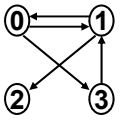
1.204 Lecture 7

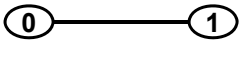
Data structures: graphs, sets

Graphs and Networks

- A graph contains:
  - Nodes
  - Arcs, or pairs of nodes ij, i !=j
- Graphs can be directed or undirected



Directed
<0, 1>
<1, 0>

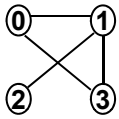


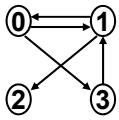
Undirected
(0, 1)

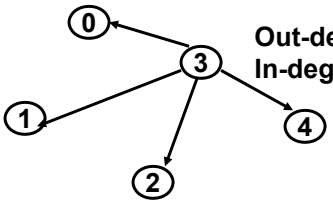
- A network is a graph with a cost associated with each arc
- There are two kinds of networks in this world…
  - Electrical and its kin…and traffic and its kin…

Networks

- **In an undirected network:**
  - **Node i is adjacent to node j if arc ij exists**
  - **Degree of node is number of arcs it terminates**
- **In a directed network:**
  - **In-degree of node is number of arcs in**
  - **Out-degree of node is number of arcs out**



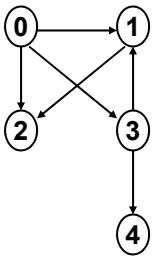


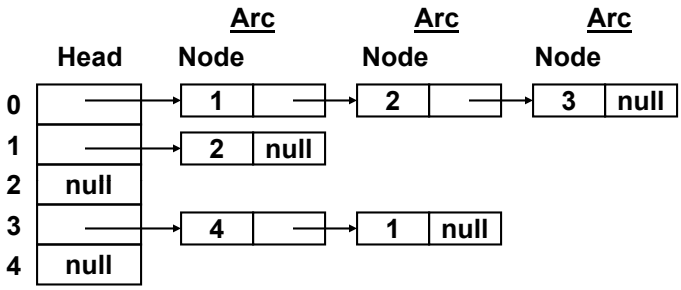


Out-degree of node 3= 4
In-degree of node 3= 0

Adjacency list representation of graphs

- **Adjacency list of graph is n lists, one for each node i**
  - **Adjacency list contains node(s) adjacent to i**



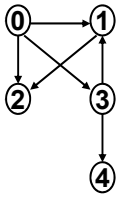


Head holds reference from each node i to the adjacent node list. Arc order arbitrary

Adjacency array representation of graphs

- If no insertion/deletion of nodes and arcs is to be done (or graph is large), we dispense with the links and list.
  - If we read the arcs from input and sort by 'from' node, we get:

From	To	Cost	(Arc number)
0	1	43	0
0	2	52	1
0	3	94	2
1	2	22	3
3	4	71	4
3	1	37	5



  - The 'from' node repeats when out-degree > 1
- We recast this structure as arrays H, To, Cost:

(Node)	H
0	0
1	3
2	4
3	4
4	6
5	6 (sentinel)

(Arc)	To	Cost
0	1	43
1	2	52
2	3	94
3	2	22
4	4	71
5	1	37

The H array

- H[i] holds the index of the first arc out of node i
  - Arcs must be sorted in order of origin (from) node
- Special case: If there are no arcs out of a node i,
  - H[i]= H[i+1]
  - This ensures that the inner for-loop below executes zero times in this special case

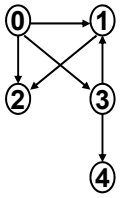
```
for (int node= 0; node < nodes; node++)
   for (int arc= H[node]; arc < H[node+1]; arc++)
     System.out.println("Arc from node " + node +
     " to node "+ to[arc] + " cost " + cost[arc]);
```

- This is creating two entities, nodes and arcs
  - And normalizing the data, which is a key principle

The H array

Node	H
0	0
1	3
2	
3	4
4	
5	6 (sentinel)

Fill in the first arc out of each node

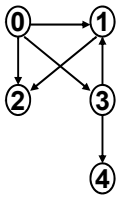


Arc	From	To
0	0	1
1	0	2
2	0	3
3	1	2
4	3	1
5	3	4

The H array

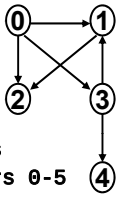
Node	H
0	0
1	3
2	4
3	4
4	6
5	6 (sentinel)

