CSci 231 Homework 9

Graph Algorithms

CLRS Chapter 22, 24

1. [CLRS 22.1-5] Give and analyse an algorithm for computing the square of a directed graph G given in (a) adjacency-list representation and (b) adjacency-matrix representation.

   Solution: To compute G^2 from the adjacency-list representation Adj of G, we perform the following for each $Adj[u]$:

   ```
   for each vertex v in Adj[u]
       for each vertex w in Adj[v]
           edge(u, w) ∈ E^2
           insert w in Adj2(u)
   ```

   where $Adj2$ is the adjacency-list representation of G^2. For every edge in Adj we scan at most $|V|$ vertices, thus we compute $Adj2$ in time $O(VE)$.

   After we have computed $Adj2$, we have to remove any duplicate edges from the lists (there may be more than one two-edge path in G between any two vertices). Removing duplicate edges is done in $O(V + E')$ where $E' = O(VE)$ is the number of edges in $Adj2$ (see for instance problem CLRS 22.1-4). Thus the total running time is $O(VE)+O(V + E')= O(VE)$.

   Let A denote the adjacency-matrix representation of G. The adjacency-matrix representation of G^2 is the square of A. Computing A^2 can be done in time $O(V^3)$ (and even faster, theoretically; Strassen's algorithm for example will compute A^2 in $O(V^{\lg 7})$).

2. (CLRS 22.2-8) Consider an undirected connected grah G. Give an $O(V + E)$ algorithm to compute a path that traverses each edge of G exactly once in each direction.

   Solution: Perform a DFS of G starting at an arbitrary vertex. The path required by the problem can be obtained from the order in which DFS explores the edges in the graph. When exploring an edge (u, v) that goes to an unvisited node the edge (u, v) is included for the first time in the path. When DFS backtracks to u again after v is made BLACK, the edge (u, v) is included for the 2nd time in the path, this time in the opposite direction (from v to u). When DFS explores an edge (u, v) that goes to a visited node (GRAY or BLACK) we add $(u, v)(v, u)$ to the path. In this way each edge is added to the path exactly twice.

3. (CLRS 22.4-3) Given an undirected graph $G = (V, E)$ determine in $O(V)$ time if it has a cycle.

   Solution: There are two cases:

(a) $E < V$: Then the graph may or may not have cycles. To check do a graph traversal (BFS or DFS). If during the traveral you meet an edge (u, v) that leads to an already visited vertex (GRAY or BLACK) then you've gotten a cycle. Otherwise there is no cycle. This takes $O(V + E) = O(V)$ (since $E < V$).

(b) $E \geq V$: In this case we will prove that the graph must have a cycle.

Claim 1: A tree of n nodes has $n - 1$ edges.

Proof of claim 1: By induction. Base case: a tree of 1 vertex has 0 edges. ok. Assume inductively that a tree of n vertices has $n - 1$ edges. Then a tree T of $n + 1$ vertices consists of a tree T' of n vertices plus another vertex connected to T' through an edge. Thus the number of edges in T is the number of edges in T' plus one. By induction hyopthesis T' has $n - 1$ edges so T has n edges. qed.

Coming back to the problem: Assume first that the graph G is connected. Perform a DFS traversal of G starting at an arbitrary vertex. Since the graph is connected the resulting DFS-tree will contain all the vertices in the graph. By Claim 1 the DFS-tree of G has $V - 1$ edges. Therefore since $E \geq V$ there will be at least an edge in G which is not in the DFS-tree of G. This edge gives a cycle in G.

If the graph G is not connected: If G has 2 connected components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Then it is easy to prove, by contradiction, that $E \geq V$ implies that either $E_1 \geq V_1$ or $E_2 \geq V_2$ (or both). In either case either G_1 will have a cycle or G_2 will have a cycle (or both).

(If the graph G is not connected and has k connected components then the same argument as above works, except that formally we need induction on k).

CLRS 22-3 (a) Prove that a directed graph has an Euler circuit if and only of for all v in G, $indeg(v) = outdeg(v)$.

Solution: First note that the proof must have two parts:

$\Longrightarrow$: If G has an Euler circuit C, then C is either a simple cyle (does not intersect itself), or not. If C is a simple cycle, each vertex in a simple cycle has indeg=outdeg=1, so the claim is true. If C is a cycle but not a simple cycle, then it must contain a simple cycle; remove it from G and from C; the remaining C is still an Euler circut for the remaining G. Repeat removing (simple) cycles until no edges left. When removing a cycle, an in-edge and out-edge of the vertices on the cycle are removed. After a cycle deletion, the in-degree and out-degree of a node on the cycle decrease by exactly 1. At the end, when no edges are left, all in-degrees and out-degrees are 0. So all vertices v must have started with $indeg(v) = outdeg(v)$.

