Recitation 10 - Minimum Spanning Trees

Sebastian Laudenschlager

sebastian.laudenschlager@colorado.edu

April 6, 2018

MST: a graph problem

- Consider an undirected graph with weighted edges.
- Goal: Find $T \subseteq E$ such that
  - T connects all vertices (T is a spanning tree).
  - $w(T) = \sum_{(u,v)\in T} w(u,v)$ is minimized.
- A tree T that satisfies these two conditions is called a Minimum Spanning Tree (MST).
- An MST has exactly $|V| - 1$ edges.
- An MST has no cycles.
- An MST may not be unique.

Basic Idea

- Basic idea of the algorithms is as follows:
  - Let A be the ultimate MST, consisting of a set of edges.
  - Initialize A to contain no edges.
  - Loop over graph while A is not a spanning tree, and find a "safe" edge to add to A.
- The algorithms to accomplish this (we'll go over two), are greedy by nature.

“Safe” edge?

- So how do we determine what a safe edge is?
- If A is some subset of an MST, an edge (u, v) is “safe” to add to A if and only if $A \cup \{(u, v)\}$ is also a subset of an MST.
- So we only add edges that maintain the property of being a subset of an MST.

Kruskal

- First MST algorithm we look at is called Kruskal's algorithm.
- It roughly works as follows:
  - Each vertex starts out as being its own component.
  - Repeatedly merges two components into one by choosing the light edge crossing the cut between them.
  - Examine the edges in increasing order of weight.

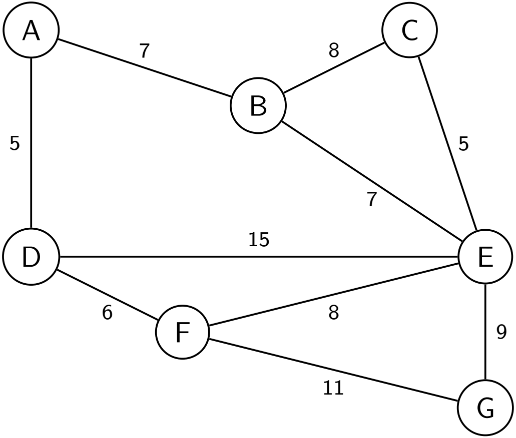
Pseudocode

```
def kruskal(G, w):
    A = {}
    for each vertex v in G.V:
        makeSet(v)
    sort edges in G.E in nondecreasing order by weight
    for each (u,v) from sorted list:
        if findSet(u) != findSet(v):
            A = A + {(u,v)}
            union(u, v)
    return A
```

Analysis

- First for loop sets each vertex to be its own component.
- Sort the edges by weight.
- Scan through the edges in order.
- Check if the edge is safe to add.
  - Check whether two vertices are in the same tree by using `findSet`.
- If it is, merge the components that the edge connects and add the edge to A.
- This algorithm uses a disjoint set data structure to implement the `makeSet` and `findSet` operations.
- Check out Ch. 21 of CLRS for more info.

Example

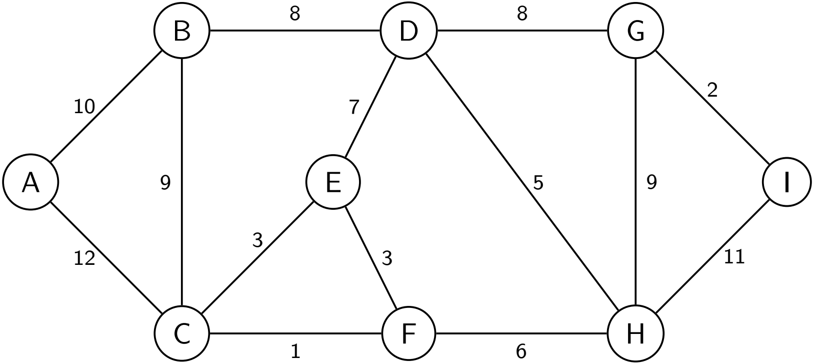


- Use Kruskal's algorithm to find a MST.

Example, cont.

- Add edges to MST in order of weight; break ties arbitrarily.
- (A, D): safe $\rightarrow A = \{(A, D)\}$
- (C, E): safe $\rightarrow A = \{(A, D), (C, E)\}$
- (D, F): safe $\rightarrow A = \{(A, D), (C, E), (D, F)\}$
- (B, E): safe $\rightarrow A = \{(A, D), (C, E), (D, F), (B, E)\}$
- (A, B): safe $\rightarrow A = \{(A, D), (C, E), (D, F), (B, E), (A, B)\}$
- (B, C): reject: `findSet(B)` $=$ `findSet(C)`
- (F, E): reject: `findSet(F)` $=$ `findSet(E)`
- (E, G): safe $\rightarrow A = \{(A, D), (C, E), (D, F), (B, E), (A, B), (E, G)\}$

Your turn



- Find the MST that Kruskal's algorithm would produce.

