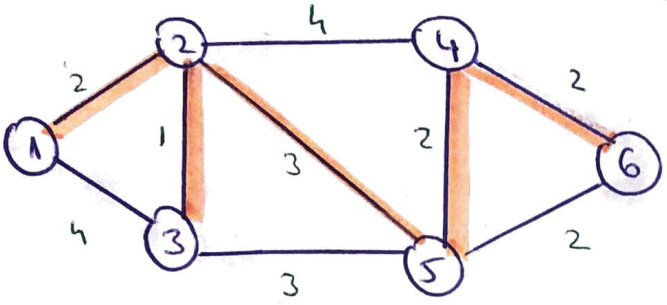
Kruskal's algorithm

MINIMUM SPANNING TREE

V-1 edges

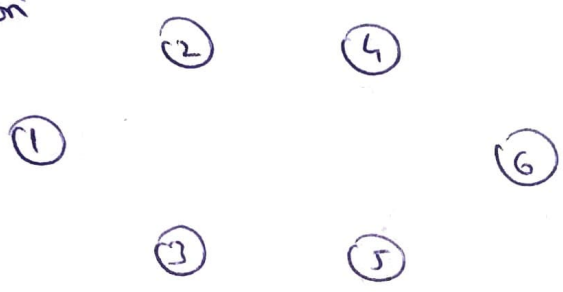
$O(E \log V)$



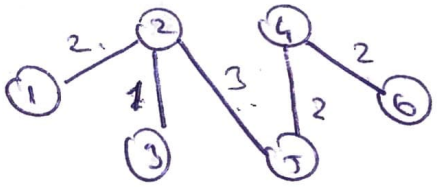
sorted edges

(2,3): 1 ✓
(1,2): 2 ✓
(4,5): 2 ✓
(4,6): 2 ✓
(5,6): 2 ✗
(2,5): 3 ✓
(3,5): 3
(1,3): 4
(2,5): 4

Initialisation



iteration 1) (2,3) selected



iteration 2) (1,2) selected

iteration 3) (4,5) selected

iteration 4) (4,6) selected

iteration 5) (5,6) not selected (it forms a cycle)

iteration 6) (2,5) selected

iteration 7) — We have 6 vertices, 5 needed edges

cost = 10

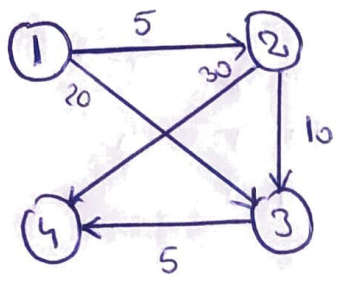
The final tree:
It contains all the vertices

Greedy's algorithm example

Floyd-Warshall algorithm – lowest cost walk between the given vertices

(no negative costs)

$O(|V|^3)$



$$D_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 5 & 20 & \infty \\ 2 & \infty & 0 & 10 & 30 \\ 3 & \infty & \infty & 0 & 5 \\ 4 & \infty & \infty & \infty & 0 \end{array} \qquad P_0 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$k=1$ using vertex 1 as intermediate vertex

$$D_1 = \begin{pmatrix} 0 & 5 & 20 & \infty \\ \infty & 0 & 10 & 30 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad P_1 = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$k=2$ using vertex 2 as intermediate vertex

$$D_2 = \begin{pmatrix} 0 & 5 & 15 & 35 \\ \infty & 0 & 10 & 30 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$k=3$ using vertex 3 as intermediate vertex

$$D_3 = \begin{pmatrix} 0 & 5 & 15 & 20 \\ \infty & 0 & 10 & 15 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad P_3 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$k=4$ using vertex 4 as intermediate vertex

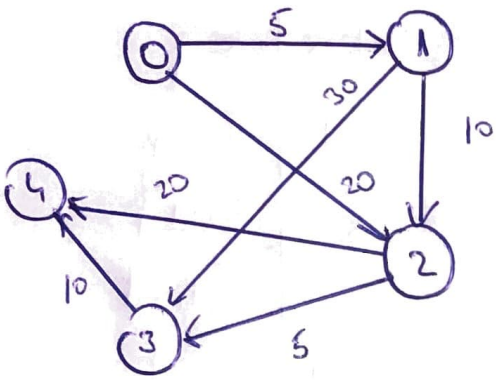
$$D_4 = \begin{pmatrix} 0 & 5 & 15 & 20 \\ \infty & 0 & 10 & 15 \\ \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 0 \end{pmatrix} \qquad P_4 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$V=4$ stop

