Topic 19: All Pairs Shortest Paths - Solutions

Solutions are provided for conceptual questions on Floyd-Warshall (Q1) and Johnson (Q3) and a run of Dijkstra's algorithm (Q2). Solutions are not provided for the simulation of Johnson's Algorithm (Q4): students should compare solutions to each others'.

1. Floyd-Warshall and Negative Weights

a. What do the main diagonal entries of the D matrix constructed by the Floyd-Warshall algorithm represent?

The <u>shortest</u> distance from a vertex to itself.

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4      for i = 1 to n
5          for j = 1 to n
6              $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$
7  return D
```

b. How can we use the diagonal entries to detect negative weight cycles?

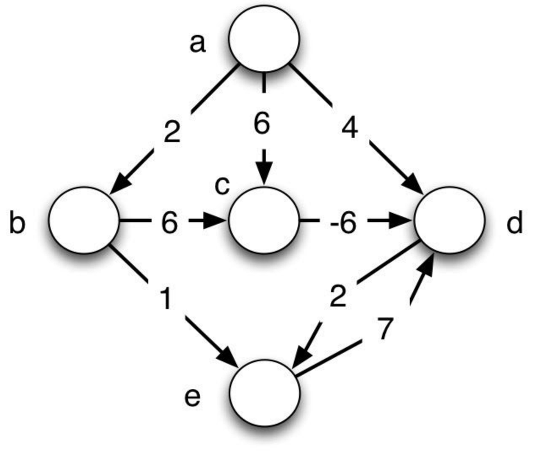
The distance of vertices to themselves are initialized to 0. Due to the "min" in line 6, the only way these distances can change is if they go negative. If a diagonal entry ever goes negative, then there must be a negative weight cycle involving the vertex, as that is the only way to start at the vertex and get back to itself at cost less than 0. Furthermore, if there is a negative weight cycle, at least one vertex on that cycle will receive a negative distance to itself, because the nested loops ensure that even the longest paths are explored.

So, simply <u>check whether any diagonal values are negative</u>. If any are, shortest paths are undefined for those vertices. Alternatively, modify the algorithm at line 6 to test whether there will be a change for any i == j, and exit immediately if there is, indicating a negative weight cycle.

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

2. Dijkstra's algorithm and Negative Weights

In this problem you will trace Dijkstra's algorithm on a small graph with negative weights to show how it fails. (Later you will continue with Johnson's algorithm to show how it succeeds.)



```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

The CLRS version of Dijkstra's algorithm shown above implicitly invokes DecreaseKey to reorder the queue. Let's assume that it is called whenever Relax changes a value v.d. But are we allowed to change v.d for a vertex already in S? It doesn't say. If we are, will that solve the negative weight problem? In this problem we'll find out whether it matters by tracing the algorithm both ways.

a. Run **Dijkstra's algorithm** on the graph shown, using vertex **a** as the start vertex. Edge (c,d) has a negative weight -6. For each step of the algorithm, show what vertex is dequeued and the distance estimate *v.d* it had at the time it is put into *S*. Use the final column to record any updates to the distance if Relax/DecreaseKey is allowed to change v.d for v in S. The first step is filled in.

Step	v put in S	v.d entering S	final v.d
1.	a	0	0
2.	b	2	2
3.	e	3	3
4.	d	4	0
5.	c	6	6

b. Which distance estimate(s) are in error, if any, if we do not allow DecreaseKey to update vertices already in S (column "`v.d entering S`")?

The estimate for d is in error, being 4 rather than 0 (via a-c-d).
The estimate for e is in error, being 3 rather than 2 (via a-c-d-e).

c. Which distance estimate(s) are in error, if any, if we allow DecreaseKey to update vertices already in S (column "`final v.d`")?

The estimate for vertex e, e.d, remains in error. It should be 2 (via a-c-d-e).

d. What do you conclude about why Dijkstra's fails to work with negative weights in general?

This shows that allowing updates of vertices already in S does *not* solve the problem of negative weight edges. Updates to vertices already in S do not propagate to other "downstream" connected vertices that are also already in S, because such propagation only happens when a vertex is removed from the queue Q in line 5 and processed in lines 7-8. In the above example, when we removed c we updated d's value, but d was already in S, so is not subsequently removed from the queue to update e's value.

---

3. Johnson's Algorithm Concepts

These questions are related to each other. Answering one might help answer the others, so read them all first. You may also want to look at the example graph on the next page.

a. We put 0-weighted links from *s* to every other vertex *v*, so isn't δ(*s*, *v*) always 0? When is it not, and why?

If there is a negative weight in the graph for some edge (u,v) then δ(*s*, *v*) can be less than 0, because we can reach u by (s,u) at cost 0 and then traverse (u,v) to get to v at a cost less than 0. (Thus this is a way of detecting negative weight edges.)

b. Suppose that *G*=(*V*,*E*) has no negative weight edges. What is the relationship between *w* and *ŵ* for *G*, and why?

If there are no negative weight edges then the functions w and *ŵ* will be identical, as *ŵ* is constructed to be different from w precisely only when the situation described above arises (based on negative δ(*s*, *v*)).

c. What is the primary purpose of adding the new vertex *s* to *V*, to construct *V'*? More specifically,

- why don't we just pick an arbitrary vertex in *V* to start from?
- for what kinds of graphs would it be unnecessary to add *s*?

We add s *with an edge to every vertex in V* to ensure that a path can be found from s to every v when we run Bellman-Ford. This is necessary to compute δ(*s*, *v*) that is needed for *ŵ*.

If we started from a vertex already in V there is no guarantee we can reach all other vertices.

Therefore the addition of s would be unnecessary for strongly connected graphs.

---

4. Running Johnson's Algorithm (Optional)

Do this problem if you have time, and continue after class to ensure you understand how Johnson's algorithm works.

Solutions not provided. Students should compare their solutions to each others'.