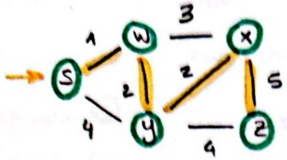
Valentina Siwaia

## Question 1)

Part a) Prim's Algorithm




$V = \{s, w, x, y, z\}$

$E = \{(s, w, 1)$
$(w, y, 2)$
$(s, y, 4)$
$(w, x, 3)$
$(x, y, 2)$
$(y, z, 4)$
$(x, z, 5)\}$

**Initialize**

Start at vertex s as the initial vertex

initialize keys for all vertices: $s = 0$

$w = \infty$

$x = \infty$

$y = \infty$

$z = \infty$

**Iteration 1** - s is starting vertex

update keys and $\pi$ values for adjacent vertices

$w$: key $= 1$, $\pi = s$

$y$: key $= 4$, $\pi = s$

**Iteration 2** - w (key $= 1$) as next vertex to add to the MST

update keys and $\pi$ values for adjacent vertices.

$x$: key $= 3$, $\pi = w$

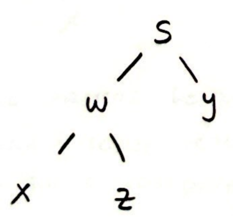
**Iteration 3** - x (key $= 3$) as the next vertex to add MST

update keys and add $\pi$ values for adjacent vertices.

$y$: key $= 2$, $\pi = x$

$z$: key $= 5$, $\pi = x$

**The minimum spanning tree rooted at S: (MST)**



part b) Dijkstra's Algorithm:

start with vertex s as the initial vertex.

$\boxed{\text{Iteration 1}}$ - choose s as the starting vertex.

update d values for adjacent vertices:

w: d = 1, π = s

y: d = 4, π = s

$\boxed{\text{Iteration 2}}$ - choose w (d = 1) as the next vertex to add to the shortest-path tree.

update d values for adjacent vertices:

x: d = 4, π = w

$\boxed{\text{Iteration 3}}$ - choose x (d = 4) as the next vertex

update d values for adjacent vertices:

y = d = 3, π = x

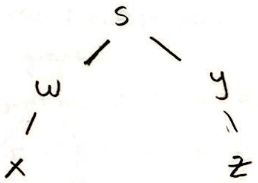
z = d = 8, π = x

$\boxed{\text{Iteration 4}}$ - choose y (d = 3) as the next vertex.

update d values for adjacent vertices:

z: d = 7, π = y

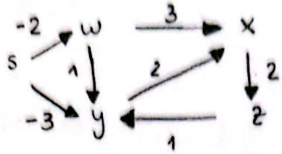
**shortest-paths tree at S:** it will be similar to the MST from part a.




part c) The difference happens because Dijkstra's algorithm aims to find the shortest path from a source vertex to all other vertices, which ends in a different edge of choice compared to Prim's algorithm, which seeks the minimum spanning tree for the entire graph. BUT in specific cases, especially when edge weights satisfy specific conditions the SPT can coincide with the MST.

## Question 2)

part a) Dijkstra's Algorithm on G2




Set the distance of all vertices with $d = \infty$ and $\pi$ = NIL

set $d[s] = 0$ to select starting vertex.

**Iteration 1**

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | ∞ | NIL |
| z | ∞ | NIL |

→

**Iteration 2**

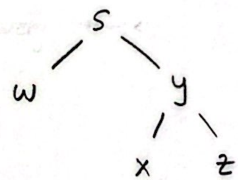
| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | -5 | y |
| z | -4 | y |

→

**Iteration 3**

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | -5 | y |
| z | -4 | y |

The final shortest paths




while Q is not empty → Extract the vertex u with the minimum d value of Q

for each vertex v adjacent to u:

if the distance to v through u is less than v's current d:

- update v's d to the distance through u.
- set v's π value to u
- update the priority queue

Repeat until Q is empty.

part b)

Bellman-Ford's
Algorithm on G2

Initialize all vertices with d = ∞ and π = NIL, except for the source vertex s with d[s] = 0

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | ∞ | NIL |
| x | ∞ | NIL |
| y | ∞ | NIL |
| z | ∞ | NIL |

→

Iteration 1

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | ∞ | NIL |
| z | ∞ | NIL |

for each edge (u,v) in the graph
if d[u] + weight(u,v) < d[v]:
- update d[v] to d[u] + weight(u,v)
- set π[v] to u

Negative-weight cycles
for each edge (u,v) in the graph:
if d[u] + weight(u,v) < d[v]
return FALSE

→

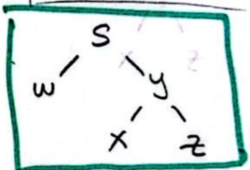
Iteration 2

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | -5 | y |
| z | -4 | y |

→

Iteration 3

| vertex | d | π |
|---|---|---|
| s | 0 | NIL |
| w | -2 | s |
| x | -3 | s |
| y | -5 | y |
| z | -4 | y |




Since there are no changes to the distances, that indicates that the algorithm has converged. thus, the algorithm will return TRUE

part c) Both, part a and b algorithms produce correct results for G2
They correctly identify the shortest paths from the source vertex s to all other vertices in the graph.
However, Dijkstra's algorithms is more efficient when dealing with graphs without negative-weight edges, while Bellman's Ford's algorithm is more versatile and can handle graphs with negative-weight edges and detect negative-weight cycles. Since in part a and b have no cycles, both algorithms provide accurate results.

```
[Running] cd "c:\Users\Valentina\Desktop\A4_Silveira\" && javac BellmanFord.java && java BellmanFord
Graph contains a negative weight cycle.
Graph G3:
Number of vertices: 8
Number of edges: 13
Vertex  d-value   Pi-value
S       0         null
A       6         S
B       5         S
C       -4        S
D       -5        G
E       -7        D
F       -2        C
G       -9        E

Graph G4:
Number of vertices: 10
Number of edges: 17
Vertex  d-value   Pi-value
S       0         null
A       6         S
B       -1        S
C       1         B
D       2         A
E       2         B
F       -1        C
G       INF       null
H       7         E
I       2         F

[Done] exited with code=0 in 0.835 seconds
```