The minimum cost walk from $s=1$ to $t=4$ has the cost $D_4(1,4) = 20$ and it is obtained from P_4 backwards; $P_4(1,4) = 3$ $P_4(1,3) = 2$ $P_4(1,2) = 1$

The minimum cost walk $1 \to 2 \to 3 \to 4$

Dijkstra's algorithm — the minimum cost walk from a vertex s to all the other vertices (non-negative costs)



s = 0 t = 4

	x	y	dist dictionary	Q: priority queue	prev dictionary
init			0 | | | | | 0 1 2 3 4	(0,0)	
	0	1	0 | 5 | | | | 0 1 2 3 4	(1,5)	0 | | | | 0 1 2 3 4
		2	0 | 5 | 29 | | 0 1 2 3 4	(1,5) (2,20)	0 | 0 | | | 0 1 2 3 4
	1	2	0 | 5 | 15 | | 0 1 2 3 4	(2,20)	0 | 1 | 1 | 0 1 2 3 4
		3	0 | 5 | 15 | 35 | 0 1 2 3 4	(2,15) (3,35)	0 | 1 | 1 | 1 | 0 1 2 3 4
	2	3	0 | 5 | 15 | 20 | 0 1 2 3 4	(3,35) (3,20)	0 | 1 | 1 | 2 | 0 1 2 3 4
		4	0 | 5 | 15 | 20 | 35 0 1 2 3 4	(3,20) (4,35)	0 | 1 | 0 | 1 | 2 | 2 0 1 2 3 4
	3		0 1 2 3 4	(4,35)	0 1 2 3 4
		4	0 | 5 | 15 | 20 | 30 0 1 2 3 4	(4,30)	0 | 1 | 1 | 2 | 3 0 1 2 3 4
	4		STOP	t = 4 = x	

Nout

0	[1,2]
1	[2,3]
2	[3,4]
3	[4]
4	[ ]

The minimum cost walk from 0 to 4 is 30 = d[4]

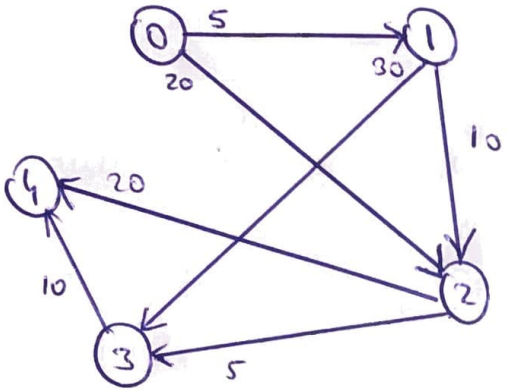
And the path is prev[4] = 3 prev[3] = 2 prev[2] = 1 prev[1] = 0 = s

0 → 1 → 2 → 3 → 4 cost = dist[4] = 30

built backwards

Dijkstra's algorithm

BACKWARDS FROM t TO s



s=0 t=4

Nin

0	[]
1	[0]
2	[0,1]
3	[1,2]
4	[2,3]

t=4 → s=0

	x	y	dist dictionary	q: priority queue	next dictionary
			0 1 2 3 4 ∞ ∞ ∞ ∞ 0	(4,0)	
	4	2 3	0 1 2 3 4 ∞ ∞ 20 ∞ 0 ∞ ∞ 20 10 0	(2,20) → P.Q. (3,10) (2,20)	0 1 2 3 4 ∞ ∞ 4 ∞ ∞ ∞ ∞ 4 4 ∞
	3	2 1	0 1 2 3 4 ∞ ∞ 15 10 0 ∞ 30 15 10 0	(2,20) (2,15) (2,15) (1,30)	0 1 2 3 4 ∞ ∞ 3 4 ∞ ∞ 3 3 4 ∞
	2	1 0	0 1 2 3 4 ∞ 25 15 10 0 35 15 15 10 0	(1,30) (1,25) (1,25) (0,35)	0 1 2 3 4 2 2 3 4 ∞
	1	0	0 1 2 3 4 30 25 15 10 0	(0,35) (0,30)	0 1 2 3 4 1 2 3 4 ∞
	0		s=0	STOP	

The minimum cost walk from s=0 to t=4 has the cost dist[0] = 30

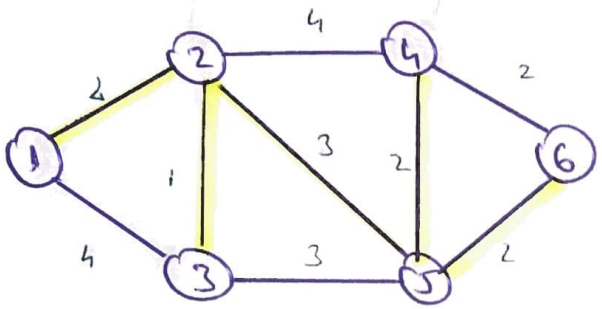
s=0 next[0]=1 next[1]=2 next[2]=3 next[3]=4 ↑ t

