Lecture 14: Greedy Algorithms and the Minimum Spanning Tree

These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- **Lab 13** – Pre-lab due tonight, code and post-lab due Friday

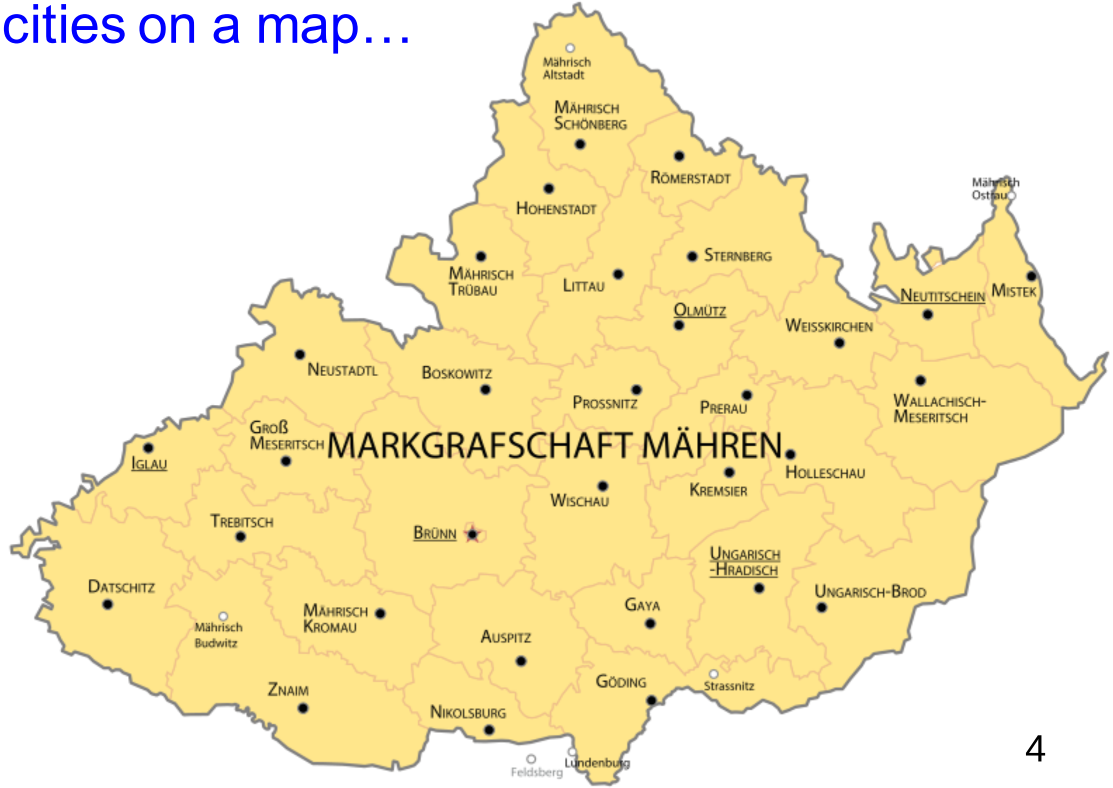
- **Exam 3** – May 1st, 10 am – 12 pm
  - Similar procedure to previous exams; stay tuned to Piazza
  - Exam review **Sun. 4/28** 2-5 pm Louderman 458
  - **Course eval:** don't forget
    - Easy 1% of final grade, feedback extremely helpful
- Stay tuned to Piazza for any TA office hours next week. (Prof. Cole will hold his as usual)

Problem du Jour – Network Design

- You have a collection of cities on a map…

Problem du Jour – Network Design

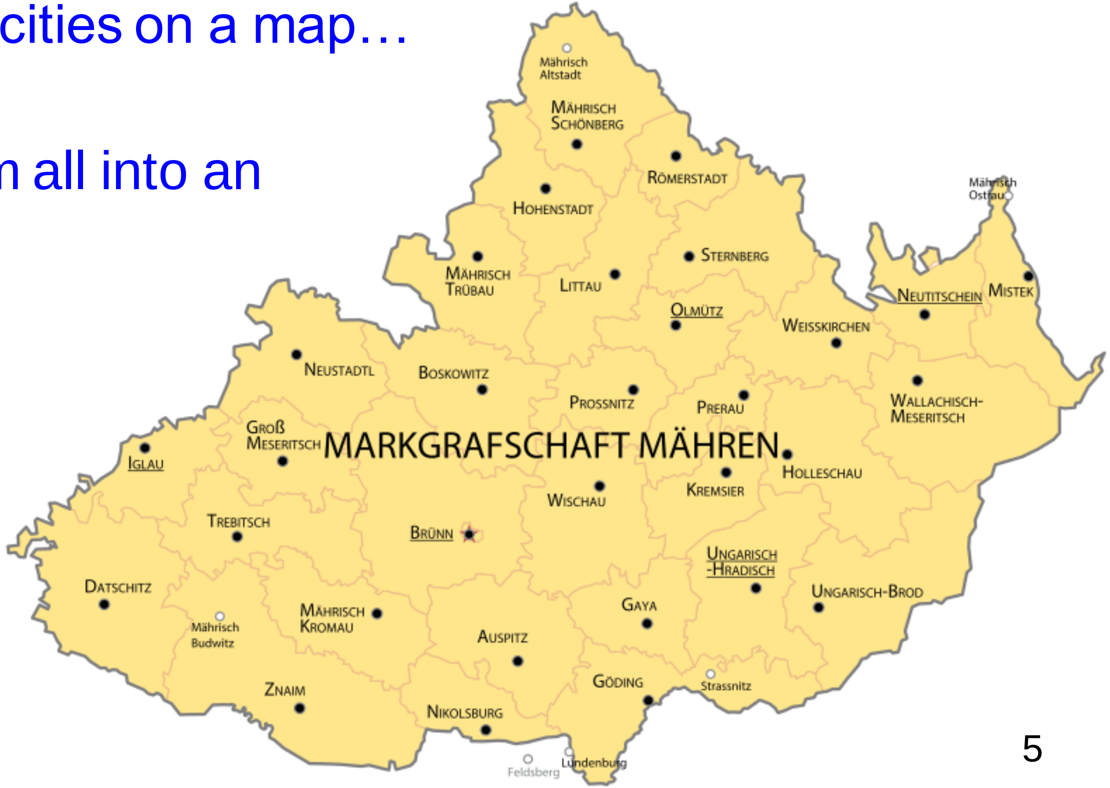
- You have a collection of cities on a map…



Problem du Jour – Network Design

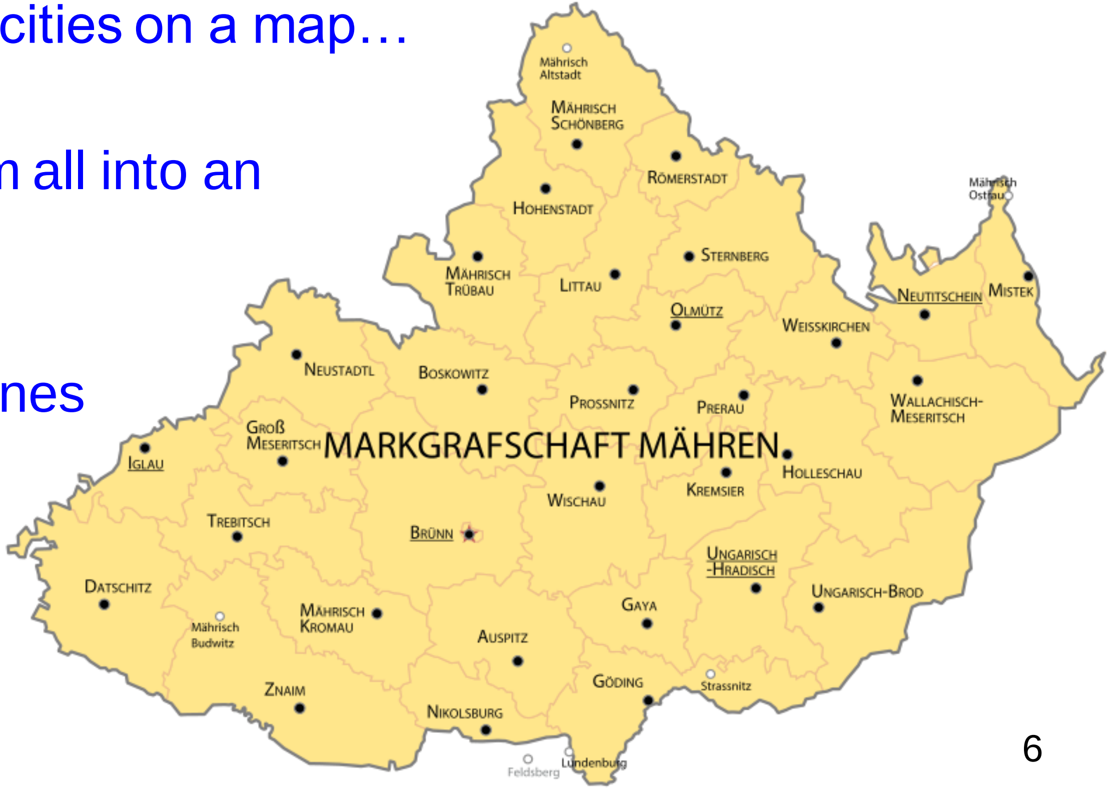
- You have a collection of cities on a map…

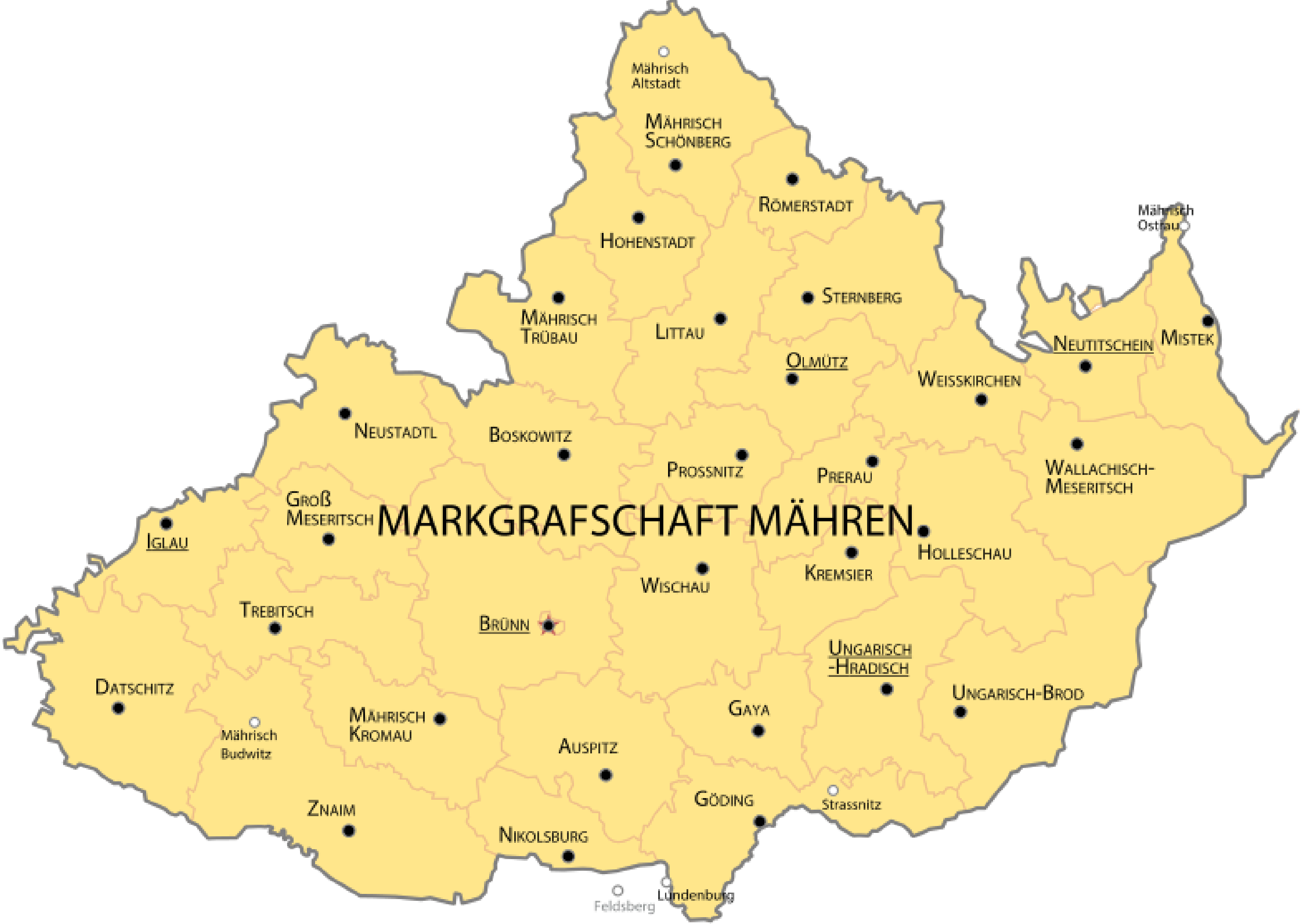
- You want to connect them all into an electric power grid.

