CSE 373 HW5 Writeup

By Stephen Hung

1. **Topological Sort**

   Queue is bottom on the left and top on the right

vertex	A	B	C	D	E	F	G
in-degree	0	1	1	2	2	1	0

→ Queue: $\overline{\text{G}\quad\text{A}}$ → Output "A" →

vertex	B	C	D	E	F	G
in-degree	0	1	1	2	1	0

→ Queue: $\overline{\text{B}\quad\text{G}}$ → Output "G" →

vertex	B	C	D	E	F
in-degree	0	1	1	2	0

→ Queue: $\overline{\text{F}\quad\text{B}}$ → Output "B" →

vertex	C	D	E	F
in-degree	1	0	1	0

→ Queue: $\overline{\text{D}\quad\text{F}}$ → Output "F" →

vertex	C	D	E
in-degree	1	0	1

→ Queue: $\overline{\text{D}}$ → Output "D" →

vertex	C	E
in-degree	1	0

→ Queue: $\overline{\text{E}}$ → Output "E" →

vertex	C
in-degree	0

→ Queue: $\overline{\text{C}}$ → Output "C" →

Output: A, G, B, F, D, E, C

2. **Minimum Spanning Trees**

   (a) Kruskal's Algorithm

Cost	Edges		
1:	(B,E)	(C,G)	(D,G)
2:	(A,B)	(C,D)	(E,F)
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #1 edge (B,E)

Cost	Edges		
1:	~~(B,E)~~	(C,G)	(D,G)
2:	(A,B)	(C,D)	(E,F)
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #2 edge (C,G)

Cost	Edges		
1:	~~(B,E)~~	~~(C,G)~~	(D,G)
2:	(A,B)	(C,D)	(E,F)
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #3 edge (D,G)

Cost	Edges		
1:	~~(B,E)~~	~~(C,G)~~	~~(D,G)~~
2:	(A,B)	(C,D)	(E,F)
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #4 edge (A,B)

Cost	Edges		
1:	~~(B,E)~~	~~(C,G)~~	~~(D,G)~~
2:	~~(A,B)~~	(C,D)	(E,F)
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #5 edge (E,F)

Edge (C,D) is not outputted because it contains the vertices C and D which have already been processed before.

Cost	Edges		
1:	~~(B,E)~~	~~(C,G)~~	~~(D,G)~~
2:	~~(A,B)~~	~~(C,D)~~	~~(E,F)~~
3:	(A,E)	(B,F)	(E,G)
4:	(A,D)	(F,G)	

→ Output #6 edge (E,G)

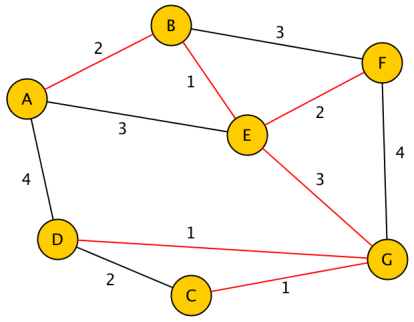
Edge (A,E) and (B,F) are not outputted because the vertices A,B,E, and F have already been processed before.

Cost	Edges		
1:	~~(B,E)~~	~~(C,G)~~	~~(D,G)~~
2:	~~(A,B)~~	~~(C,D)~~	~~(E,F)~~
3:	~~(A,E)~~	~~(B,F)~~	~~(E,G)~~
4:	(A,D)	(F,G)	

The amount of edges $= |V| - 1 = 6$ so Kruskal's algorithm stops.

Edge Number and Output: #1 :**(B,E)**, #2 :**(C,G)**, #3 :**(D,G)**, #4 :**(A,B)**, #5 :**(E,F)**, #6 :**(E,G)**

Minimum Spanning Tree:



(b) Prim's Algorithm

Vertex	Known	Cost	Path
A	Y	0	
B		2	A
C		∞	
D		4	A
E		3	A
F		∞	
G		∞	

→ Output Edge #1 :(A,B)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C		∞	
D		4	A
E		1	B
F		3	B
G		∞	

→ Output Edge #2 :(B,E)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C		∞	
D		4	A
E	Y	1	B
F		2	E
G		3	E

→ Output Edge #3 :(E,F)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C		∞	
D		4	A
E	Y	1	B
F	Y	2	E
G		3	E

→ Output Edge #4 :(E,G)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C		1	G
D		1	G
E	Y	1	B
F	Y	2	E
G	Y	3	E

→ Output Edge #5 :(G,C)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C	Y	1	G
D		1	G
E	Y	1	B
F	Y	2	E
G	Y	3	E

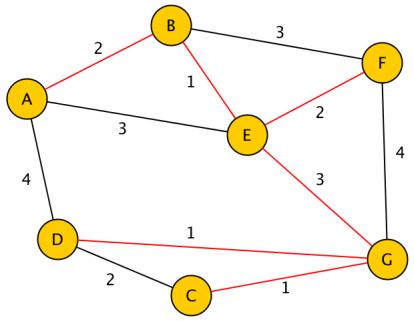
→ Output Edge #6 :(G,D)

Vertex	Known	Cost	Path
A	Y	0	
B	Y	2	A
C	Y	1	G
D	Y	1	G
E	Y	1	B
F	Y	2	E
G	Y	3	E

Because all of the nodes have been processed, the algorithm stops.

