References

Collaborated with Crystal Huang.

Question 1: Dijkstra

Show the running of Dijkstra's algorithm for finding the distance from A to all other vertices in the following graph. You should show the updated distances to all the vertices (i.e., the key of the vertex in the priority queue) after each step in the algorithm.

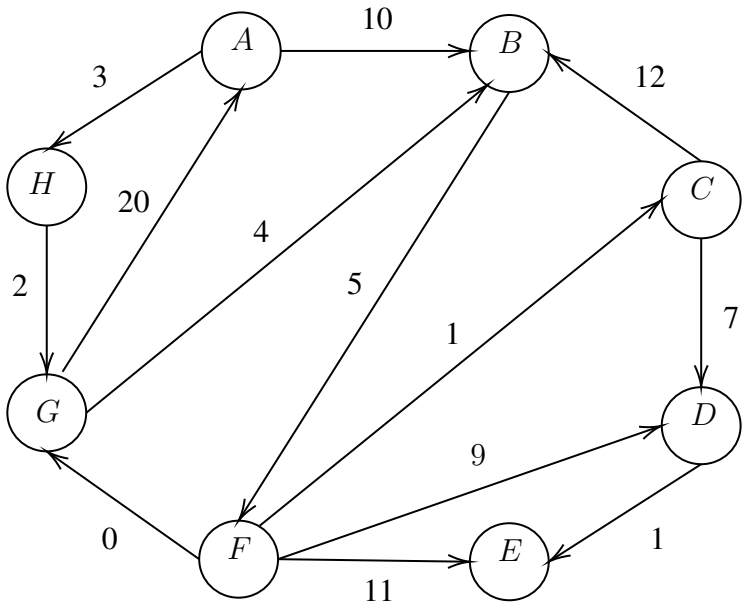

Figure 1: Directed and weighted graph G

For ease of representing your solution, simply complete the following table. The first row (Step 0) represents the initial values.

Step	Vertex Removed from Q	$A.d$	$B.d$	$C.d$	$D.d$	$E.d$	$F.d$	$G.d$	$H.d$
0	-	0	∞	∞	∞	∞	∞	∞	∞
1	A	0	10	∞	∞	∞	∞	∞	3
2	H	0	10	∞	∞	∞	∞	5	3
3	G	0	9	∞	∞	∞	∞	5	3
4	B	0	9	∞	∞	∞	14	5	3
5	F	0	9	15	23	25	14	5	3
6	C	0	9	15	22	25	14	5	3
7	D	0	9	15	22	23	14	5	3
8	E	0	9	15	22	23	14	5	3

Table 1: Execution trace of Dijkstra on G.

Solution: By stepping through Dijkstra's algorithm, we obtain the distances above.

Question 2: Floyd-Warshall

Consider again the graph from Question 1. In the following, complete the first four steps of the Floyd-Warshall algorithm by filling four copies of the following table corresponding to $DP[*,*,1]$, $DP[*,*,2]$, $DP[*,*,3]$, and $DP[*,*,4]$. In other words, compute the shortest distances between every pair of vertices when only using nodes $\{A\}$, $\{A,B\}$, $\{A,B,C\}$, and only using $\{A,B,C,D\}$ as intermediate vertices.

	A	B	C	D	E	F	G	H
A								
B								
C								
D								
E								
F								
G								
H								

Table 2: Shortest distance when only using first i vertices as intermediates.

Solution: Stepping through the first four iterations of the outer loop of the Floyd–Warshall algorithm gives the results below.

	A	B	C	D	E	F	G	H
A	0	10	∞	∞	∞	∞	∞	3
B	∞	0	∞	∞	∞	5	∞	∞
C	∞	12	0	7	∞	∞	∞	∞
D	∞	∞	∞	0	1	∞	∞	∞
E	∞	∞	∞	∞	0	∞	∞	∞
F	∞	∞	1	9	11	0	0	∞
G	20	4	∞	∞	∞	∞	0	23
H	∞	∞	∞	∞	∞	∞	2	0

Table 3: Shortest distance when only using first 1 vertex as an intermediate.

	A	B	C	D	E	F	G	H
A	0	10	∞	∞	∞	15	∞	3
B	∞	0	∞	∞	∞	5	∞	∞
C	∞	12	0	7	∞	17	∞	∞
D	∞	∞	∞	0	1	∞	∞	∞
E	∞	∞	∞	∞	0	∞	∞	∞
F	∞	∞	1	9	11	0	0	∞
G	20	4	∞	∞	∞	9	0	23
H	∞	∞	∞	∞	∞	∞	2	0

Table 4: Shortest distance when only using first 2 vertices as intermediates.

