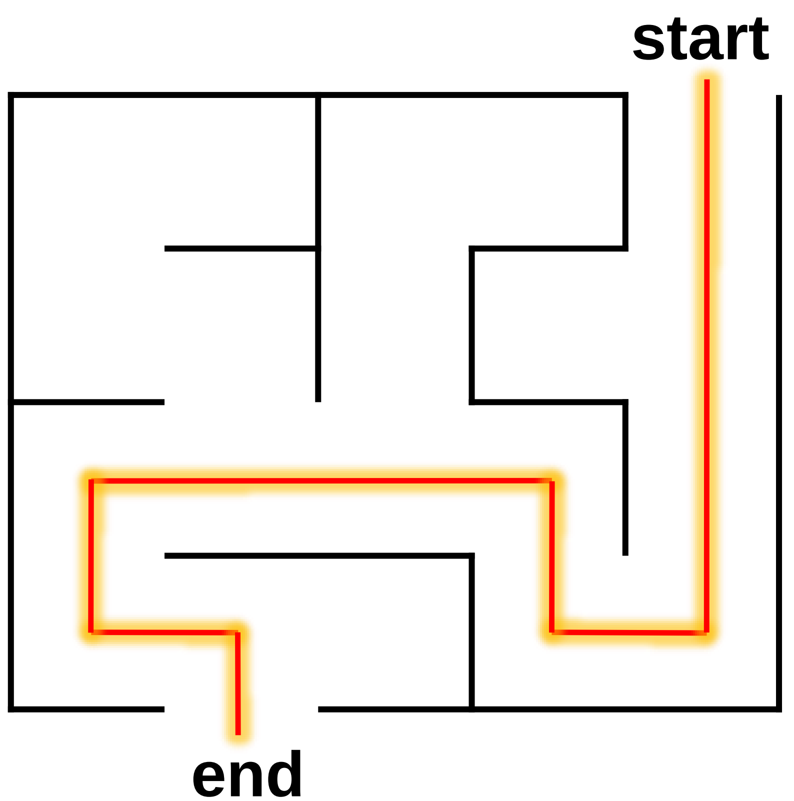


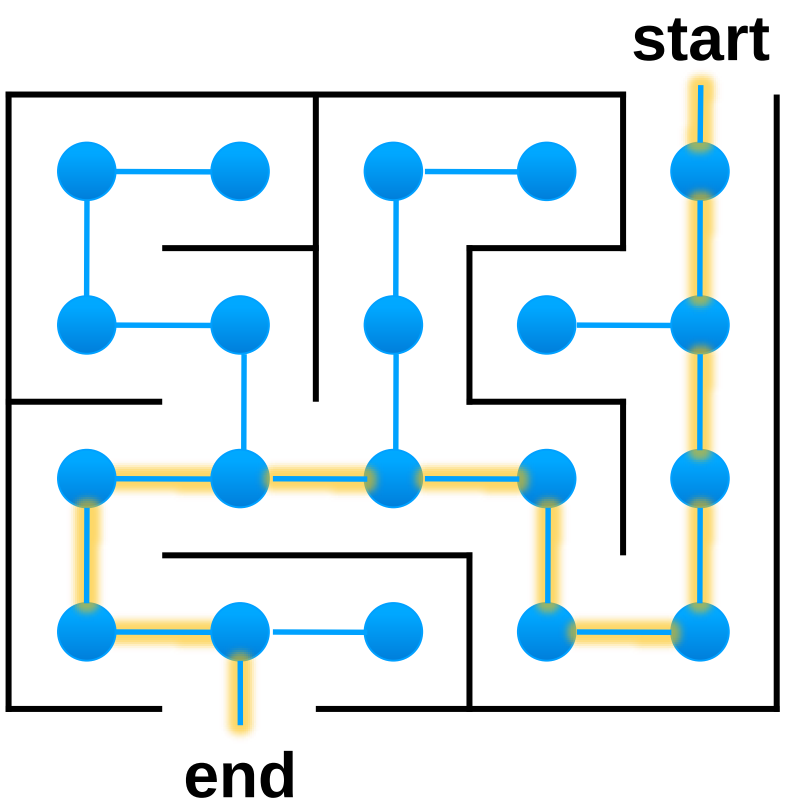
Get rid of the graph



How to *Solve* a Screensaver Maze?

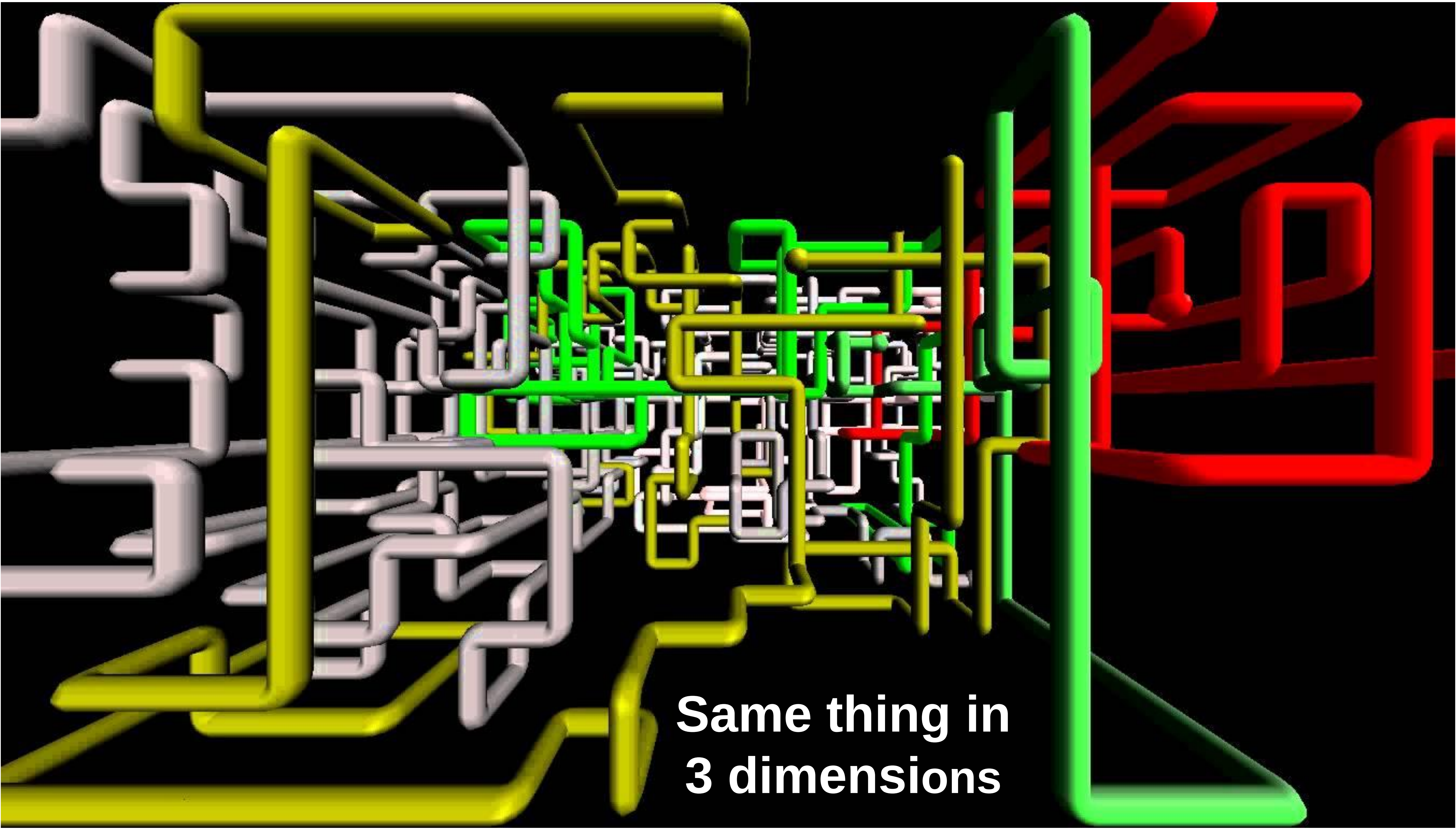
Run DFS on the spanning tree





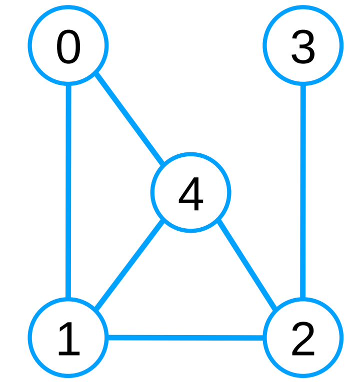
The animation is DFS exploring vertices and backtracking

What about Pipe Screensavers?



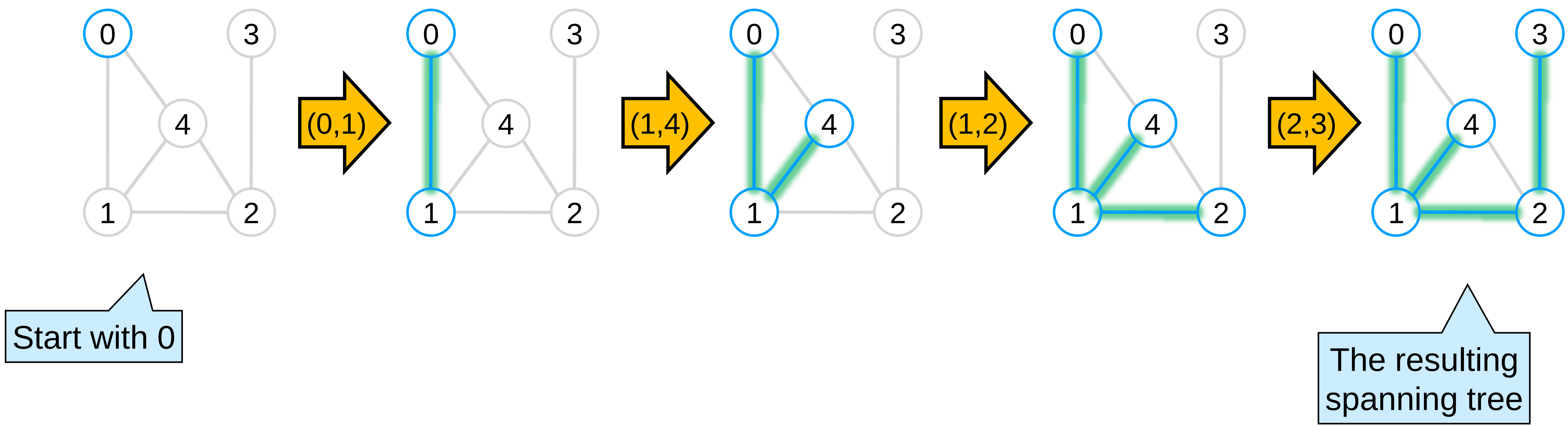
Vertex-centric Algorithm

The Vertex-centric Algorithm



Start with a single vertex in the tree and add edges to vertices not in the tree

- Let’s run it on the example graph



Towards an Algorithm

Start with a single vertex in the tree and add edges to vertices not in the tree

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

1. Pick an arbitrary vertex **start** in G and put it in T
2. Repeat until all vertices are in T
   - find an edge (u,v) in G between a vertex **u in T** and a vertex **v not in T**
   - add (u,v) to T

Assume G has a single connected component

- *How do we find (u,v)?*
  - Mark the vertices we add to T
  - Keep track of their **neighbors**

Towards an *Actual* Algorithm

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

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a work list
2. Repeat until the work list is empty
   - pick an edge **(u,v)** from the work list
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the work list all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges

Consider the neighbors of the vertices added to T

This is our early exit condition

Assume G has a single connected component

Towards an Actual Algorithm

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a work list
2. Repeat until the work list is empty
   - pick an edge **(u,v)** from the work list
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the work list all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges

- This looks just like BSF and DFS
  - depending on the work list
- The edges followed by BFS and DFS form a spanning tree!