$\Longleftarrow$: If every vertex v has $indeg(v) = outdeg(v)$, the first observation is that for any vertex v, there must be a path starting from v that comes back to v (need to prove this, see below). Assuming this is true, pick a random vertex v and find a cycle C that comes back to v. Delete all the edges on C from G. Each vertex in the new G still has $indeg(v) = outdeg(v)$, so we pick a vertex v' on C that has edges incident (such a vertex must exist) and repeat. Overall we find a cycle C, then another cycle C' that has (at least) a common vertex with C, and so on. We can build a big cycle that goes arund C, jumps into C' ad goes around C', then comes back to C and finishes C.

Proof of the claim that we made above: For any vertex v, there must be a cycle that contains v. Start from v, and chose any outgoing edge of v, say (v, u). Since $indeg(u) = outdeg(u)$ we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v, at the time we visit an entering edge, there must be an outgoing edge left unvisited, since $indeg = outdeg$ for all vertices. The only vertex for which there may not be an unvisited outgoing edge is v—because we started the cycle by visiting one of v's outgoing edges. Since there's always a leaving edge we can visit for any vertex other than v, eventually the cycle must return to v, thus proving the claim.

(b) Describe how to find an Euler circuit in G.

Solution: First we can check in $O(|E|)$ time whether $indeg(v) = outdeg(v)$ is true for every vertex. If yes, then we can q find the Euler circuit by finding and deleting cycles as above. Let's argue that it all takes $O(|E|)$ time.

Pick a vertex v and perform DFS from it until finding a back edge that links back to v. Once you find this cycle, traverse all edges of the cycle, and delete the corresponding edge in the adjacency list of G; to delete edges of G quickly assume we midify DFS so that we store, for each edge that we traverse, a pointer to the corresponding edge in the adjacency list of G. With this information, deleteion of an edge can be done in constant time, basically because we don't need to "search" for the edge in G. Then we repeat. Overall this $O(|E|)$ time.

4. (CLRS 22-4) Let $G = (V, E)$ be a directed grah in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, ..., |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V | u \text{ reaches } v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e. $\min(u)$ is the vertex v such that $L(v) = \min\{L(w)|W \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

   Solution: One solution is to compute the strongly connected components of the graph and erase all but the smallest label vertex in each component C; let this vertex be denoted $w(C)$. For every edge (u, v) with u not in the C and v in C add an edge (u, w). For every edge (v, u) with v in C and u not in C add an edge (w, u).(this process is called *contracting* C to a single vertex w). The resulting graph is a DAG. This DAG can be computed in $O(V + E)$ time (since strongly connnected components can be computed in $O(V + E)$ time). So we reduced the problem to the same problem on a DAG. Now it is simple: traverse the graph in reverse topological order. Initially every vertex has $min(u) = u$. For every vertex u look at its outgoing edges (u, v) and update $min(u) = \min\{min(v)|(u, v)\}$. Since We traverse vertices in reverse topological order all outgoing vertices (u, v) of u will have already found their final label $min(v)$.

   A much simpler way to solve this problem (without worrying about strongly connected compoenents) is to traverse the graph (either BS or DFS) but looking at the *incoming* edges rather than at outgoing edges, while processing vertices in increasing order of their label. The formal way to say this is as follows: compute a reverted graph G^T which is the same as G but with the direction of every edges reverted. This graph can be easily computed in linear time $O(V + E)$. Then

```
sort vertices in increasing order of their label
for each v in order do
    if v not black then BFS(v)
```

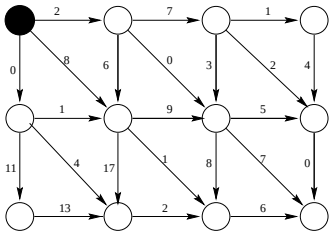
   That is, first perform $BFS(1)$; this will visit all vertices reachable from 1 in G^T (that is, which can reach 1 in G) and set their $min(u) = 1$. Then find the next smallest node that has not been reached in the previous BFS and start BFS from it, and so on.

   This in total takes $O(V)$ to sort the vertices (using a linear-time sorting algrithm) and $O(V + E)$ to do the graph traversal (BFS or DFS).

5. **Shortest path for Directed Acyclic Graphs (DAGs):** Let $G = (V, E)$ be a DAG and let s be a vertex in G. Find a linear time $O(|V| + |E|)$ algorithm for computing SSSP(s). What vertices are reachable from s? Sketch a proof that your algorithm is correct. Does your algorithm need the constraint that the edge weights are non-negative?

   Solution: Read the textbook.