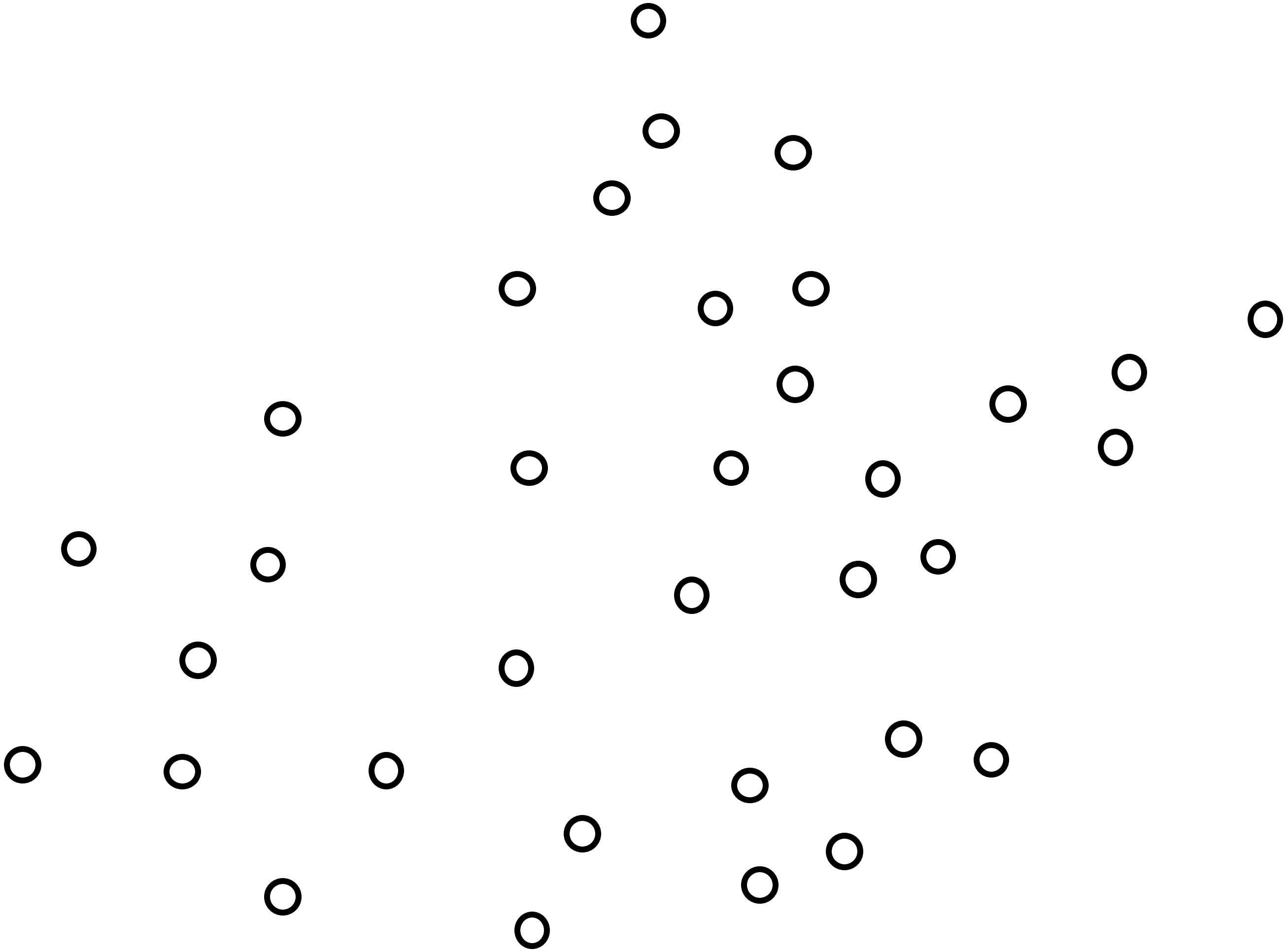


Problem du Jour – Network Design

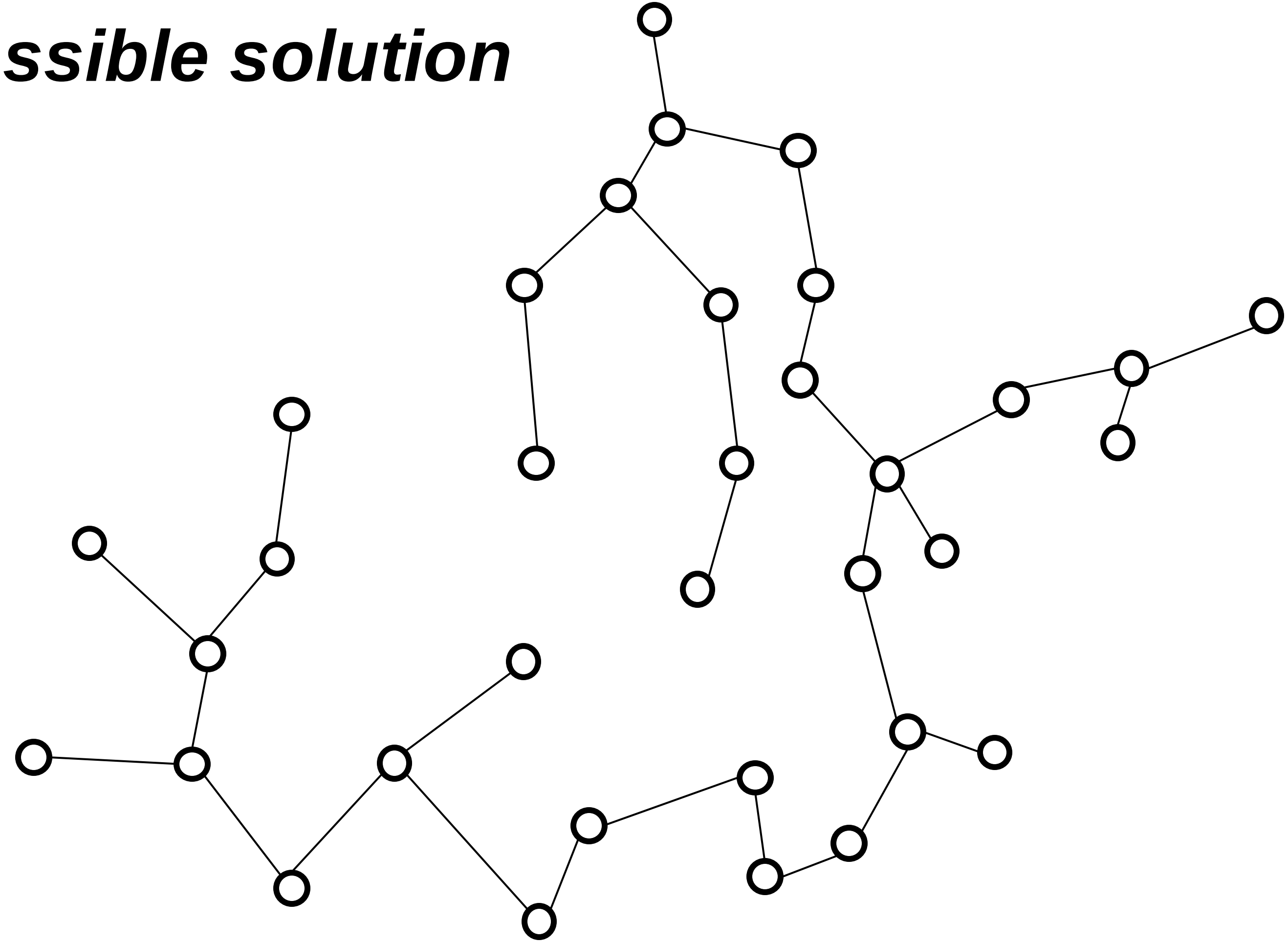
- You have a collection of cities on a map…
- You want to connect them all into an electric power grid.
- Can string transmission lines between cities
- Every city must be connected!




Mährisch Altstadt
Mährisch Schönberg
Römerstadt
Hohenstadt
Mährisch Ostrau
Sternberg
Mährisch Trübau
Littau
Neutitschein
Mistek
Olmütz
Weisskirchen
Neustadtl
Boskowitz
Prossnitz
Prerau
Wallachisch-Meseritsch
Groß Meseritsch
MARKGRAFSCHAFT MÄHREN
Iglau
Holleschau
Kremsier
Wischau
Trebitsch
Brünn
Ungarisch -Hradisch
Datschitz
Ungarisch-Brod
Mährisch Kromau
Gaya
Mährisch Budwitz
Auspitz
Göding
Strassnitz
Znaim
Nikolsburg
Feldsberg
Lundenburg



One possible solution



Abstract Graph Problem

- Cities form set of vertices
- *All possible* transmission lines are edges between vertices
- Goal is to pick a subset of edges that “spans” graph (that is, subset that *connects all vertices*
- **So why not just add all possible edges?**

Abstract Graph Problem

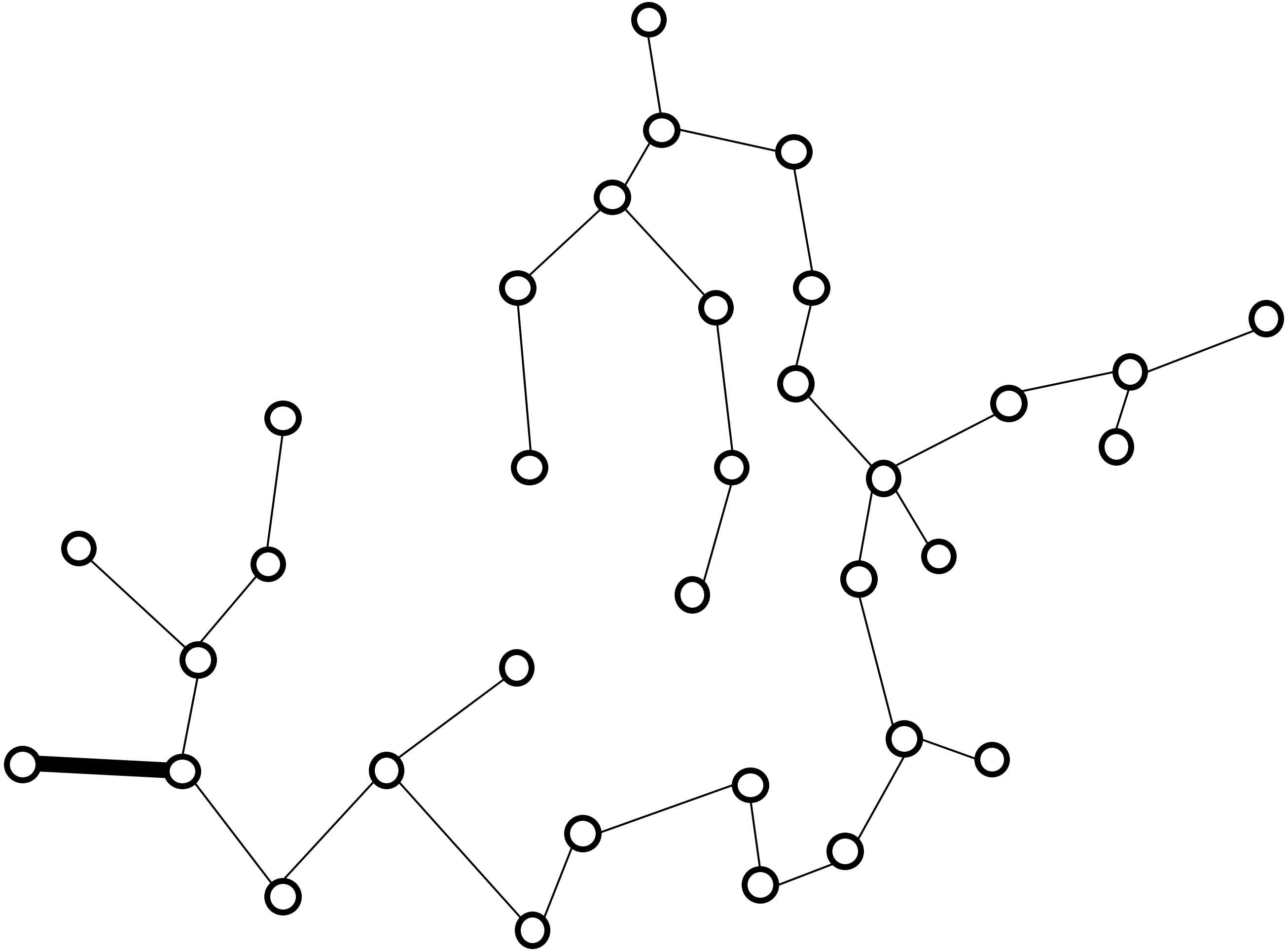
- Cities form set of vertices
- *All possible* transmission lines are edges between vertices
- Goal is to pick a subset of edges that "spans" graph (that is, subset that *connects all vertices*)
- **So why not just add all possible edges?**

COST!

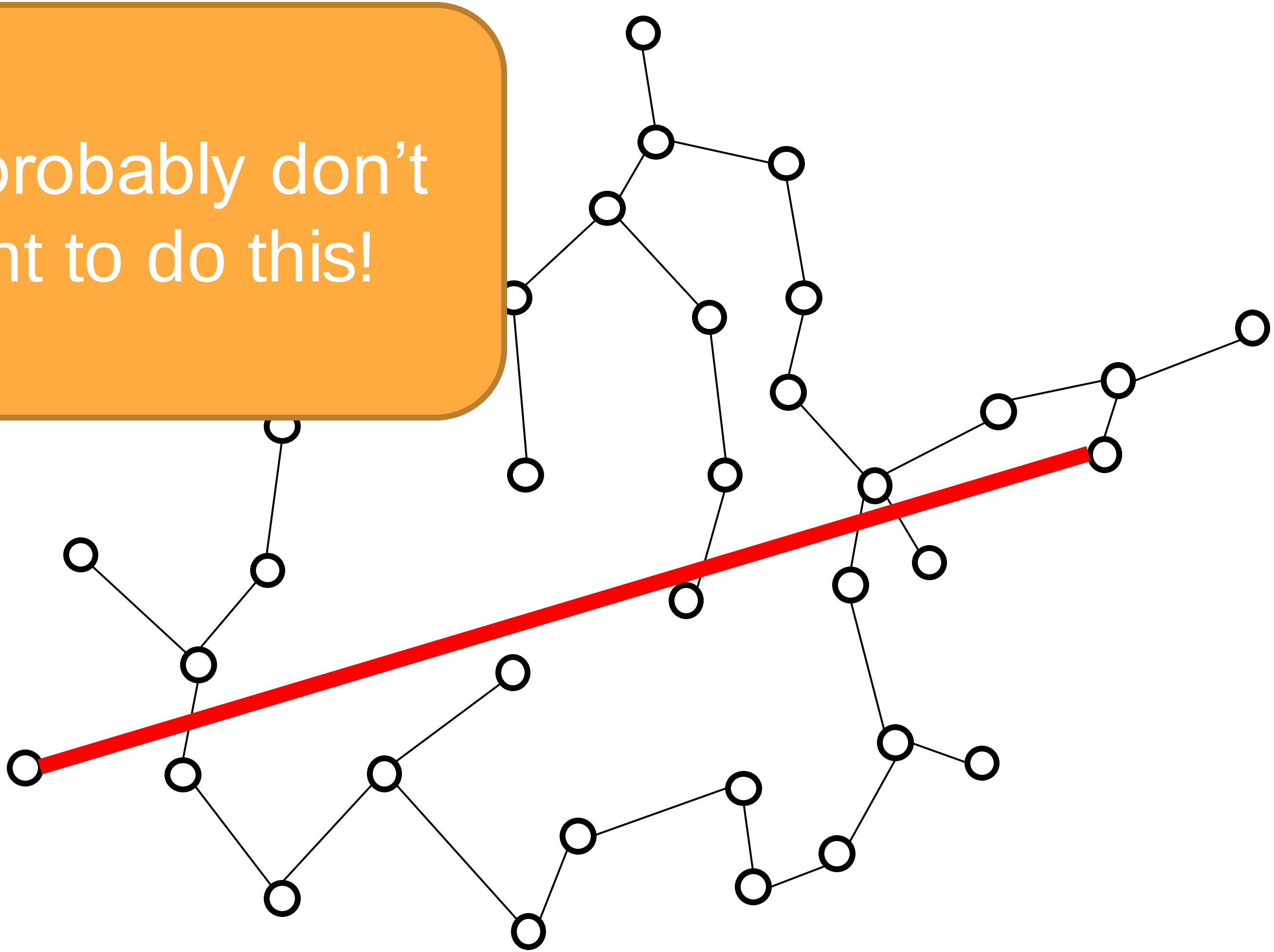
Adding Construction Costs

- Using edge between vertices u,v has cost **w(u,v) ≥ 0**
- Want to minimize total cost to connect all vertices
- Hence, pick a set **T** of edges that spans graph s.t.

