Graphs

CSC 173

Definitions

- Graph – a set of N nodes and E edges (or arcs) where each element of E is a pair of nodes
  - Directed vs. undirected (ordered vs. unordered pairs)
  - Weighted (each edge has an associated value/weight)
- Path in a directed graph
  - List of nodes such that there is an arc from the ith to the i+1th node for all 1 <=i<n
  - Cost of the path is the sum of the costs on each arc
  - A simple path visits no node more than once
- A cycle in a directed graph is a path of length >=1 that begins and ends with the same node
  - Cyclic graph – one with at least one cycle
  - Acyclic graph – one with no cycles

Graph Implementations

- Adjacency lists
  - Is (u,v) an edge? – O(E/N) on average
  - Successors(u) – O(E/N) on average
  - Predecessors (u) – O(N+E)
  - Space – O(N+E)
  - Best for sparse graphs (E << N^2)
- Adjacency matrix
  - Is (u,v) an edge? – O(1)
  - Successors(u) – O(N)
  - Predecessors(u) – O(N)
  - Space – O(N^2)
  - Best for dense graphs (E ~= N^2)

Operations on Graphs

- Breadth-first and depth-first search
- Finding cycles
- Connected components of undirected graphs
- Minimal spanning tree: find a tree that connects all nodes in a weighted graph with minimal cost
- Single-source shortest path
- All-pairs shortest path

Breadth-First Search Algorithm

```
BFS(vertex u)
   queue Q
   u.marked  = true
   // perform required operation
   Q.enqueue(u)
   while not Q.empty()
      v = Q.dequeue()
      for all neighbors w of v
         if not w.marked
            w.marked = true
            // perform required operation
            Q.enqueue(w)
main
   for all nodes u
      u.marked = false
   for all nodes u
      if not u.marked
         BFS(u)
```

Running Time – O(N+E)

Depth-First Search Algorithm

```
DFS(vertex u)
   u.marked = true
   //perform required operation
   for all neighbors v of u
      if not v.marked
         DFS(v)
main
   for all nodes u
      u.marked = false
   for all nodes u
      if not u.marked
         DFS(u)
```

Running Time – O(N+E)

Finding Cycles

```
Cycle_test_DFS(vertex u, p)  //p is parent, needed only for undirected case
   u.marked = true; u.onpath = true;
   //perform required operation
   for all neighbors v of u
      if v.onpath and (graph is directed or v != p)
         announce cycle
         halt
      if not v.marked
         cycle_test_DFS(v,u)
   u.onpath = false
main
   for all nodes u
      u.marked = false; u.onpath = false
   for all nodes u
      if not u.marked
         cycle_test_DFS(u, nil)
   announce no cycle
```

Running Time – O(N+E)

Post-Order DFS

```
postorder_DFS(vertex u, ref int nextnum)
   u.marked = true
   for all neighbors v of u
      if not v.marked
         postorder_DFS(v, nextnum)
   u.ponum = nextnum++
postorder_main
   for all nodes u
      u.marked = false
   int nextnum = 1
   for all nodes u
      if not u.marked
         postorder_DFS(u, nextnum)
```

Running Time – O(N+E)

Testing for Cycles: Alternative

```
cycle_test_alternate_main
   postorder_main()
   for all nodes u
      for all neighbors v of u
         if u.ponum <= v.ponum
                                  // = catches self-loops
            announce cycle
            halt
   announce no cycle
```

Topological Sort

- Assign a linear ordering to the vertices in a DAG such that if (i,j)is an edge, i appears before j in the ordering
  - Use a stack to get the order right, pushing prior to exiting the DFS call
  - Use the reverse of the postorder numbers to order nodes (there could be other sorts as well due to unordered nodes)

Reachability

- Given a directed graph G and a vertex v in G, find all vertices in G that can be reached from v by following arcs
  - Set of nodes explored from v using depth-first search

Single Source Shortest Path (SSSP)

- Find the cost of the least cost path from a source node v to each other node in G