Then set H[i]= H[i+1] for any nodes with no arcs out of them



Arc	From	To
0	0	1
1	0	2
2	0	3
3	1	2
4	3	1
5	3	4

Traversing a graph




```
public class GraphSimple {
   public static void main(String[] args) {
     int[] H= {0, 3, 4, 4, 6, 6};     // 5 actual nodes
     int[] to= {1, 2, 3, 2, 4, 1};    // 6 arcs, numbers 0-5
     int[] cost= {43, 52, 94, 22, 71, 37};
     int nodes= H.length - 1;         // Don't count sentinel

     for (int node= 0; node < nodes; node++)
       for (int arc= H[node]; arc < H[node+1]; arc++)
         System.out.println("Arc from node " + node +
               " to node "+ to[arc] + " cost " + cost[arc]);

   }
}
// This traverses all the arcs in the graph
// To traverse (visit only once) the nodes, create a boolean array
// visit; set it true when node is visited first time; and check
// it to output all nodes in graph only once
```

Graph class

```
public class Graph {
   private int to[];
   private int dist[];
   private int H[];
   private int nodes;
   private int arcs;

   // Constructor follows:
   //  1. Reads network arcs from file (or database)
   //  2. Sorts arcs by origin node, then destination node
   //  3. Fishes out to[] and dist[] arrays from sorted arcs
   //          (Done for convenience in later algorithms)
   //  4. Constructs head array H[] of references to 1st arc
   //          out of each node
```

Graph: constructor

```
public Graph(String filename) {
     Arc[] graph= null;            // graph is discarded at end
     try {                         // Step 1: Read arcs from file
       FileReader fin= new FileReader(filename);
       BufferedReader in= new BufferedReader(fin);
       graph= readData(in);        // also sets number of nodes
       in.close();
     } catch(IOException e) {
         System.out.println(e);
     }
     Arrays.sort(graph);           // Step 2: Sort in origin order
     arcs= graph.length;           // Step 3: Fish out to[], dist[]
     to= new int[arcs];
     dist= new int[arcs];
     for (int i=0; i < arcs; i++) {
       to[i]= graph[i].dest;
       dist[i]= graph[i].cost;
     }                             // continues on next slide
```

Graph: constructor, p.2

```
     // Create H from the array of Arcs. Length= nodes+1 (sentinel)
     H= new int[nodes+1];  // Step 4: Construct head array
     int prevOrigin= -1;
     for (int i=0; i < arcs; i++) {
       int o= graph[i].origin;
       if (o != prevOrigin) {
         for (int j= prevOrigin+1; j < o; j++)
           H[j]= i;                // Nodes with no arcs out
         H[o]= i;
         prevOrigin= o;
       }
     }
     // Sentinel, and nodes before it with no arcs out.
     for (int i= nodes; i > prevOrigin; i--)
       H[i]= arcs;
}    // End constructor
```

Graph: read input

```
public Arc[] readData(BufferedReader in) throws IOException {
     int n= Integer.parseInt(in.readLine()); // Number of arcs
     Arc[] arcArr= new Arc[n];
     for (int i=0; i < n; i++) {
       arcArr[i]= new Arc();
       String str = in.readLine();
       StringTokenizer t = new StringTokenizer(str, ",");
       arcArr[i].origin= (Integer.parseInt(t.nextToken()));
       arcArr[i].dest= (Integer.parseInt(t.nextToken()));
       arcArr[i].cost= (Integer.parseInt(t.nextToken()));
       if (arcArr[i].origin > nodes)
         nodes= arcArr[i].origin;
       if (arcArr[i].dest > nodes)
         nodes= arcArr[i].dest;
     }
     nodes++;       // Started at 0, add one to get number of nodes
     return arcArr;
}
```

Graph: traverse(), main()

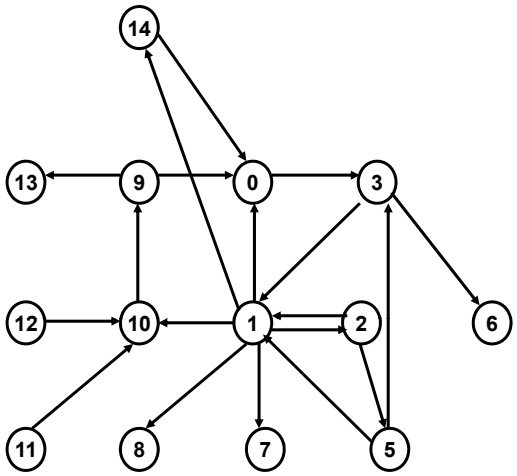
```
  public void traverse() {    // Traverses arcs
    for (int node= 0; node < nodes; node++) {
      for (int arc= H[node]; arc < H[node+1]; arc++)
         System.out.println("("+ node +
               ", "+ to[arc]+", "+ dist[arc]+")");
    }
  }

  public static void main(String[] args) {
    Graph g= new Graph("src/dataStuctures/graph.txt");
    g.traverse();
  }
}

// There is one other type of graph traverse, depth-first.
// It traverses the nodes of the graph recursively
// See Horowitz text if interested
```