$$W(T) = \sum_{e \in T} w(e) \text{ is minimized.}$$



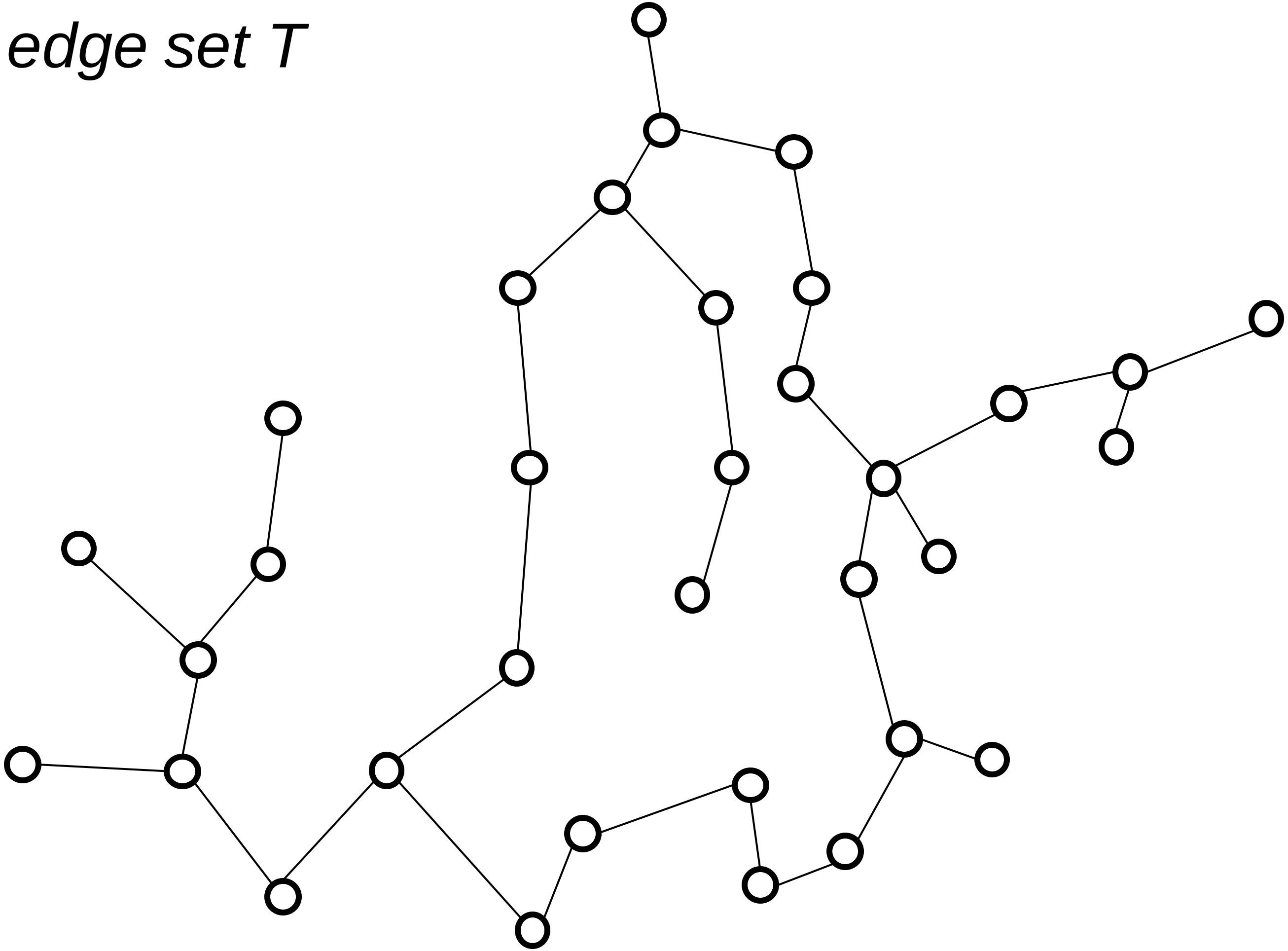
We probably don’t want to do this!



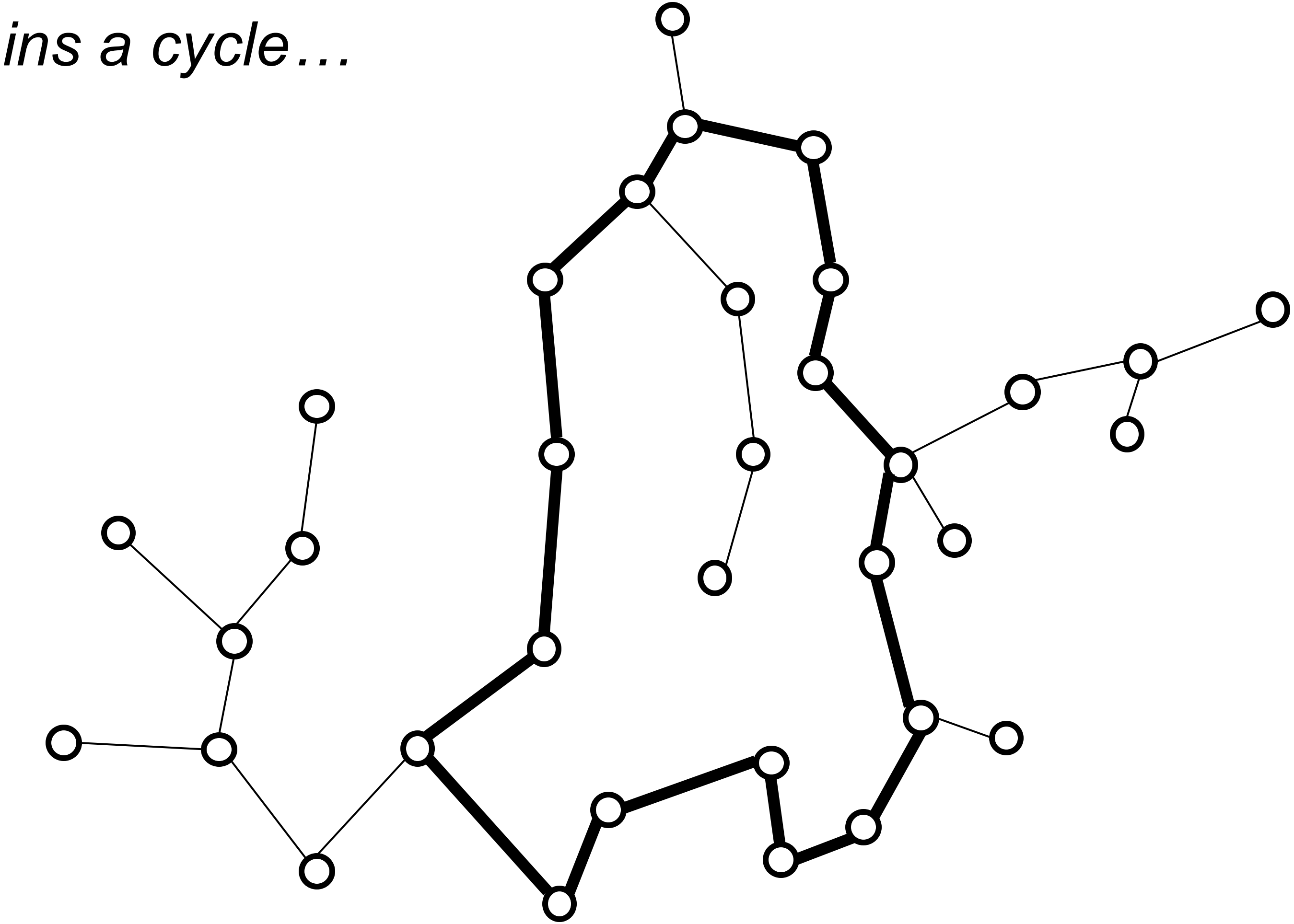
Observation: Desired T is a Tree!

- We just need to connect all vertices.
- If any cycle exists, *some edge can be removed* without disconnecting any vertex.
- Since edges have non-negative cost, this can only improve W(T).
- Hence, T is an (undirected) acyclic graph, also known as a **tree**.

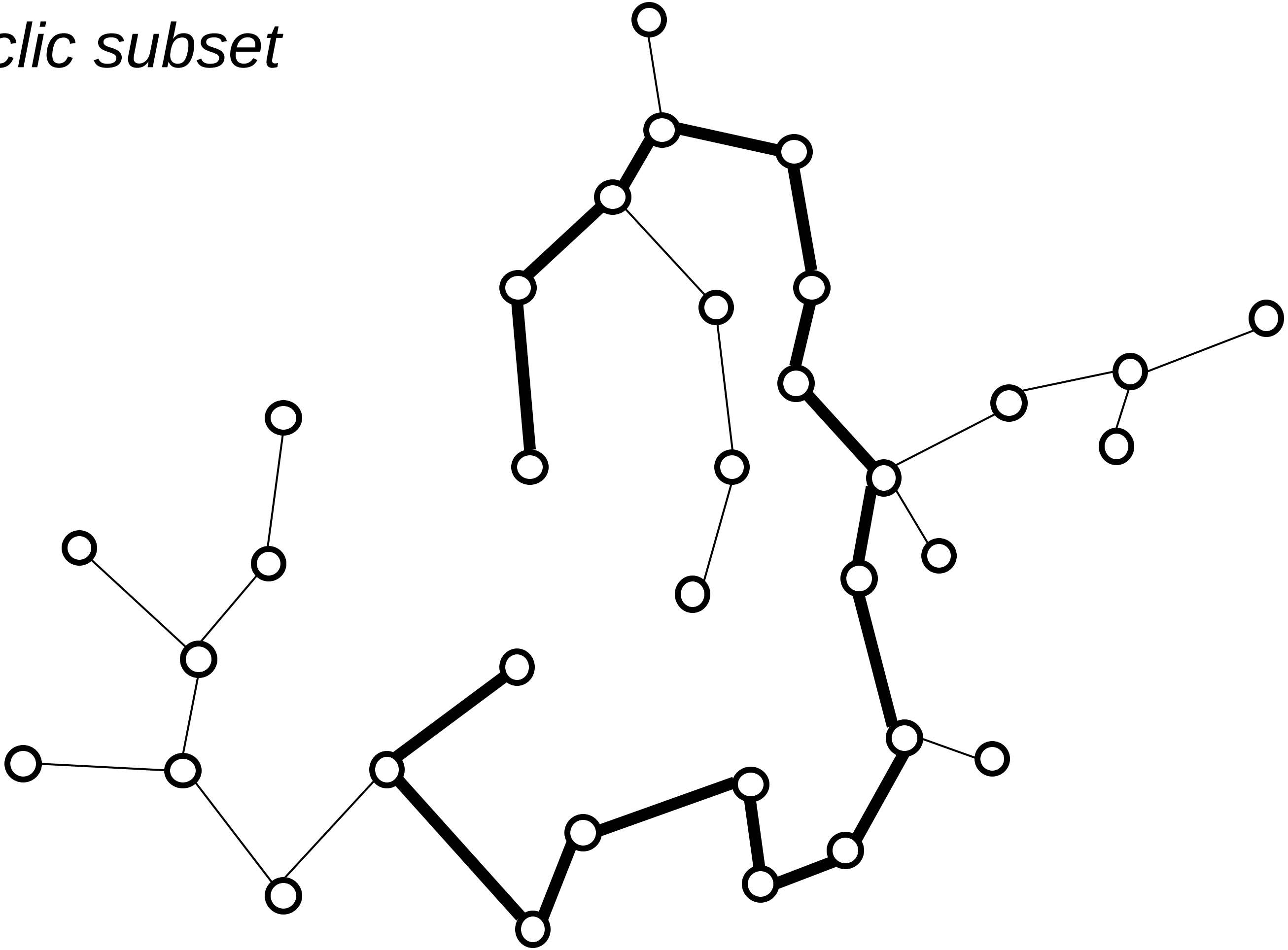
Possible edge set T



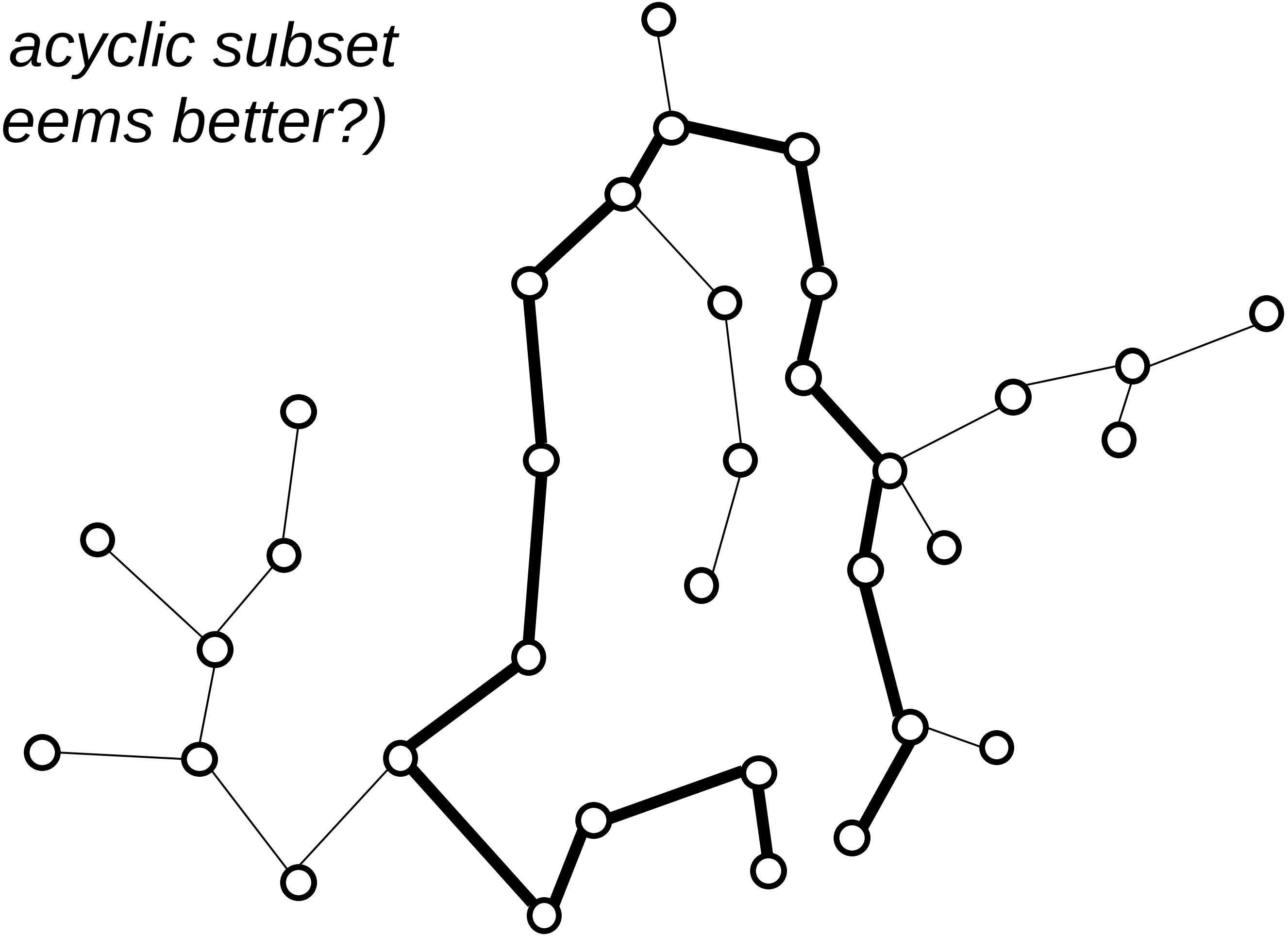
T contains a cycle…



One acyclic subset



Another acyclic subset
(which seems better?)



