1.204 Lecture 11

Greedy algorithms:
Minimum spanning trees

Minimum spanning tree

- If G is an undirected, connected graph, a subgraph T of G is a spanning tree iff T is a tree with n nodes (or, equivalently, n-1 arcs)
  - A minimum spanning tree is the spanning tree T of G with minimum arc costs

Figure by MIT OpenCourseWare.
Applications of minimum spanning trees

- Building wiring, mechanicals
- Water, power, gas, CATV, phone, road distribution networks
- Copper (conventional) phone networks
  - MST algorithms not needed, done heuristically
- Wireless telecom networks
  - Cell tower connectivity with microwave ‘circuits’
  - Cost is not a function of distance, but reliability is
  - East-west links preferred to north-south (ice, sun,…)
  - Topography matters: DEM data
  - Move to fiber optics as better technology
  - Problem is to have a cost-effective, reliable network
    - Not to find the minimum spanning tree
- System engineer looks at entire issue
  - MST is one component of a broader solution

Prim’s algorithm

- Greedy method to build minimum spanning tree
  - Start at an arbitrary node (root)
  - The set of arcs selected always form a tree T
    - Initially the tree T is just the root. No arcs added to it yet.
  - The next arc (u,v) to be included in T is:
    - Minimum cost arc such that
    - Both nodes u and v are not in T already
  - Add arc (u,v) and node v to T
    - Mark node v as being in T, or visited (u is already in the tree)
    - T ∪ {(u,v)} is now the new tree T
  - End when all nodes in tree have been visited, or
    - Equivalently, when (n-1) arcs have been put in the spanning tree
Prim’s algorithm example

Stages in Prim’s Algorithm

Figure by MIT OpenCourseWare.

Standard Prim: data members, constructor

```java
public class Prim {  // Assumes connected graph; not checked
    private int nodes;  // Assumes consecutive node numbers
    private int[] head; // Assumes consecutive node numbers
    private int[] to;
    private int[] dist;
    private int[] P;  // Predecessor node back to root
    private boolean[] visited;  // Has node been visited
    private int MSTcost;

    Prim(int n, int[] h, int[] t, int[] d) {  
        nodes = n;  // Or set nodes = head.length - 1
        head = h;
        to = t;
        dist = d;
    }
```
public int prim(int root) {
    P = new int[nodes];  // Predecessor node in MST
    visited = new boolean[nodes];  // Has node been visited
    for (int i = 0; i < nodes; i++) {  // Initialize
        P[i] = -1;  // No predecessor on path
    }
    visited[root] = true;  // Initialize root node

    // Continued on next slide

    for (int i = 0; i < nodes-1; i++) {  // Add nodes-1 arcs
        int minDist = Integer.MAX_VALUE;
        int nextNode = -1;  // Next node to be added to MST
        int pred = -1;  // Predecessor of next node added to MST
        // Find node w/ min distance via arc from already visited set
        for (int node = 0; node < nodes; node++) {
            for (int arc = head[node]; arc < head[node + 1]; arc++) {
                int dest = to[arc];
                if (!visited[dest] & & dist[arc] < minDist) {
                    minDist = dist[arc];
                    nextNode = dest;
                    pred = node;
                }
            }
        }
        visited[nextNode] = true;
        P[nextNode] = pred;
        MSTcost += minDist;
    }
    return MSTcost;
}
Standard Prim: print(), main()

```java
public void print() {
    System.out.println("i \tP");
    for (int i = 0; i < nodes; i++) {
        if (P[i] == -1)
            System.out.println(i + " -");
        else
            System.out.println(i + " \t" + P[i]);
    }
    System.out.println("MST cost: " + MSTcost);
}

public static void main(String[] args) {
    // Create test data (H&S p. 237–see download)
    Prim p = new Prim(nodes, hh, tt, dd);
    p.prim(root);
    p.print();
}
```

Prim’s algorithm code, standard version, output

```
0 28
10 14 16
5 24 18 12
22 3

Node numbers start at 0, not 1, compared to first example
```

i P
0 -
1 2
2 3
3 4
4 5
5 0
6 1
MST cost: 99
Better Prim algorithm

• In each node iteration in the standard version:
  – We go through the arcs out of every visited node each time a node is added to the tree, looking for the shortest arc from any node
  – This is a lot of repetitive work: We look at each arc about n/2 times to see if it’s the shortest, and it almost never is
  – Standard Prim is O(na), for number of nodes n and arcs a

• If we keep the arcs out of visited nodes in a heap, we can just add arcs from a newly visited node to the heap, an O(lg n) operation, rather than the O(n) standard scan
  – In each iteration we then delete the shortest arc from the heap:
    • If its destination has been visited, ignore it and delete the next arc from the heap
    • Otherwise, add the arc to the MST
  – This is O(a lg n), where a is the number of arcs
  – Complexity proof easy except whether to use ‘lg n’ or ‘lg a’
  – Since ‘n’ and ‘a’ usually proportional, it’s not a major issue
  – Also, sorting to create the network takes O(a lg a) steps

PrimHeap: arc class

```java
public class MSTArc implements Comparable {
    int from;   // Package access
    int to;     // Package access
    int dist;   // Package access

    public MSTArc(int f, int t, int d) {
        from = f;
        to = t;
        dist = d;
    }

    public String toString() {
        return " from: " + from + " to: " + to + " dist: " + dist;
    }

    public int compareTo(Object o) {
        MSTArc other = (MSTArc) o;
        if (dist > other.dist) return -1; // Ascending sort with
        else if (dist < other.dist) return 1;
        else return 0;
    }
}
```
PrimHeap: data members, constructor

```java
public class PrimHeap {  // Assumes connected graph; not checked
    private int nodes;  // Assumes consecutive node numbers
    private int arcs;
    private int[] head;
    private int[] to;
    private int[] dist;
    private boolean[] visited;  // Has node been visited in Prim
    private int MSTcost;
    private Heap g;
    private MSTArc[] inMST;  // Arcs in MST

    PrimHeap(int n, int[] a, int[] h, int[] t, int[] d) {
        nodes = n;
        arcs = a;
        head = h;
        to = t;
        dist = d;
        g = new Heap(arcs);
        inMST = new MSTArc[nodes];
    }
}
```

