CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 11
Submit via Gradescope

Name:Shri Datta Madhira (NUID: 001557772)
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(25 points)*** *Following the notes/slides/book, explain All-Source-Shortest-Paths by edges DP using matrix multiplication trick. Write pseudocode for ASSP-Fast and the corresponding Extending-SP procedures.*

Solution:

Step 1: Characterizing the problem

For the all-pairs-shortest-paths problem on a graph G = (V,E) we have proven that all sub-paths of a shortest path are shortest paths.
Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most m edges. If vertices i and j are distinct, then we decompose path p into i $\rightsquigarrow$ k -> j (assume this is p'), where path p' now contains at most m - 1 edges.

Step 2: Recurrence Objective

Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. Then,

$$l_{ij}^{(m)} = \begin{cases} \infty & \text{if i} \neq \text{j}, \\ 0 & \text{if i = j} \end{cases}$$

For m≥1, we compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j. We recursively define this as,

$$l_{ij}^{(m)} = min(l_{ij}^{(m-1)}, min(l_{ik}^{(m-1)} + w_{kj}))$$

Now we can see the relation to matrix multiplication. Suppose we wish to compute the matrix product C = A.B of two nxn matrices A and B. Then, for i, j = 1,2,...,n; we compute

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

Thus, if we make these changes to EXTEND-SHORTEST-PATHS and also replace ∞ (the identity for min) by 0(the identity for +), we obtain the same $\Theta(n^3)$ time procedure for multiplying square matrices.

Our goal, however, is not to compute all the $L^(m)$ matrices: we are interested only in matrix $L^{(n-1)}$. Just as traditional matrix multiplication is associative, so is matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure. Therefore, we can compute $L^{(n-1)}$ with only $\lceil(\lg n)\rceil$ matrix products by computing the sequence. This technique is called "repeated squaring". The FAST-ASSP Program below will implement this idea.

Step 3: Algorithms for the problem

Algorithm 1: Fast ASSP and Exted-SP Procedure.

```
Function FAST-ASSP(W):
    n = W.rows
    L^(1) = W
    m = 1
    While m < n-1 :
        let L^(2m) be a new nxn matrix
        L^(2m) = EXTEND-SHORTEST-PATH(L^(m), L^(m))
        m = 2m
    Return L^(m)
Function EXTEND-SHORTEST-PATH(A, B):
    n = A.rows
    let C be a new nxn matrix
    For i = 1 to n :
        For j = 1 to n :
            c_ij = 0
            For k = 1 to n :
                c_ij += (a_ik.b_kj)
    Return C
```

2. ***(25 points)*** *Exercise 24.1-3.*

Solution:

In the original algorithm we can put a flag to check whether the relaxation values changed from the previous iteration to the current iteration. If at least one of those values changed, then we continue the process. If everything remains the same, we immediately terminate. Because, if no value changes in the iteration, then there won't be any change in the successive iterations as well.

Algorithm 2: Modified Bellman-Ford Algorithm.

```
Function BELLMAN-FORD-MODIFIED(G,w,s):
    Initialize-Single-Source(G, s)
    For i = 0 to |G.V|-1 :
        count = 0
        For (u,v) in G.E :
            change = RELAX(u,v,w)
            If change :
                continue
            Else
                count += 1
        If count == G.E.length :
            break
Function RELAX-MODIFIED(u,v,w):
    changed = False
    If v.d > u.d + w(u,v) :
        v.d = u.d+w(u,v)
        v.π = u
        changed = True
    Return changed
```

We did not include the negative cycle check as there won't be any of those kind as mentioned in the question itself.

The algorithm works by incrementing the count by one for every time an edge relaxation isn't changed. If that count equals the number of edges, then none of the edges changed, which means the relaxation state has been achieved, so we break out of the outside loop.

3. ***(25 points)*** *Exercise 24.2-2.*

Solution:

Since we are topologically sorting the vertices, the vertices which have an outgoing path will come before the vertices with 0 outgoing paths. So, the vertex without any outgoing edge will end up in last place in the list. So, we are visiting the first $|V|-1$ vertices to get the relaxed stage.

Therefore, even if you change line 3 from ***for each vertex u, taken in topologically sorted order*** to ***for the first $|V|-1$ vertices, taken in topologically sorted order***, it will not change the working of the algorithm because they mean the same thing as discussed above.

4. ***(25 points)*** *Exercise 24.3-4.*

Solution:

We first check the base conditions, s.d = 0, s.π = NIL and that v.d = v.π + w(v.π, v) $\forall$ u where (u,v) $\in$ E and v $\neq$ s. If these conditions fail, return false(algorithm is wrong). If these conditions pass, then we have to check if d and π attributes match those of some shortest-paths tree.

For this we can run the Bellman Ford Algorithm for a single iteration to see if any of the edges relax in that iteration. If the output of Professor Gaedel's algorithm is correct, then the edges should not relax. In other words, d and π attributes should not change. If they didn't then the algorithm is correct. Else, the algorithm is wrong. The algorithm below runs in O(V+E) time.