Formal Problem: Minimum Spanning Tree

- Given undirected graph **G = (V,E)** with weights **w(e) ≥ 0** for all e ϵ E

- Find a *tree* **T** that spans G, s.t.

$$W(T) = \sum_{e \in T} w(e) \text{ is minimized.}$$

- T is called a **minimum spanning tree** of G.

Other Applications of Minimum Spanning Tree

- Other network design problems (phone, Internet, road, …)
- Clustering data points by proximity
  [remove k-1 largest MST edges to form k clusters]
- Approximate answers to much harder problems (e.g. *travelling salesperson problem*)

General Approach

- Start with empty edge set T

- Keep adding edges to T, *without creating a cycle*, until T spans G.

- **Question**: how do we know *which edge to add next* to ensure that W(T) ends up being minimal?

Greedy Principle

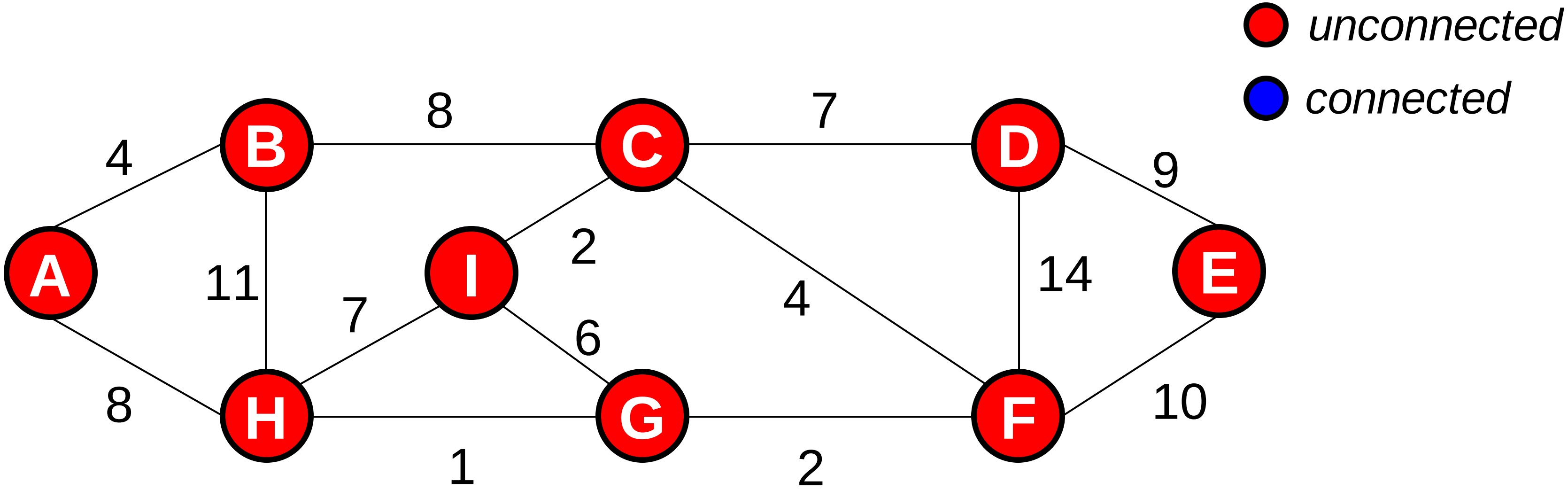
- Define a “local” criterion to apply when picking each edge

- At each step, *pick the edge that is currently best by this criterion* and add it to T.

- Keep picking edges until T spans G.

Greedy Principle Applied to MST (Prim's Algo)

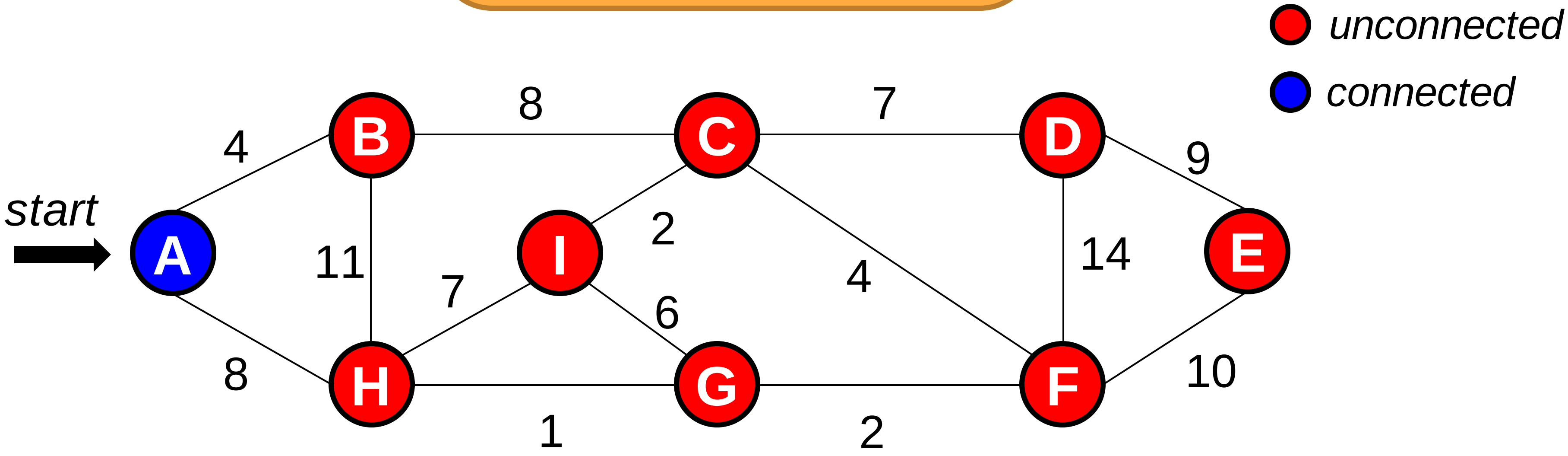
- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



Greedy Princip (Prim's Algo)

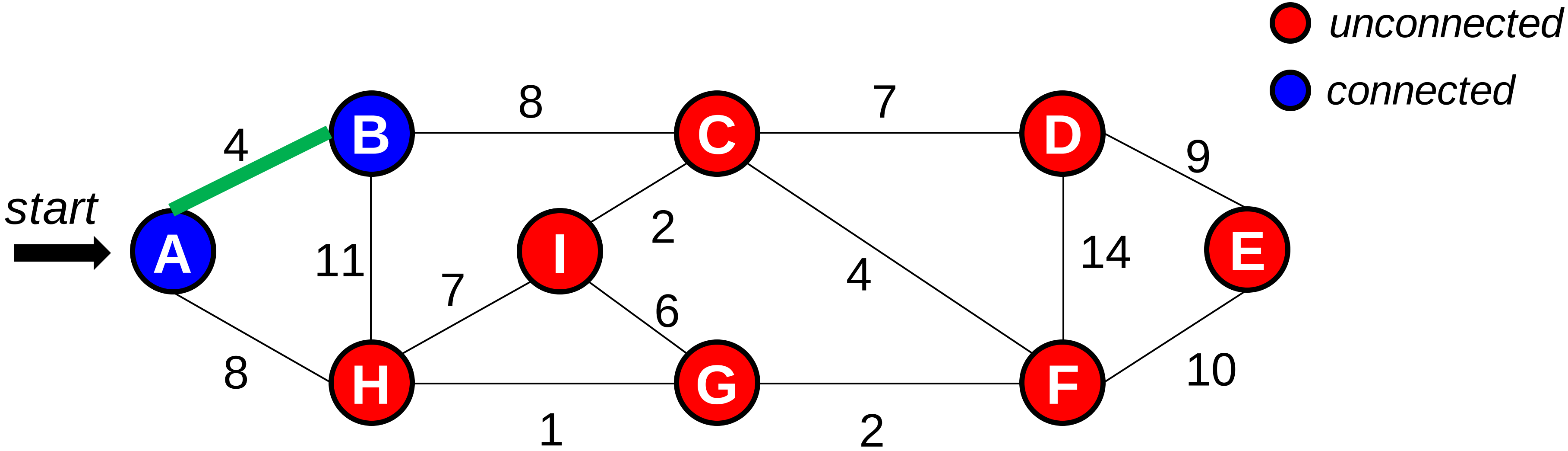
- **Prim's criteri** nimum w(e) that connects a ver et in T.

Starting vertex for building T is arbitrary.



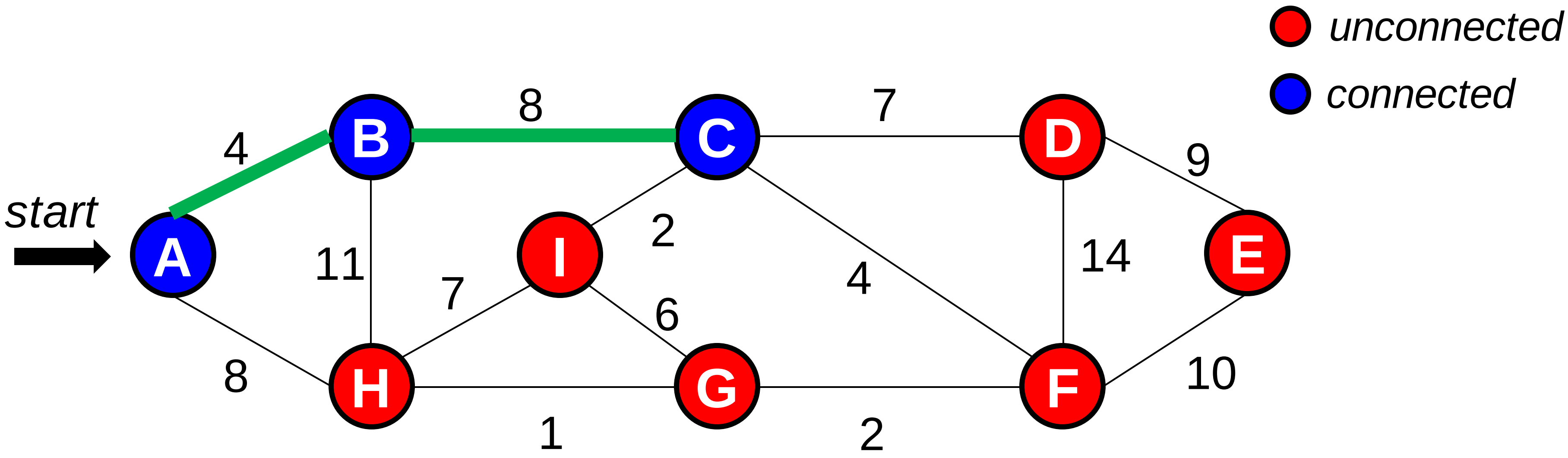
Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



Greedy Principle Applied to MST (Prim's Algo)

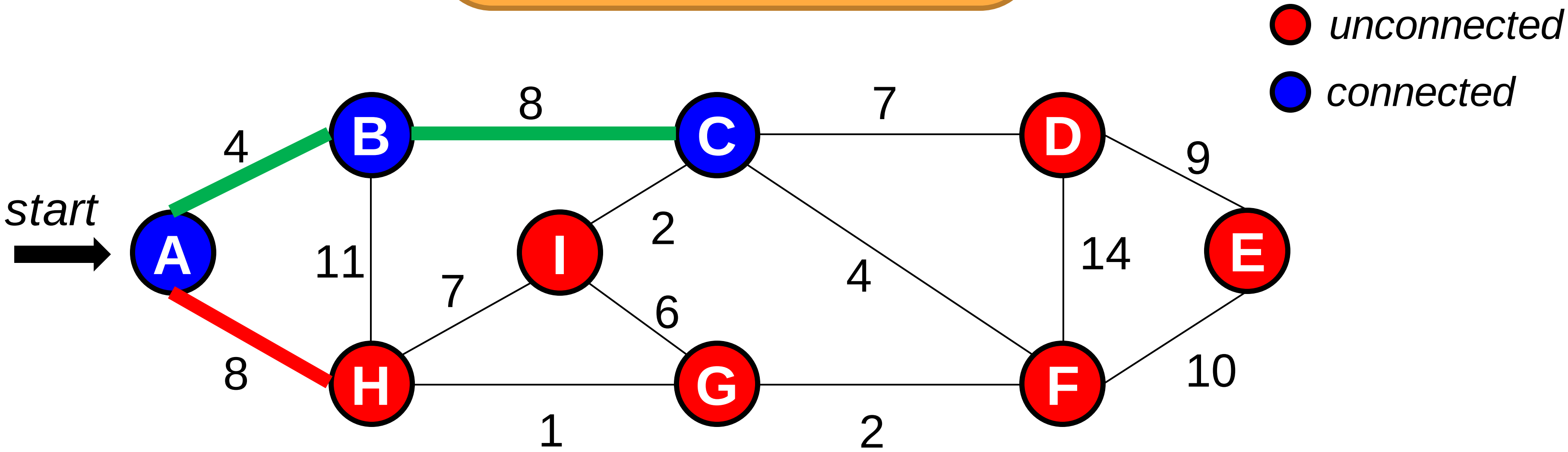
- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



