CSE332: Data Abstractions
Lecture 22: Minimum Spanning Trees

James Fogarty
Winter 2012
Making Connections

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-5
4-2
1-6
5-7
4-8
3-7

Q: Are nodes 2 and 4 connected? Indirectly?
Q: How about nodes 3 and 8?
Q: Are any of the paired connections redundant due to indirect connections?
Q: How many sub-networks do you have?
Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

Start: \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\}

3-5 \{1\} \{2\} \{3, 5\} \{4\} \{6\} \{7\} \{8\} \{9\}

4-2 \{1\} \{2, 4\} \{3, 5\} \{6\} \{7\} \{8\} \{9\}

1-6 \{1, 6\} \{2, 4\} \{3, 5\} \{7\} \{8\} \{9\}

5-7 \{1, 6\} \{2, 4\} \{3, 5, 7\} \{8\} \{9\}

4-8 \{1, 6\} \{2, 4, 8\} \{3, 5, 7\} \{9\}

3-7

Q: Are nodes 2 and 4 connected? Indirectly?
Q: How about nodes 3 and 8?
Q: Are any of the paired connections redundant due to indirect connections?
Q: How many sub-networks do you have?
Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
  - **Union(5,1)**
    - Result: \{3,5,7,1,6\}, \{4,2,8\}, \{9\}
    - To perform the union operation, we replace sets x and y by \((x \cup y)\)

- **Find(x)** – return the name of the set containing x.
  - Given sets: \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do Union in constant time.
- We can get Find to be *amortized* constant time (worst case \(O(\log n)\) for an individual Find operation).
Cute Application

- Build a random maze by erasing edges.
Cute Application

- Pick Start and End
Cute Application

- Repeatedly pick random edges to delete.
Number the Cells

Disjoint sets $S = \{ \{1\}, \{2\}, \{3\}, \{4\}, \ldots \{36\} \}$, each cell is unto itself. We have all edges $W = \{(1,2), (1,7), (2,8), (2,3), \ldots \}$ 60 walls total.

<table>
<thead>
<tr>
<th>Start</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>
Maze Building with Disjoint Union/Find

- Algorithm sketch:
  - Choose wall at random.
    - Boundary walls are not in wall list, because we cannot delete them
  - Erase wall if the neighbors are in disjoint sets
    - Avoids cycles
  - Take union of those sets
  - Repeat until there is only one set
    - Every cell reachable from every other cell
A Hidden Tree
Up-Tree for Disjoint Union/Find

Initial State

1  2  3  4  5  6  7

Intermediate state

1  3  7

2  5  4

6

Roots are the names of each set
Find Operation

- Find(x):
  follow x to the root and return the root

Find(6) = 7
Union Operation

- Union(i,j):
  assuming i and j roots, point i to j.
Simple Implementation

- Array of indices

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Up[x] = 0 means x is a root
Weighted Union

• Weighted Union
  – Instead of arbitrarily joining two roots, always point the smaller root to the larger root

Union(1,7)
Elegant Array Implementation

Up weight

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>up weight</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Path Compression

• On a Find operation point all the nodes on the search path directly to the root.

Before:

```plaintext
    1
   /|
  2 5
 /|
3 6
```

After Find(3):

```plaintext
    1
   /|
  2 5
 /|
3 6
```

```plaintext
    3
   |
  6
```

```plaintext
    8
   |
  9
```

```plaintext
    7
```

```plaintext
    4
```

```plaintext
    10
```

```plaintext
    1
   |
  2
```

```plaintext
    3
   |
  6
```

```plaintext
    5
   |
  4
```

```plaintext
    10
```

```plaintext
    8
   |
  9
```
Analyzing Disjoint Sets

• For $n$ elements, total cost of $m$ finds, at most $n-1$ unions

• Total work is: $O(m+n)$, i.e. $O(1)$ amortized
  – With $O(1)$ worst-case for union
  – And $O(\log n)$ worst-case for find

• Find and union cannot both be worst-case $O(1)$
Spanning Trees

- A simple problem: Given a connected graph $G=(V,E)$, find a minimal subset of the edges such that the graph is still connected
  - A graph $G_2=(V,E_2)$ such that $G_2$ is connected and removing any edge from $E_2$ makes $G_2$ disconnected
Observations

