Chapter 19: Binomial Heaps
Chapter 22: Elementary Graph Algorithms

19-2) Chapter 23 presents two algorithms to solve the problem of finding a minimum spanning tree of an undirected graph. Here, we shall see how binomial heaps can be used to devise a different minimum-spanning-tree algorithm.

We are given a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. We call $w(u, v)$ the weight of edge (u, v). We wish to find a minimum spanning tree for G: an acyclic subset $T \subseteq E$ that connects all the vertices in V and whose total weight

$w(T) = \sum_{(u,v)\in T} w(u, v))$ is minimized.

The following pseudocode, which can be proven correct using techniques from Section 23.1, constructs a minimum spanning tree T. It maintains a partition $\{V_i\}$ of the vertices of V and, with each set V_i, a set

$E_i \subseteq \{(u, v) : u \in V_i \text{ or } v \in V_i\}$

of edges incident on vertices in V_i.

```
MST(G)
1  T ← ∅
2  for each vertex v_i ∈ V[G]
3    do V_i ← {v_i}
4    E_i ← {(v_i, v) ∈ E[G]}
5  while there is more than one set V_i
6    do choose any set V_i
7       extract the minimum-weight edge (u, v) from E_i
8       assume without loss of generality that u ∈ V_i and v ∈ V_j
9       if i ≠ j
10        then T ← T ∪ {(u, v)}
11             V_i ← V_i ∪ V_j, destroying V_j
12             E_i ← E_i ∪ E_j
```

Describe how to implement this algorithm using binomial heaps to manage the vertex and edge sets. Do you need to change the representation of a binomial heap? Do you need to add operations beyond the mergeable-heap operations given in Figure 19.1? Give the running time of your implementation.

Solution:

first for loop

Create binomial heaps for each of the V_i's and E_i's. The keys of the nodes of the V_i's are the vertex labels and the keys of the nodes of the E_i's is the pair u, the adjacent vertex, and $w(v_i, u)$, the weight of the edge. [(1)]This takes O(2V) = $O(V)$.

The edges must then be inserted into the heaps. Since each of the vertices in the graph can have at most degree $|V| - 1$, each E_i heap has at most $|V| - 1$ nodes.

BINOMIAL-HEAP-INSERT(H,x) takes $O(lg\ n)$ to compute (pg 468) for a heap of size n, which must be called to insert each of the edges. $^{(2)}$ This will take $O(E\ lgV)$.

while

Line 7 BINOMIAL-HEAP-EXTRACT-MIN(E_i) runs $O(lgV)$ and will excute a maximum of $|E|$ number of times $\implies$ run time is $O(ElgV)$.
Line 8 node u must be searched within V_i, which runs in O(lgV) and will run as many times as line 7 runs $\implies$ run time is $(ElgV)(lgV) = O(ElgV)$.
Line 9 runs in $O(1)$ and runs as many times as line 7 as well.
$^{(3)}$Lines 7-9 will run in $O(ElgV)$

Execution inside the if statement will occur $|V-1|$ times , O(V):
Line 10 will computes in $O(1)$.
Line 11 BINOMIAL-HEAP-UNION(V_1,V_2) computes in $O(lgV)$
Line 12 BINOMIAL-HEAP-UNION(E_1,E_2) computes in $O(lgE)$
$^{(4)}$Lines 10-12 will run in $O(V + VlgV + VlgE) = O(VlgV)$

The total runtime is $^{(1)+(2)+(3)+(4)} = O(V + ElgV + ElgV + VlgV) = O(ElgV)$. [2]

There is no need to change the representatin of the binomial heap. The operations used are BINOMIAL-HEAP-INSERT(H,x), BINOMIAL-HEAP-EXTRACT-MIN(H), BINOMIAL-HEAP-UNION(H_1,H_2), and BINOMIAL-HEAP-MERGE(H_1,H_2). The total running time: $O(ElgV)$

22.1-1) Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Solution:
An adjacency-list representation of a directed graph is an array, Adj, of size $|V|$, where each entry, $Adj[u]$, is a linked list containig all the vertices adjacent to $u \in V$.

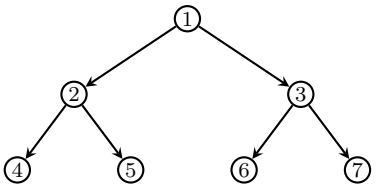
(a) *out-degree*: the out-degree of a vertex u is the number of elements in the linked list $Adj[u]$, which takes $O(E)$ to compute. The sum of the out-degrees of all the vertices is $|E|$. To compute the out-degree for each vertex, we must count the number of out-degrees for each, which will take $O(V+E)$ because each edge and each vertex is referenced once.

(b) *in-degree*: the in-degree of a vertex u is the number of nodes pointing to it, which is the total number of times u appears in an adjacency list. To compute the in-degree of each vertex, we look at the $Adj[u]$ for each vertex u and increment the in-degree of a vertex everytime it appears in $Adj[u]$. Since each edge and each vertex is referenced once, this takes $O(V+E)$.

22.1-2) Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

Solution:

Binary heap



Adjacency-list

1	$\rightarrow 2 \rightarrow 3$
2	$\rightarrow 4 \rightarrow 5$
3	$\rightarrow 6 \rightarrow 7$
4	
5	
6	
7	

Adjacency-matrix

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

22.1-3) The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

Solution:

(a) Adjacency-matrix

G-TRANSPOSE-ADJ-MATRIX(G)

```
1 for i ← 2 to G.length
2   for j ← 1 to i−1
3     swap(G[i,j] G[j,i])
```

This iterates through the entries below the diagonal of the adjacency-matrix and swaps each entry below the diagonal with the corresponding entry above the diagonal. There are $\frac{V^2}{2}$ entries below the diagonal. Thus, it takes $O(V^2)$ to compute the transpose of the adjacency-matrix.

(b) Adjacency-list

G-TRANSPOSE-ADJ-LIST(G)

```
1 allocate array G^T of size |V|
2 for each v ∈ V
3   for each u ∈ Adj[v] in G
4     insert v to Adj[u] in G^T
5 return ← G^T
```

The transpose of a G flips all the in-edges to out-edges (note that if the graph is undirected, then fliping the order of the edges does not change the graph, thus, the matrix is symmetric and running the transpose does not change the either the adj-list or the adj-matrix representation of the graph). This algorithm references the edges of each vertex for all vertices. Since each edge and each vertex is referenced once, this algorithm computes in $O(V + E)$.

22.1-4) Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$-time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

Solution:

case 1: Undirected multigraph

SIMPLIFY-UNDIRECTED-GRAPH(G)

```
1  for each v ∈ V
2    RADIX–SORT(Adj[v])        \\sorting and eliminating self loops
3
4  for each v ∈ V
5    current ← Adj[v]
6    while(current.next() ≠ null)
7      next ← current.next()
8      while(next.next()≠null AND next.next()=current)\\skip duplicate edges
9        next ← next.next()
10     current.next() ← next
11     current ← next
```

The first part goes through each Adjacency-list and sorts the entries while checking for self loops. Each vertex is referenced once, which takes $|V|$. RADIX-SORT runs linearly so it sorts $Adj[v]$ in O($|Adj[v]|$). This is done for each vertex, which is a total running time $\sum_{v \in V} |Adj[v]| = |E|$. Thus, the running time of the first for loop is $O(V + E)$. The second loop goes through each $Adj[v]$ list and gets rid of duplicate nodes by skipping the duplicates, breaking the linked list, and updating the pointer to the next adjacent vertex. This requires only one pass through the whole Adjacency-list, which runs in $O(V + E)$. The total running time of SIMPLIFY-UNDIRECTED-GRAPH() is $O(V + E)$.

[3]

case2: Directed multigraph

SIMPLIFY-DIRECTED-GRAPH(G)

```
1 G^T ← G–TRANSPOSE–ADJ–LIST(G)
2 G ← G ∪ G^T
3 SIMPLIFY–UNDIRECTED–GRAPH(G)
```

For a directed multigraph, we must make sure that any edge between vertex u and v must be in both $Adj[u]$ and $Adj[v]$. The transpose flips all the directed edges and takes $O(V + E)$ to compute. Combining G and G^T guarantees that each edge is bidirectional (whole graph is now undirected). This takes $2|V| + |E| = O(V+E)$ to compute. We now have twice as many edges. Running SIMPLIFY-UNDIRECTED-GRAPH will simplifiy the graph as explained above, taking $|V|+2|E| = O(V+E)$ time to compute. The total running time of SIMPLIFY-DIRECTED-GRAPH(G) is $O\big(3(V + E)\big) = O(V + E)$.

References

[1] Cormen, Thomas. H., Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, MA, 2009.

[2] http://www.slideshare.net/champguru/solution-of-cormen

[3] http://www.cs.nyu.edu/courses/summer04/G22.1170-001/fin_samp_04.pdf