Disconnected Graphs

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a work list
2. Repeat until the work list is empty
   - pick an edge **(u,v)** from the work list
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the work list all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges

- If G has more than one connected component, this will find a spanning tree only for **start**'s component
- We need to repeat with a start vertex from each connected component

Disconnected Graphs

Given a graph G, construct a spanning forestT for it

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a work list
2. Repeat until the work list is empty
   - pick an edge **(u,v)** from the work list
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the work list all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges
3. If T has fewer than v-1 edges
   - add an arbitrary unmarked vertex and continue with (1)

Complexity

- The vertex-centric algorithm has the same complexity as DFS and BFS
  - if we use a stack or a queue as the work list

O(v + e)

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a work list
2. Repeat until the work list is empty
   - pick an edge **(u,v)** from the work list
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the work list all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges
3. If T has fewer than v-1 edges
   - add an arbitrary unmarked vertex and continue with (1)

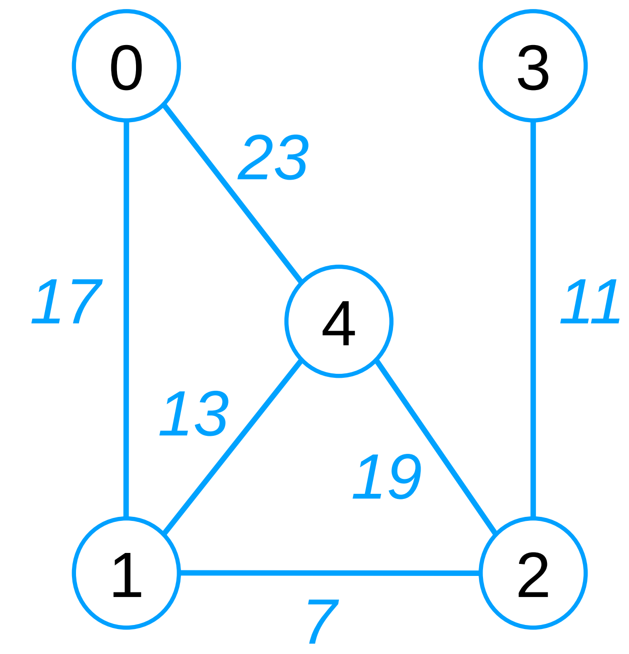
This too is a greedy algorithm

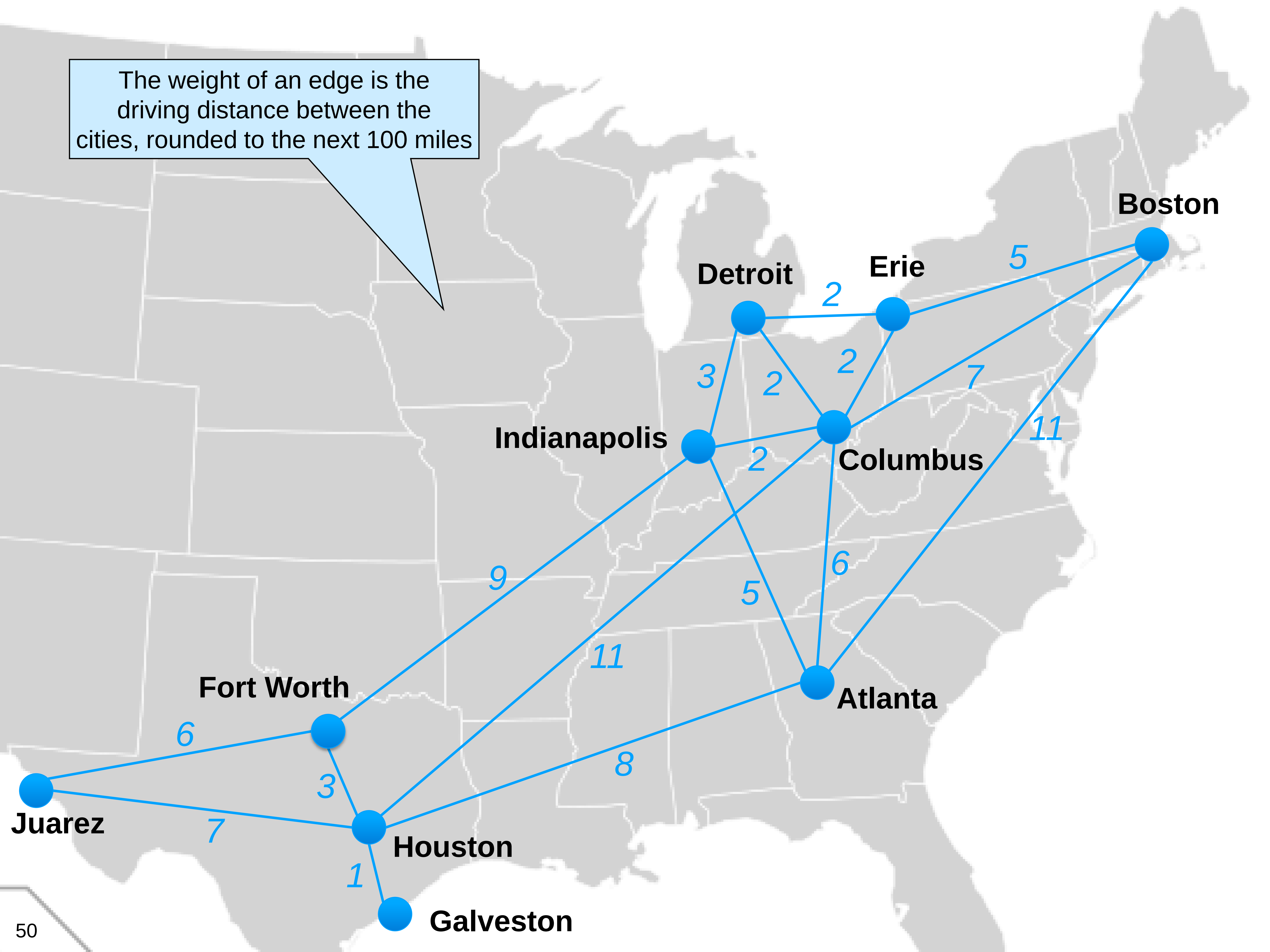
Minimum Spanning Trees

Weighted Graphs

- A graph with measures associated with the edges is a **weighted graph**
  - the measures are called **weights**
  - for us, they will be integers

- The weights represent some kind of cost or value of using that edge
  - time
  - distance
  - power, …




The weight of an edge is the driving distance between the cities, rounded to the next 100 miles
Boston
Erie
Detroit
Indianapolis
Columbus
Atlanta
Fort Worth
Juarez
Houston
Galveston
5
2
2
3
2
7
2
11
9
5
6
11
6
3
8
7
1

Minimum Spanning Tree

- Of all the spanning trees for a weighted graph, one with the least total weight
  - the sum of weights of all its edges

  is called a **minimum spanning tree**

- A graph may have several minimum spanning trees
  - if all the weights are the same, every spanning tree is a minimum spanning tree

They should be called minim**al** spanning trees

Computing MSTs

- The algorithms for computing spanning trees are easily adapted to minimum spanning trees

  - The edge-centric algorithm for MSTs is called **Kruskal's algorithm**

  - The vertex-centric algorithm for MSTs is called **Prim's algorithm**

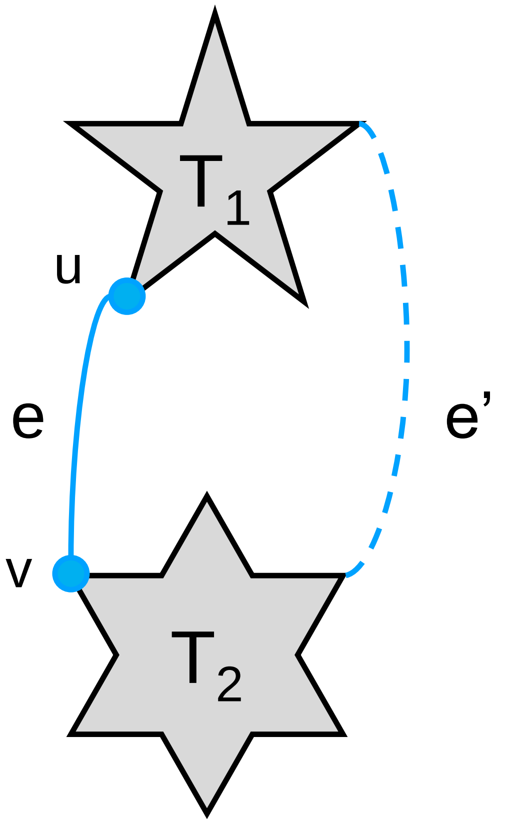
Kruskal’s Algorithm

The Cycle Property

If C is a simple cycle in graph G, and e is an edge of maximal weight in C, then there is some MST of G that does not contain e

Proof

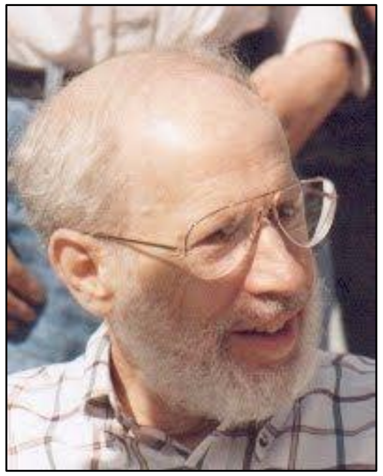
- Assume e is the edge (u,v) and T is a spanning tree
  - either e is not in T, and we are done
  - or e is in T
    - if we remove e, we obtain **two** spanning trees T_1 and T_2
    - because e is part of a cycle in G, there is another edge e' we can add to connect T_1 and T_2
    - let T' be the resulting tree
    - since e had maximal weight, the total weight of T' is ≤ the total weight of T



The Cycle Property

If C is a simple cycle in graph G, and e is an edge of maximal weight in C, then there is some MST of G that does not contain e

- If we construct a spanning tree by adding the edges of lowest weight that won't create a simple cycle **first**, we will obtain a minimum spanning tree
  - This is the basic insight of Kruskal's algorithm



Joseph Kruskal

Kruskal's Algorithm

- Add a preliminary step to the **edge-centric algorithm**:

  sort the edges in increasing weight order

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?*
     - **yes**: discard the edge
     - **no**: add it to T
   - Stop once T has v-1 edges

