CS 521: Data Structures and Algorithms I
Homework 4

Dustin Ingram

November 22, 2011

1. **Solution:** Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, $m + 1$ would also represent the number of iterations of the Bellman-Ford algorithm, as this would mean that some shortest path from s to any vertex v has already been found, and any subsequent iterations would not modify the current paths. The RELAX function can be modified to report changes as follows:

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v) then
    v.d = u.d + w(u, v)
    v.π = u
    return TRUE
  return FALSE
```

   Therefore, modifying the Bellman-Ford algorithm as follows to terminate after an iteration where no edges were relaxed would terminate the algorithm after m total iterations.

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for each edge (u, v) ∈ G.E do
    if RELAX(u, v, w) == FALSE then
      BREAK
```

2. **Solution:** For a graph which has a negative-weight cycle, we can run the Bellman-Ford algorithm once to produce the "distance" values and parent values for each node in the graph.
   If we were to run the Bellman-Ford algorithm again on the graph, with the $v.d$ and $v.\pi$ values set from the previous iteration (instead of being re-initialized), these values would not change. However, if there is some negative-weight cycle in the graph, some vertex's $v.d$ and $v.\pi$ will change.

Once this is detected (by a successful call to RELAX, which makes a change to these values), we can travel back through the negative-weight cycle from v to $v.\pi$, etc., until we return to v, at which point we have listed all the vertices in the cycle.
The complexity of this modified algorithm is bounded by running the Bellman-Ford algorithm twice, then traversing a cycle of at most V vertices, for an overall runtime of $O(2(VE) + V) = O(VE)$.

3. **Solution:** This solution iterates over each edge $(u, v) \in G(V, E)$. Each vertex holds a variable $v.p$, which represents the number of paths to that vertex at any given moment. As each new edge is added, the number of paths to the 'destination' edge increases by the number of paths to the 'origin' edge, plus one (for the current edge). After the algorithm finishes, the total number of paths is the sum of all paths to each vertex. The algorithm instead maintains *total* as it iterates over the edges; therefore the complexity is $O(E)$.

TOTAL-DAG-PATHS(G)

```
total = 0
for each edge (u, v) ∈ G(V, E) do
    v.p += u.p + 1
    total += u.p + 1
return total
```

4. **Solution:** Here, we can use Dijkstra's algorithm (modified to use EXTRACT-MAX instead) and a modified version of RELAX as follows:

RELAX(u, v, r)

```
if v.r < u.d × r(u, v) then
    v.r = u.r × r(u, v)
    v.π = u
```

Instead of minimizing the weight of a path (sum of weight of edges), this modified algorithm will maximize the reliability of a path (product of reliabilities of edges).

5. **Solution:** The solution is trivial: Instead of creating a new matrix for every iteration of the `while` loop, we keep two copies, $L^{(a)}$ and $L^{(b)}$, and use the latter to store the result of a call to EXTEND-SHORTEST-PATHS using the former, then update it with the result.

SMALLER-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(a)} = W$
$L^{(b)} = W$
$m = 1$
while $m < n - 1$ **do**
  $L^{(b)} =$ EXTEND-SHORTEST-PATHS$(L^{(a)}, L^{(a)})$
  $L^{(a)} = L^{(b)}$
  $m = 2m$
return $L^{(a)}$

This uses only two matrices for a space complexity of $\Theta(n^2)$.

6. **Solution:** This modification of equation (25.7) in fact reveals a third case for the assignment of $\pi_{ij}^{(k)}$, which can be interpreted as follows:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} \textbf{ OR } \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

The last case simply represents the case where the existing path $i \rightsquigarrow j$ is the exact same distance as the new path $i \rightsquigarrow k \rightsquigarrow j$. In the original version of the equation, the algorithm kept the shortest path which was first found. The modified equation updates the path to use the edge $k \rightsquigarrow j$. The alternative definition of the predecessor matrix Π is correct, but it will produce a different predecessor matrix if the third case is encountered.

7. **Solution:** We can generate the strongly connected component graph G^{SCC} from the general directed graph G in $O(V+E)$ using the STRONGLY-CONNECTED-COMPONENTS algorithm. Since G^{SCC} is a DAG by definition, we can then use the $f(|V|, |E|)$-time algorithm to compute G^+, the transitive closure of G^{SCC} (which holds, because G^{SCC} can have at most V vertices and E edges). Then, assuming that we have maintained knowledge of which original vertices in G belong to which strongly connected component in G^{SCC} (now represented by vertices in G^+), for each vertex v, we draw an edge between v and each vertex in its strongly connected component, as well as between v and each vertex in an SCC neighboring its own SCC. Since there are at most V vertices and E^* possible edges in the resulting graph, this leaves us with a running time of $f(|V|, |E|) + O(V + E^*)$.

8. **Solution:** This algorithm maintains an array S' as it iterates through all possible subsequences, where $S'[k]$ represents the size of the longest monotonically increasing subsequence ending at $S[k]$.

```
LONGEST-MONOTONICALLY-INCREASING-SUBSEQUENCE(S)
  max = 0
  S'[0...(n-1)] = 0
  for k = 0 to (n-1) do
    max' = 0
    for j = 0 to (k-1) do
      if S[k] > S[j] then
        max' = MAX(max', S'[j])
    S'[k] = max' + 1
    max = MAX(max, S'[k])
  R[0...(max-1)] = 0
  for i = (n-1) to 0 do
    if S'[i] = max then
      max = max - 1
      R[max] = S[i]
  return R
```

Once S' is computed, it is necessary to traverse backwards through S' to fill R, which is the result array.