Greedy Princip Prim's Algo)

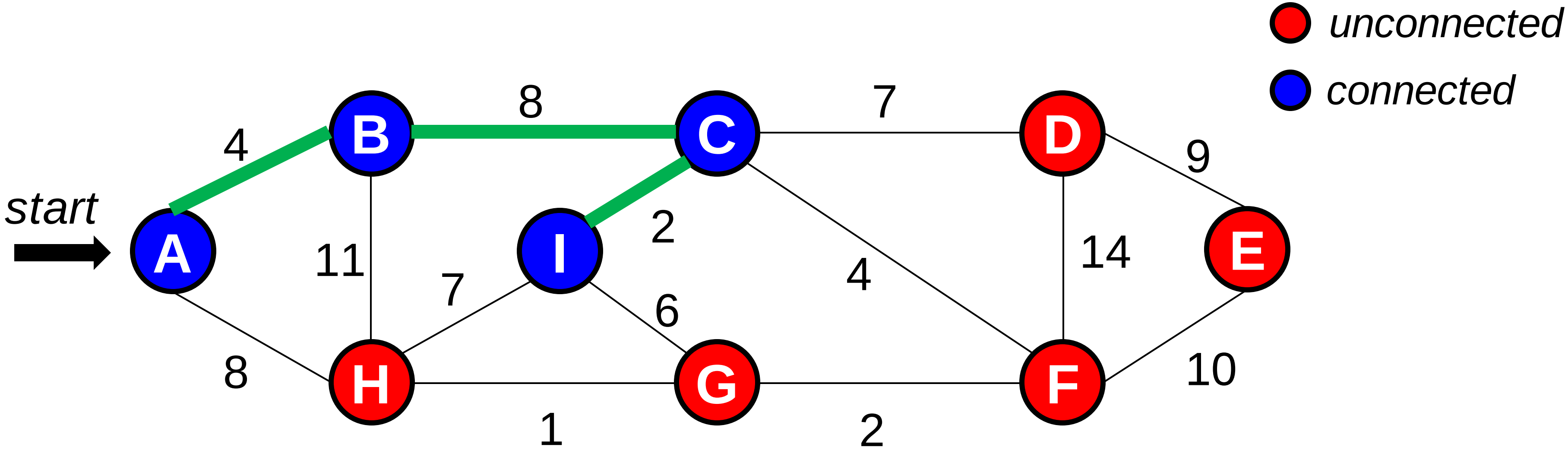
- **Prim's criteri** nimum w(e) that connects a ver et in T.

Break ties arbitrarily between edges of equal weight.



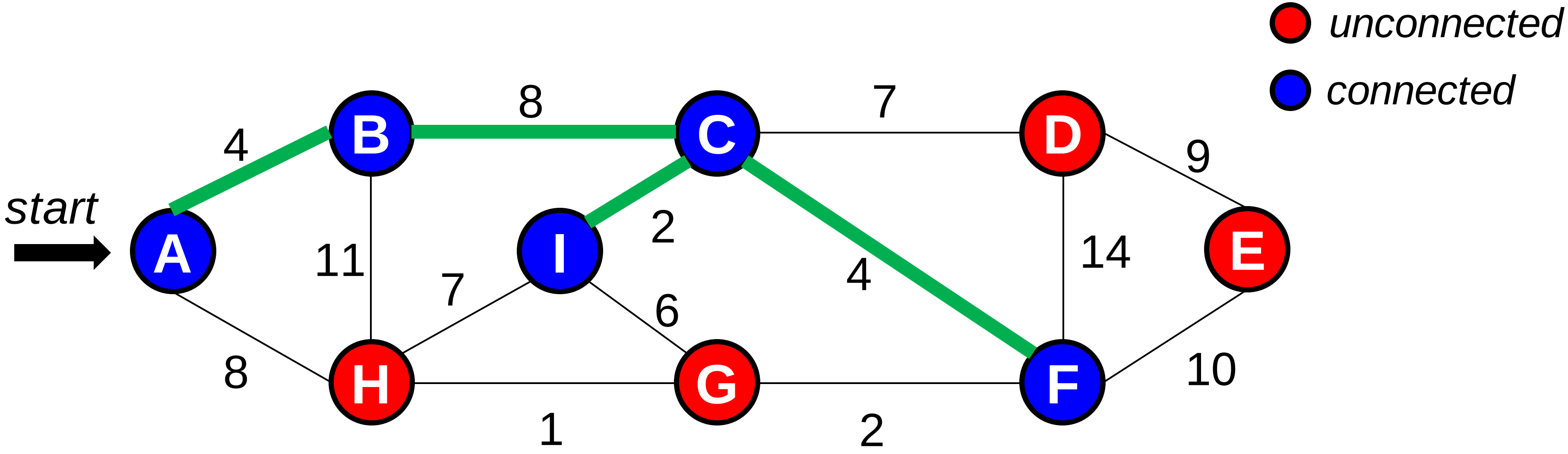
Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



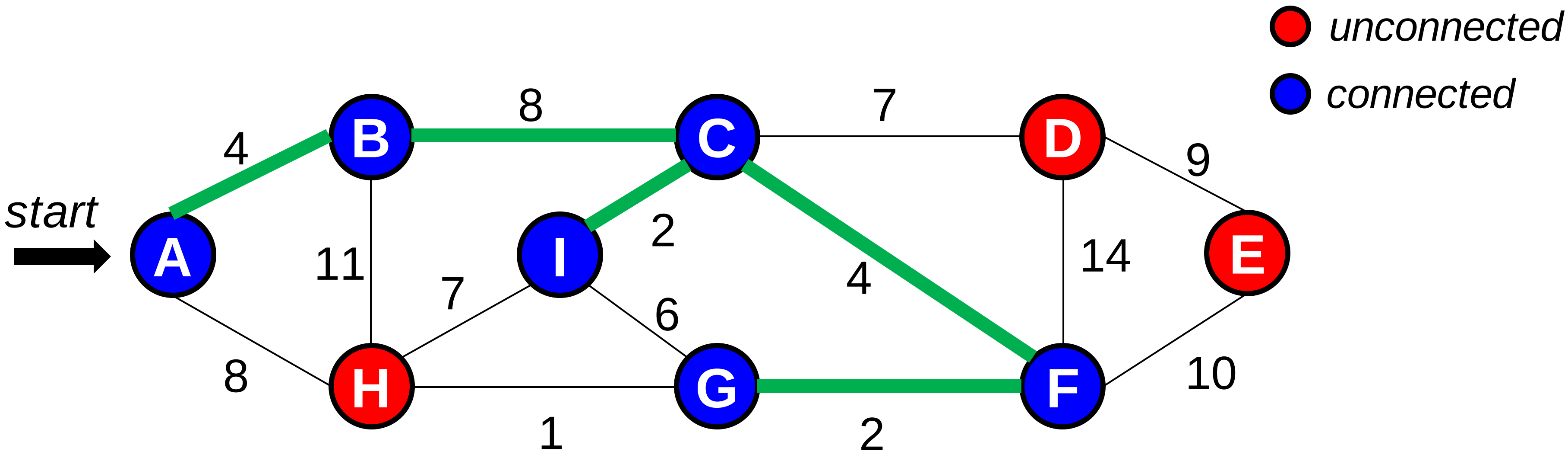
Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



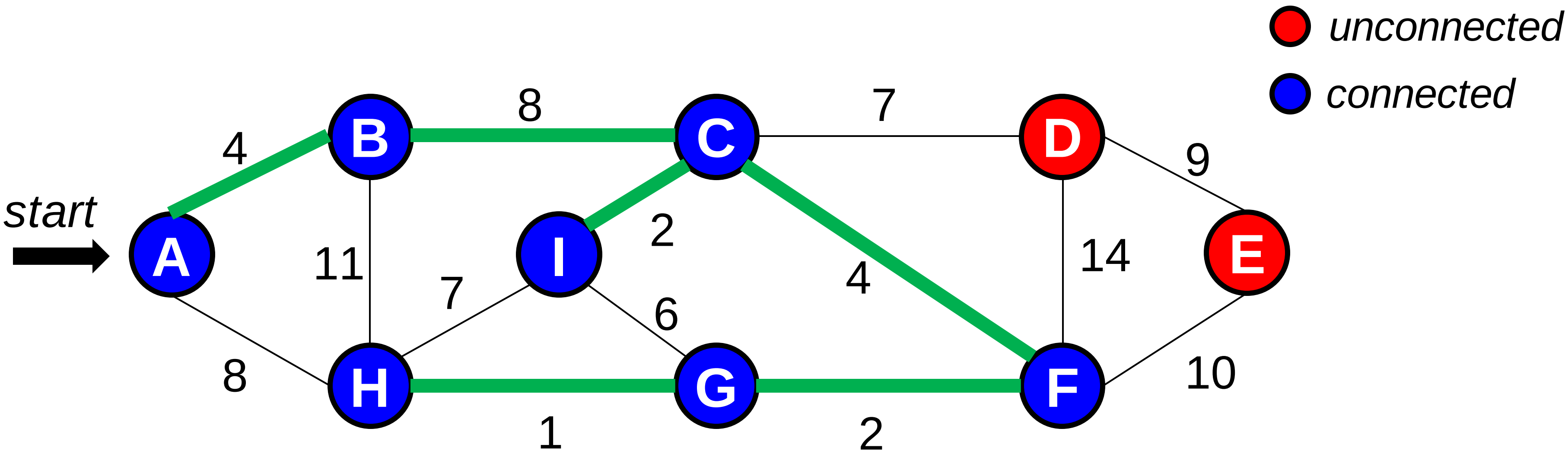
Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



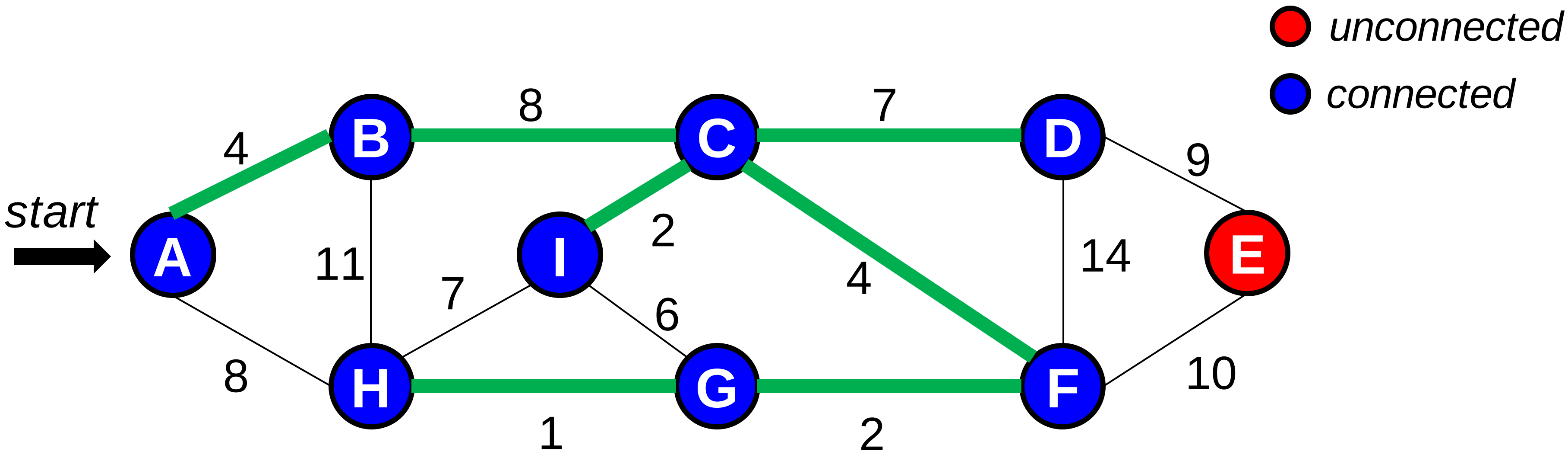
Greedy Principle Applied to MST (Prim’s Algo)

- **Prim’s criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



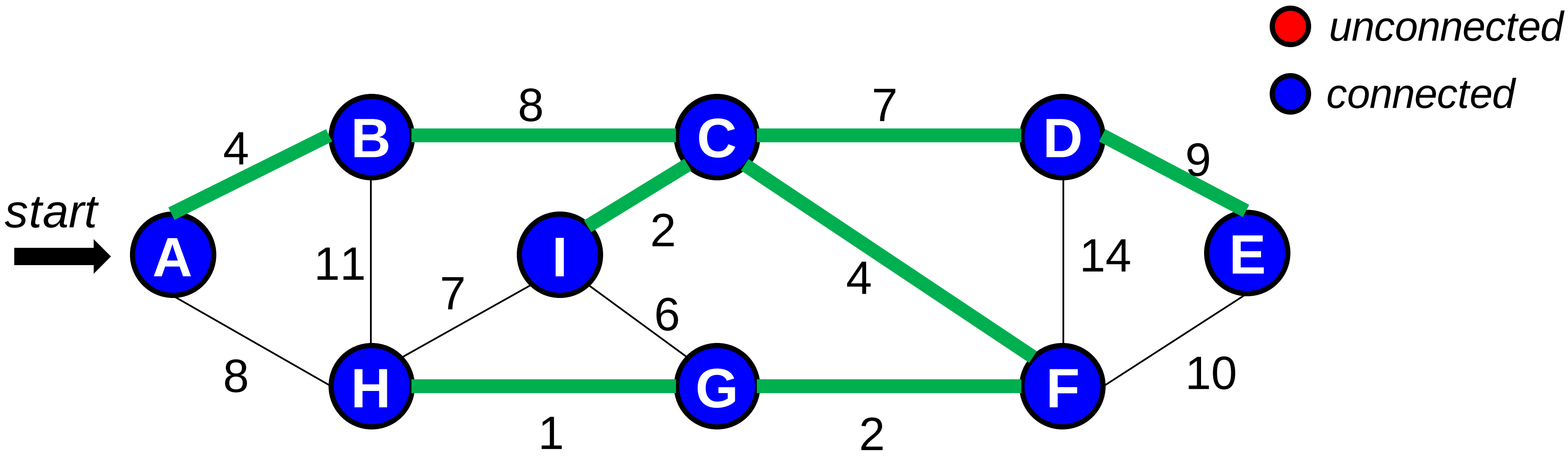
Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



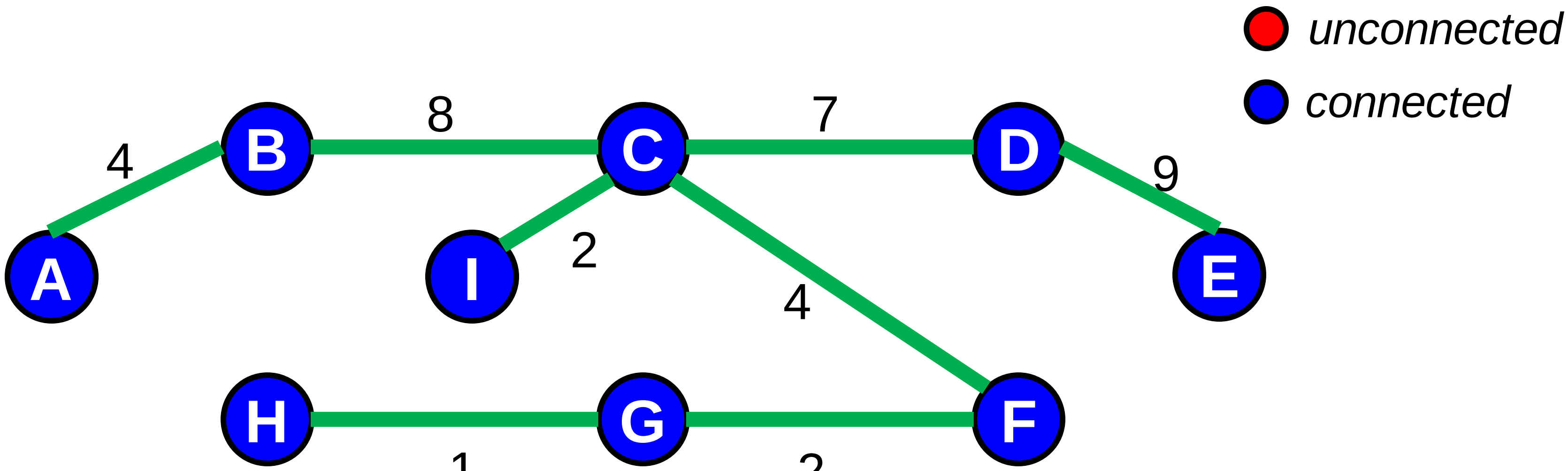
Greedy Principle Applied to MST (Prim’s Algo)

- **Prim’s criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.



Greedy Principle Applied to MST (Prim's Algo)

- **Prim's criterion**: pick the edge e of minimum w(e) that connects a vertex in T to a vertex not yet in T.

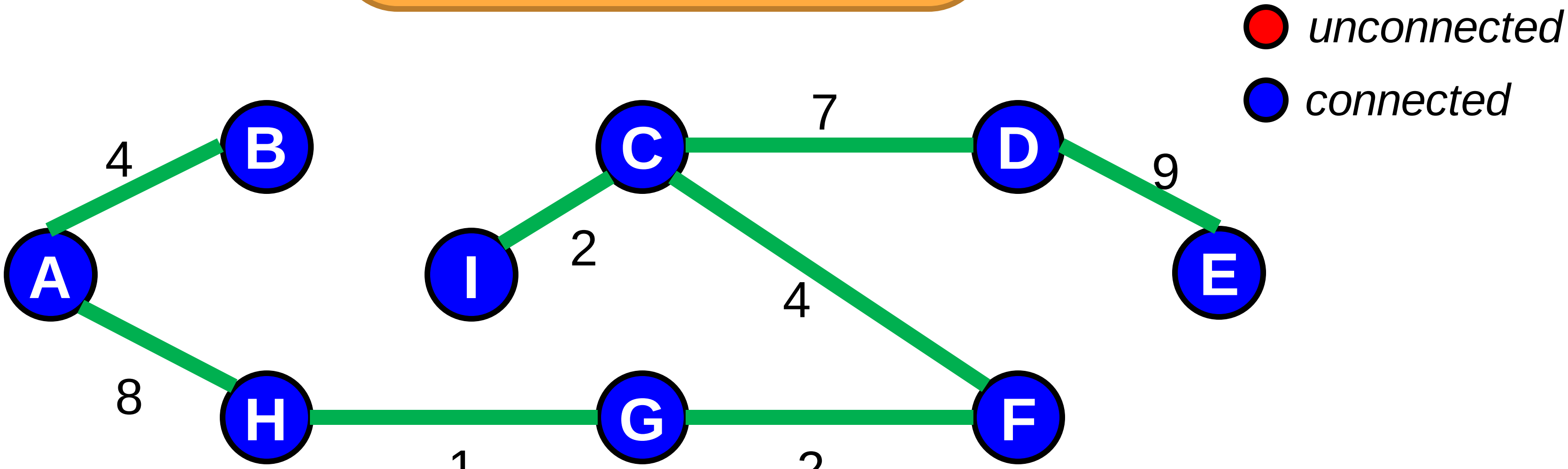


W(T) = 37

Greedy Princip (Prim’s Algo)

Note that G may not have a unique MST!

- **Prim’s criteri** nimum w(e) that connects a ver et in T.



W(T) = 37

Why Does Prim's Greedy Criterion Work?

- **Claim**: After any number of edges are chosen, algorithm's current edge set T is a ***subset*** *of some minimum spanning tree* for G.

- *(Hence, once T spans all of G, T is itself an MST for G.)*

Why Does Prim’s Greedy Criterion Work?

- **Claim**: After any number of edges are chosen, algorithm’s current edge set T is a ***subset*** *of some minimum spanning tree* for G.

- *(Hence, once T spans all of G, T is itself an MST for G.)*

- **Pf**: by induction on # of edges chosen so far.

- **Bas**: before any edges are chosen, T is empty, so is a subset of every MST for G.

Why Does Prim's Greedy Criterion Work?

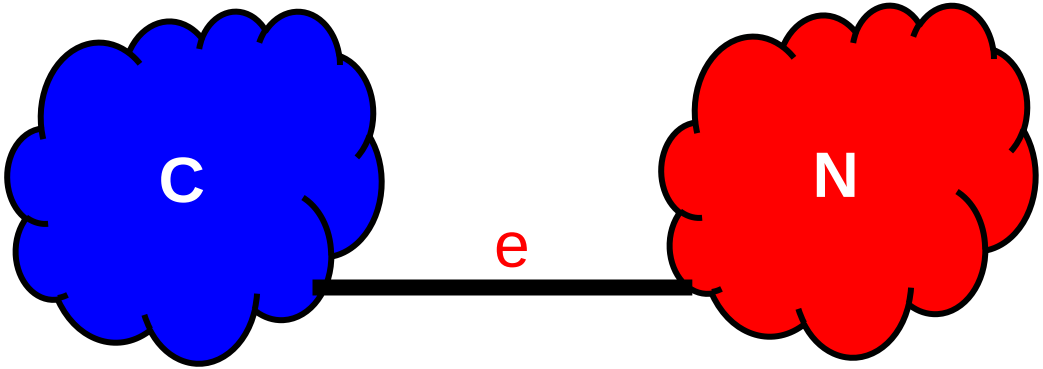
- **Claim**: After any number of edges are chosen, algorithm's current edge set T is a ***subset*** *of some minimum spanning tree* for G.

- **Ind**: Suppose Prim's criterion picks a next edge **e**.

- Let **C** and **N** be the connected and unconnected vertices of G after picking edge set T.

Why Does Prim's Greedy Criterion Work?

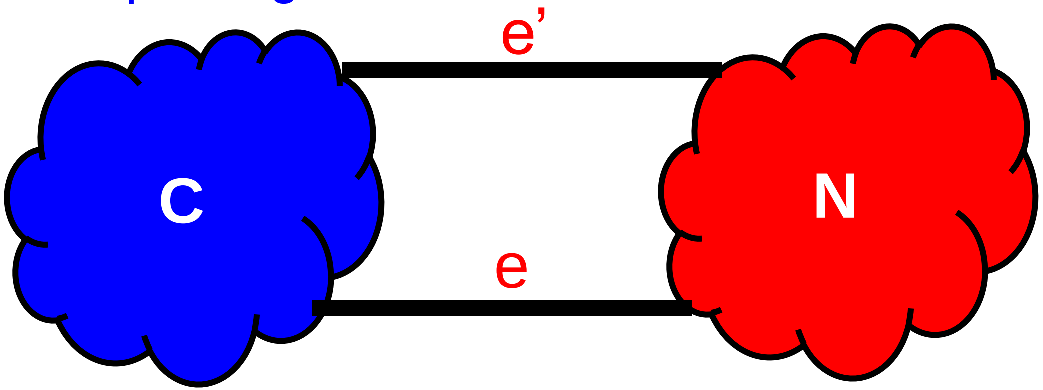
- **Claim**: After any number of edges are chosen, algorithm's current edge set T is a ***subset*** *of some minimum spanning tree* for G.



vertices connected to T *vertices not connected to T*

Why Does Prim's Greedy Criterion Work?

- **Claim**: After any number of edges are chosen, algorithm's current edge set T is a ***subset*** *of some minimum spanning tree* for G.
- By IH, T is a subset of some MST **T*** for G.
- Some unique edge **e'** of T* connects C and N, as does edge e.

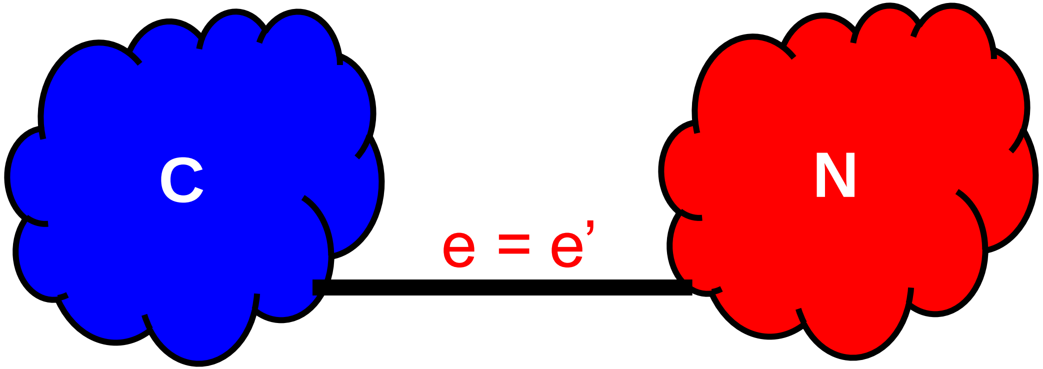


vertices connected to T

vertices not connected to T

Why Does Prim's Greedy Criterion Work?

- **Claim**: After any number of edges are chosen, algorithm's current edge set T is a ***subset*** *of some minimum spanning tree* for G.
- By IH, T is a subset of some MST **T*** for G.
- Some unique edge **e'** of T* connects C and N, as does edge e.

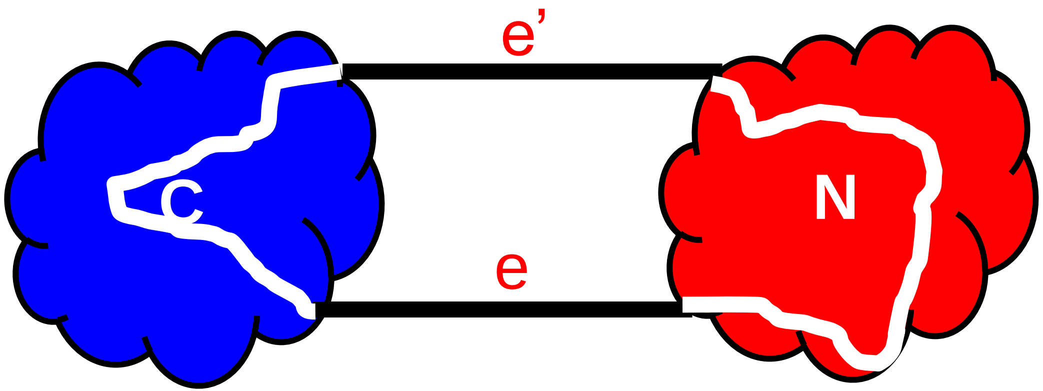


If e = e', then T U {e} is a subset of T*, and we are done.

vertices connected to T   *vertices not connected to T*

Why Does Prim's Greedy Criterion Work?

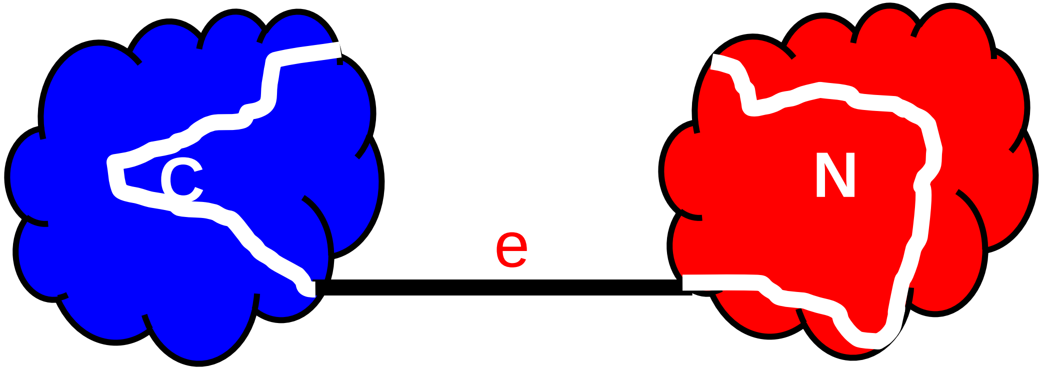
- Some unique edge **e'** of T* connects C and N, as does edge e.
- If e ≠ e', then T* U {e} (spanning tree + 1 edge) forms a **cycle** in G.



vertices connected to T   *vertices not connected to T*

Why Does Prim’s Greedy Criterion Work?

- Some unique edge **e’** of T* connects C and N, as does edge e.
- If e ≠ e’, then T* U {e} (spanning tree + 1 edge) forms a **cycle** in G.
- Hence, **T’ = T* U {e} – {e’}** *is another spanning tree for G.*



vertices connected to T

vertices not connected to T

Why Does Prim’s Greedy Criterion Work?