PrimHeap: prim()

```java
public int prim(int root) {
    visited = new boolean[nodes];
    MSTArc inArc = null;
    int k = 0;  // Index of arcs in MST
    visited[root] = true;  // Initialize root node
    for (int arc = head[root]; arc < head[root + 1]; arc++)
        g.insert(new MSTArc(root, to[arc], dist[arc]));

    for (int i = 0; i < nodes - 1; i++) { // Add (nodes-1) arcs
        do { // Find shortest arc to node not yet visited
            inArc = (MSTArc) g.delete();
            while (!visited[inArc.to]);
            inMST[k++] = inArc;
            int inNode = inArc.to;
            visited[inNode] = true;
            MSTcost += inArc.dist;
            for (int arc = head[inNode]; arc < head[inNode + 1]; arc++)
                g.insert(new MSTArc(inNode, to[arc], dist[arc]));
        } while (visited[inArc.to]);
    }
    return MSTcost;  // O(aln)
}
```
PrimHeap: print(), main()

```java
public void print() {
    System.out.println("ArCs iM MST");
    for (int i = 0; i < nodes - 1; i++) {
        System.out.println(inMST[i]);
    }
    System.out.println("MST cost:\n + MSTcost");
}

public static void main(String[] args) {
// Create test data (H&S p. 237) – see download
    PrimHeap p = new PrimHeap(nodes, arcs, hh, tt, dd);
    p.print(root);
    p.print();
}
```

Prim's algorithm code, heap version, output

ArCs iM MST
- from 0 to: 5 dist: 10
- from 5 to: 4 dist: 25
- from 4 to: 3 dist: 22
- from 3 to: 2 dist: 12
- from 2 to: 1 dist: 16
- from 1 to: 6 dist: 14
MST cost: 99

Node numbers start at 0, not 1, compared to first example
Kruskal’s algorithm

- A different greedy method to build minimum spanning tree:

  Tree T is empty;
  Heap A contains all arcs, from lowest to highest cost
  While (T has fewer than n-1 arcs) & (A has more arcs) {
    Delete arc (v,w) from A
    If arc (v,w) does not create a cycle in T
      Add arc (v,w) to T
    Else
      Discard arc (v,w)
  }

- To detect cycles, we need to know if the origin and destination nodes of the candidate entering arc are already connected
  - Doing this efficiently is key to Kruskal’s algorithm
  - We place connected nodes in the same Set
  - The arcs will be a forest (set of disconnected subtrees) until the end
- We place the arcs in a Heap
  - We only need the minimum arc in each iteration, not a complete sort

Kruskal’s algorithm example

Figure by MIT OpenCourseWare.
**Kruskal: data members, constructor**

```java
public class Kruskal {    // Assumes connected graph; not checked
    private int nodes;    // Assumes consecutive node numbers
    private int arcs;
    private MSTArc[] inMST;    // Arcs in MST
    private int MSTcost;
    private Heap g;
    private Set s;

    Kruskal(int n, int a, MSTArc[] arcList) {
        nodes = n;
        arcs = a;
        inMST = new MSTArc[nodes];
        s = new Set(nodes);
        g = new Heap(arcList);
    }
}
```

**Kruskal: kruskal()**

```java
public void kruskal() {
    int i = 0;    // Index in inMST array where arcs are placed
    for (int arc = 0; arc < arcs; arc++) {
        MSTArc d = (MSTArc) g.delete();
        int j = s.collapsingFind(d.from);
        int k = s.collapsingFind(d.to);
        if (j != k) {
            inMST[i++] = d;
            MSTcost += d.dist;
            s.weightedUnion(j, k);
        }
        if (i == nodes - 1)
            break;
    }
}
```

// print() and main() same as PrimHeap (except call kruskal() in main)
// MSTArc class same as in PrimHeap
// Once you're comfortable with the MST codes, move them to Graph class
// KruskalAdjArray class in download uses adjacency array
Kruskal's algorithm code, output

Arcs in MST
from: 0 to: 5 dist: 10
from: 3 to: 2 dist: 12
from: 1 to: 6 dist: 14
from: 1 to: 2 dist: 16
from: 3 to: 4 dist: 22
from: 4 to: 5 dist: 25
MST cost: 99

Node numbers start at 0, not 1, compared to first example

Improving Kruskal: Boruvka steps

A Boruvka step

Figure by MIT OpenCourseWare.
Summary: Minimum spanning trees

• *Prim*:
  – Without heap: \(O(na)\), where \(n\) is number of nodes
  – With heap: \(O(a \lg n)\), where \(a\) is number of arcs

• *Kruskal*:
  – Standard: \(O(a \lg n)\), where \(a\) is number of arcs
  – Randomized: \(\Omega'(n + a)\), where \(\Omega'\) is ‘high probability’ running time of random element
    • See text, p. 53-54

• *Prim with heap and standard Kruskal* are usual implementation choices
  – Fast, straightforward

• *Add these to your Graph class* if you wish
  – Use symmetric directed graph in implementation
  – Minor changes to constructor for add’l data members