0 → 1 → 2 → 3 → 4

Prim's algorithm

MINIMUM SPANNING TREE

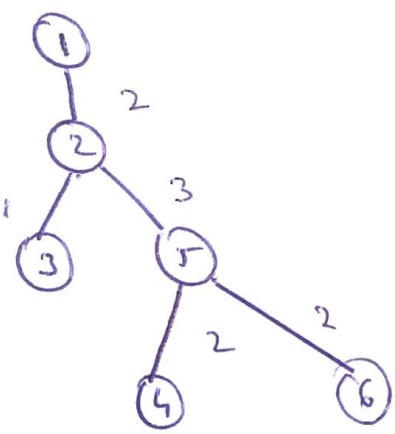
$O(E \log V)$



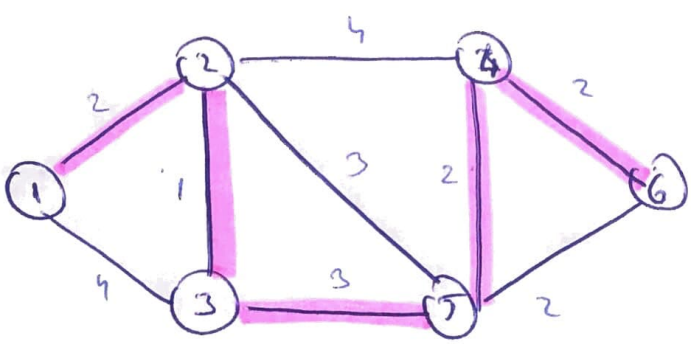
sorted edges

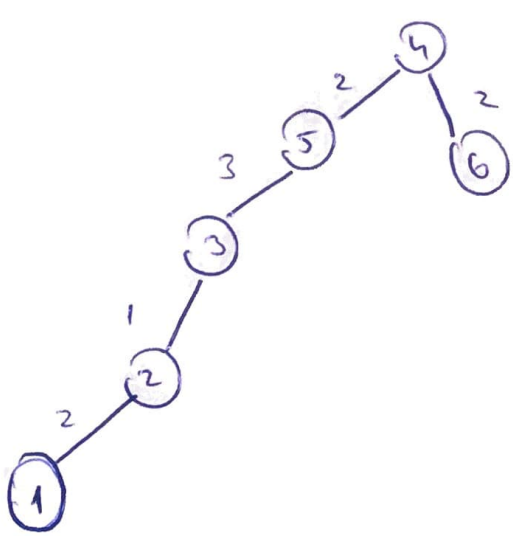
(2,3) : 1 ✓
(1,2) : 2 ✓
(4,5) : 2 ✓
(4,6) : 2
(5,6) : 2
(2,5) : 3 ✓
(3,5) : 3
(2,4) : 4
(1,3) : 4

	selected edge	Vnew	Enew
initialization		{1}	{}
iteration 1	(1,2)	{1,2}	{(1,2)}
iteration 2	(2,3)	{1,2,3}	{(1,2)(2,3)}
iteration 3	(2,5)	{1,2,3,5}	{(1,2)(2,3)(2,5)}
iteration 4	(4,5)	{1,2,3,4,5}	{(1,2)(2,3),(2,5),(4,5)}
iteration 5	(5,6)	{1,2,3,4,5,6}	{(1,2)(2,3),(2,5),(4,5),(5,6)}



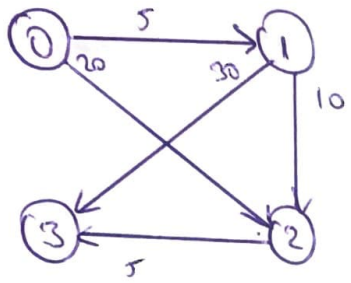
Another minimum spanning tree with the same cost = 10





Bellman - Ford's algorithm (based on principle of relaxation)