- Some unique edge **e’** of T* connects C and N, as does edge e.
- If e ≠ e’, then T* U {e} (spanning tree + 1 edge) forms a **cycle** in G.
- Hence, **T’ = T* U {e} – {e’}** *is another spanning tree for G.*
- Prim’s criterion picked e instead of e’, so **w(e) ≤ w(e’).**
- Conclude that **W(T’)** = W(T*) – w(e’) + w(e) **≤ W(T*)**, and so T’ is a *minimum spanning tree* that contains T U {e}, as claimed. **QED**

Implementing Prim’s Algorithm

- Maintain set of unconnected vertices.
- For each unconnected vertex v, maintain v.conn, weight *of lowest-weight edge* connecting v to any vertex in T.
- When we add an edge (u,v) to T, update connections to each x adjacent to v:

$$\text{If } w(v,x) < x.\text{conn}, \ \text{ then } x.\text{conn} \leftarrow w(v,x)$$

Prim’s MST Algorithm (*Adding to T Not Shown*)

```
starting vertex v gets v.conn ← 0; all other u get u.conn ← ∞
mark all vertices as unconnected

while (any vertex unconnected)
    v ← unconnected vertex with smallest v.conn
    for each edge (v,u)
        if (u.conn > w(u,v))
            u.conn ← w(u,v)
    mark v connected      // augment partial MST with edge from T to v
```

Prim’s MST Algorithm (*Adding to T Not Shown*)

```
• starting vertex                     et u.conn ← ∞
• mark all vertic

• while (any ver
•     v ← unconn                      n
•     for each ed
•         if (u.conn
•             u.conn ← w(u,v)
•     mark v connected    // augment partial MST with edge from T to v
```

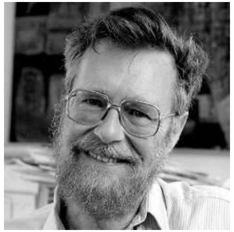
Does this pseudocode look familiar?

Dijkstra’s Shortest Path Algorithm

```
• starting vertex v gets v.dist $\leftarrow$ 0; all other u get u.dist $\leftarrow$ $\infty$
• mark all vertices as unfinished

• while (*any vertex unfinished*)
•     v $\leftarrow$ unfinished vertex with smallest v.dist
•     for each edge (v,u)
•         if (u.dist > v.dist + w(u,v))
•             u.dist $\leftarrow$ v.dist + w(u,v)
•     mark v finished
```

Prim vs Dijkstra



- Prim's MST algorithm is *nearly identical* to Dijkstra's shortest-path algorithm

- Only difference is in *greedy criterion* for next vertex to process.
  - Dijkstra – **total weight of path** from start to unfinished vertex v
  - Prim – **weight of last edge on path** from start to unconnected vertex v

- **We can use *same min-first priority queue trick* to efficiently select next vertex to connect to T; for Prim's algo, use u.conn as vertex's key.**

Prim’s MST Algorithm w/Queue

```
v.conn ← 0; D[v] ← PQ.insert(starting vertex v)
For all other vertices u
   u.conn ← ∞; D[u] ← PQ.insert(u)

while (PQ not empty)
    v ← PQ.extractMin()
    for each edge (v,u)
        if (u.conn > w(v,u))
            u.conn ← w(v,u)
            D[u].decrease(u)
```

Prim’s MST Algorithm w/Queue

```
v.conn ← 0; D[v] ← PQ.insert(starting vertex v)
For all other vertices u
    u.conn ← ∞; D[u] ← PQ.insert(u)

while (PQ not empty)
    v ← PQ.extractMin()
    for each edge (v,u)
        if (u.conn > w(v,u))
            u.conn ← w(v,u)
            D[u].decrease(u)
```

Note: book’s pseudocode uses common variable names, so that Prim & Dijkstra code, *including tree maintenance*, differ by only **one line**.

Running Time of Prim’s Algorithm

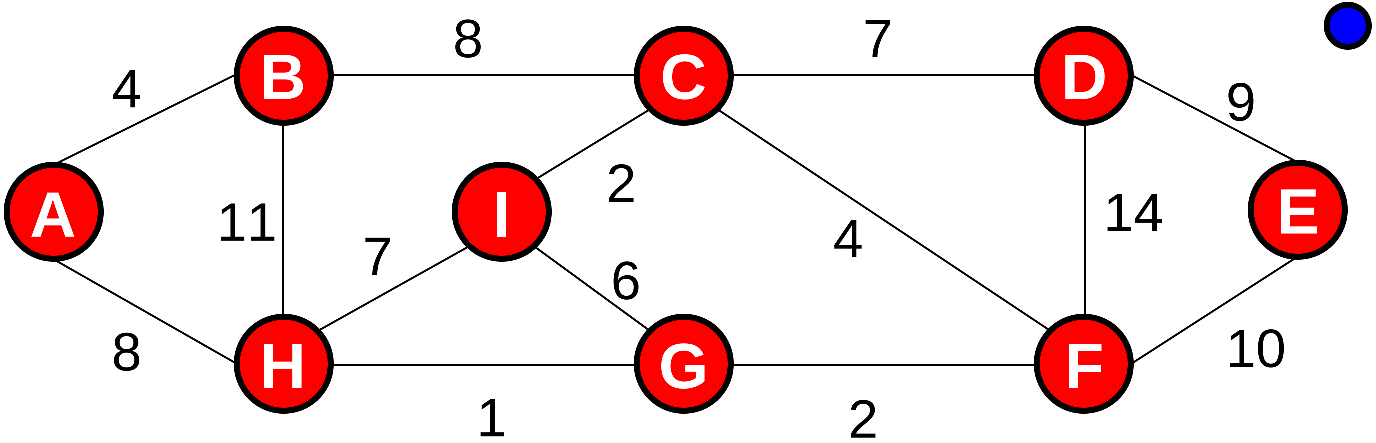
- Exactly the same analysis as for Dijkstra’s algorithm!
- Dominant cost is again heap operations.
- **Algorithm runs in time Θ((|V| + |E|) log |V|)** using a binary heap.

Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.

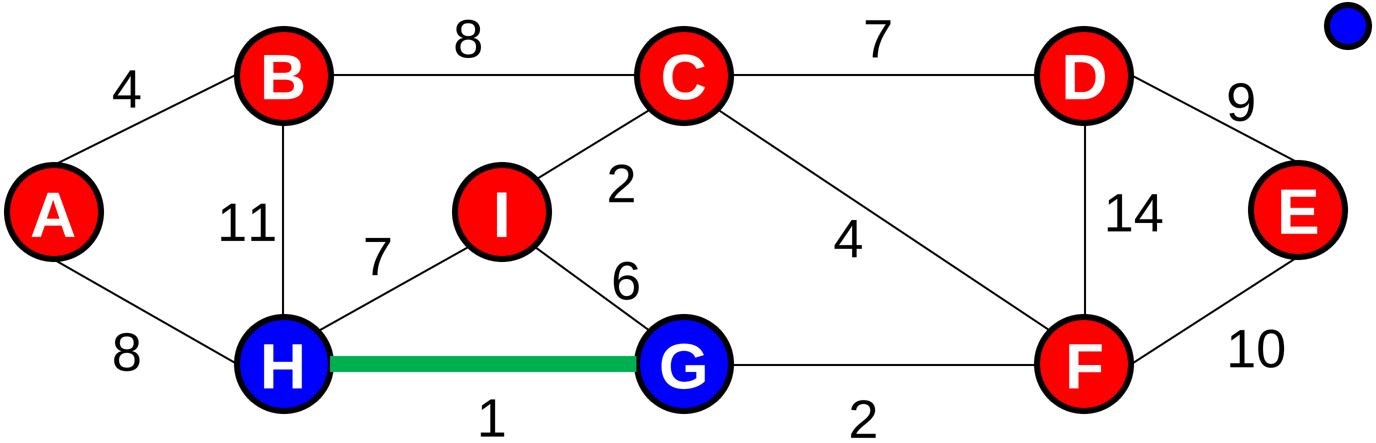
🔴 *unconnected*

🔵 *connected*



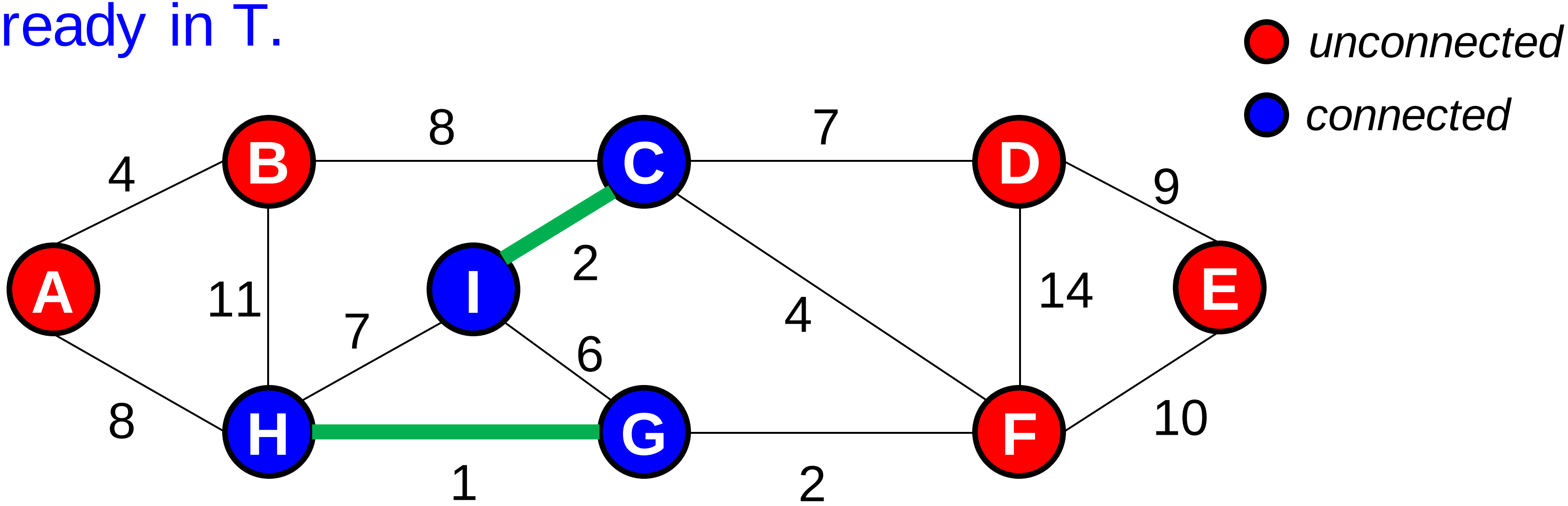
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



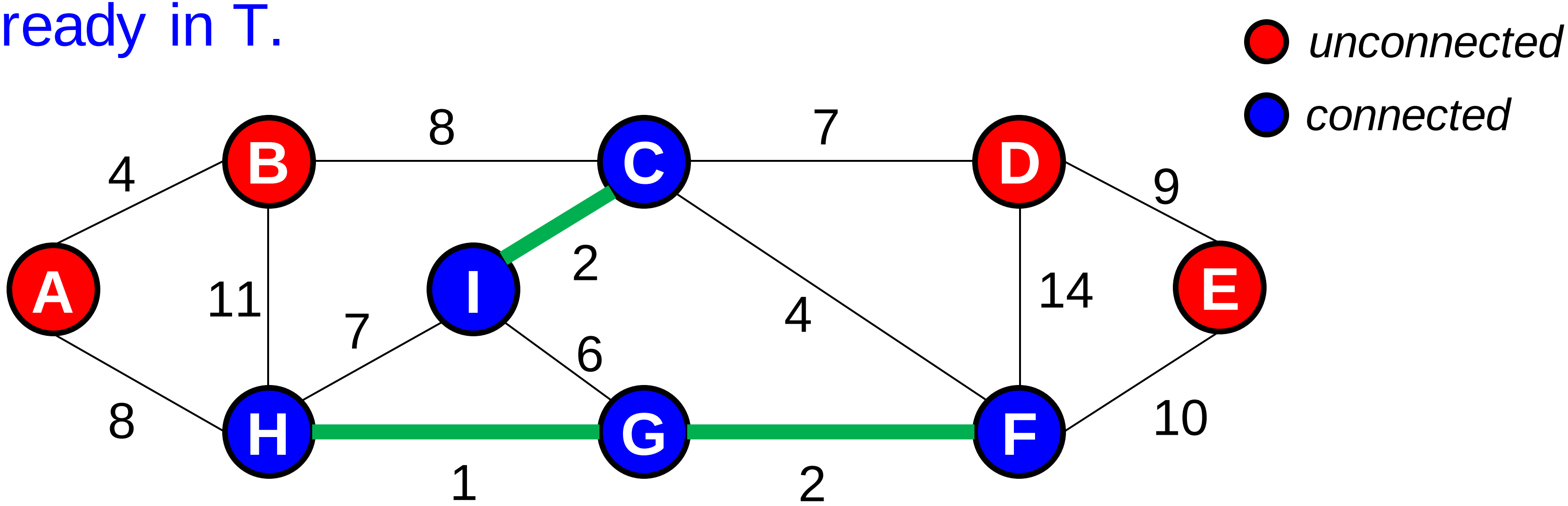
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



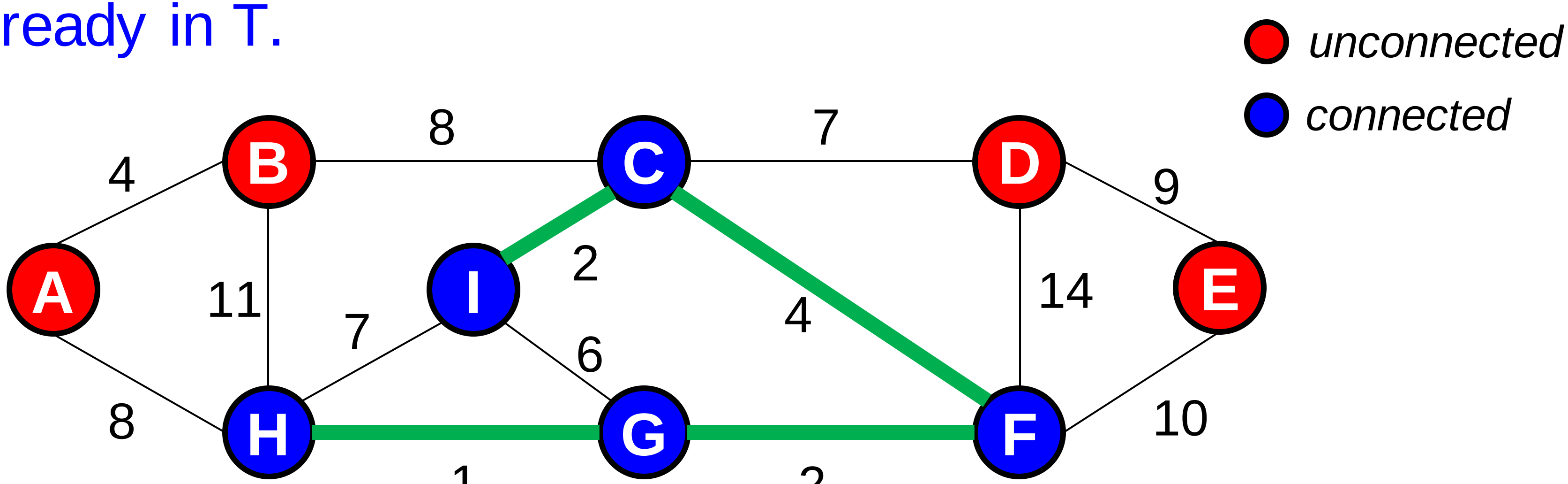
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