	A	B	C	D	E	F	G	H
A	0	10	∞	∞	∞	15	∞	3
B	∞	0	∞	∞	∞	5	∞	∞
C	∞	12	0	7	∞	17	∞	∞
D	∞	∞	∞	0	1	∞	∞	∞
E	∞	∞	∞	∞	0	∞	∞	∞
F	∞	13	1	8	11	0	0	∞
G	20	4	∞	∞	∞	9	0	23
H	∞	∞	∞	∞	∞	∞	2	0

Table 5: Shortest distance when only using first 3 vertices as intermediates.

	A	B	C	D	E	F	G	H
A	0	10	∞	∞	∞	15	∞	3
B	∞	0	∞	∞	∞	5	∞	∞
C	∞	12	0	7	8	17	∞	∞
D	∞	∞	∞	0	1	∞	∞	∞
E	∞	∞	∞	∞	0	∞	∞	∞
F	∞	13	1	8	9	0	0	∞
G	20	4	∞	∞	∞	9	0	23
H	∞	∞	∞	∞	∞	∞	2	0

Table 6: Shortest distance when only using first 4 vertices as intermediates.

Question 3: Shortest path

Sam, who lives in a node s of a weighted *undirected* graph $G = (V, E)$ with non-negative weights, is invited to a birthday party located at a node h. Naturally, Sam wants to get from s to h as soon as possible, but they are told to buy some beer on the way over. They can get beer at any supermarket, and the supermarkets form a subset of the vertices $B \subseteq V$. Thus, starting at s, they must go to some node $b \in B$ of their choice, and then head from b to h using the shortest total route possible (assume they waste no time in the supermarket). Help Sam to reach h as soon as possible, by solving the following sub-problems.

1. Compute the shortest distance from s to all supermarkets $b \in B$.

> **Solution:** Sam can use Dijkstra's algorithm (denoted DIJKSTRA(G, s)) to compute the shortest distance from s to all supermarkets $b \in B$.
>
> **Algorithm I.** TOSUPERMARKETS(G, s, B) requires graph $G = (V, E)$, source vertex $s \in V$, and supermarket vertices $B \subseteq V$.
>
> $(D, P) \leftarrow$ invoke DIJKSTRA(G, s) for graph $G = (V, E)$ and source vertex s; for all $v \in V$, the value at $D[v]$ represents the shortest distance from s to v and $P[v]$ represents the shortest path from s to v.
>
> Return (D, P), noting that for all $b \in B$, the value at $D[b]$ represents the shortest distance from s to b and $P[b]$ represents the shortest path from s to b.

2. Compute the shortest distance from every supermarket $b \in B$ to h. Note that this is the dual of the single-source shortest path where we are now asking for the shortest path from every node to a particular destination.

> **Solution:** Again, Sam can use Dijkstra's algorithm to compute the shortest distance from h to all supermarkets $b \in B$, leveraging the fact that G is an undirected graph to reverse the direction of the search without impacting the results.
>
> **Algorithm II.** FROMSUPERMARKETS(G, B, h) requires graph $G = (V, E)$, supermarket vertices $B \subseteq V$, and target vertex $h \in V$.
>
> $(D, P) \leftarrow$ invoke DIJKSTRA(G, h) for graph $G = (V, E)$ and source vertex h; for all $v \in V$, the value at $D[v]$ represents the shortest distance from h to v, and $P[v]$ represents the shortest path from h to v. Since G is undirected, $D[v]$ also represents the distance from v to h, and $P[v]$ also represents the shortest path from v to h.
>
> Return (D, P), noting that for all $b \in B$, the value at $D[b]$ represents the shortest distance from b to h and $P[b]$ represents the shortest path from b to h.

3. Combine parts 1 and 2 to solve the full problem.

Solution: Sam should take the shortest overall path; that is, Sam should choose the supermarket $b \in B$ which minimizes the sum of the distances from s to b and b to h.

Algorithm III. ToBirthdayParty(G, s, B, h) requires graph $G = (V, E)$, source vertex $s \in V$, supermarket vertices $B \subseteq V$, and target vertex $h \in V$:

Let $b^* \leftarrow$ nil.

Let $d^* \leftarrow +\infty$.

Let $p^* \leftarrow$ nil.

Assign $(D_s, P_s) \leftarrow$ invoke ToSupermarkets(G, s, B).

Assign $(D_h, P_h) \leftarrow$ invoke FromSupermarkets(G, B, h).

For $b \in B$:

- if $D_s[b] + D_h[b] < d^*$, then:
  - assign $b^* \leftarrow b$;
  - assign $d^* \leftarrow D_s[b] + D_h[b]$;
  - assign $p^* \leftarrow$ concatenation of $P_s[b]$ and $P_h[b]$.

Return p^*, the shortest path from s to h via some $b \in B$.

