CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 11
Submit via Gradescope

Name: Xuran Feng
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:ASSP by edges adopts dynamic programming technique. The optimal substructure is defined as subpaths of shortest paths are shortest paths. Let $l_{ij}^{(m)}$=weight of shortest path from i to j that contains ≤m edges. When m==1, $L^{(1)} = W$; when m≥2, $l_{ij}^{(m)} = min_{1 \le k \le n}\{l_{ij}^{(m-1)} + w_{kj}\}$ when k is all possible predecessors of j. For each edge, we iterate the whole matrix and update each vertex based on all possible predecessors.

EXTEND(L,W,n)
create L'
for i=1:n
    for j=1:n
        $l'_{ij} = \infty$
        for k=1:n
            $l'_{ij} = min(l'_{ij}, l_{ik} + w_{ij})$
return L'

If we do a normal outer loop iteration:

SLOW-APSP(W,n)
$L^{(1)} = W$
for m=2 to n-1:
    $L^{(m)}$=EXTEND($L^{(m-1)}, W, n$)
return $L^{(n-1)}$

Our goal is to obtain $L^{(n-1)}$ and we don't need to compute $L^{(3)}$, $L^{(5)}$... to just obtain $L^{(n-1)}$, instead we can just continue computing the square to speed up the process to obtain $L^{(n-1)}$.

FAST-APSP(W,n)
$L^{(1)} = W$
m=1
while m<n-1:
    $L^{(2m)}$=EXTEND($L^{(m)}, L^{(m)}, n$)
    m=2m
return $L^{(n-1)}$

The idea to use matrix multiplication trick is because EXTEND can be considered as a multiplication of two matrix L and W, since the SLOW-APSP each time passes a constant matrix W into the EXTEND to do multiplication with previous L, we could accumulate the multiplication result since matrix multiplication are associative to speed up the process.

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

Solution: For Bellman-Ford algorithm, the reason why outer loop needs to run $|V|-1$ times is because the longest path from source to a vertex is at most $|V|-1$ edges and $|V|-1$ times of running RELAX-ALL-EDGES will ensure all vertices find their own shortest path from source; however, we may don't need to run as many as $|V|-1$ times to know all vertices are done given that no negtive-weight cycle exist. When all vertices are done, the next running of RELAX-ALL-EDGES will never change the v.d, in other words, the v.d will remain unchanged for all vertices for the following RELAX-ALL-EDGES runnings, by checking this, we can know this is true end of Bellman-Ford

algorithm. If there exists negative-weight cycle, some v.d will keep changing because they can always get updated, so in this case we have to run as many as $|V|-1$ times. But when there is no negative-weight cycle, we can stop iterating in (m+1) passes.

```
Bellman-Ford(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    stillUpdate=true
    while stillUpdate==true
        stillUpdate=false
        for each edge(u,v) belonging to G.E
            RELAX-ALL-EDGES(u,v,w)
RELAX-ALL-EDGES(u,v,w)
    if v.d>u.d+w(u,v)
        v.d=u.d+w(u,v)
        v.p=u
        stillUpdate=true
```

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

Solution: The algorithm is still correct, because the last vertex doesn't have any successor so there are no vertices needing to be updated when this last vertex is taken out, in other words, no other vertices will rely on the last vertex to get a cheaper path because all other vertices are already updated before the last vertex, so taking out the last vertex or not will make no difference. And it is known that when the second last vertex is taken out, if there is a cheaper path from the second last to the last, the last vertex will automatically updated; if not, the last vertex will remain as before the second last vertex being taken out, no matter which case, the edges of last vertex will be relaxed completely so the last vertex will get its correct predecessor and distance information. So the last vertex in fact needs not to be taken out because no other information it will provide in finding a shorter path.

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

Solution: For each vertex v, among its all incoming edges, finding the minimum w(u,v)+u.d and verify w(u,v)+u.d==v.d and verify v.p=u; there should be only one vertex which has no predecessor and its d==0 since it is the source vertex. We may worry that the v.p may not have correct v.p.d, however at least we know v is appropriately updated because we iterate every possibility of v; so now we can move focus to v.p, and since we iterate every possibility of v.p too, we continue moving focus to v.p.p...; for each iteration, if current v.d!=min(w(u,v)+u.d) for all incoming edges or v.p!=u if u is indeed the predecessor, we return false; otherwise we can continue iterating all vertices. As long as the current phase is correct, then checking the predecessor phase is all we need to check. Iterating all vertices and their own incoming edges will give a $O(V+E)$ time complexity.

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

Solution:

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

Solution: Transitive closure of a graph describes reachability between any two vertices using a matrix. If the matrix cell[i][j]==1, this means there is a path between vertex i and vertex j; else the cell value is 0, then there is no such a path between vertex i and vertex j. The transitive closure can be used to find connected component quickly. We can use Floyd-Warshall algorithm to fill in the reachability matrix by assigning each edge weight 1 since Floyd-Warshall algorithm overshoots All-Pairs reachability.

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: To enumerate the whole matrix, we need two nested loops, which takes $O(n^2)$ running time, for each cell (i,j), we need to enumerate each vertex k again to find the vertex that satisfies $L_{i,k} + w(k, j) = L_{k,j}$, and mark this k as the cell value of $\Pi(i, j)$, so this will add the third layer nested loop for each (i,j) and thus total time is $O(n^3)$.

```
COMPUTE-PREDECESSOR(L,W)
for i=1:n
    for j=1:n
        for k=1:n
            if(L[i][k]+W(k,j)==L[k][j])
                Π[i][j] = k
                break;
ENUMERATE-PATH(i,j)
    if(i==j)
        output(i)
    ENUMERATE(i,Π[i][j])        output(j)
```

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

Solution: The screenshot below shows how Floyd-Warshall algorithm works for each iteration k=1:6, the highlighted cell marks the updates of each iteration.

k=0	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	∞	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

k=1	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	-5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

k=2	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

k=3	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

k=4	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	-2	0	∞	2	-3	∞
3	0	2	0	4	-1	-8
4	-4	∞	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

k=5	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	-2	0	∞	2	-3	∞
3	0	2	0	4	-1	-8
4	-4	2	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

k=6	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	-2	0	∞	2	-3	∞
3	-5	-3	0	-1	-6	-8
4	-4	2	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

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

Solution: If we don't drop the superscripts, the equation is $d_{i,j}^{(k)} = min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$; if we want to drop the supercripts, we are worrying that $d_{i,k}^{(k-1)}$ or $d_{k,j}^{(k-1)}$ may be overwritten to $d_{i,k}^{(k)}$ or $d_{k,j}^{(k)}$ so we lost the (k-1)th information to update $d_{i,j}^{(k)}$. However, $d_{i,k}^{(k-1)}$=$d_{i,k}^{(k)}$ because $d_{i,k}$ itself already represents using kth vertex and it's impossible to put kth vertex itself again in the intermediate

vertex set {1,2,...,k} , it can only be {1,2,...,k-1}, thus $d_{i,k}^{(k-1)}=d_{i,k}^{(k)}$ and there is no need to worry about overwritting, we can simply drop one dimension to get a $\theta(n^2)$ space complexity.

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

Solution: If there a negative cycle, we could run Floyd-Warshall algorithm again to see if any matrix cell value changes because negative weight cycle will make some paths cost lower and some weights will be updated.