Dijkstra's Algorithm

- Greedy algorithm for the SSSP problem
- Abstractions used
  - Adjacency lists for neighbors of a node and the cost of edges
  - Priority queue of nodes ("unsettled" nodes) for which the cheapest path has not yet been identified
  - A notion of the lowest current known cost to each "unsettled" node

Dijkstra's Algorithm

```
DijkstraSSSP(vertex u)
   vertex_set unsettled = V – {u}
   for all nodes v                                  //O(N)
      v.cost = infinity
   for all neighbors v of u                         //O(N)
      v.cost = weight(u,v)
   while not unsettled.empty()                      //O(N)
      find v in unsettled s.t. v.cost is minimal    //O(NlogN) if partially ordered tree
      unsettled -= {v}
      for all neighbors w of v                          O(E)
         if v.cost + weight(v,w) < w.cost
            //shorter path from u to w through v
            w.cost = v.cost + weight(v,w)
            unsettled.adjust()    // re-order heap, O(ElogN)
```

Dijkstra's Algorithm

- Why does it work?
  - On each iteration of the main loop, remove vertex v with least cost from unsettled. v.cost is the lowest cost path from u to v through known nodes. If there is a lower cost path through as yet unknown node x
    - x.cost would be less than v.cost
    - x would be selected before v
    - x would be in known

All Pairs Shortest Path – Floyd's Algorithm

- Uses adjacency matrix

```
for u in 0..n-1
   for v in 0..n01
      // initialize with direct arcs
      D[u,v] = A[u,v]  // A is infinity if there is no edge
for u in 0..n-1
   for v in 0..n-1
      for w in 0..n-1
         if (D[v,u] + D[u,w] < D[v,w])
            D[v,w] = D[v,u]+D[u,w]
```

Running time – O(N*N*N)

Transitive Closure (Warshall's Algorithm)

- Determine if there is a path between any two nodes

```
for u in 0..n-1
   for v in 0..n01
      // there's a path if there's an edge
      P[u,v] = A[u,v]  // A is 0 if there is no edge
for u in 0..n-1
   for v in 0..n-1
      for w in 0..n-1
         if (!P[v,w])
            P[v,w] = P[v,u] && P[u,w]
```

Minimum Cost Spanning Tree (MCST)

- A graph G is connected if every pair of vertices is connected by a path
- A spanning tree for G is a free (unrooted) tree that connects all vertices in G
- The cost of the spanning tree is the sum of the cost of all edges in the tree

Prim's Algorithm for MCST

```
// Initially, one node in the spanning tree and no edges
PrimMCST(ref edge_set T) // T = set of edges in spanning tree
   int closest[N] // closest[v] = vertex u in U closest to v
   int lowcost[N] // lowcost[v] = weight[v,u]
   // U is, implicitly, node 0 and the nodes connected to it by edges of T

   T = empty
   for v in 1..N-1
      lowcost[v] = weight[0,v]
      closest[v] = 0
   N-1 times do
      // find the node closest to U and add it to U
      min = lowcost[1]
      k = 1
      for j in 2..N-1
         if lowcost[j] < min
            min = lowcost[j]
            k = j
      //k is now the node outside U closest to something in U; add it to U
      T += {(closest[k],k)}
      lowcost[k] = infinity
      for j in 1..N-1
         if weight[k,j] < lowcost[j] && lowcost[j] < infinity
            lowcost[j] = weight[k,j]
            closest[j] = k
```

Kruskal's Algorithm for MCST

- Initially, each node is its own MCST
- Merge two MCSTs at each step that can be connected together with the least cost adding the lowest cost edge
- Terminate when there is only 1 MCST that contains all vertices
- Running time – O(ElogE)

Other Operations of Interest

- Minimal graph coloring: assign a color to each node so that no two nodes sharing an edge have the same color, and the total number of distinct colors is as small as possible
- Hamiltonian circuit: Find a cycle, if there is one, on which every node appears exactly once
- Euler circuit: Find a cycle, if there is one, on which every edge appears exactly once
- Traveling salesman problem (TSP): find a minimum-cost cycle that visits every node exactly once

The above do not have known polynomial time solutions