Complexity of 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

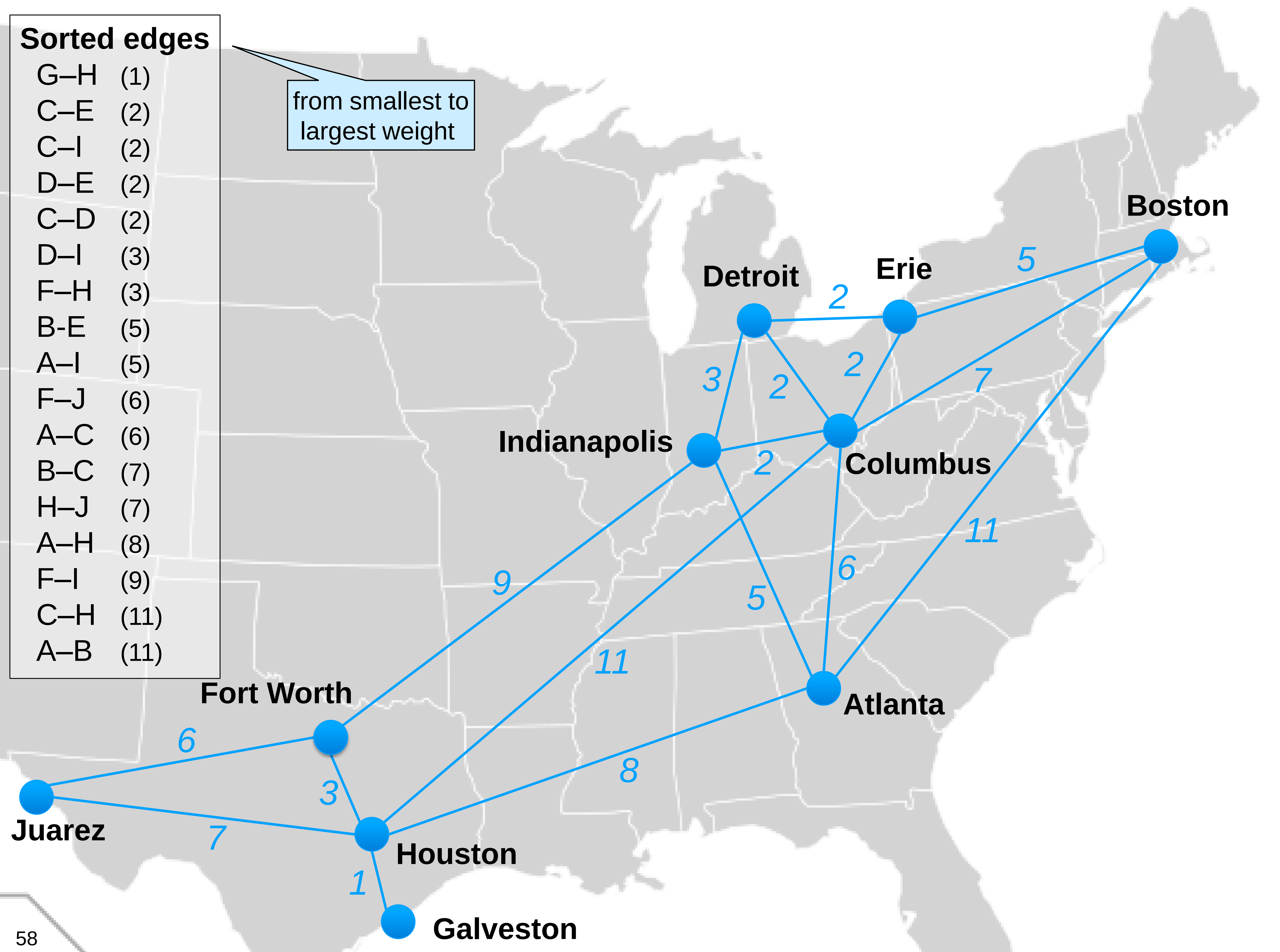
Using mergesort for example

- Kruskal's algorithm has complexity **O(ev)**
  - That's O(e log e + ev) above
  - but log e $\in$ O(v)
    - because e $\in$ O(v^2), so log e $\in$ O(log v)
    - and log v $\in$ O(v)

Sorted edges

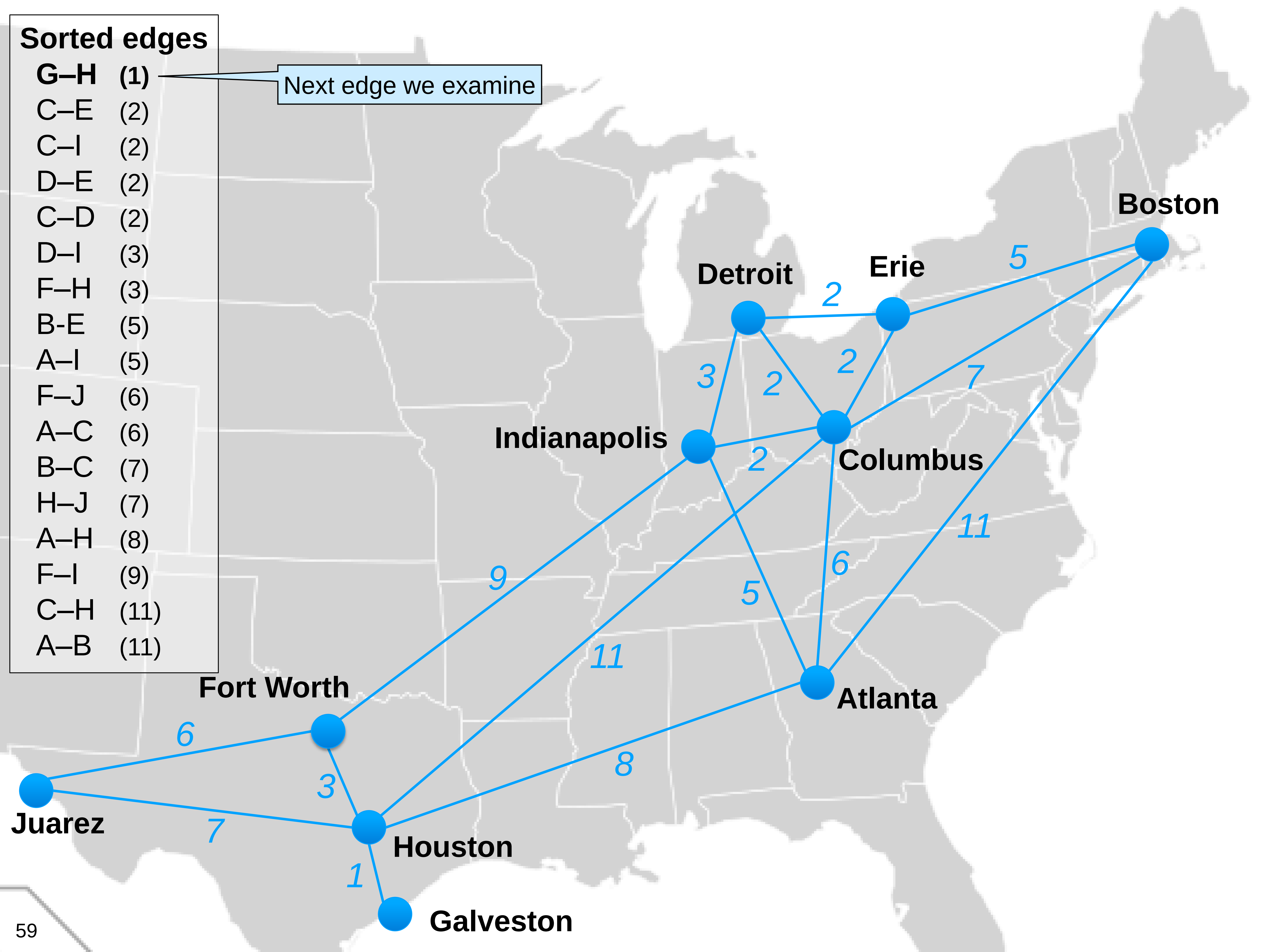
from smallest to largest weight

G–H	(1)
C–E	(2)
C–I	(2)
D–E	(2)
C–D	(2)
D–I	(3)
F–H	(3)
B-E	(5)
A–I	(5)
F–J	(6)
A–C	(6)
B–C	(7)
H–J	(7)
A–H	(8)
F–I	(9)
C–H	(11)
A–B	(11)



Sorted edges

G–H (1)
C–E (2)
C–I (2)
D–E (2)
C–D (2)
D–I (3)
F–H (3)
B-E (5)
A–I (5)
F–J (6)
A–C (6)
B–C (7)
H–J (7)
A–H (8)
F–I (9)
C–H (11)
A–B (11)



Sorted edges

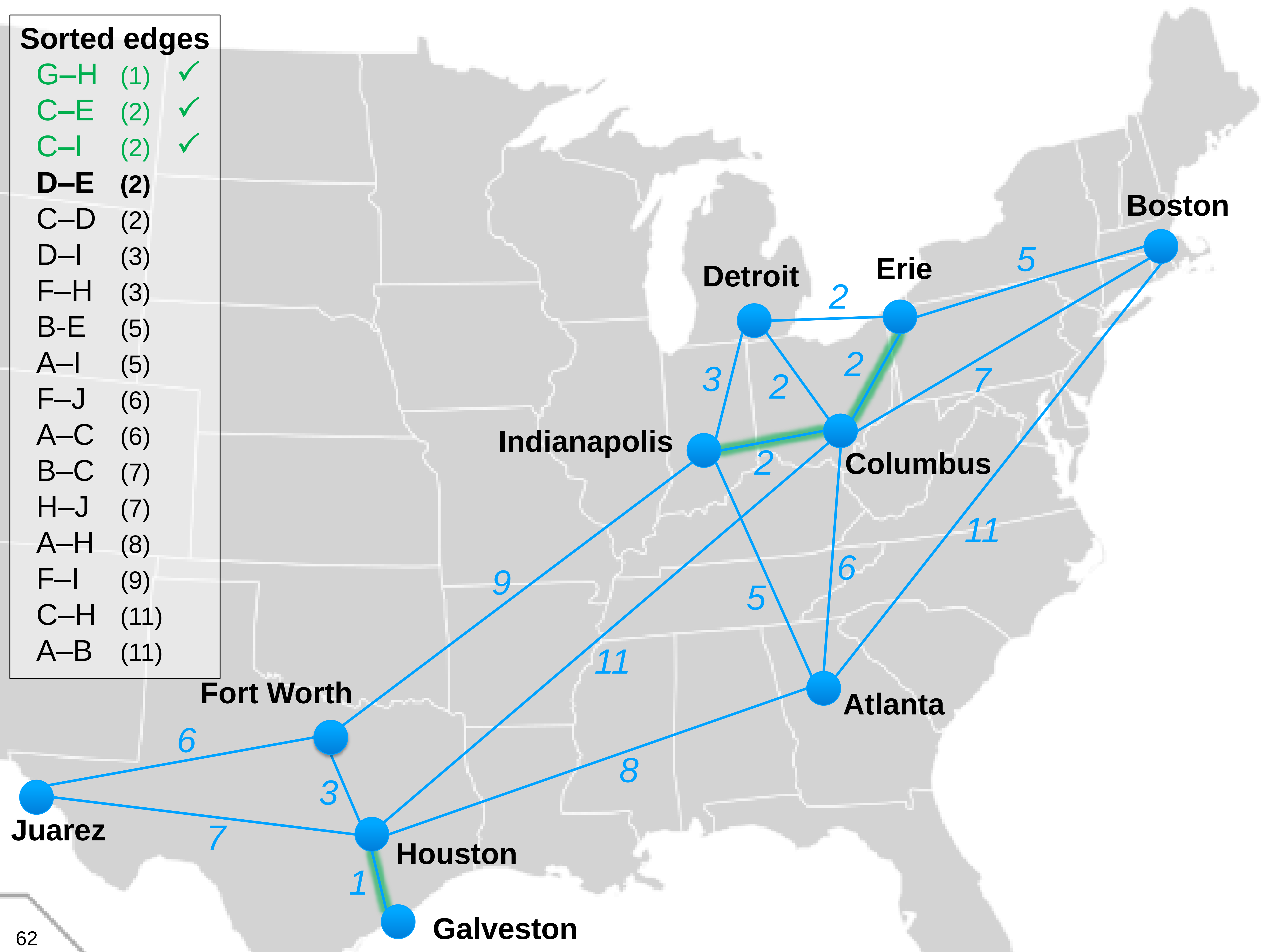
G–H	(1)	✓
C–E	**(2)**	
C–I	(2)	
D–E	(2)	
C–D	(2)	
D–I	(3)	
F–H	(3)	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	

Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	**(2)**	
D–E	(2)	
C–D	(2)	
D–I	(3)	
F–H	(3)	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	

