Mat 3770

Spring 2014

Week 9 — Student Responsibilities

- Reading: Chapter 3.3–3.4 (Tucker), 10.4–10.5 (Rosen)
- Homework

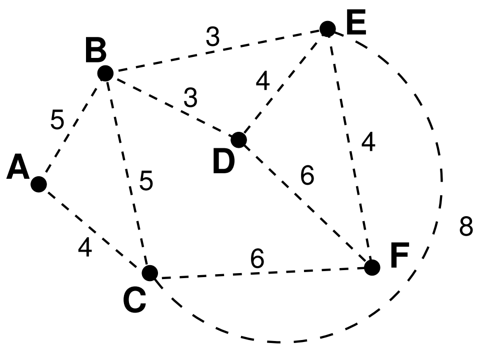
Due date	Tucker	Rosen
3/21	3.2	10.3
3/21	DFS & BFS	Worksheets
3/26	3.3	10.4, 10.5
3/28	Heapify	worksheet

- Attendance **Truly, Madly, Deeply** Encouraged

3.3 Spanning Trees

- A **spanning tree** of a graph G is a subgraph of G that is a tree containing all vertices of G.
- A **minimal spanning tree** is a spanning tree whose sum of the edge weights (lengths) is as small as possible.
- **Problem Statement**: Given a graph G = (V, E) with positive edge weights (cost: E $\to \Re^+$), find the cheapest connected spanning subgraph H of G.
- **Note**: If H = (V, E_H), then cost(H) $= \sum_{e \in E(H)}$ cost(e), i.e., cost of subgraph is sum of costs of edges in subgraph.

Example: Minimal Spanning Tree



Observations

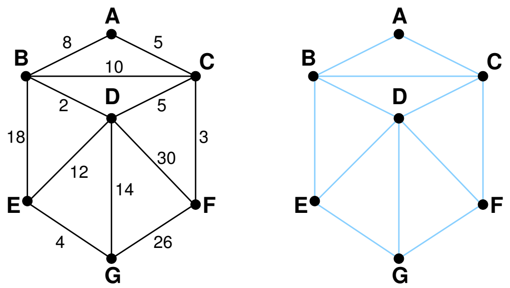
- H must be a tree (if H exists). Why?
  1. must span and be connected
  2. if cycle, then extra edge with positive weight, which could be removed to reduce cost
- If G has n vertices ($|V| =$ n), then **any** minimal spanning tree of G has $N - 1$ edges.
- A graph with no cycles is called a **forest**
- A connected forest is called a **tree**

Prim's Minimal Spanning Tree

Idea: "Grow" a Tree

1. Pick an arbitrary vertex in the graph, place in V_H
2. From among the edges going from V_H to vertices not in V_H, choose a cheapest one, say edge e to vertex x
3. Add vertex x to V_H and e to E_H
4. Repeat process from step 2 until no more vertices remain to be added to V_H, which is equivalent to saying $|E_H| = |V_G| - 1$

Prim's Minimal Spanning Tree, Start at **A**



Prim's MST Algorithm

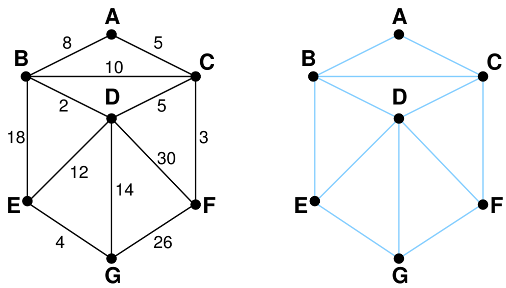
```
Procedure Prim (G): H
// PRE:  G is connected
// POST: H is an MST of G

begin
  pick an arbitrary vertex x in the vertices of G,
     and add it to VH, the vertices in H
  from among the edges incident to x, select the
     cheapest and add it to EH, the edges in H
  while |EH| <  |VG| - 1
     find the cheapest edge <a, b>
          where a is in VH, and b is in VG - VH
     add <a, b> to EH, add b to VH
end
```


Kruskal's MST

Idea: connect forests until one tree

1. Place the vertices of V_G into $|V_G| = n$ individual subtrees
2. Find the minimum cost edge, $e \in E_G$, which doesn't cause a cycle in the spanning forest
3. Add e to E_H, joining two of the subtrees
4. repeat process from step 2 until a single spanning tree exists, which is equivalent to saying $|E_H| = |V_G| - 1$

Kruskal's MST



Edge weights: 2, 3, 4, 5, 5, 8, 10, 12, 14, 18, 26, and 30

Kruskal's MST Algorithm

```
Procedure Kruskal (G): H
// PRE:  G is connected
// POST: H is an MST of G

begin
  put n vertices into n singleton trees
  H = { }

  // repeat until tree has n-1 edges
  while |EH| < |VG| - 1
     a) find the min_cost_edge e in EG
     b) if H remains a forest when e is added
           add e to VH
     c) delete e from EG
end
```


Implementing Kruskal's Algorithm

- We need to be able to (**quickly**) find the **next cheapest edge**
  - Using a **min-heap** vs **sorted list**
  - Heap is better since not every edge may be examined / removed
  - But, what's a heap?

