Computer Science & Engineering 423/823 Design and Analysis of Algorithms

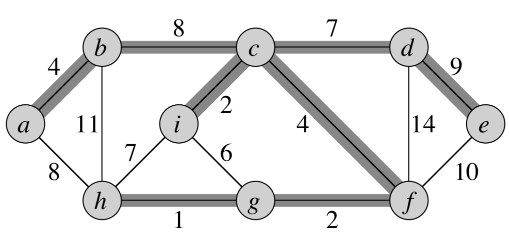
Lecture 04 — Minimum-Weight Spanning Trees (Chapter 23)

Stephen Scott
(Adapted from Vinodchandran N. Variyam)

Introduction

- Given a connected, undirected graph $G = (V, E)$, a **spanning tree** is an acyclic subset $T \subseteq E$ that connects all vertices in V
  - T acyclic $\Rightarrow$ a tree
  - T connects all vertices $\Rightarrow$ **spans** G
- If G is weighted, then T's weight is $w(T) = \sum_{(u,v)\in T} w(u, v)$
- A **minimum weight spanning tree** (or **minimum spanning tree**, or MST) is a spanning tree of minimum weight
  - Not necessarily unique
- Applications: anything where one needs to connect all nodes with minimum cost, e.g. wires on a circuit board or fiber cable in a network

MST Example



Kruskal's Algorithm

- Greedy algorithm: Make the locally best choice at each step
- Starts by declaring each vertex to be its own tree (so all nodes together make a forest)
- Iteratively identify the minimum-weight edge (u, v) that connects two distinct trees, and add it to the MST T, merging u's tree with v's tree

MST-Kruskal(G, w)

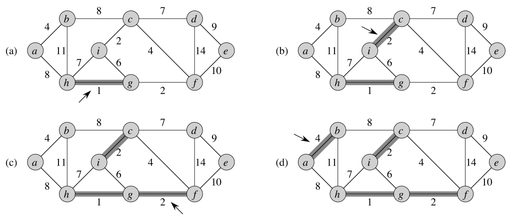
```
A = ∅ ;
for each vertex v ∈ V do
    MAKE-SET(v) ;
end
sort edges in E into nondecreasing order by weight w ;
for each edge (u, v) ∈ E, taken in nondecreasing order do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A = A ∪ {(u, v)} ;
        UNION(u, v) ;

end
return A
```


MST-Kruskal(G, w), Part 2

- FIND-SET(u) returns a representative element from the set (tree) that contains u
- UNION(u, v) combines u's tree to v's tree
- These functions are based on the **disjoint-set data structure**
- More on this later

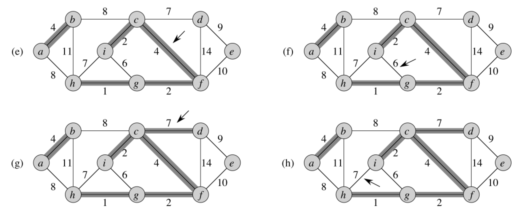
Example (1)

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm



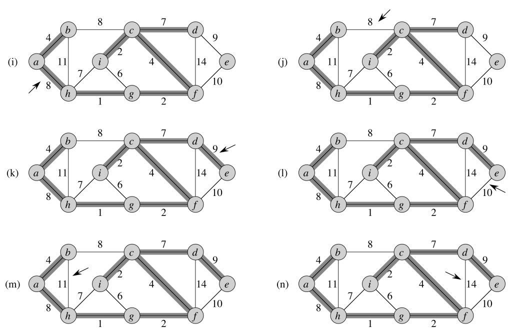
Example (2)

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm



Example (3)

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm



Disjoint-Set Data Structure

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm

- Given a **universe** $U = \{x_1, \ldots, x_n\}$ of elements (e.g. the vertices in a graph G), a DSDS maintains a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint sets of elements such that
  - Each element x_i is in exactly one set S_j
  - No set S_j is empty
- Membership in sets is dynamic (changes as program progresses)
- Each set $S \in \mathcal{S}$ has a **representative element** $x \in S$
- Chapter 21

Disjoint-Set Data Structure (2)