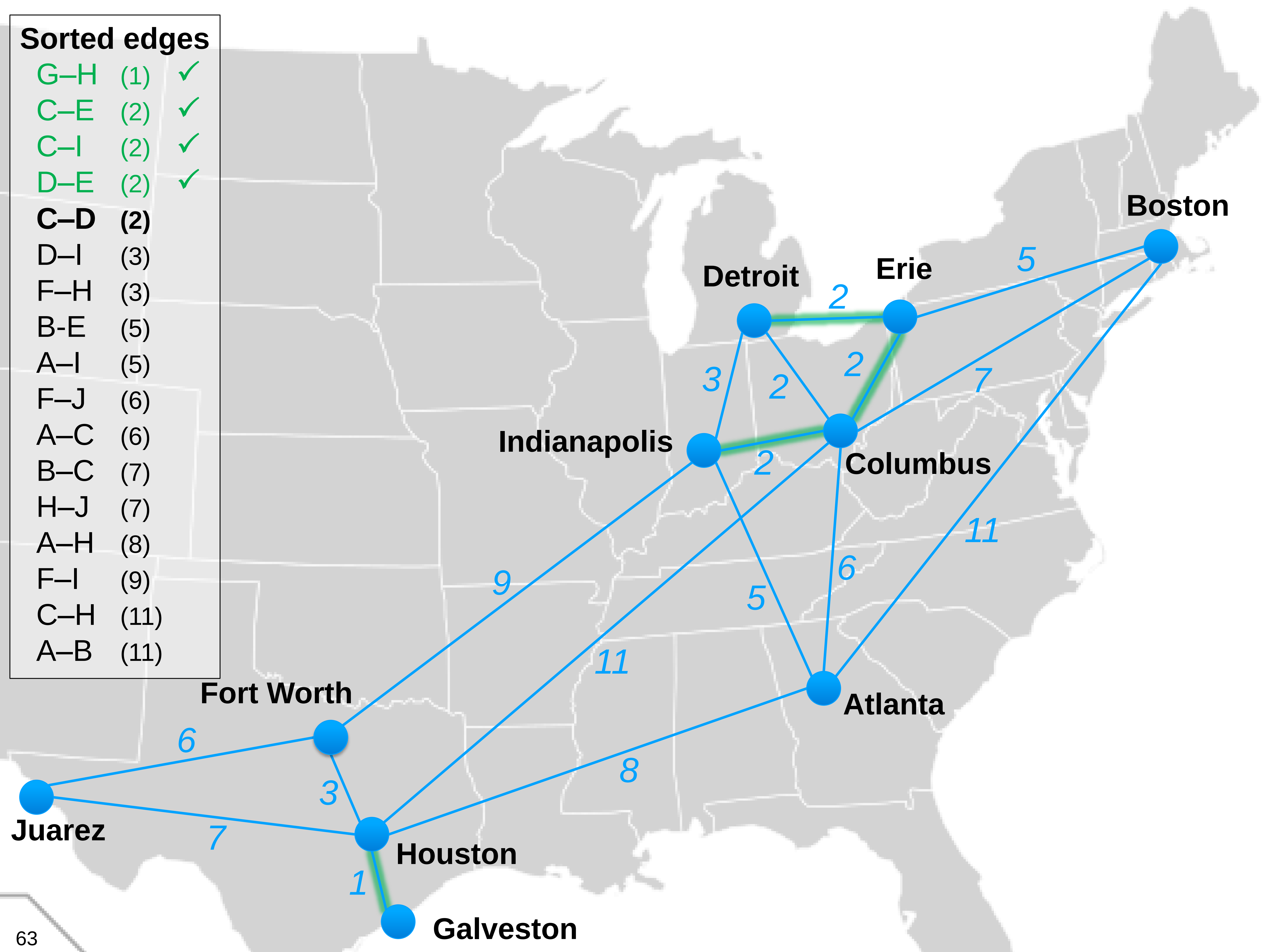
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	**(2)**	
C–D	(2)	
D–I	(3)	
F–H	(3)	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



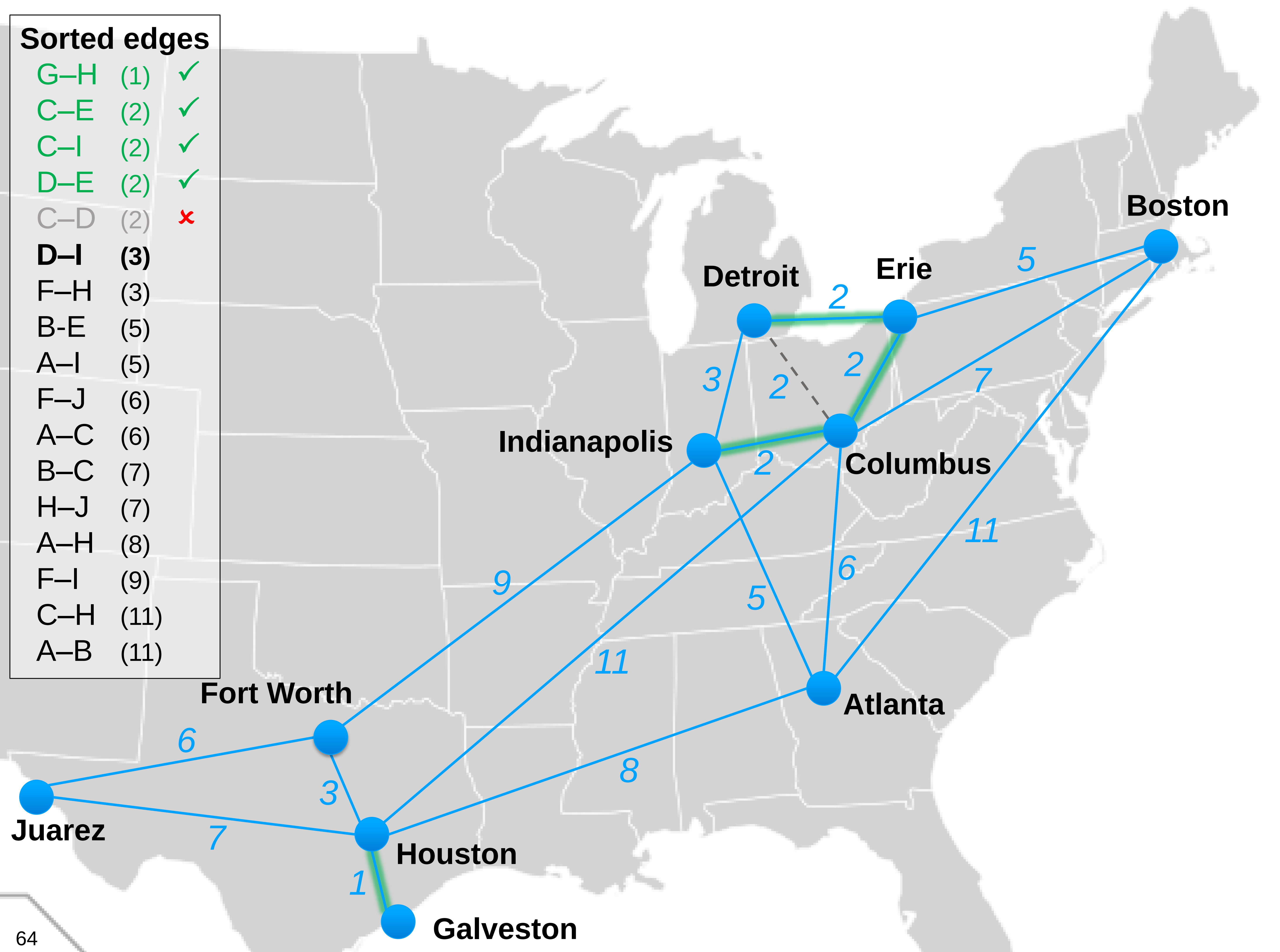
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	**(2)**	
D–I	(3)	
F–H	(3)	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



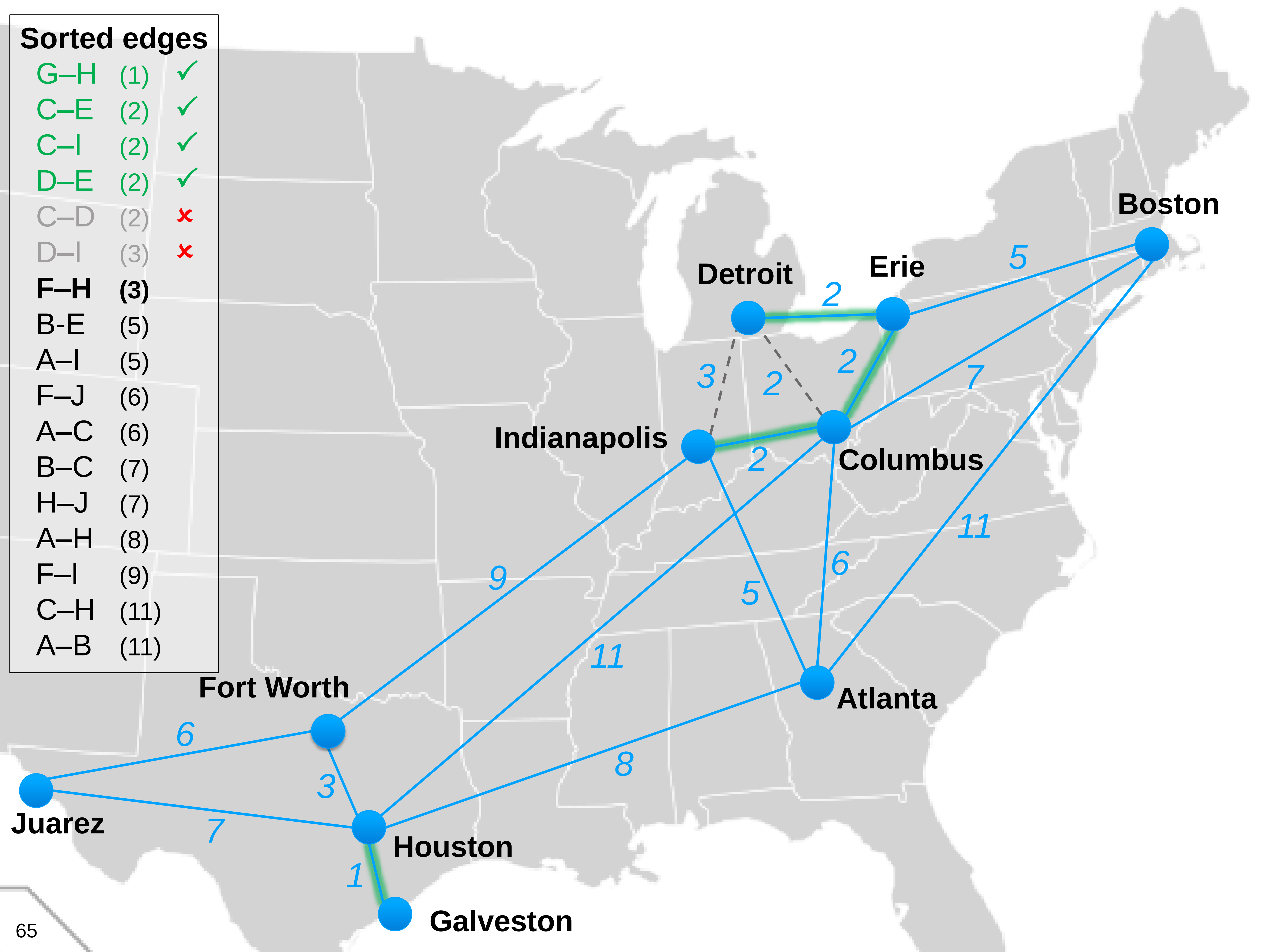
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	**(3)**	
F–H	(3)	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