6. Consider a directed weighted graph with non-negative weights and V vertices arranged on a rectangular grid. Each vertex has an edge to its southern, eastern and southeastern neighbours (if existing). The northwest-most vertex is called the root. The figure below shows an example graph with V=12 vertices and the root drawn in black:



Assume that the graph is represented such that each vertex can access **all** its neighbours in constant time.

(a) How long would it take Dijkstra's algorithm to find the length of the shortest path from the root to all other vertices?

Solution: Dijkstra's algorithm has running time $O(E \log V)$. In the graph described above, each vertex has at most three outgoing edges, so that the number of edges in the graph is at most $3V$, that is, $E = O(V)$. In this case, Dijkstra's algorithm will run in $O(V \log V)$.

(b) Describe an algorithm that finds the length of the shortest paths from the root to all other vertices in $O(V)$ time.

Solution: This question is easily answered if you realize the graph is a directed acyclic graph (or dag), since the SSSP problem can be solved on dags in $\Theta(V + E)$ time using e.g. the DAG-SHORTEST-PATHS algorithm. Because in this case we have $E = O(V)$, we can compute SSSP using DAG-SHORTEST-PATHS in $O(V)$.

You can also design your own $O(V)$ algorithm, in which case you must analyze its running time and prove correctness. Here is one example algorithm. The paths from the root to all vertices with in-degree 1 (first row, first column) are unique, so we can find the shortest paths traveling along the path from the root and summing the weights of the edges encountered along the way. Otherwise a vertex u has in-degree 3, that is, there are three predecessors $p_{u_1}, p_{u_2}, p_{u_3}$ of u. If we already know the shortest paths $\delta(r, p_{u_1})$, $\delta(r, p_{u_2})$, $\delta(r, p_{u_3})$ from the root r to the predecessors of u then the shortest path $\delta(r, u)$ from r to u is given by

$$\min\left\{\delta(r, p_{u_1}) + w(p_{u_1}, u),\ \delta(r, p_{u_2}) + w(p_{u_2}, u),\ \delta(r, p_{u_3}) + w(p_{u_3}, u)\right\},$$

where $w(p_{u_{(\cdot)}}, u)$ denotes the weight of the edge from a predecessor of u to u. Thus if all $\delta(p_{u_{(\cdot)}}, u)$ are already known, we can find the shortest path from the root at any vertex u in $O(1)$ time, and we perform this computation once for each vertex. To ensure we perform computations in the correct order (i.e. we know $\delta(r, p_{u_{(\cdot)}})$ before attempting to compute $\delta(r, u)$) we must first perform a topological sort of the vertices.

A topological sort takes time $O(V + E) = O(V)$ and our algorithm performs $O(1)$ work at each vertex, so the total running time is $O(V)$.

We now need to prove our algorithm correctly solves SSSP. For any vertex with in-degree 1 the path from the root is unique and therefore must be the shortest path. For all other vertices, we prove correctness by induction on a vertex v in the topological ordering of the vertices. The first vertex is the root, and the path from the root to itself is zero and therefore the shortest path. Assume that at a vertex v the shortest paths from the root to all vertices before v in the topological order are known. In particular, the shortest paths to all predecessors of v (of which there are three) are known. The only possible paths from the root to v must pass through a predecessor $p_{v_{(\cdot)}}$ of v, and from each predecessor there is only one possible (shortest) path to v (i.e. $\delta(p_{v_{(\cdot)}}, v) = w(p_{v_{(\cdot)}}, v)$, so that the possible shortest paths to v are in the set

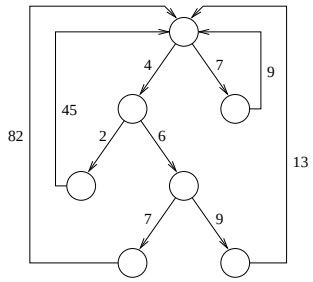
$$\left\{\delta(r, p_{u_1}) + w(p_{u_1}, u),\ \delta(r, p_{u_2}) + w(p_{u_2}, u),\ \delta(r, p_{u_3}) + w(p_{u_3}, u)\right\}.$$

The minimum value in this set must be the shortest path from r to v.

(c) Describe an efficient algorithm for solving the all-pair-shortest-paths problem on the graph (it is enough to find the length of each shortest path).

Solution: We can solve APSP by computing SSSP for each vertex in the graph. In (b) we gave an $O(V)$ algorithm to solve SSSP, thus we can solve APSP in $O(V^2)$. This is optimal, since there are $O(V^2)$ pairs in the graph.

7. Consider a directed weighted graph with non-negative weights which is formed by adding an edge from every leaf in a binary tree to the root of the tree. Let the graph/tree have n vertices. An example of such a graph with $n = 7$ could be the following:



We want to design an algorithm for finding the shortest path between two vertices in such a graph.

(a) How long time would it take Dijkstra's algorithm to solve the problem?

Solution: Dijkstra's algorithm has running time $O(E \log V)$. In this graph, each vertex has at most two outgoing edges, so that the number of edges in the graph is at most $2V$. Thus we have $E = O(V)$ and Dijkstra's algorithm will run in $O(V \log V)$.

(b) Describe and analyze a more efficient algorithm for the problem.

Solution: We first make some observations: There is exactly one vertex in the graph with in-degree > 1, call it r. We can identify r in $O(V)$ time by reading the adjacency-list of G. The shortest path $\delta(s, t)$ between two vertices s and t will then either not pass through r or pass through r. We can check which of these cases apply in $O(V)$ time using a modified graph search (BFS, DFS) at s which 'ignores' all edges (u, r).

Consider the two cases:

- $\delta(s, t)$ does not pass through r:
  If a path from s to t does not pass through r then it is unique, and moreover we can find it in $O(V)$ time – $\delta(s, t)$ is the sum of edge weights on the path from t back up to s.
  The algorithm is clearly correct as a unique path between two vertices must be the shortest one.
  We know the path for s to t passes through r, and that the path from r to t is unique. Then $\delta(s, t) = \delta(s, r) + \delta(r, t)$. This must give the shortest path from s to t, because $\delta(s, t)$ must go through r and subpaths of shortest paths are shortest paths. we already know how to find $\delta(r, t)$ in $O(V)$ from case (i).
  So we only need to find $\delta(s, r)$. There are at least two ways to do this. One way is to consider only the part of G consisting of the subtree rooted at s and the root node r. This subgaph is acyclic. We can apply the DAG-SHORTEST-PATHS algorithm at vertex s in $O(V)$ time.
  Another way is to compute $\delta(s, r)$ for node s from the values of its children in the tree in a dynamic-programming fashion, bottom-up starting from r.

Thus the total running time of the algorithm is $O(V + E) = O(V)$.

8. **All-Pair-Shortest-Paths with dynamic programming:** In the APSP problem, we want to compute the shortest path between any two vertices $u, v \in V$. Note that the output is of size $O(|V|^2)$ so we cannot hope to design a better than $O(|V|^2)$ time algorithm.

   (a) We can solve the problem simply by running Dijkstra's algorithm $|V|$ times. What is the running time of this approach? What does the running time become for sparse graphs ($E = \theta(V)$) and for dense graphs ($E = \theta(V^2)$)?

   Solution: Running Dijkstra from every vertex in the graph takes $V \cdot O(E \lg V)$ which is $O(EV \lg V)$. For a sparse graph this is $O(V^2 \lg V)$, and for a dense graph this is $O(V^3 \lg V)$.

   We can obtain another algorithm by working on adjacency matrix A. For weighted graphs, a_{ij} is equal to the weight w_{ij} of the edge (v_i, v_j); w_{ij} is assumed to be ∞ is the edge does not exist. Let A, B be two matrices, and let $C = A \cdot B$. Remember that

   $$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

   We redefine the $\sum$ and $\cdot$ operators in matrix multiplication to mean *minimum* and $+$ respectively. That is,

   $$c_{ij} = min_{k=1..n}\{a_{ik} + b_{kj}\}$$

   (b) What does $A \cdot A$ represent in terms of paths in graph G? What about $\min\{A, A \cdot A\}$?

   Solution: You can also find this in the textbook, in the section for all-pairs shortest paths. An entry c_{ij} in A^2 represents the weight of the shortest path consisting of exactly two edges from v_i to v_j. An entry c_{ij} in $\min\{A, A \cdot A\}$ represents the weight of the shortest path consisting of ≤ 2 edges from v_i to v_j.

   (c) Sketch an algorithm for computing APSP using this approach and estimate its running time.

   Solution: You can also find this in the textbook. The shortest path between two vertices cannot visit a vertex more than once (if it did, it would create a loop; by eliminating the loop from the path we would get a shorter path—contradiction). So a shortest path can have at most V vertices, or V-1 edges. In the spirit of part (b) above, compute:
   $B_2 = min\{A, A^2\}$: B_2 represents shortest distances among all paths of ≤ 3 edges.

   $B_3 = min\{B_2, A^3\}$: B_3 represents shortest distances among all paths of ≤ 3 edges.

   ...

   min $\{B_{V-1}, A^V\}$: represents shortest distances among all paths of $\leq V$ edges. Since a shortest path has $\leq V - 1$ edges, this matrix represents APSP.
   Analysis: Computing each of $A^2, A^3, ...$ can be done in $O(V^3)$ time. So overall we get $O(V \cdot V^3) = O(V^4)$ time. It is also possible, with a little trick, to compute A^k by computing only $\lg_2 k$ powers of A; thus computing A^V and B_{V-1} can be done in $V^3 \lg v$ time.