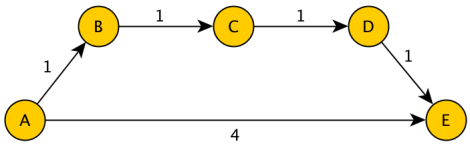
Edge Number and Output: #1 :**(A,B)**, #2 :**(B,E)**, #3 :**(E,F)**, #4 :**(E,G)**, #5 :**(G,C)**, #6 :**(G,D)**

Minimum Spanning Tree:



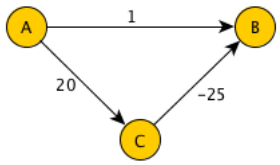
3. **Dijkstras and Negative Edges:**

   (a) No. Dijkstra's Algorithm only cares about the weight of the edges instead of the number of the edges. For example in the following graph the shortest path from A to E will give A - B - C - D - E



However, the path A - E has the same weight but fewer edges yet it is not chosen. A way to modify Dijkstra's Algorithm is to keep track of both the total cost from the source and the number of edges from the source. Therefore when deciding which node to process next, find the minimum cost and the minimum number of edges from the source.

(b) The following is a graph with a negative cost edge where Dijkstra's Algorithm gives the wrong answer for the path from A to B.



Dijkstra's Algorithm fails this example because when starting from A, Dijkstra's Algorithm chooses the adjacent edge with the lowest cost. This is A to B with a cost of 1 since A to C has a cost of 20. Once it has done so, the vertex B is marked known like so:

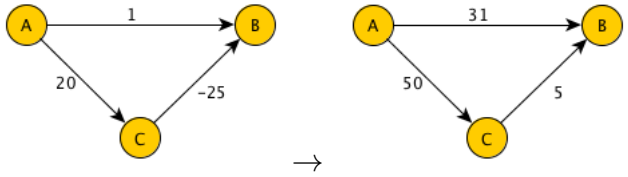
Vertex	Known	Cost	Path
A	Y	0	
B	Y	1	A
C		20	A

Afterwards, Dijkstra's Algorithm processes vertex C and quits, since there are no more unknown vertices.

Vertex	Known	Cost	Path
A	Y	0	
B	Y	1	A
C	Y	20	A

This results in the path A to B with a cost of 1, but the path A to C to B has a cost of -5 which is actually the lowest cost path. The reason why when processing C, the vertex B's cost and path are not updated is because B is already marked known and thus Dijkstra fails.

(c) For this problem I use the same graph from part 3b since it has negative-cost edges but has no negative-cost cycles. I then add 30 to all the edges in order to make sure the edge costs are positive.



After running Dijkstra's Algorithm and subtracting 30 cost from each the cost and path for each vertex is as follows:

Vertex	Known	Cost	Path
A	Y	30	
B	Y	31	A
C	Y	50	A

$\rightarrow$

Vertex	Known	Cost	Path
A	Y	0	
B	Y	1	A
C	Y	20	A

Thus Dijkstra's Algorithm would still return the path A to B. As shown in the previous part (3b), the actual correct path is A to C to B, but Dijkstra's Algorithm still returns A to B. This is because, B is "known" first and thus when C is processed, the negative edge from C to B is ignored thus ignoring the A to C to B path.

A general explanation for why this technique does not work is that the negative weight of an edge was why it would be a part of the original least cost path. This is due to the fact that no matter the weight of the edge before, the negative weight of the edge can reduce the total cost till it is the smallest. Therefore, when all edges become positive, the ability of the negative weight dissapears and thus the path is no longer the least cost path.

4. **Testing:**

   Most of my testing was done using the provided edge.txt and vertex.txt. First I manually computed several paths of varying lengths. An example of this is the path from ORD to SEA which should be a path containing five vertices including the source and destination and has a cost of 541. Afterwards, I ran Dijkstra's Algorithm in order to determine that the paths I manually computed and the paths that were returned were the same.

   I also tested for boundary cases such as when a vertex is not connected to any other vertices. So for example, I added SJC to the vertex.txt and tried to find the shortest path from SEA to SJC which should return an error. Another case I tested for was the IllegalArgumentException thrown when a vertex doesn't exist.

   I also tested for cyclic graphs and when the source and destination vertex are the same.