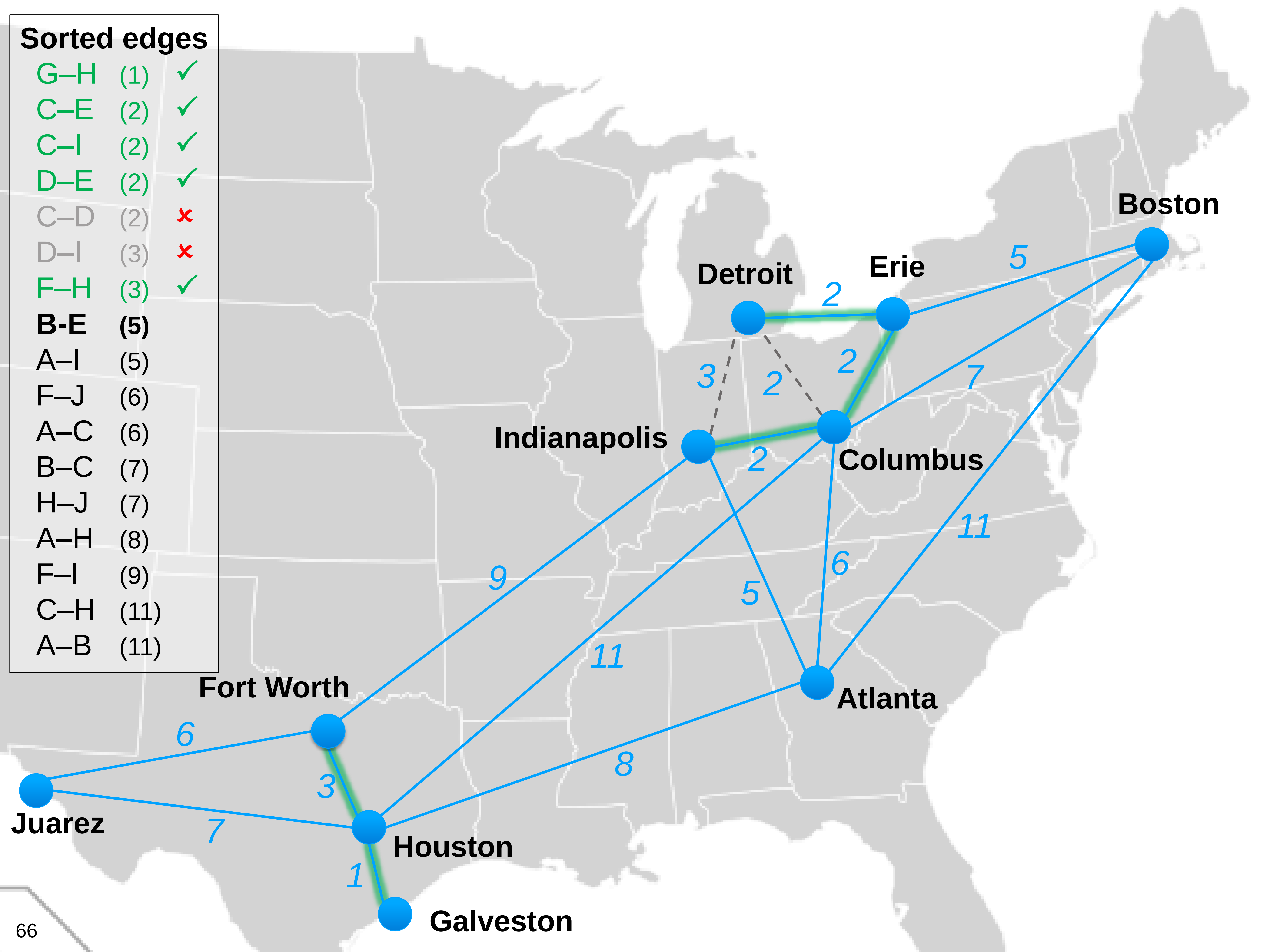
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	**(3)**	
B-E	(5)	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



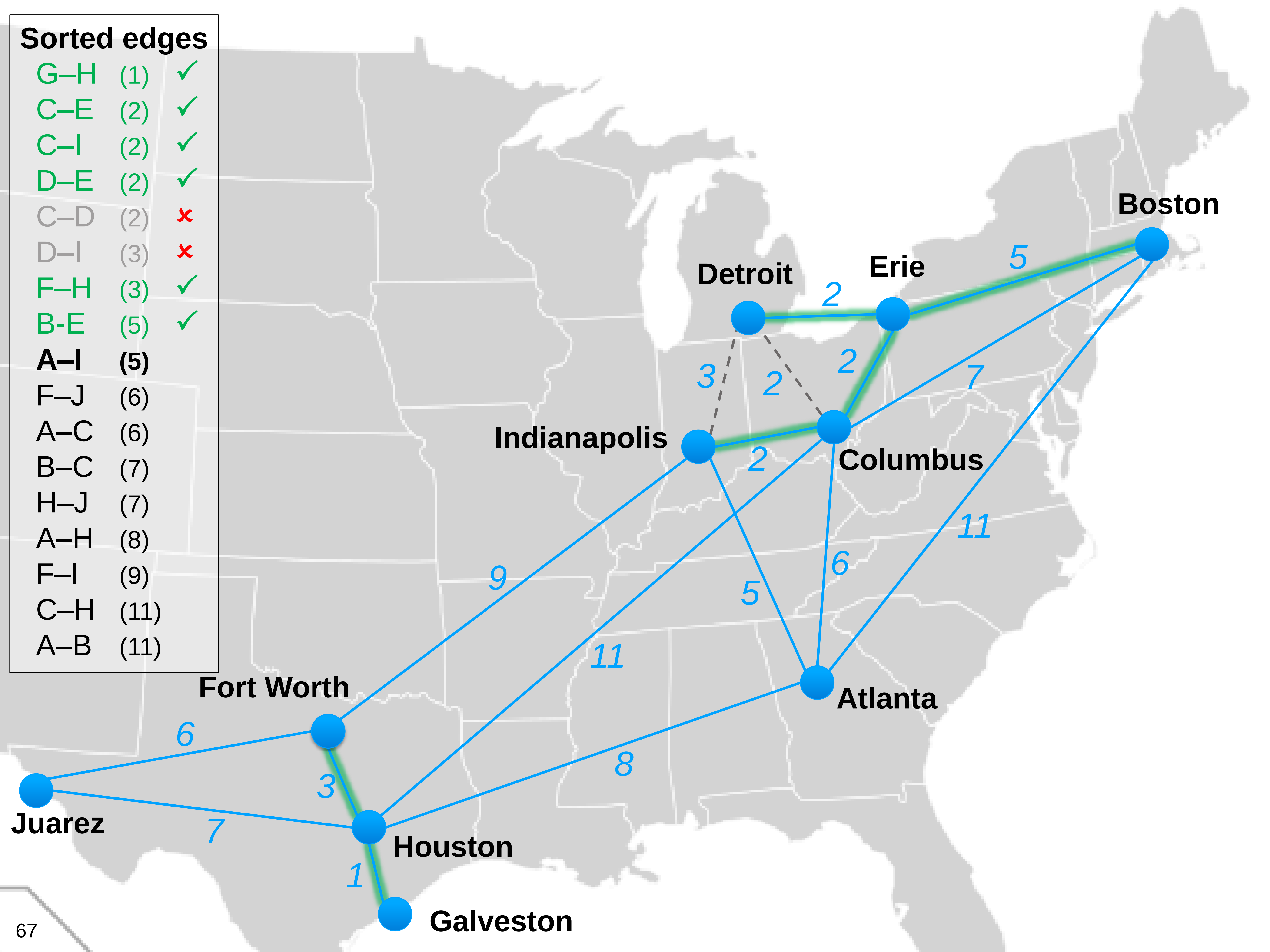
Sorted edges

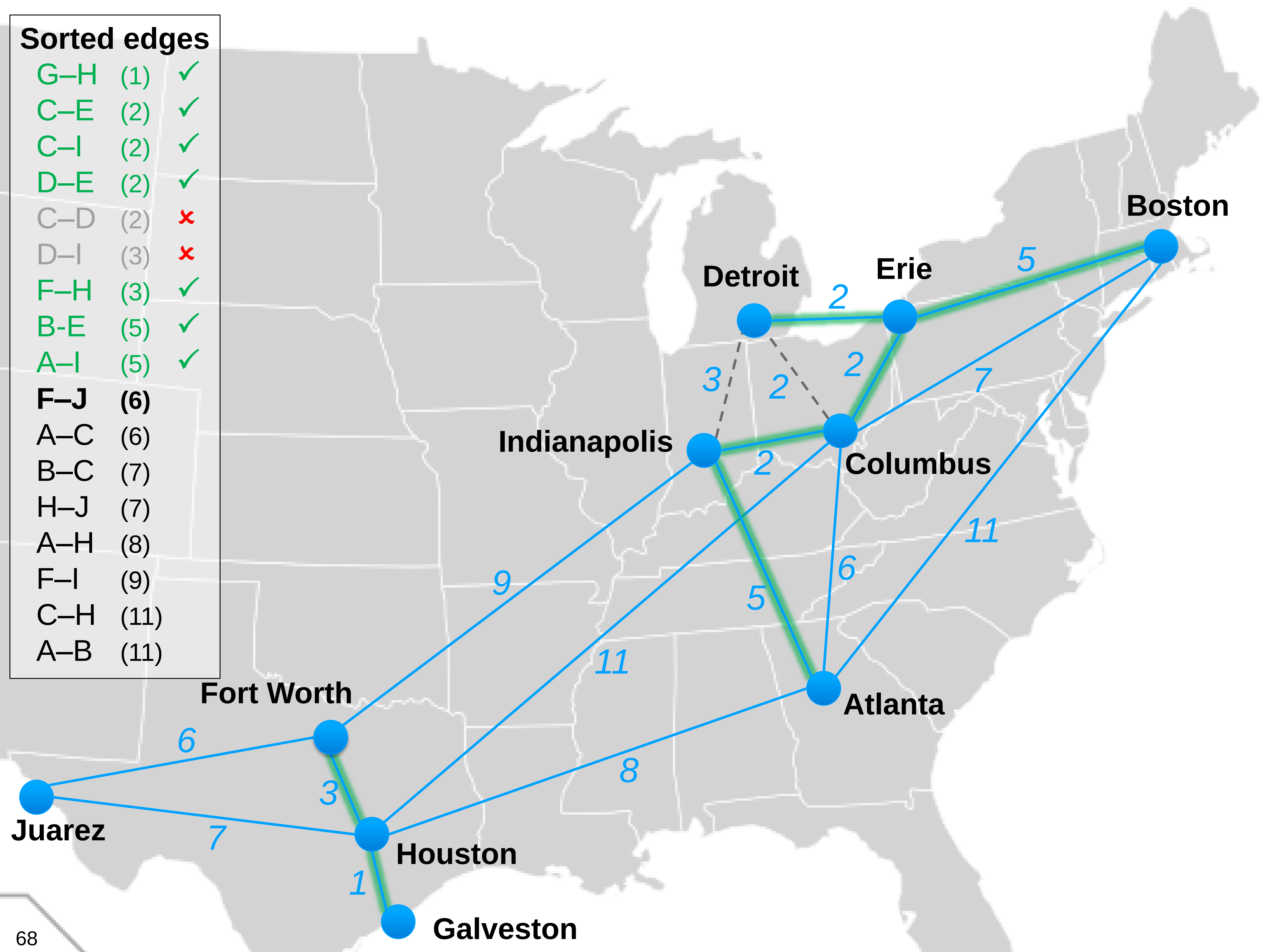
G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	(3)	✓
B-E	**(5)**	
A–I	(5)	
F–J	(6)	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



Sorted edges

- G–H (1) ✓
- C–E (2) ✓
- C–I (2) ✓
- D–E (2) ✓
- C–D (2) ✗
- D–I (3) ✗
- F–H (3) ✓
- B-E (5) ✓
- **A–I (5)**
- F–J (6)
- A–C (6)
- B–C (7)
- H–J (7)
- A–H (8)
- F–I (9)
- C–H (11)
- A–B (11)