Arc

```
public class Arc implements Comparable {
   int origin;         // Package access
   int dest;           // Package access
   int cost;           // Package access

   public Arc() {}
   public Arc(int o, int d, int c) {
     origin= o;
     dest= d;
     cost= c;
   }

   public int compareTo(Object other) {
     Arc o= (Arc) other;
     if (origin < o.origin || (origin == o.origin && dest < o.dest))
       return -1;
     else if (origin> o.origin || (origin== o.origin && dest> o.dest))
       return 1;
     else
       return 0;
    }

    public String toString() {
      return (origin+" "+dest+" "+cost);
} }
```

Example

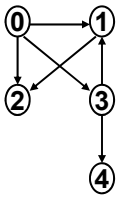
```
19
5,3,42
9,0,98
9,13,21
10,9,39
11,10,32
12,10,43
5,1,33
1,0,54
1,2,45
0,3,32
2,1,22
2,5,26
3,1,17
3,6,18
14,0,11
1,10,32
1,14,22
1,7,24
1,8,37
```



Bad ways to represent graphs

- Node-node incidence matrix

	0	1	2	3	4
0	0	1	1	1	0
1	0	0	1	0	0
2	0	0	0	0	0
3	0	1	0	0	1
4	0	0	0	0	0



  - Does not scale
    - 1,000,000 node network (10^6) requires 10^{12} storage (TB)
    - Average node degree ~ 4 → matrix density ~ 10^{-6}
  - Adjacency array (or "forward star") is a sparse matrix technique
- Any representation with Arc or Node objects allocated one at a time as the graph is built
  - Allocating memory in small pieces is VERY slow
  - Use arrays; determine array size first
- Linked lists
  - Same memory allocation problem
- These bad ideas limit problem size to toy problems only
  - Even though the underlying algorithms are very fast
  - Same issues apply to dynamic programming (use virtual graph!)

Disjoint sets

- Assume we have objects, each with an integer identifier
- Sets are disjoint
  - If S_i and S_j, i != j, are two sets, there is no element in both S_i and S_j
- Operations:
  - Union: all elements in both sets
  - Find: find set containing an element
  - (Intersection is null)
  - (Difference is all members)
- Model each set as tree
  - Choose one node in set as root
  - Use array to store root:
    - p[i] is parent of node I
    - p[i] = -1 indicates root

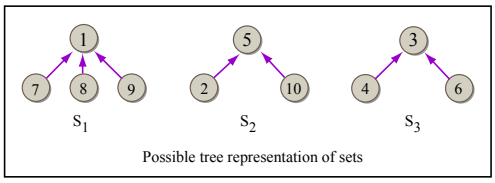


Figure by MIT OpenCourseWare.

i:	1	2	3	4	5	6	7	8	9	10
p[i]:	-1	5	-1	3	-1	3	1	1	1	5

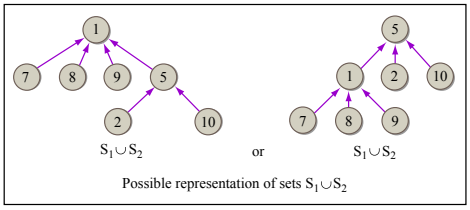


Figure by MIT OpenCourseWare.

i:	1	2	3	4	5	6	7	8	9	10
p[i]:	-1	5			1		1	1	1	5
	or 5				or -1					

Set: simple union and find

- **Simple implementations**
  - **Union(i,j) sets p[i]= j**
  - **Find(i) looks up p[i] until p[i] = -1**
- **These have poor worst-case performance**
  - **We can get degenerate trees, e.g.,**
  - **union(1,2), union(2,3), union(3,4), … produces:**

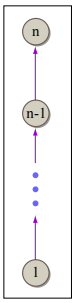


Figure by MIT OpenCourseWare.

