Graphs and Networks

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

  \[
  \begin{align*}
  &\text{Directed} \\
  &\{0, 1\} \\
  &\{1, 0\}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{Undirected} \\
  &\{0, 1\}
  \end{align*}
  \]

- 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

Adjacency list representation of graphs

• Adjacency list of graph is $n$ lists, one for each node $i$
  – Adjacency list contains node(s) adjacent to $i$
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:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Cost</th>
<th>(Arc number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>94</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>71</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>37</td>
<td>5</td>
</tr>
</tbody>
</table>

- The ‘from’ node repeats when out-degree > 1

- We recast this structure as arrays H, To, Cost:

<table>
<thead>
<tr>
<th>(Node)</th>
<th>H</th>
<th>(Arc)</th>
<th>To</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>43</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>94</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>37</td>
</tr>
</tbody>
</table>

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
    ```java
    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

<table>
<thead>
<tr>
<th>Node</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6  (sentinel)</td>
</tr>
</tbody>
</table>

Fill in the first arc out of each node

<table>
<thead>
<tr>
<th>Arc</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Arc</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Traversing a graph

```java
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

```java
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

```java
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: Flush 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

```java
// 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

```java
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++;
    return arcArr;
}
```

Graph: traverse(), main()

```java
public void traverse() { // Traverse 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/dataStructures/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 1 1 1 0 1
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 \(\approx 4\) \(\rightarrow\) matrix density \(\approx 10^{-6}\)
- 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 \(S_j\), \(i \neq j\), 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.
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:

In the worst case
• Each union is O(1)
• Average find is O(n)
We can do better

Set: simple version

```java
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;
    }
}
```

Figure by MIT OpenCourseWare.
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 \( p[j]=i \); otherwise set \( p[i]=j \) (smaller tree is child)

- 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 \( O(\log n) \) [p.115]

Trees achieved worst-case bound

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]

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

Set: efficient version

```java
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

```java
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
```