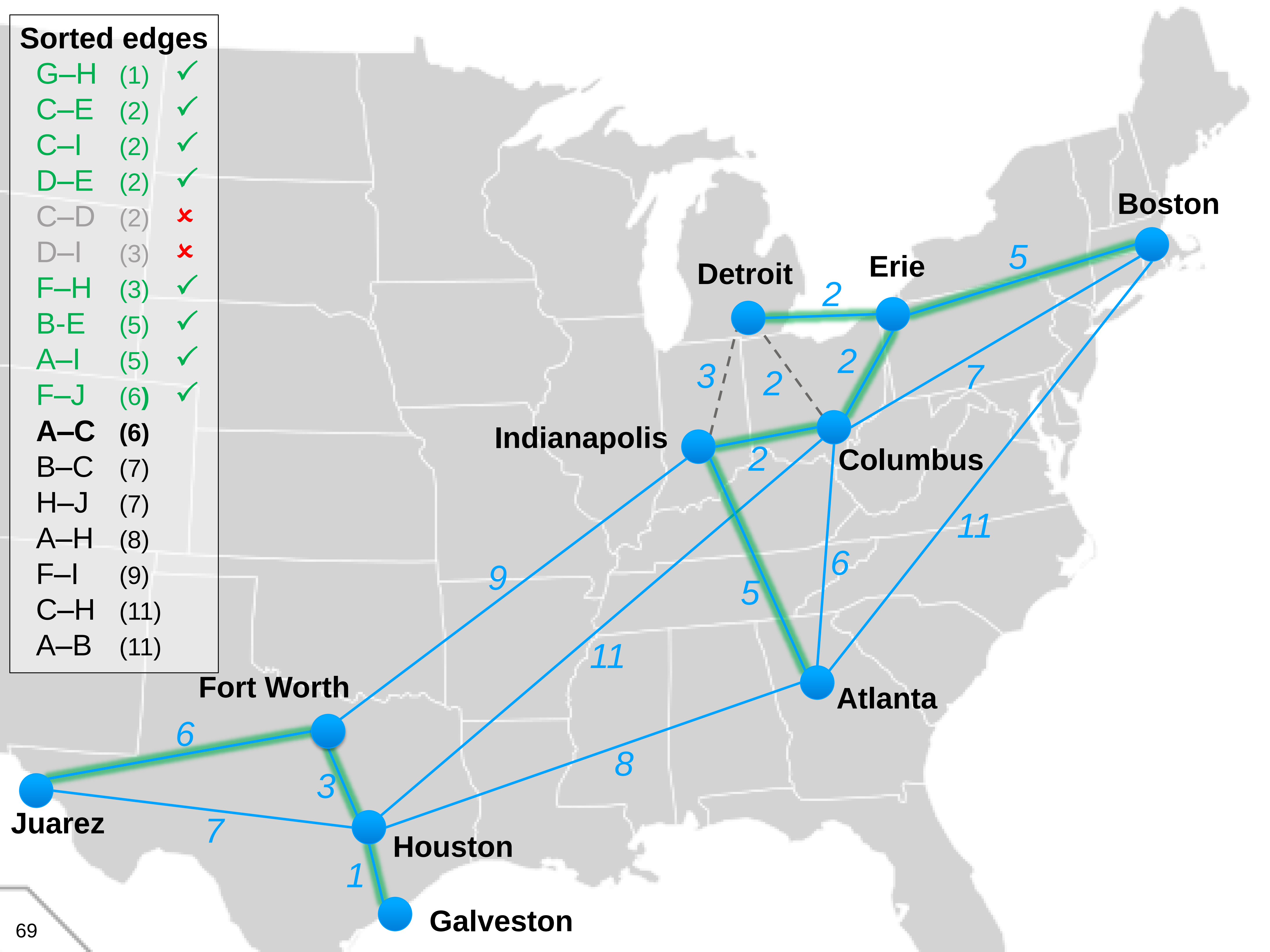


Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	**(6)**	
A–C	(6)	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	

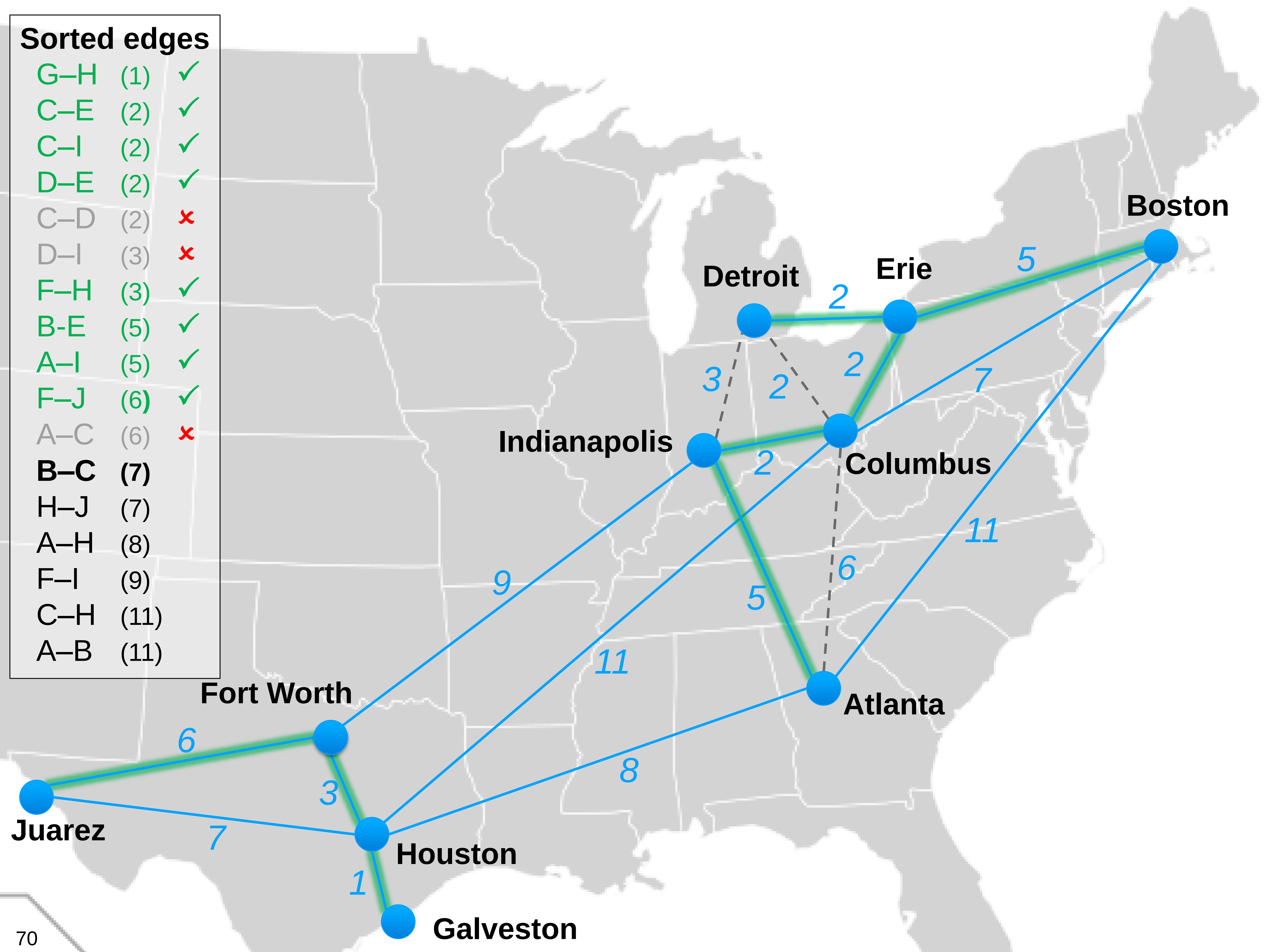
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	(6)	✓
A–C	**(6)**	
B–C	(7)	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



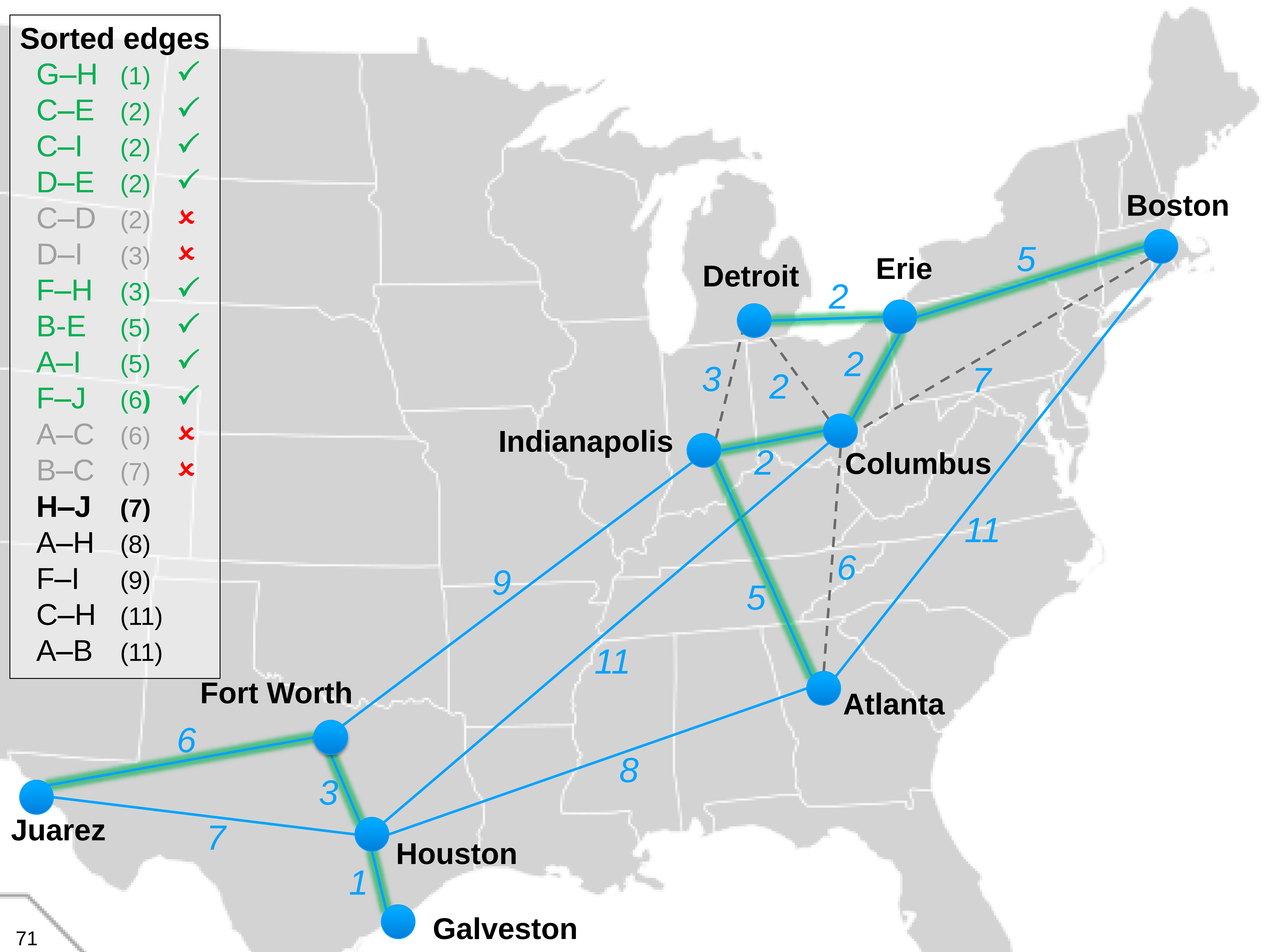
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	×
D–I	(3)	×
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	(6)	✓
A–C	(6)	×
B–C	**(7)**	
H–J	(7)	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