Priority Queues and Heaps

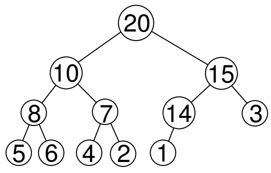
- A **Priority Queue** is a **data structure** supporting the operations:
  1. **insert()** and
  2. **removeMin()** (or **removeMax()**, depending upon the problem)
- One way to implement priority queues is with a **heap**
- A **heap** is an **essentially complete** binary tree, i.e.:
  1. all levels are full except possibly bottom level (leaves)
  2. all bottom nodes are in left–most positions.

Heap Implementation

Heaps can be efficiently implemented with arrays, which form an implicit (vs explicit) representation of a tree. For example:

i =	1	2	3	4	5	6	7	8	9	10	11	12
L	20	10	15	8	7	14	3	5	6	4	2	1

Where Children of L[i] are in positions 2*i and (2*i)+1.



The Min–heap Property

An array L[k..n] has the **Min–heap property** if

$$\forall\, i \ni k \le i < \frac{n}{2}: \quad L[i] \le L[2i] \;\; and \;\; L[i] \le L[2i+1]$$

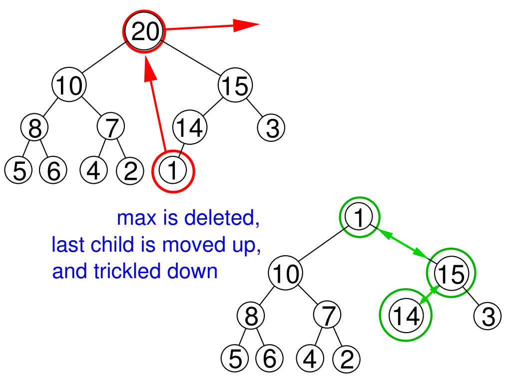
if n is even, then $L[\frac{n}{2}] \le L[n]$

In other words: **Parents are smaller than their children**, or child in the case n is even.

`removeMin() (Max)` Algorithm

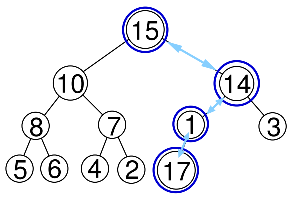
- Send out the first value in heap as min (max)
- Put last value (x) of heap in position 1: $L[1] = L[size]$
- Decrement heap size
- Trickle–down x through heap by swapping it with the smaller (larger) child until smaller (larger) child is larger (smaller) than x.

Example — `removeMax()`



`Insert()` Algorithm

- Increment heap size
- Put new value (x) in L[size]
- While x is smaller (bigger) than its parent, swap them (aka percolate– or bubble–up)



Complexity of Heap Operations

- Observation: if a heap of height h has n nodes, then

$$2^h \leq n \leq 2^{h+1}$$

  one leaf at level h versus a complete tree
- Take the log of each part of the inequality:

$$h \leq \log n \leq h+1$$

- Subtracting 1, we find:

$$\log n - 1 \leq h \leq \log n$$

- Hence, to traverse the heap from root to leaf, or in reverse, takes $O(\log n)$ time.
- Thus, both **Insert()** and **Delete()** take $O(\log n)$ time.

What if we merely kept an **unordered** list?

- Delete: find, delete item, and fill in
  - array: $O(n) + O(1) + O(1) = O(n)$
  - linked list: $O(n) + O(1) = O(n)$
- Insert: array / linked list: $O(1)$

An **ordered** list?

- Delete: find, delete item, and fill in
  - array: $O(1) + O(1) + O(n) = O(n)$
  - linked list: $O(1) + O(1) = O(1)$
- Insert: find position, insert (move)
  - array: $O(\log n) + O(n) = O(n)$
  - linked list: $O(n) + O(1) = O(n)$

An Aside: Heapsort

- An array, L, can be sorted as follows:
  1. Turn L[1..n] into a heap (aka **heapify**)
  2. `Remove()` n times, storing the removed (min or max) value at the end of the heap, then decrement size of heap
- How fast is this sort?
  - Step 2 takes time:

$$\log(n) + \log(n-1) + \cdots + \log(1) = \sum_{i=1..n} \log i \in O(n \log n)$$

  - Step 1? It depends ...