In the worst case
- **Each union is O(1)**
- **Average find is O(n)**

We can do better

Set: simple version

```
public class Set {
   private int[] p;
   private static final int DEFAULT_CAPACITY = 10;

   public Set(int size) {
     p = new int[size];
     for (int i = 0; i < size; i++)
       p[i] = -1;
   }
   public Set() {
     this(DEFAULT_CAPACITY);
   }
   public void simpleUnion(int i, int j) {
     p[i] = j;
   }
   public int simpleFind(int i) {
     while (p[i] >= 0)
       i = p[i];
     return i;
   }
```

Set: efficient union

- **Avoid degenerate tree by using weighting rule**
  - **Let n_i be number of nodes in set i**
  - **If $n_i < n_j$ set p[i]= j; otherwise set p[j]= i (smaller tree is child)**

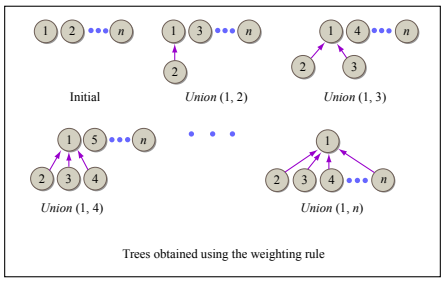


Figure by MIT OpenCourseWare.

  - **To implement this:**
    - **Keep number of nodes in root of each tree as negative number (-1 used to indicate root in simple version)**
    - **Union complexity is still O(1), simple find complexity O(lg n) [p.115]**

Set: efficient union

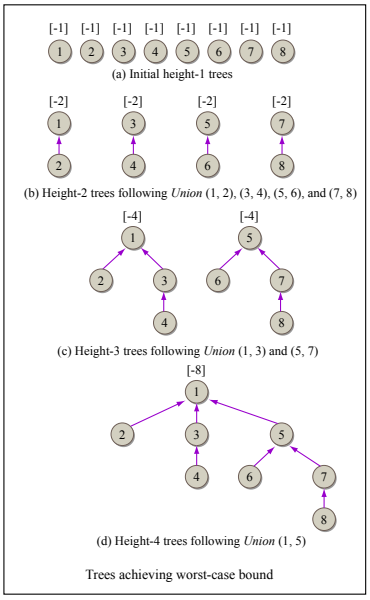
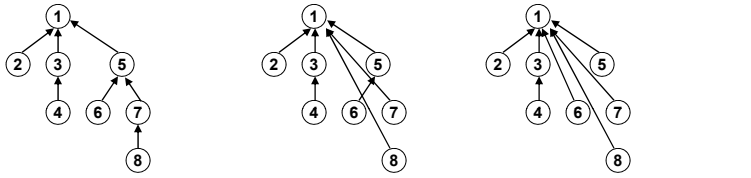


Figure by MIT OpenCourseWare.

Set: efficient find

- **Find can use collapsing rule:**
  - **If j is node on path from i to its root and p[i] != root[i],**
  - **Then set p[j]= root[i]**



Original set **After find(8)** **After find(8) and find(6)**

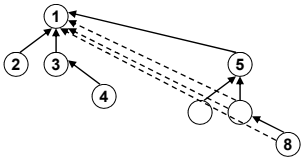
Sequence of unions and finds is O(Ackermann's function), nearly constant

Set: efficient version

```
public void weightedUnion(int i, int j) {
  // Could check p[i]<0, p[j]<0, throw exception if not
  int nodes = p[i] + p[j]; // negative
  if (p[i] > p[j]) {       // i has fewer nodes
    p[i] = j;
    p[j] = nodes;
  } else {                 // j has fewer or equal nodes
    p[j] = i;
    p[i] = nodes;
  }
}

public int collapsingFind(int i) {
  int r = i;
  while (p[r] >= 0)
    r = p[r];
  while (i != r) {     // Collapse nodes from i to root r
    int s = p[i];
    p[i] = r;
    i = s;
  }
  return r;
}
```

Set: example

```
  public static void main(String[] args) {
      Set s= new Set();
      s.weightedUnion(1,2);
      s.weightedUnion(3,4);
      s.weightedUnion(5,6);
      s.weightedUnion(7,8);
      s.weightedUnion(1,3);
      s.weightedUnion(5,7);
      s.weightedUnion(1,5);
      for (int i= 1; i < 9; i++)
             System.out.print(" "+ s.p[i]);
      System.out.println();
      s.collapsingFind(8);
      for (int i= 1; i < 9; i++)
             System.out.print(" "+ s.p[i]);
      System.out.println();
      s.collapsingFind(6);
      for (int i= 1; i < 9; i++)
             System.out.print(" "+ s.p[i]);
      System.out.println();
}  }
```




```
 -8 1 1 3 1 5 5 7

 -8 1 1 3 1 5 1 1

 -8 1 1 3 1 1 1 1
```

1.204 Computer Algorithms in Systems Engineering

Spring 2010