Prim's Algorithm

- It works roughly as follows:
  - Build a single tree A.
  - Start from some arbitrary root.
  - At each step, find a light edge crossing the cut $(V_A, V - V_A)$, where V_A is the set of vertices that A is incident on. In other words, find a light edge that connects A to an isolated vertex.
  - Add this edge to A.
  - This edge is safe to add for the same reason an edge in Kruskal's algorithm is safe to add.

Details

- How do we find such a light edge efficiently?
- Use a min-priority queue (min-heap).
- During execution, all vertices not in A reside in a min-priority queue based on a key attribute.
- For each vertex v, the attribute $v.key$ is the minimum weight edge connecting the vertex to A.

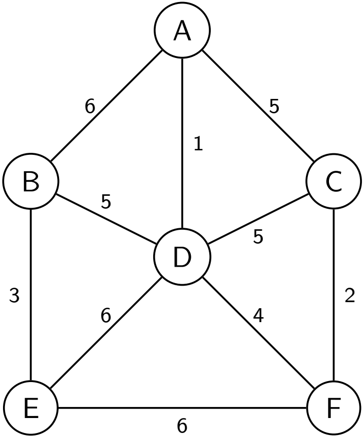
Pseudocode

```
def prim(G, w, r):
    for each u in G.V:
        u.key = Inf
        u.parent = NIL
    r.key = 0
    Q = G.V
    while Q not empty:
        u = extractMin(Q)
        for each v in G.adj[u]:
            if v in Q and w(u, v) < v.key:
                v.parent = u
                v.key = w(u, v)
```

How does it find MST?

- Each vertex v knows its parent in the tree by its attribute $v.parent$.
- So in the end, all vertices are in the tree (implicitly), and we know the parent of each one, thus giving us the MST.

Example



- Use Prim's algorithm to find a MST.

Example, cont.

- At the start, $Q = \{A, B, C, D, E, F\}$. Start at A:
- `extractMin(A)` $\Rightarrow Q = \{B, C, D, E, F\}$.
  - `D.parent = A, D.key = 1`
  - `B.parent = A, B.key = 6`
  - `C.parent = A, C.key = 5`
- `extractMin(D)` $\Rightarrow Q = \{B, C, E, F\}$.
  - `B.parent = D, B.key = 5`
  - `F.parent = D, F.key = 4`
  - `E.parent = D, E.key = 6`
- `extractMin(F)` $\Rightarrow Q = \{B, C, E\}$.
  - `C.parent = F, C.key = 2`
- `extractMin(C)` $\Rightarrow Q = \{B, E\}$.
- `extractMin(B)` $\Rightarrow Q = \{E\}$.
  - `E.parent = B, E.key = 3`
- `extractMin(E)` $\Rightarrow Q = \{\}$.

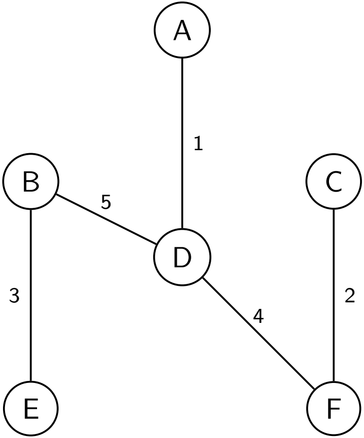
Example result

- Implicitly, we now have our MST. To see this:

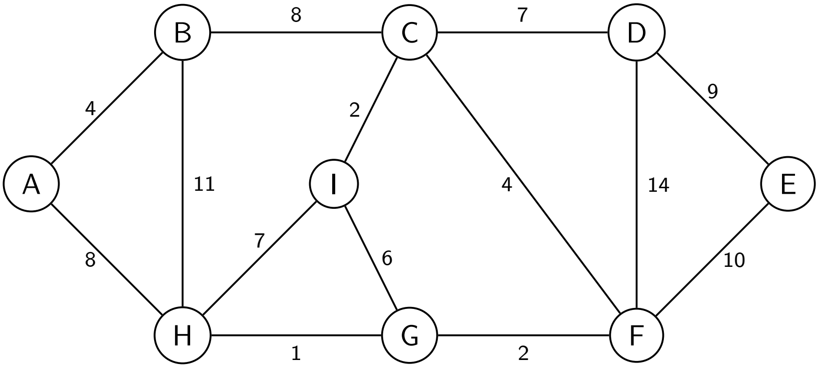
-	A	B	C	D	E	F
parent	-	D	F	A	B	D
key	0	5	2	1	3	4

Example result, cont.

- This leads to the following MST:



Your turn



- Find the MST that Prim's would (implicitly) produce.