Heapifying — **Method 1: Top–down**

```
for i = 2 to n
   BubbleUp(L[1..i])
   // invariant:  L[1..i] is a heap
```

(Max)–Heapify: 1, 2, 3, 4, 5, 6, 7, 8, 10, 14, 15, 20

Method 1 Complexity Analysis

- Complexity? All nodes at depth j take j swaps in worst case, so:

$$T(n) = \sum_{j=0..h} j \times \textit{number of nodes at depth } j$$

- We have at most 2^j nodes at depth j, so:

$$T(n) \leq \sum_{j=0..h} (j \times 2^j) = (h-1)2^{h+1} + 2$$

- We know $h \leq \log n$, so:

$$\begin{aligned} T(n) &\leq (\log(n-1))2^{\log n + 1} + 2 \\ &\leq \log n(n) + 2 \\ &\leq n \log n + 2 \\ &\in O(n \log n) \end{aligned}$$

Can We Create Heaps Any Faster?

- **Method 1**: **Top–down** moves many (about $\frac{n}{2}$) elements by $\log n$ in the worst case (if already in order, all $\frac{n}{2}$ last inserts must percolate to the top!)
- Consider **Method 2**: **Bottom–up**, where we assume the leaves are in place and use sift–down on the top $\frac{n}{2}$ elements.

  This moves fewer elements by the height of the tree.

Heapifying — **Method 2: Bottom–up**

```
// PRE:  L[n/2 + 1 .. n] already a heap
for i = n/2 downto 1
   SiftDown(L[i..n])
   // invariant:  L[i..n] satisfies the
   //             heap property
```

(Max)–Heapify: 1, 2, 3, 4, 5, 6, 7, 8, 10, 14, 15, 20

Method 2 Complexity Analysis

- We have the recurrence relation:

$$T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + \log n$$

- We brilliantly guess that

$$\begin{aligned} T(n) &= O(n - \log n), \quad so \\ T(n) &\leq cn - d \log n \end{aligned}$$

- Thus

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + \log n \\ &\leq 2(c(\frac{n}{2}) - d \log \frac{n}{2}) + \log n \\ &= cn - 2d \log \frac{n}{2} + \log n \\ &= cn - 2d(\log n - \log 2) + \log n \\ &= cn - 2d \log n + 2d + \log n \\ &= cn - 2d \log n + \log n + 2d \end{aligned}$$

- And we would like:

$$cn - 2d \log n + \log n + 2d \leq cn - d \log n$$

- So we want:

$$\begin{aligned} -d \log n + \log n + 2d &\leq 0 \\ (1-d) \log n + 2d &\leq 0 \\ 2d &\leq -(1-d) \log n \\ 2d &\leq (d-1) \log n \\ \frac{2d}{d-1} &\leq \log n \end{aligned}$$

- Pick $d = \frac{1}{2}$, then

$$\begin{aligned} \frac{2d}{d-1} &= \frac{2(\frac{1}{2})}{(\frac{1}{2}-1)} \\ &= \frac{\frac{2}{2}}{\frac{-1}{2}} \\ &= \frac{1}{(\frac{-1}{2})} \\ &= -2 \end{aligned}$$

- Oops, $d-1$ must be positive ... Pick $d = 2$, then

$$\frac{2(2)}{(2-1)} = \frac{4}{1} = 4, \quad and$$

$$\frac{2d}{(d-1)} \leq \log n, \quad for \quad n \geq 2^4 = 16$$

- We would also need to show that $T(n) \leq cn - d \log n$, which is no problem.

Homework: Max–heapify (both methods):
3, 21, 19, 8, 7, 13, 24, 16, 31, 22, 14, 1, 12, 81, 5

Back to Implementing Kruskal's Algorithm

- We needed to be able to find the next cheapest edge
  - Use a min-heap of edges. (All $|E_G|$ of them.)
  - Fastest heapify? O(n) for n keys, thus it takes O($|E_G|$) time to make the heap
  - In worst case, O($|E_G|$) edges may be removed.
  - Let n = $|V_G|$ and p = $|E_G|$
    Then total heap processing / activity time is:
    $O(p + p \log_2 p)$ for the heapify + $O(p * delete\_min)$ deletes

- Note: since $p \le \binom{n}{2} = \frac{(n^2-n)}{2}, \quad p \in O(n^2)$, so:

$$\begin{aligned} \log p &\in O(\log n^2) \\ &\in O(2 \log n) \\ &\in O(\log n) \end{aligned}$$

Thus, $p \log p \in O(p \log n)$ *or* $O(n^2 \log n)$.

Finishing Up Kruskal's Algorithm

- What else needs to be done? We need to be able to tell whether adding an edge to the subgraph H forms a forest or results in a cycle.
- If not cycle is formed, then we need to be able to **merge** the two trees that the edge connects.
- That is, H represents a set of trees. Given an edge $e = < u, v >$, we need to know if vertices u and v are in the same tree. If not (no cycle), merge the trees in which they're found.

The Union–Find Data Structure

- What we need is a data structure which will hold a collection of **disjoint sets** (vertices in trees), and allow efficient implementation of:
  1. Union(i, j): merge sets i and j
  2. Find(x): determine which set contains x
- Such a data structure is called a **Union–Find** or **Disjoint–Set** data structure.