1. Any solution to this problem is a tree
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected

4. A tree with $|V|$ nodes has $|V|-1$ edges
   - Every spanning tree solution has $|V|-1$ edges
Motivation

A spanning tree connects all the nodes with as few edges as possible

- Example: A “phone tree” so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

In most compelling uses, we have a weighted undirected graph and we want a tree of least total cost

- Example: Electrical wiring for a house or clock wires on a chip
- Example: Road network if you cared about asphalt cost

This is the minimum spanning tree problem

- Will do that next, after intuition from the simpler case
Two Approaches

Different algorithmic approaches to the spanning-tree problem:
1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree
2. Iterate through edges; add to output any edge that doesn’t create a cycle
Spanning Tree via DFS

```java
spanning_tree(Graph G) {
    for each node i: i.marked = false
    for some node i: f(i)
}
f(Node i) {
    i.marked = true
    for each j adjacent to i:
        if(!j.marked) {
            add(i,j) to output
            f(j) // DFS
        }
}
```

Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|E|)$
Example

Stack

Output:
Example

Stack
f(1)
f(2)

Output: (1,2)
**Example**

**Stack**
- $f(1)$
- $f(2)$
- $f(7)$

Output: $(1,2), (2,7)$
Example

Stack
f(1)
f(2)
f(7)
f(5)

Output: (1,2), (2,7), (7,5)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)

Output: (1,2), (2,7), (7,5), (5,4)
Example

Stack
\begin{align*}
  f(1) \\
  f(2) \\
  f(7) \\
  f(5) \\
  f(4) \\
  f(3)
\end{align*}

Output: (1,2), (2,7), (7,5), (5,4), (4,3)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4)  f(6)
f(3)

Output:  (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)
Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4) f(6)
f(3)

Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)
Second Approach

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):
- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

Efficiency:
- Depends on how quickly you can detect cycles
- Reconsider after the example
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output:
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6),
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5)
Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Can stop once we have |V| - 1 edges.
Cycle Detection

• To decide if an edge could form a cycle is $O(|V|)$ because we may need to traverse all edges already in the output

• So overall algorithm would be $O(|V||E|)$

• But there is a faster way using the disjoint-set ADT
  – Initially, each item is in its own 1-element set
  – $\text{find}(u)$: what set contains $u$?
  – $\text{union}(u,v)$: union (combine) the sets containing $u$ and $v$
Aside: Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: \{3,5,7\}, \{4,2,8\}, \{9\}, \{1,6\}
  - **Union(5,1)**
    Result: \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
    To perform the union operation, we replace sets x and y by \((x \cup y)\)
  
- **Find(x)** – return the name of the set containing x.
  - Given sets: \{3,5,7,1,6\}, \{4,2,8\}, \{9\},
  - **Find(1)** returns 5
  - **Find(4)** returns 8

- We can do Union in constant time.
- We can get Find to be *amortized* constant time
  (worst case \(O(\log n)\) for an individual Find operation).
Using Disjoint-Set

Can use a disjoint-set implementation in our spanning-tree algorithm to detect cycles:

Invariant: \( u \) and \( v \) are connected in output-so-far
iff
\( u \) and \( v \) in the same set

- Initially, each node is in its own set
- When processing edge \((u,v)\):
  - If \( \text{find}(u) == \text{find}(v) \), then do not add the edge
  - Else add the edge and \( \text{union}(u,v) \)
Summary so Far

The spanning-tree problem
- Add nodes to partial tree approach is $O(|E|)$
- Add acyclic edges approach is $O(|E| \log |V|)$
  - Using the disjoint-set ADT “as a black box”

But really want to solve the minimum-spanning-tree problem
- Given a weighted undirected graph,
  give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be $O(|E| \log |V|)$
Getting to the Point

Algorithm #1

Shortest-path is to Dijkstra’s Algorithm as
Minimum Spanning Tree is to Prim’s Algorithm
(Both based on expanding cloud of known vertices, basically using a priority queue instead of a DFS stack)

Algorithm #2

Kruskal’s Algorithm for Minimum Spanning Tree is
Exactly our forest-merging approach to spanning tree but process edges in cost order
Prim’s Algorithm Idea

Idea: Grow a tree by adding an edge from the “known” vertices to the “unknown” vertices. *Pick the edge with the smallest weight that connects “known” to “unknown.”*