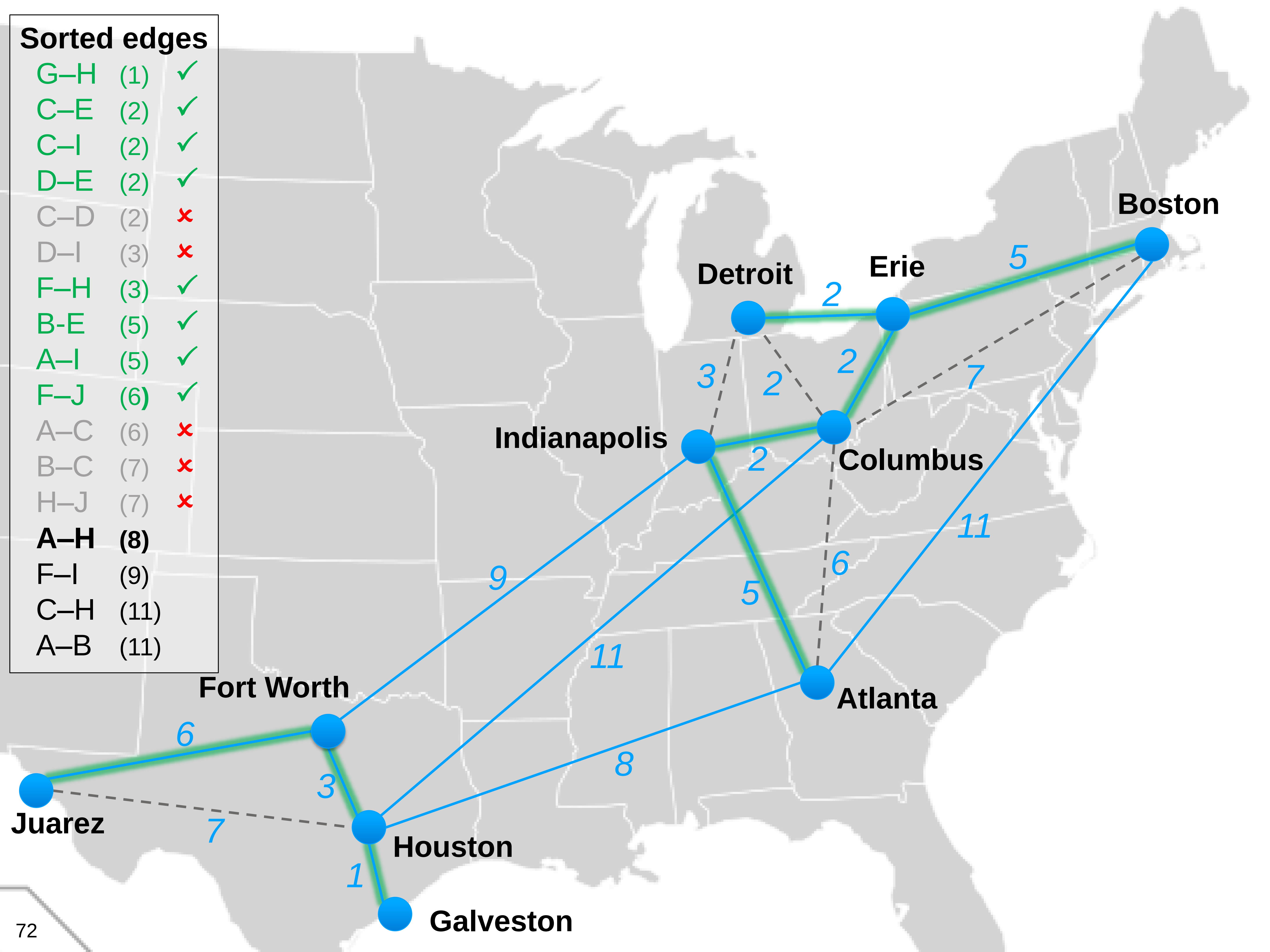
Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	×
D–I	(3)	×
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	(6)	✓
A–C	(6)	×
B–C	(7)	×
H–J	**(7)**	
A–H	(8)	
F–I	(9)	
C–H	(11)	
A–B	(11)	



Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	(6)	✓
A–C	(6)	✗
B–C	(7)	✗
H–J	(7)	✗
A–H	**(8)**	
F–I	(9)	
C–H	(11)	
A–B	(11)	

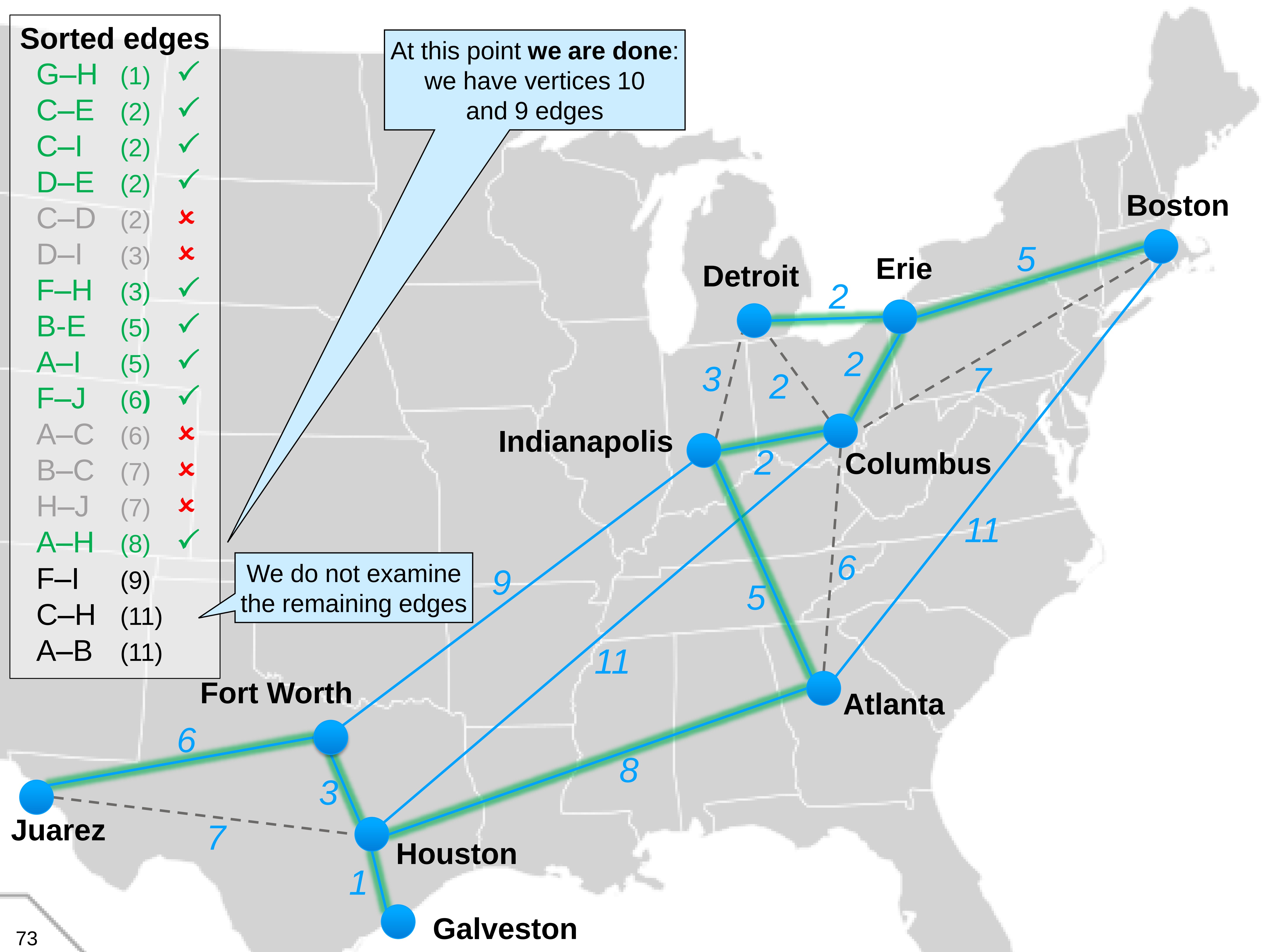


Sorted edges

G–H	(1)	✓
C–E	(2)	✓
C–I	(2)	✓
D–E	(2)	✓
C–D	(2)	✗
D–I	(3)	✗
F–H	(3)	✓
B-E	(5)	✓
A–I	(5)	✓
F–J	(6)	✓
A–C	(6)	✗
B–C	(7)	✗
H–J	(7)	✗
A–H	(8)	✓
F–I	(9)	
C–H	(11)	
A–B	(11)	

At this point **we are done**: we have vertices 10 and 9 edges

We do not examine the remaining edges



Prim's Algorithm

Prim’s Algorithm

Robert Prim

- In the **vertex-centric algorithm**, use a priority queue with lower-weight edges having higher priority

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

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a priority queue
2. Repeat until the priority queue is empty
   - pick an edge **(u,v)** from the priority queue
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the priority queue all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges
3. If T has fewer than v-1 edges
   - add an arbitrary unmarked vertex and continue with (1)


Priority Queue
Lower-weight edges having higher priority
Boston
Erie
Detroit
Indianapolis
Columbus
Atlanta
Start vertex
Fort Worth
Juarez
Houston
Galveston
5
2
2
3
2
7
2
11
6
5
9
11
8
6
3
7
1

Priority Queue

A–I	(5)
A–C	(6)
A–H	(8)
A–B	(11)

Lower-weight edges having higher priority

Adding the edges going out of A


Priority Queue
C–I (2)
D–I (3)
A–C (6)
A–H (8)
I–F (9)
A–B (11)
Lower-weight edges having higher priority
Boston
Detroit
Erie
Indianapolis
Columbus
Atlanta
Fort Worth
Juarez
Houston
Galveston
5
2
3
2
2
7
2
11
6
9
5
11
8
6
3
7
1
Adding the edges going out of I
We skip the edges to marked vertices

Priority Queue

C–D	(2)
C–E	**(2)**
D–I	(3)
A–C	(6)
B–C	**(7)**
A–H	(8)
I–F	(9)
A–B	(11)
C–H	**(11)**

Lower-weight edges having higher priority

Adding the edges going out of C

Priority Queue

C–E	(2)
D–E	**(2)**
D–I	(3)
A–C	(6)
B–C	(7)
A–H	(8)
I–F	(9)
A–B	(11)
C–H	(11)

Lower-weight edges having higher priority

Adding the edges going out of D

Priority Queue

D–E	(2)
D–I	(3)
B–E	**(5)**
A–C	(6)
B–C	(7)
A–H	(8)
I–F	(9)
A–B	(11)
C–H	(11)

Lower-weight edges having higher priority

The next 2 edges have marked endpoints; we skip them

Adding the edges going out of E


Priority Queue
B–E (5)
A–C (6)
B–C (7)
A–H (8)
I–F (9)
A–B (11)
C–H (11)
Lower-weight edges having higher priority
Boston
Detroit
Erie
Indianapolis
Columbus
Atlanta
Fort Worth
Juarez
Houston
Galveston
5
2
3
2
2
7
2
11
6
9
5
11
6
8
3
7
1

Priority Queue

A–C	(6)
B–C	(7)
A–H	(8)
I–F	(9)
A–B	(11)
C–H	(11)

