CS521 Fall 2011 \ Assignment #4

Ariel Stolerman

1) CLRS Page 654, Exercise 24.1-3

Let $G = (V, E)$ be a weighted directed graph with no negative-weight cycles over a weight function $w: E \to \mathbb{R}$.

Let $m = \max\{\min - \text{num} - \text{of} - \text{edges}(w - shortest - paths(s, v)) \mid v \in V\}$.

Following is a suggestion of a change in the Bellman-Ford algorithm that terminates in $m + 1$ passes without knowing m in advance:

Since we know the maximum size of any shortest path from s to any $v \in V - \{s\}$ is m, it is sufficient to stop after m passes as no d-values will change after m passes – that is all shortest paths have been found. In that case we are guaranteed that no d (and π) values will be changed after m iterations. Since m is not known in advance we can simply track if changes are done anywhere along current iteration, and if no changes to any of the d-values occur, we can stop – that will happen of course at the $m + 1$ iteration.

The changes are as follows:

- In the initialization process add another field $changes - occured$ and initialize it to $true$.
- Change $Relax(u, v, w)$ as follows:

  $if\ v.d > u.d + w(u, v)$:
  $\quad v.d = u.d + w(u, v)$
  $\quad v.\pi = u$
  $\quad changes - occured = true$
- Change the relaxation for loop in the main procedure of Bellman-Ford to the following $while$ loop:

  $while\ changes - occured == true$:
  $\quad changes - occured = false$
  $\quad for\ each\ (u, v) \in G.E$:
  $\quad\quad relax(u, v, w)$

At the $m + 1$ iteration no changes will occur by any of the modified $relax$ calls, thus the next condition check in the $while$ loop will fail and the passes will stop.

2) CLRS Page 655, Exercise 24.1-6

Let $G = (V, E)$ be a weighted directed graph with a negative-weight cycle over a weight function $w: E \to \mathbb{R}$. Following is an efficient algorithm to list the vertices of one such cycle (a variation of the original Bellman-Ford algorithm):

- In the initialization process add another bit $v.c$ for all $v \in V$ that will later hold 1 if this node is part of a negative-cycle and 0 otherwise. Initialize all $v.c = 0$ for all $v \in V$.

  In addition initialize an empty negative-cycle list $cycle = \{\}$.

- Change the part in Bellman-Ford that checks for a negative cycle (as we know for sure there will be one) as follows:

```
for each (u,v) ∈ G.E:
    if v.d > u.d + w(u,d):
        v.c = 1
        curr = v
        cycle.append(v)
        while curr.π.c == 0:
            curr = curr.π
            curr.c = 1
            cycle.append(curr)
        return cycle
```

(if the graph doesn't contain a negative cycle, then at this level we can return $cycle == \{\}$).

Correctness:

By the correctness of Bellman-Ford we know that after the $n-1$ passes over all $e \in E$ and relaxations, if there isn't a negative cycle in the graph then $v.d > u.d + w(u,d)$ should be $false$ for all $(u,v) \in E$. However, if there is a negative cycle, the first $(u,v) \in E$ that evaluates $v.d > u.d + w(u,v)$ to $true$ must be an edge in a negative cycle:

- If (u,v) is not in a negative cycle or affected by one, the condition will evaluate to $false$.
- If (u,v) is not in a negative cycle but IS affected by one, since we stopped updating d-values, the condition cannot evaluate to $true$ (because last time when $v.d > u.d + w(u,v)$, we updated $v.d = u.d + w(u,v)$, so no matter the value of $w(u,v)$, the ">" will not hold again).
- The condition may apply only when (u,v) is on a negative cycle, as it is guaranteed that if we have a negative cycle the condition will be $true$ (derives from the correctness of the finding negative-cycles in the original BF), and since it cannot be any other edge (as presented above), it must be on a negative cycle.

Furthermore, it is guaranteed that by the end of the BF relaxation passes, for each negative cycle $C \subset V$: $\forall v \in C: v.\pi \in C$ since going through the cycle as many times as we want assures reducing $v.d$ for each $v \in C$, thus $v.\pi$ must be the predecessor of v in the cycle (could be overlapping cycles). Therefore once we find (u,v) that satisfies the condition of the if, going through the π path from v will assure:

- We will go through the entire negative cycle.
- Marking $v.c = 1$ will assure we will stop when all the cycle is covered.

Thus $cycle$ will eventually contain all vertices in a negative cycle (or stay empty if none exists).

Running-time:

The initialization process and BF passes stay with the same running time as the original BF. Finding (u,v) that satisfies the if condition is $O(m)$ and finding the cycle's members is then $O(n)$. The total running time is therefore like BF, which is $\Theta(nm)$ (which is $O(n^3)$).