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm

- DSDS implementations support the following functions:
  - $\textsc{Make-Set}(x)$ takes element x and creates new set $\{x\}$; returns pointer to x as set's representative
  - $\textsc{Union}(x, y)$ takes x's set (S_x) and y's set (S_y, assumed disjoint from S_x), merges them, destroys S_x and S_y, and returns representative for new set from $S_x \cup S_y$
  - $\textsc{Find-Set}(x)$ returns a pointer to the representative of the unique set that contains x
- Section 21.3: can perform d D-S operations on e elements in time $O(d\,\alpha(e))$, where $\alpha(e) = o(\lg^* e) = o(\log e)$ is *very* slowly growing:

$$\alpha(e) = \begin{cases} 0 & \text{if } 0 \le e \le 2 \\ 1 & \text{if } e = 3 \\ 2 & \text{if } 4 \le e \le 7 \\ 3 & \text{if } 8 \le e \le 2047 \\ 4 & \text{if } 2048 \le e \le 16^{512} \end{cases}$$

Analysis of Kruskal's Algorithm

Introduction
Kruskal's Algorithm
Introduction
The Algorithm
Example
Disjoint-Set Data Structure
Analysis
Prim's Algorithm

- Sorting edges takes time $O(|E| \log |E|)$
- Number of disjoint-set operations is $O(|V| + |E|)$ on $O(|V|)$ elements, which can be done in time $O((|V| + |E|)\,\alpha(|V|)) = O(|E|\,\alpha(|V|))$ since $|E| \ge |V| - 1$
- Since $\alpha(|V|) = o(\log |V|) = O(\log |E|)$, we get total time of $O(|E| \log |E|) = O(|E| \log |V|)$ since $\log |E| = O(\log |V|)$

Prim's Algorithm

- Greedy algorithm, like Kruskal's
- In contrast to Kruskal's, Prim's algorithm maintains a single tree rather than a forest
- Starts with an arbitrary tree root r
- Repeatedly finds a minimum-weight edge that is incident to a node not yet in tree

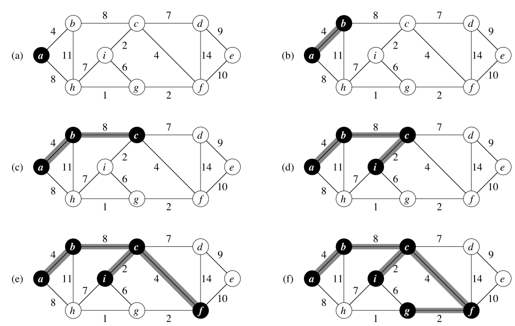
MST-Prim(G, w, r)

```
A = ∅ ;
for each vertex v ∈ V do
    key[v] = ∞ ;
    π[v] = NIL ;
end
key[r] = 0 ;
Q = V ;
while Q ≠ ∅ do
    u = EXTRACT-MIN(Q) ;
    for each v ∈ Adj[u] do
        if v ∈ Q and w(u, v) < key[v] then
            π[v] = u ;
            key[v] = w(u, v) ;

    end
end
```

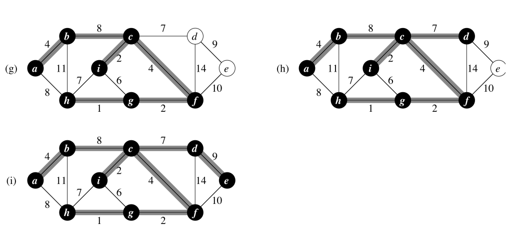

MST-Prim(G, w, r), Part 2

- $key[v]$ is the weight of the minimum weight edge from v to any node already in MST
- Extract-Min uses a **minimum heap** (minimum priority queue) data structure
  - Binary tree where the key at each node is $\leq$ keys of its children
  - Thus minimum value always at top
  - Any subtree is also a heap
  - Height of tree is $\lfloor \lg n \rfloor$
  - Can build heap on n elements in $O(n)$ time
  - After returning the minimum, can filter new minimum to top in time $O(\log n)$
  - Based on Chapter 6

Example (1)



Example (2)



Analysis of Prim's Algorithm

- **Invariant:** Prior to each iteration of the while loop:
  1. Nodes already in MST are exactly those in $V \setminus Q$
  2. For all vertices $v \in Q$, if $\pi[v] \neq \text{NIL}$, then $key[v] < \infty$ and $key[v]$ is the weight of the lightest edge that connects v to a node already in the tree
- Time complexity:
  - Building heap takes time $O(|V|)$
  - Make $|V|$ calls to Extract-Min, each taking time $O(\log |V|)$
  - For loop iterates $O(|E|)$ times
    - In for loop, need constant time to check for queue membership and $O(\log |V|)$ time for decreasing v's key and updating heap
  - Yields total time of $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$
  - Can decrease total time to $O(|E| + |V| \log |V|)$ using Fibonacci heaps