Lower-weight edges having higher priority

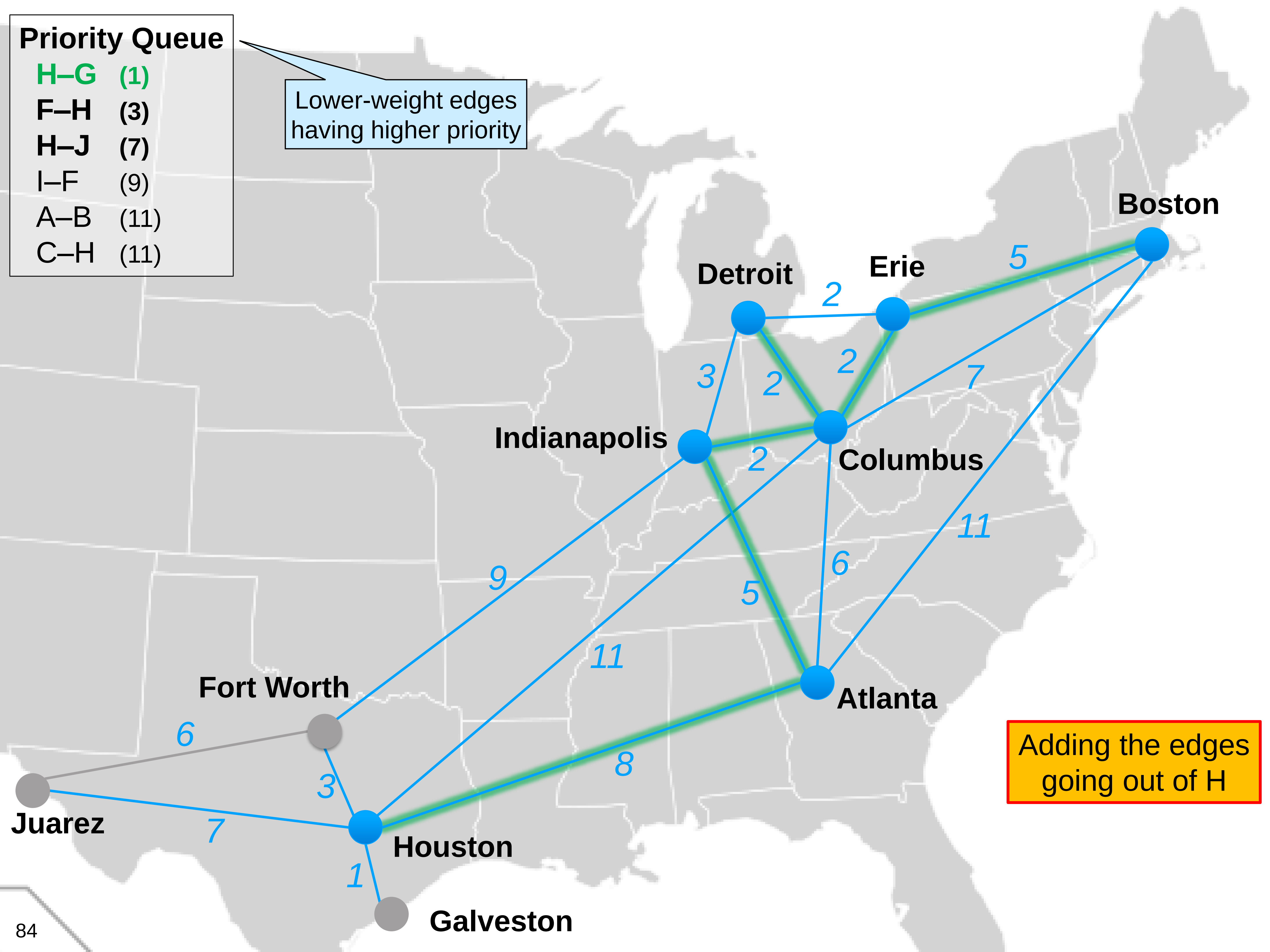
The next 2 edges have marked endpoints; we skip them

Adding the edges going out of B

Priority Queue

- **H–G (1)**
- **F–H (3)**
- **H–J (7)**
- I–F (9)
- A–B (11)
- C–H (11)

Lower-weight edges having higher priority

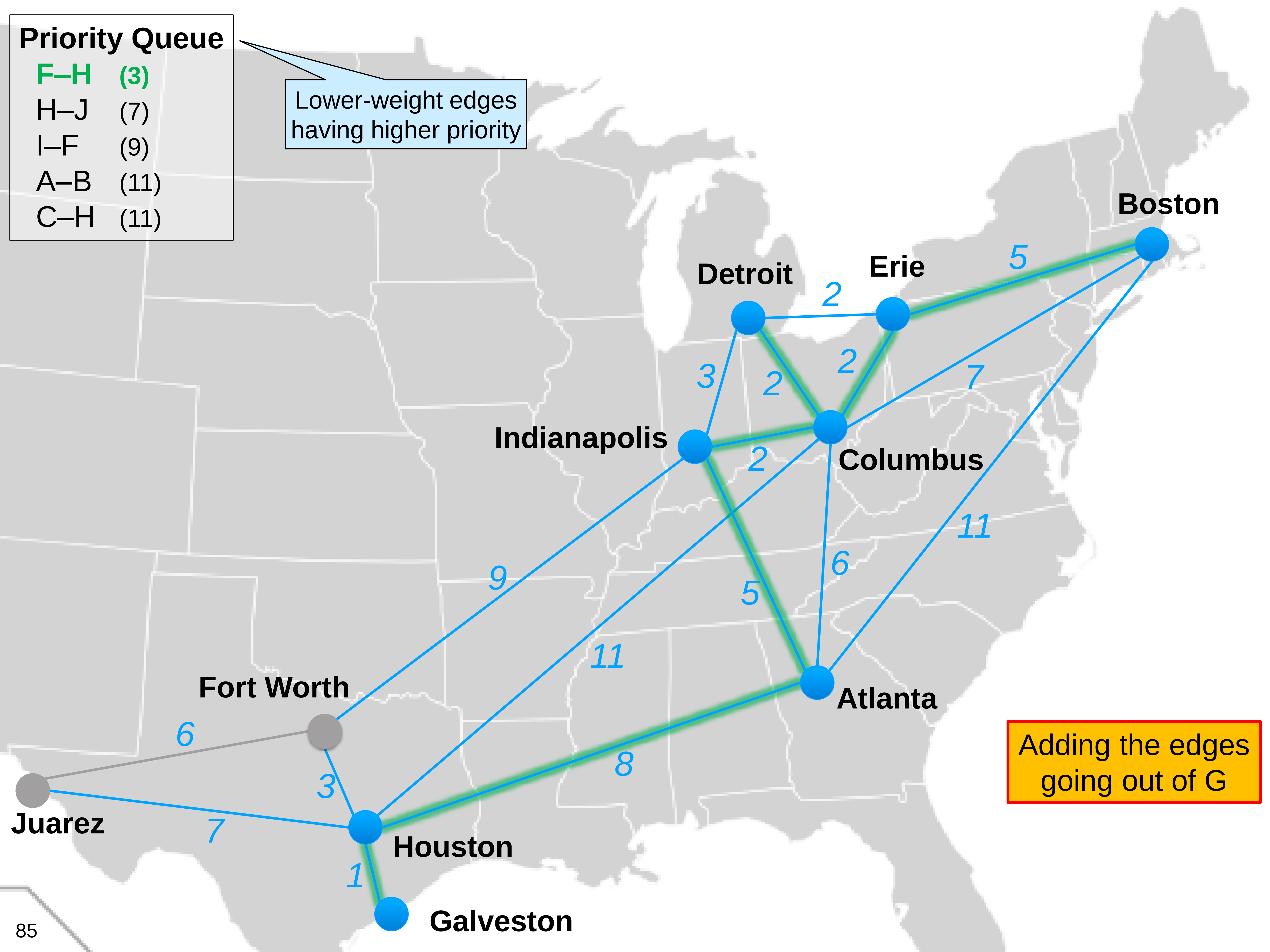


Adding the edges going out of H

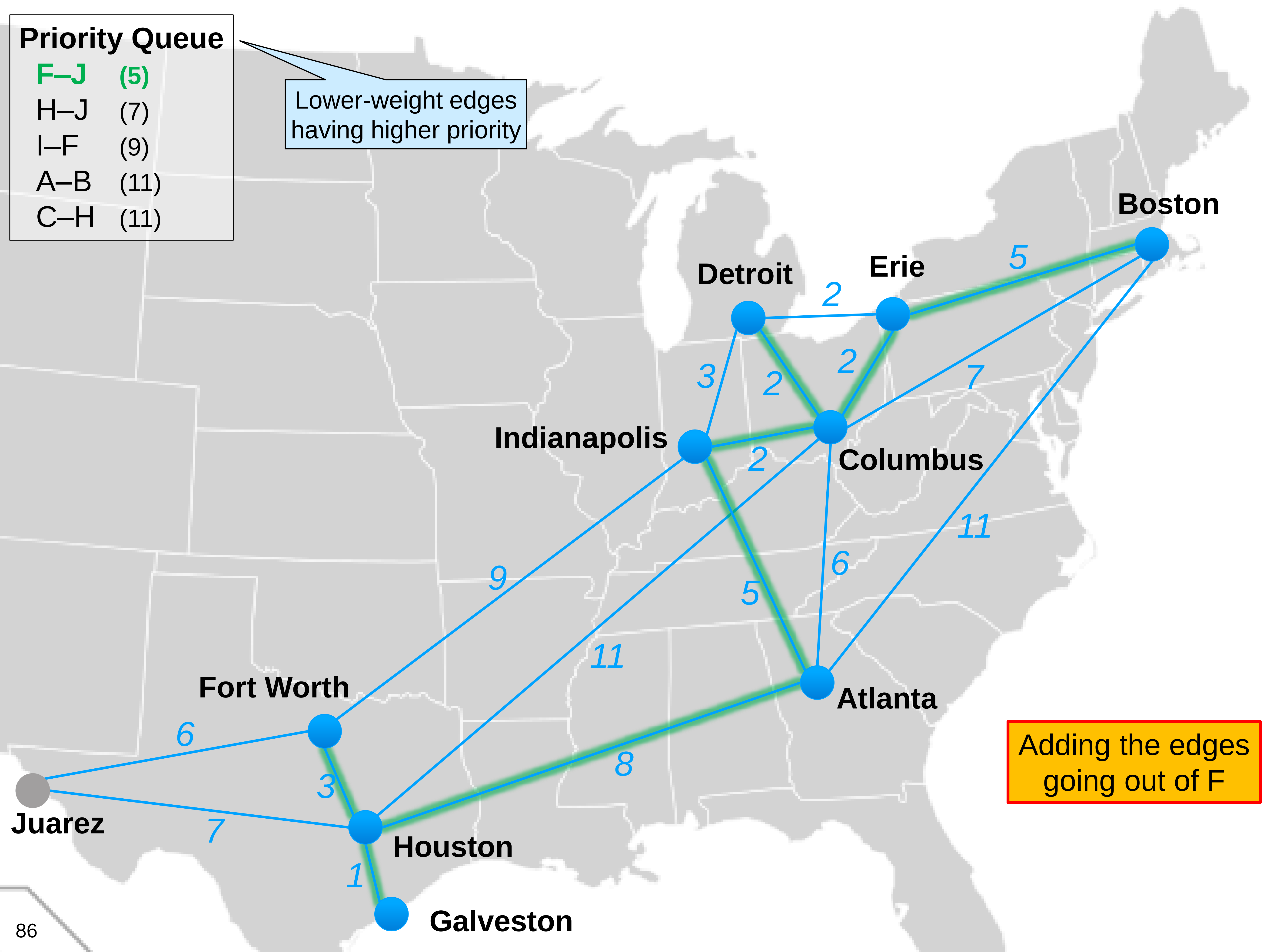
Priority Queue

F–H	**(3)**
H–J	(7)
I–F	(9)
A–B	(11)
C–H	(11)

Lower-weight edges having higher priority



Adding the edges going out of G


Priority Queue
F–J (5)
H–J (7)
I–F (9)
A–B (11)
C–H (11)
Lower-weight edges having higher priority
Boston
Erie
Detroit
Indianapolis
Columbus
Atlanta
Fort Worth
Juarez
Houston
Galveston
5
2
2
2
3
2
7
11
6
5
9
11
8
6
3
7
1
Adding the edges going out of F

Priority Queue

H–J	(7)
I–F	(9)
A–B	(11)
C–H	(11)

Lower-weight edges having higher priority

At this point **we are done**: there are 10 vertices and 9 edges

Complexity

1. Pick an arbitrary vertex **start** in G and put it in T
   - mark **start**
   - add all edges **(start,w)** in G to a priority queue
2. Repeat until the priority queue is empty
   - pick an edge **(u,v)** from the priority queue
     - if **v** is marked, discard it
     - add **(u,v)** to T
     - mark **v**
     - add to the priority queue all edges **(v,w)** in G such that **w** is unmarked
   - stop once T has v-1 edges
3. If T has fewer than v-1 edges
   - add an arbitrary unmarked vertex and continue with (1)

- At most, Prim's algorithm puts every edge of G in the priority queue
  - once from each endpoint

  that's 2e steps
- At each step, the most expensive operation is adding/removing edges to/from the priority queue
  - O(log e)
- Eventually adds all v vertices
  - O(v)

- The complexity of Prim's algorithm is **O(v + e log e)**

Summary

- 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

But can we **improve** Kruskal's algorithm?