3) CLRS Page 658, Exercise 24.2-4

Following is an algorithm to count the total number of paths in a DAG:

$\underline{Find - total - paths(G)}$:

$topologically\ sort\ all\ vertices\ of\ G.V\ and\ then\ REVERSE\ order$

$for\ all\ v \in V$:
    $v.p = 0$ // will hold the total number of paths from this node and on
$for\ each\ u \in V\ (taken\ in\ reversed\ topologically\ sorted\ order)$:
    $for\ each\ v \in G.Adj[u]$:
        $u.p = u.p + (v.p + 1)$
$total = 0$
$for\ each\ u \in V$:
    $total = total + u.p$
$return\ total$

Correctness:

When topologically sorted, for an edge (u, v) the total number of paths from u that go through this edge is the total number of paths from v plus 1 – the path $u \to v$ which is the edge (u, v) itself. Summing from the last node in topological order (with value 0, as no paths start from it), each node will eventually hold as p-value the total number of paths that start from it and go through each of its edges. Since we go in reversed-topological order, it is guaranteed that we miss no paths to count for the current u checked. Eventually, summing all p-values will give the total number of paths in the graph.

If the graph would have had a cycle, we could immediately determine the answer is ∞ (go through a cycle as many times as we like to generate as many paths as we like).

Running-time:

Topologically sort and reverse the order takes $\Theta(n + m)$ (as shown for instance for the DAG-shortest-paths algorithm). The initialization of the p-values is $\Theta(n)$. The update of all p-values for all nodes is $\Theta(m)$. Finally the creation of $total$ is $\Theta(n)$. Therefore the total is $\Theta(n + m)$.

4) CLRS Page 663, Exercise 24.3-6

Let $G = (V, E)$ be a directed graph, and let $r: E \to [0,1]$ be a *reliability* function from the source node to the destination node of each edge, i.e. $\forall (u, v) \in E: r(u, v) = \Pr[channel\ from\ u\ to\ v\ will\ not\ fail]$. Furthermore, these probabilities are independent.

Following is an efficient algorithm to find the most reliable path between two given vertices. For that we first define a modified Dijkstra's algorithm as follows:

$modified - Dijkstra(G, w, s)$:
$for\ each\ v \in G.V$:
    $v.d = 0$
    $v.\pi = NIL$
$s.d = 1$
$S = \emptyset$
$Q = G.V$

$while\ Q \neq \emptyset$:
    $u = extract - \min(Q)$
    $S = S \cup \{u\}$
    $for\ each\ v \in G.Adj[u]$:
        $if\ v.d < u.d \times w(u,v)$:
            $v.d = u.d \times w(u,v)$
            $v.\pi = u$

Then our algorithm would be:

$find - reliable - path(G, r, s, t)$:

    $run\ modified - Dijkstra(G, r, s)$

    $find\ the\ \pi - path\ from\ t \to s\ and\ return\ its\ inverse$

Correctness:

First, since the probabilities r are independent, then the reliability of a path $p =< v_1, \dots, v_k >$ equals the product of the probabilities of each edge in the path, i.e. $\prod_{i=1}^{k-1} r(v_i, v_{i+1})$. The modified Dijkstra does exactly the same as the original Dijkstra, only instead of keeping minimums, it keeps maximums, and instead of keeping sums – it keeps products. Note that since all r-values are non-negative, the correctness stays the same as for the original Dijkstra.

The initialization of $v.d = 0$ to all $v \in V - \{s\}$ adjusts the initial values to fit the task of finding a maximum rather than a minimum, and initializing $s.d = 1$ makes sure the first product won't be 0, and affect the followings (also, the reliability of a path from $s \to s$ is 1: no path at all).

After applying the modified Dijkstra, all is left is to find the π-path from t to s and return its inverse – as that is the path from s to t that produces the maximum reliability.

Another approach is to use the original Dijkstra with the weight function $w(u,v) = -\log\big(r(u,v)\big)$, as minimizing $\sum -\log\big(r(u,v)\big)$ is the same as maximizing $\sum \log\big(r(u,v)\big)$ which is the same as maximizing $\log \prod r(u,v)$ which is the same as maximizing $\prod r(u,v)$ – what we're looking for.

Running-Time:

When using a Fibonacci heap for the priority queue implementation, the modified Dijkstra runs the same as the original Dijkstra, i.e. $O(n \lg n + m)$. Finding the inverse π-path is another $O(n)$ (no cycles can contribute the reliability of a path). Therefore the total running time is $O(n \lg n + m)$.

I don't give a tight bound since the book shows an O-bound for the original Dijkstra algorithm.

5) CLRS Page 692, Exercise 25.1-8

Following is a modification of the $faster - all - pairs - shortest - paths$ algorithm to use only $\Theta(n^2)$ space. First we define the procedure $extend - shortest - paths(L', L, W)$ in a similar way as the original $extend - shortest - paths(L, W)$, only instead of allocating a new $n \times n$ matrix L', it uses the given L' and updates its values. Since we have $l'_{ij} = \infty$ as an initialization as part of the original $extend - shortest - paths$, no need to do anything further to L' when sent to $modified - extend - shortest - paths$.