Recall Dijkstra picked “edge with closest known distance to source.”
- But that is not what we want here
- Otherwise identical
- Feel free to look back and compare
The Algorithm

1. For each node \( v \), set \( v.\text{cost} = \infty \) and \( v.\text{known} = \text{false} \)

2. Choose any node \( v \).
   a) Mark \( v \) as known
   b) For each edge \((v,u)\) with weight \( w \),
      set \( u.\text{cost} = w \) and \( u.\text{prev} = v \)

3. While there are unknown nodes in the graph
   a) Select the unknown node \( v \) with lowest cost
   b) Mark \( v \) as known and add \((v, v.\text{prev})\) to output
   c) For each edge \((v,u)\) with weight \( w \),
      \[
      \text{if}(w < u.\text{cost}) \{
      \begin{align*}
      u.\text{cost} &= w; \\
      u.\text{prev} &= v;
      \end{align*}
      \}
      \]
Example

![Graph](image)

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```
vertex  | known? | cost | prev
------|-------|-----|-----
A      | Y     | 0   |     
B      |       | 2   | A   
C      |       | 2   | A   
D      |       | 1   | A   
E      |       | ??  |     
F      |       | ??  |     
G      |       | ??  |     
```
Example

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>6</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
Example

vertex | known? | cost | prev |
-------|--------|------|------|
A      | Y      | 0    |      |
B      |        | 1    | E    |
C      | Y      | 1    | D    |
D      | Y      | 1    | A    |
E      | Y      | 1    | D    |
F      |        | 2    | C    |
G      |        | 3    | E    |
Example

A

B

C

D

E

F

G

vertex | known? | cost | prev
--- | --- | --- | ---
A | Y | 0 | |
B | Y | 1 | E
C | Y | 1 | D
D | Y | 1 | A
E | Y | 1 | D
F | | 2 | C
G | | 3 | E
Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
Example

A
B
C
D
E
F
G

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
Analysis

- Correctness
  - Intuitively similar to Dijkstra

- Run-time
  - Same as Dijkstra
  - $O(|E| \log |V|)$ using a priority queue
**Kruskal’s Algorithm**

Idea: Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.

- But now consider the edges in order by weight

So:

- Sort edges: $O(|E| \log |E|) = O(|E| \log |V|)$
- Iterate through edges using union-find for cycle detection $O(|E| \log |V|)$

Somewhat better:

- Floyd’s algorithm to build min-heap with edges $O(|E|)$
- Iterate through edges using union-find for cycle detection and `deleteMin` to get next edge $O(|E| \log |V|)$
- Not better worst-case asymptotically, but often stop long before considering all edges
Pseudocode

1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size < |V|-1
   - Consider next smallest edge \((u,v)\)
   - if \(\text{find}(u,v)\) indicates \(u\) and \(v\) are in different sets
     • output \((u,v)\)
     • \(\text{union}(u,v)\)

Recall invariant:
\(u\) and \(v\) in same set if and only if connected in output-so-far
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)

Output:

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1:  (A,D), (C,D), (B,E), (D,E)
2:  (A,B), (C,F), (A,C)
3:  (E,G)
5:  (D,G), (B,D)
6:  (D,F)
10: (F,G)

Sets: (A, D) (B) (C) (E) (F) (G)

Output: (A,D)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1:  (A,D), (C,D), (B,E), (D,E)
2:  (A,B), (C,F), (A,C)
3:  (E,G)
5:  (D,G), (B,D)
6:  (D,F)
10: (F,G)

Sets: (A, C, D) (B) (E) (F) (G)

Output: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest
**Example**

Edges in sorted order:
1:  (A,D), (C,D), (B,E), (D,E)
2:  (A,B), (C,F), (A,C)
3:  (E,G)
5:  (D,G), (B,D)
6:  (D,F)
10: (F,G)

Sets:  (A, C, D) (B, E) (F) (G)

Output: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A, B, C, D, E) (F) (G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A, B, C, D, E) (F) (G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A, B, C, D, E, F) (G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A, B, C, D, E, F) (G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Sets: (A, B, C, D, E, F, G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest
Analysis

• Correctness
  – That it is a spanning tree
    • When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
    • The graph is connected, we consider all edges
  – That it is minimum
    • By induction
      • At every step, the output is a subset of a minimum tree

• Run-time
  – $O(|E| \log |V|)$