Question 4: Detecting negative cycles

A negative weight cycle is a cycle where the sum of the weights of the edges in the cycle is negative. In many algorithms, these negative weight cycles can prove to be problematic, so we want a way to detect them ahead of time. How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle in a directed graph?

Solution:

Algorithm 1. ContainsNegativeCycle(W, n) where $W[i, j]$ is the weight of edge (i, j) (or ∞ if the edge is absent) and n is the number of vertices; returns true if the graph represented by W and n contains a negative-weight cycle; otherwise, false:

Let $D \leftarrow$ Floyd–Warshall(W, n).

For $i = 1$ to n:

- if $W[i, i] < 0$, then return true.

Return false.

Proposition 1. *Claim.* Algorithm 1 is correct.

Proof. Algorithm 1 returns true if there is a vertex with a negative self-weight, and false otherwise. This is correct if the following statement holds: *there is a vertex with a negative self-weight if and only if there is a negative-weight cycle*.

- Suppose there is a vertex v where, at termination, $W[v, v] < 0$. The Floyd–Warshall algorithm initializes $W[v, v]$ to 0, meaning that the current shortest path from v to v is (v) with cost 0. Since v has a negative self-weight at termination, we know that the algorithm discovered a cheaper path $P = (v, \ldots, v)$ with a total cost $c < 0$. Of course, P is a negative-weight cycle. Hence, a negative self-weight implies a negative-weight cycle.
- Suppose there is a cycle $P = (v, \ldots, v)$ with a negative total cost $c < 0$. Initially, $W[v][v] = 0$ via the zero-cost path (v) from v to v. However, P provides a cheaper path. Since, at every step of the algorithm, weights are never increased, we know that the self-weight of v at termination is no greater than c. That is, $W[v][v] \leq c < 0$. Hence, a negative-weight cycle implies a negative self-weight.

The above statement holds. Ergo Algorithm 1 is correct. □

Question 5: Modified Floyd–Warshall

Consider the following (simplified) pseudocode of the Floyd-Warshall algorithm. $W[i,j]$ is the weight of the edge $i \to j$ (or ∞ if the edge is absent). At the end of the execution of this algorithm, $D[i,j]$ contains the length of the shortest path from i to j. First, convince yourself that this is indeed the case (no need to submit this part).

```
FLOYD-WARSHALL(W, n)
1    for i = 1 to n
2        for j = 1 to n
3            D[i,j] = W[i,j]
4    for k = 1 to n
5        for i = 1 to n
6            for j = 1 to n
7                cost = D[i,k] + D[k,j]
8                if cost < D[i,j]
9                    D[i,j] = cost
```

Now, consider the modification where, instead of at line 4 above, the iterator for k is nested within the loop for the variable j, i.e.:

```
FLOYD-WARSHALL-MODIFIED(W, n)
1    for i = 1 to n
2        for j = 1 to n
3            D[i,j] = W[i,j]
4    for i = 1 to n
5        for j = 1 to n
6            for k = 1 to n
7                cost = D[i,k] + D[k,j]
8                if cost < D[i,j]
9                    D[i,j] = cost
```

What is the meaning of the value stored at $D[1,2]$, at the end of the execution of this modified algorithm? Justify your answer.

Solution: At the end of the execution of this modified algorithm, the value stored at $D[1,2]$ is the shortest distance from vertex $i = 1$ to vertex $j = 2$ via at most 1 intermediate vertex. In other words, it is the shortest distance from vertex 1 to vertex 2 via a direct edge $(1,2)$ or a path $(1,k,2)$ for some other vertex k. We can confirm this by stepping through the algorithm.

First. The algorithm assigns the shortest distance from i to j to be $W[i,j]$: the weight of the edge (i,j), if it exists, or $+\infty$ if not.

Second. The second loop considers $i = 1, j = 1$. Note that FLOYD-WARSHALL is only valid for $W[i,j] \geq 0$, so any path $(1,\ldots,1)$ is no cheaper than the direct edge $(1,1)$. So the distance $D[1,1]$ remains 0.

Third. The second loop considers $i = 1, j = 2$. For each $1 \leq k \leq n$, if the path (i,k,j) has a shorter total cost than the existing $D[1,2]$, it is taken. Otherwise, it is not, and $D[1,2] = W[1,2]$. In

other words, if there is a path via 1 intermediate vertex that is cheaper than the existing path via 0 intermediate vertices, it is taken.

Thereafter. The second loop all successive pairs (i, j) after $(1, 2)$. If any value in D is ever modified, it is always position $D[i, j]$ for the current pair (i, j). Since for all successive pairs $i > 1$ or $j > 2$, we know that $D[1, 2]$ is not modified during this process.

Hence, the value stored at $D[1, 2]$ is either $W[1, 2]$ or the shortest distance from 1 to 2 via 1 other vertex, whichever is smaller.