We then we define the modification of the $faster - all - pairs - shortest - paths$ algorithm as follows:

$modified - faster - all - pairs - shortest - paths(W)$:
$n = W.rows$
$initialize\ two\ n \times n\ matrices$:
    $L_0 = W$
    $L_1 = empty$
$i = 0$
$m = 1$
$while\ m < n - 1$:
    $modified - extend - shortest - paths\left(L_{(i+1)mod\ 2}, L_{i\ mod\ 2}, L_{i\ mod\ 2}\right)$
    $m = 2m$
    $i = i + 1$
$return\ L_{(i+1)mod\ 2}$

Correctness:

The correctness derives from the correspondence to the original algorithm. At each iteration we update $L_{(i+1)mod\ 2}$ by multiplying $L_{i\ mod\ 2}$ with itself, and incrementing i makes sure the last iteration's updated matrix is the source for calculating the values of the matrix in the current iteration. We start correct since the first iteration with $i = 0$ uses L_1 as target and $L_0 = W$ is used for calculation.

Finally, the last matrix that was updated is the one returned as the answer – corresponding to the $L^{(m)}$ matrix returned in the original algorithm.

Space-wise we only use 2 $n \times n$ matrices, L_0 and L_1, as required.

6) CLRS Page 699, Exercise 25.2-5

If in the case where $d_{ij}^{(k-1)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ $(k \geq 1)$ we set $\pi_{ij}^{(k)}$ to be $\pi_{kj}^{(k-1)}$ instead of $\pi_{ij}^{(k-1)}$, the predecessor matrix Π would still be correct. Recall that when $k \geq 1$, $d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$, so in case of equality it doesn't matter which of the two paths will be taken, as the weight of both paths is the same. Essentially it means we would go from i to j through k instead of not through k, but in this case both are valid shortest paths, meaning the Π matrix would still hold shortest paths.

7) CLRS Page 700, Exercise 25.2-9

Say we have an algorithm that computes the transitive closure of a DAG in $f(n, m)$ ($|V| = n, |E| = m$) (is monotonically increasing). Let $G = (V, E)$ be a general directed graph, and let $G^* = (V, E^*)$ ($|E^*| = m^*$) be its transitive closure. Following is an algorithm to compute G^* in $f(n, m) + O(n + m^*)$. We use the strongly-connected-components graph of G, $G^{SCC} = (V^{SCC}, E^{SCC})$, where each $C \in V^{SCC}$ will actually be a set of nodes $v \in V$ that belong to that component.

$find - transitive - closure(G)$:
$construct\ for\ G\ the\ strongly\ connected\ components\ graph\ G^{SCC} = (V^{SCC}, E^{SCC})$
$initialize\ E^* = \{\}$
$for\ each\ C \in V^{SCC}$:
    $for\ each\ pair\ u \in C$:
        $for\ each\ v \in C - \{u\}$: // if we want self-edges in E^*, use C instead of $C - \{u\}$
            $E^* = E^* \cup \{(u, v)\}$

$G^{SCC*} = find - DAG - transitive - closure(G^{SCC})$
$for\ each\ (C_1, C_2) \in E^{SCC*}$:
    $for\ each\ u \in C_1$:
        $for\ each\ v \in C_2$:
            $E^* = E^* \cup \{(u, v)\}$
$return\ G^* = (V, E^*)$

Correctness:

We know that every pair of nodes in that belong to the same strongly connected component have a path from one to the other and vice versa, hence the first addition loop for E^* it correct. Furthermore we know that if $(C_1, C_2) \in E^{SCC*}$ then there's a path from C_1 to C_2 in G^{SCC}, therefore there are $u \in C_1, v \in C_2$ such that there's a path from u to v ($u, v \in V$). Since u, v are each connected to all nodes in their components C_1, C_2 respectively, then there's a path from each $u \in C_1$ to each $v \in C_2$, so the second addition loop for E^* is also correct. Finally, if there is no path from some $u \in V$ to some $v \in V$, then u, v cannot be in the same connected component, and there could not be a path from u's component C_1 to v's component C_2, hence $(C_1, C_2) \notin G^{SCC*}$ - and the algorithm above doesn't take any such edges into E^*.

Running-time:

Constructing G^{SCC} takes $O(n + m)$, and $|V^{SCC}| = O(n), |E^{SCC}| = O(m)$ thus finding the transitive closure of G^{SCC} is $O\big(f(n, m)\big)$. For each edge that is to be added to E^* we go through exactly once, and we don't check any other edges, so the first and second addition loops together take $O(m^*)$. Since $m \leq m^*$, the total running time is $f(n, m) + O(n + m^*)$, as required.

8) CLRS Page 397, Exercise 15.4-5

Following is an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers:

- Copy the input X into Y and sort Y.
- Return $LCS(X, Y)$

Correctness:

Finding the largest subsequence of X in its sorted copy Y will give the largest monotonically increasing subsequence of X since that subsequence, and any monotonically increasing subsequence, is by itself a sorted sequence. Therefore Y will actually hold that subsequence in the correct order, but might have extra elements inserted between elements of the largest subsequence (elements that originally in X were in other positions). Therefore LCS solves the problem of finding the largest match in X of a subsequence in Y, meaning the largest subsequence of monotonically increasing values.

Running-time:

The copy procedure takes $\Theta(n)$, followed by $O(n \lg n)$ for sorting Y. Then applying LCS on two inputs of size n takes $O(n^2)$. The total running time is then $O(n^2)$.