Algorithm 3: Algorithm for transitive closure of a directed graph

```
Function GAEDEL-ALGO-CHECK(L, w, s):
    For v = 1 to V :
        If v == s :
            If s.d != 0 OR s.π != NIL :
                Return False
        Else
            If v.d != v.π.d + w(v.π, v) :
                Return False
            If v.d == ∞ AND v.π == NIL :
                Return False
    For edge (u,v)∈E :
        changed = RELAX(u,v,w)
        If change :
            Return False
    Return True
Function RELAX(u,v,w):
    changed = False
    If v.d > u.d + w(u,v) :
        v.d = u.d+w(u,v)
        v.π = u
        changed = True
    Return changed
```

5. ***(Extra Credit 30 points)*** *Problem 24-2.*

Solution:

6. ***(30 points)*** *Explain in few lines the concept of transitive closure.*

Solution:

Given a directed graph G = (V,E) transitive closure of a directed graph determines whether G contains a path from i to j for all vertex pairs $(i, j) \in V$. The transitive closure of $G^* = (V, E^*)$ can be defined as,

$$E^* = (i, j) : there\ is\ a\ path\ from\ vertex\ i\ to\ vertex\ j\ in\ G$$

We can use the Floyd Warshall Algorithm to compute the transitive closure for a directed graph. The space and time complexities will be $\Theta(n^3)$. We can reduce the space complexity by replacing the $min()$ and + operations with logical OR($\vee$) and AND($\wedge$). A recursive definition of $t_{ij}^{(k)}$ in $T^{(k)}$ is given by,

$$t_{ij}^{(k)} = \begin{cases} 0 & \text{if i} \neq \text{j and (i,j)} \notin \text{E,} \\ 1 & \text{if i = j or (i,j)} \in \text{E} \end{cases}$$

and for k $\geq$ 1,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Because the direct transitive-closure algorithm uses only boolean values rather than integer values, it is better to use logical operators to improve space and time complexity of the base algorithm. The algorithm for transitive closure is as follows,

Algorithm 4: Algorithm for transitive closure of a directed graph

```
Function TRANSITIVE-CLOSURE(G,w,s):
    n = |G.V|
    T^(0) = (t_ij^(0))
    For i = 1 to n :
        For j = 1 to n :
            If i == j or (i,j) ∈ G.E :
                t_ij^(0) = 1
            Else
                t_ij^(0) = 0
    For k = 1 to n :
        T^(k) = (t_ij^(k))
        For i = 1 to n :
            For j = 1 to n :
                t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
    Return T^(n)
```

7. *(20 points)* *Exercise 25.1-6. (the book uses different notation for the matrices). Also explain how to use this result in order to display all-pair shortest paths, enumerating intermediary vertices (or edges) for each path.*

Solution:

Algorithm 5: Algorithm for calculating predecessor matrix (according to matrices in the book).

```
Function PREDECESSOR-MATRIX-CALC(L, W):
    Π[1...n, 1...n] = ∞
    For i = 1 to n :
        For j = 1 to n :
            If i ≠ j :
                For k = 1 to n :
                    If L[i,k] + w[k,j]  ==  L[i,j] and W[k,j] ≠ ∞ and W[k,j] ≠ 0 :
                        Π[i, j] = k
            Else
                Π[i, j] = i
    Return Π
```

Algorithm 6: Algorithm for calculating paths from predecessor matrix (according to matrices in the book).

```
Function GET-PATHS(Π):
    For i = 1 to n :
        For j = 1 to n :
            print(path from i->j is:)
            path = STACK()
            prev = ∞
            While True :
                pred = Π[i, j]
                If pred == prev :
                    break
                Else
                    path.PUSH(j)
                    prev = j
                    j = pred
            print(path.POP() till the path is empty)
```

8. ***(20 points)*** *Exercise 25.2-1.*

Solution:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix} \quad D^{(1)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$$D^{(6)} = \begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

EMPTY SLOT

An example calculation,

$$D^1[2,5] = min(D^0[2,5],\ D^0[2,1] + D^0[1,5]) = (\infty, 1 + (-1)) = 0$$

9. ***(20 points)*** *Exercise 25.2-4.*

Solution:

In the original Floyd Warshall Algorithm for all source shortest path, we use a seperate matrix $D^{(k)}$ for each iteration of k. By this method it takes n^2 space for n iterations of k. That means, a total of $\Theta(n^3)$ space complexity.

But we can decrease this by using a single matrix for all the iterations. This can be done just by dropping the superscript over D. So, there is only one D matrix for the entire process.

The algorithm works correctly because, even in the original process, we are using the modified matrix(matrix from the previous round) for the calculations in the current round. For example, to calculate $D^{(2)}$ matrix, we use $D^{(1)}$ matrix.

In the new method, by dropping the superscript we are modifying the values in-place, and after one complete iteration of the matrix, we essentially have the modified matrix. So, we still got the $D^{(2)}$ matrix using the $D^{(1)}$ matrix, but all the values are in D. And, if we are using 'k' as an intermediary between i and j, then the values of d_{ik} will not change because, it cannot be updated as d_{ik} is the shortest path to k. So, the algorithm given in the question works fine.

10. ***(Extra Credit 20 points)*** *Exercise 25.2-6*

Solution: