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

```plaintext
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)
    // for all nodes u
    for all nodes u
        if not u.marked
            BFS(u)

Running Time – O(N+E)
```

Depth-First Search Algorithm

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

Running Time – O(N+E)
```
Finding Cycles

```plaintext
Cycle_test_DFS(vertex u, p)
// p is parent, needed only for undirected case
u.marked = true; u.onPath = false;
for all neighbors v of u
if !onPath and (graph is directed or v = p)
    cycle_detected = true
    halt
if !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 !marked
    cycle_test_DFS(u, nil)
    announce no cycle

Running Time – $O(V+E)$

Post-Order DFS

```plaintext
postorder_DFS(vertex u, ref int num)
    u.marked = true
    for all neighbors v of u
        if v.onPath and (graph is directed or v = u)
            cycle_detected = true
            halt
    if !marked
        cycle_test_DFS(u, u)
        u.onPath = false
    for all nodes u
        if !marked
            postorder_DFS(u, num)
            num += 1
```

`main`

for all nodes u
if !marked
    postorder_DFS(u, num)

Running Time – $O(V+E)$

Testing for Cycles: Alternative

```plaintext
cycle_test_alternate_main
postorder_main()
for all nodes u
for all neighbors v of u
    if u.num <= v.num
        // 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

\[
\text{DijkstraSSP}(\text{vertex } u)
\]

\[
\text{vertex}_\text{set} \text{ unsettled} = \text{V} - \{u\} \quad /\text{O}(N)
\]

\[
\text{v} . \text{cost} = \infty \\
\text{for all nodes } v \\
\text{v} . \text{cost} = \text{weight}(u,v) \\
\text{for all neighbors } v \text{ of } u \\
\text{while not unsettled. empty()}
\]

\[
\text{find } v \text{ in unsettled s.t. v.cost is minimal} \quad /\text{O}(N\log N) \text{ if partially ordered tree unsettled} \rightarrow (v)
\]

\[
\text{unsettled} - = \{v\} \\
\text{for all neighbors } w \text{ of } v
\]

\[
\text{if } v . \text{cost} + \text{weight}(v,w) < w . \text{cost} \\
\]

\[
\text{//shorter path from } u \text{ to } w \text{ through } v \\
\text{w} . \text{cost} = v . \text{cost} + \text{weight}(v,w)
\]

\[
\text{unsettled. adjust()} \quad // \text{re-order heap, O(E log N)}
\]

Dijkstra’s Algorithm

- Why does it work?
  - On each iteration of the main loop, remove vertex \( v \) with least cost from unsettled. \( v . \text{cost} \) is the lowest cost path from \( u \) to \( v \) through known nodes. If there is a lower cost path through as yet unknown node \( x \)
    - \( x . \text{cost} \) would be less than \( v . \text{cost} \)
    - \( x \) 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..n-1
    // 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 – \( O(N^3) \)

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..n-1
      // there’s a path if there’s an edge
      \( P[u,v] = A[u,v] \) // \( A \) is 0 if there is no edge
  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

1. Initially, one node in the spanning tree and no edges
2. Prim(MCST, set edge set T), T = set of edges in spanning tree
3. closest[u][v] = edge v in U closest to v
4. lowest[u][v] = weight[v, u]

// Iniially, one node in the spanning tree and no edges
Prim(MCST, set edge set T)

1. T = empty
2. for v in 1..N
3. closest[v][v] = 0
4. N times do
5. // find the node closest to U and add it to U
6. k = closest[0][v]
7. k = closest[0][v]
8. T += { (k, v) }
9. lowest[k][v] = infinity
10. for j in 1..N
11. if weight[k][j] < lowest[j][k] & & lowest[j][k] < infinity
12. lowest[j][k] = weight[k][j]
13. closest[j] = k

Kruskal’s Algorithm for MCST

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

Other Operations of Interest

1. 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
2. Hamiltonian circuit: Find a cycle, if there is one, on which every node appears exactly once
3. Euler circuit: Find a cycle, if there is one, on which every edge appears exactly once
4. Traveling salesman problem (TSP): find a minimum-cost cycle that visits every node exactly once

The above do not have known polynomial time solutions