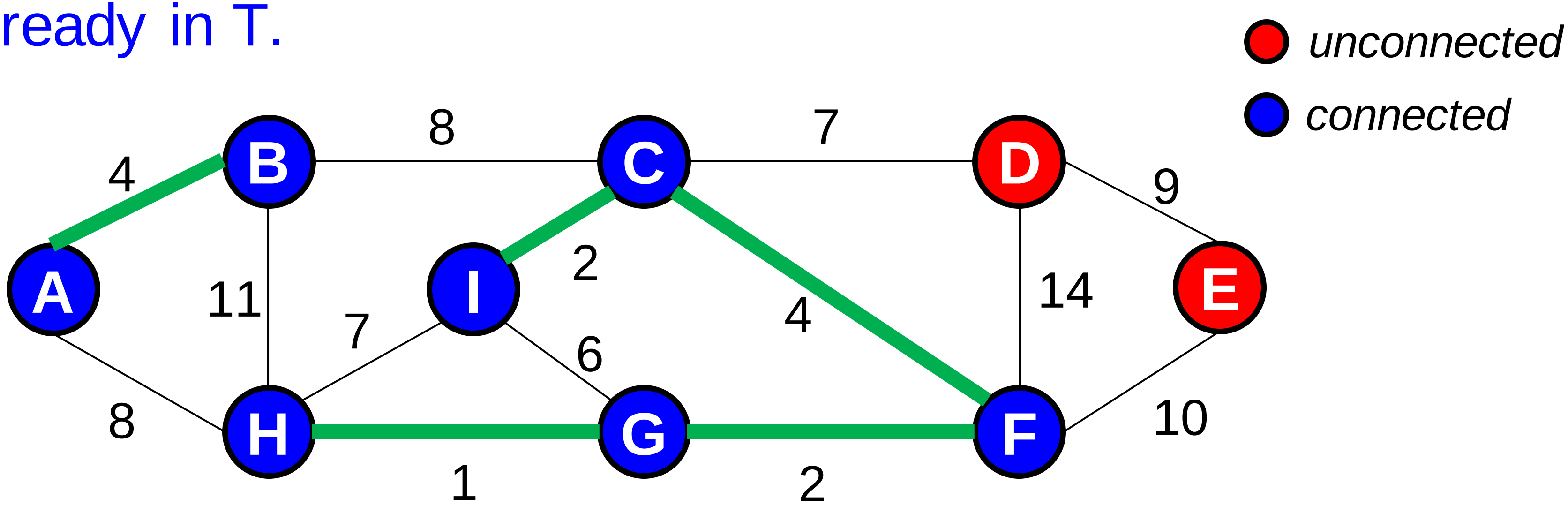
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



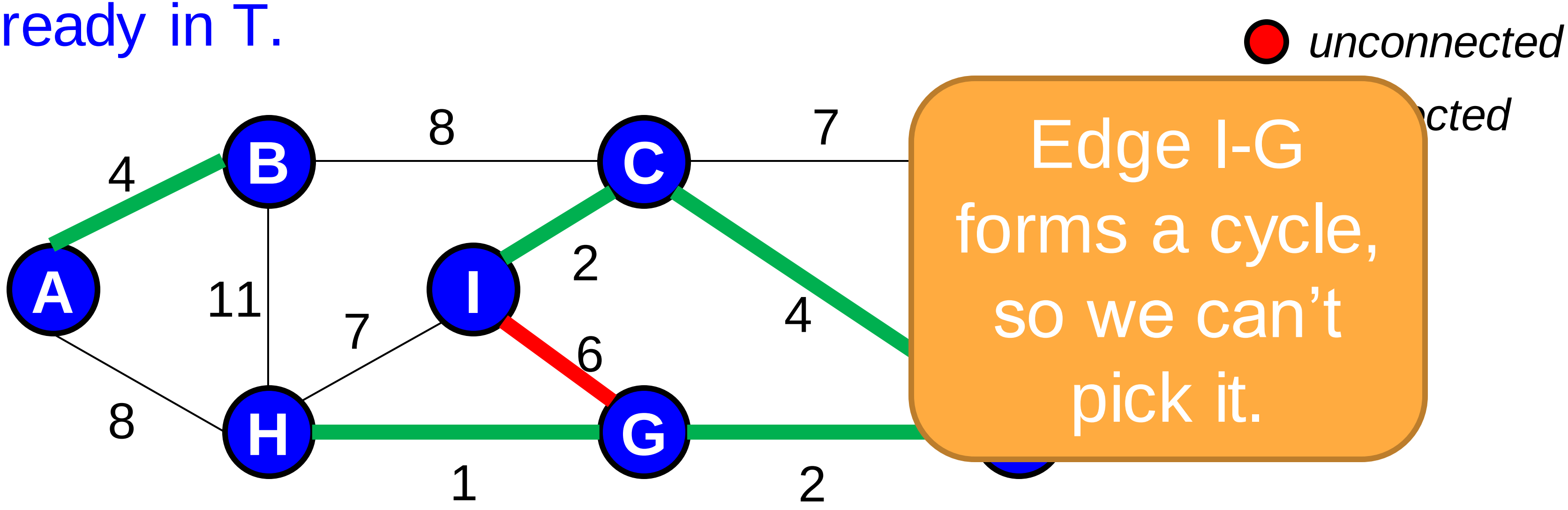
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



Another Greedy Criterion for MST

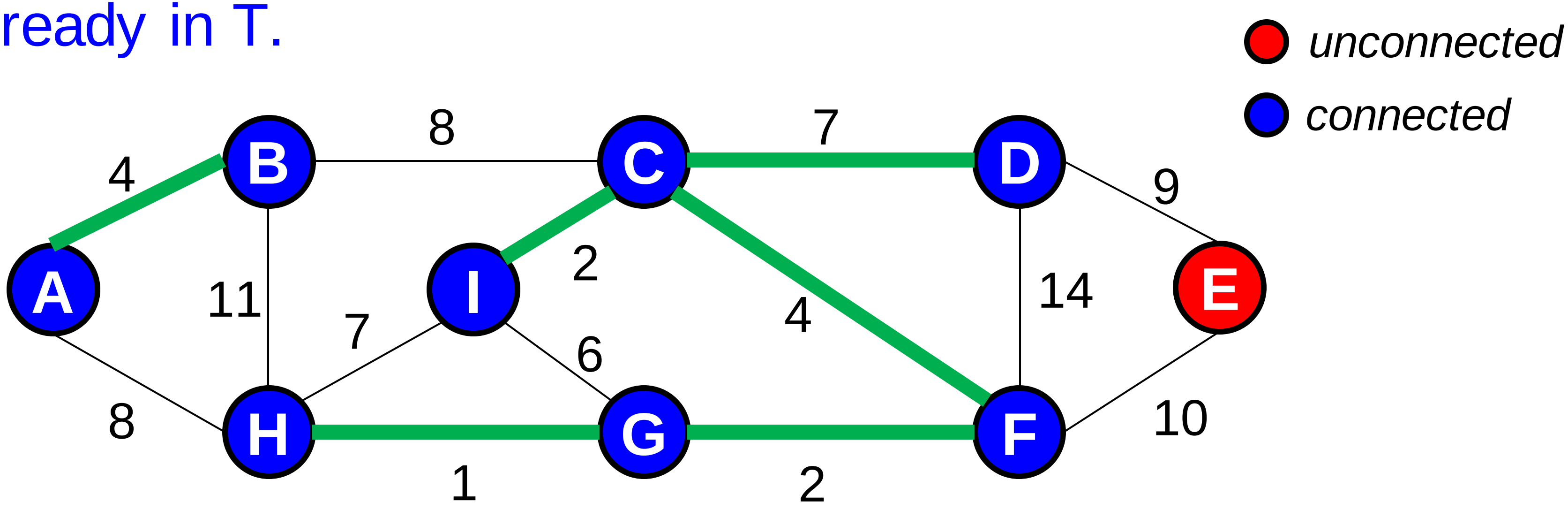
- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



Edge I-G forms a cycle, so we can't pick it.

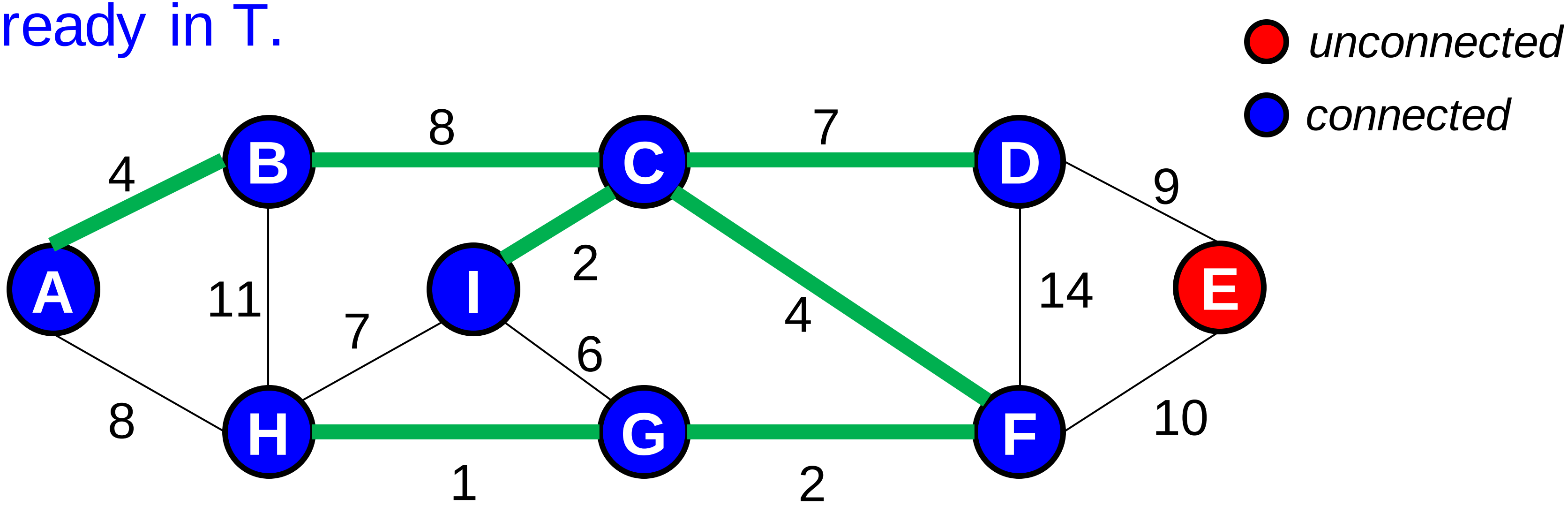
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



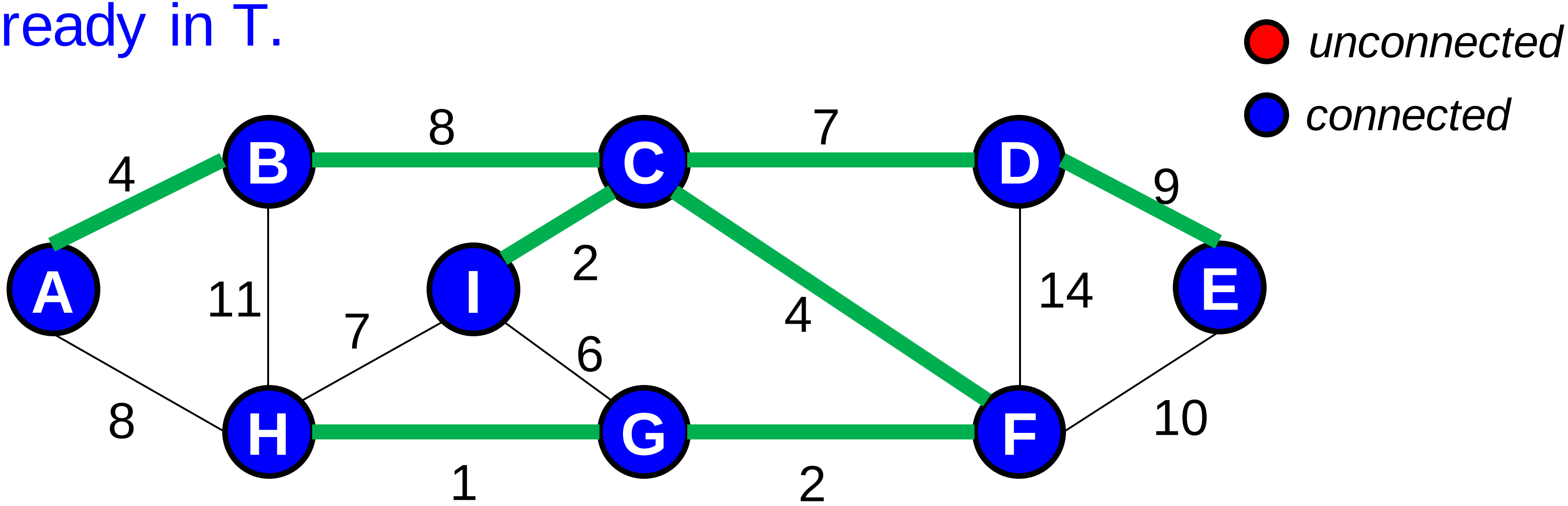
Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



Another Greedy Criterion for MST

- **Kruskal's criterion**: add to T the edge e of minimum w(e) that does not form a cycle when combined with edges already in T.



A Few More Words on Greedy Algorithms

- Greedy choice is a *design principle* for algorithms.
- Many different problems can be solved using it.
- **Does it always work?**
- *Tune in to Studio 14 to find out!*

Course wrap-up: what to do next?

- Take more CSE classes (no matter your degree program)
- Join the WashU chapter of the ACM (Association for Computing Machinery)
  - Programming competitions, tech talks, course registration discussions, social events…
- Apply to be a TA (look for e-mail about "TA draft")
- Be an active, CSE-literate member of society

Course wrap-up: thank you!

- Getting to know you as CSE thinkers and as people has been a pleasure
- We've seen you work hard, grow intellectually, work together in studio, graciously help each other and us
- We look forward to seeing you around the department and having you as CSE colleagues
- All the best!

Thank you for a great semester!