MINIMUM COST WALK



s = 0
t = 3

	changed	edge (x,y)	dist dictionary	prev dictionary
initialization	true		0 ∞ ∞ ∞ (0 1 2 3)	
iteration 1	false			
	true	(0,1)	0 5 ∞ ∞	0
	true	(0,2)	0 5 20 ∞	0 0
	true	(1,2)	0 5 15 ∞	0 1
	true	(1,3)	0 5 15 35	0 1 1
	true	(2,3)	0 5 15 20	0 1 2
iteration 2	false			
		(0,1)	0 5 15 20	0 1 2
		(0,2)	0 5 15 20	0 1 2
		(1,2)	0 5 15 20	0 1 2
		(1,3)	0 5 15 20	0 1 2
		(2,3)	0 5 15 20	0 1 2

changed = false → STOP

built backwards

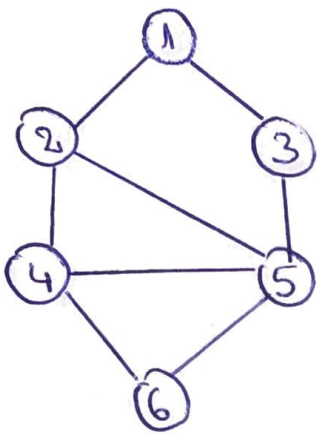
the minimum cost walk is cost = dist[3] = 20

t = 3 prev[3] = 2 prev[2] = 1 prev[1] = 0 = s

walk 0 → 1 → 2 → 3

Breadth - first search (BFS) - TRAVERSAL

$O(V+E)$
↳ edges
↳ vertices



1) Visited | 0 | 0 | 0 | 0 | 0 | 0 |
1 2 3 4 5 6

Queue:

2) Visited | 1 | 0 | 0 | 0 | 0 | 0 |
1 2 3 4 5 6

Queue: 1

3) Visited | 1 | 0 | 0 | 0 | 0 | 0 |
1 2 3 4 5 6

Queue: ~~1~~

4) Visited | 1 | 1 | 1 | 0 | 0 | 0 |
1 2 3 4 5 6

Queue: 2 3

5) Visited | 1 | 1 | 1 | 1 | 1 | 0 |
1 2 3 4 5 6

Queue ~~2~~ 3 4 5

6) Visited | 1 | 1 | 1 | 1 | 1 | 0 |
1 2 3 4 5 6

Queue ~~3~~ 4 5

7) Visited | 1 | 1 | 1 | 1 | 1 | 0 |
1 2 3 4 5 6

Queue ~~4~~ 5

8) Visited | 1 | 1 | 1 | 1 | 1 | 1 |
1 2 3 4 5 6

Queue 5 6

9) Visited | 1 | 1 | 1 | 1 | 1 | 1 |
1 2 3 4 5 6

Queue ~~5~~ 6

10) Visited | 1 | 1 | 1 | 1 | 1 | 1 |
1 2 3 4 5 6

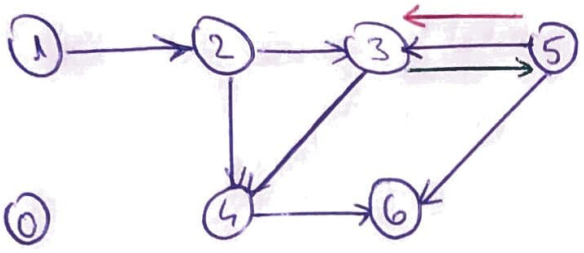
Queue ~~6~~

11) Visited | 1 | 1 | 1 | 1 | 1 | 1 |
1 2 3 4 5 6

Queue - empty

Print: 1 2 3 4 5 6

Lowest length path between s and t - BFS



FORWARD

Nout - dictionary

0	- []
1	- [2]
2	- [3,4]
3	- [4]
4	- [6]
5	- [3,6]
6	- []

s=1 t=6 path [1,2,4,6] length = 3 (dist[6])

	x	y	queue	visited (set)	dist dictionary	prev dictionary
			1	{1}	0 1 2 3 4 5 6 | 0 | | | | | |	0 1 2 3 4 5 6 | | | | | | |
	1	2	2	{1,2}	0 1 2 3 4 5 6 | 0 | 1 | | | |	0 1 2 3 4 5 6 | | 1 | | | |
	2	3	3	{1,2,3}	0 1 2 3 4 5 6 | 0 | 1 | 2 | | |	0 1 2 3 4 5 6 | | 1 | 2 | | |
		4	3 4	{1,2,3,4}	0 1 2 3 4 5 6 | 0 | 1 | 2 | 2 | |	0 1 2 3 4 5 6 | | 1 | 2 | 2 | |
	3	4	4			
	4	6	6	{1,2,3,4,6}	0 1 2 3 4 5 6 | 0 | 1 | 2 | 2 | 3 |	0 1 2 3 4 5 6 | | 1 | 2 | 2 | 4 |

The reverse path is built from prev t=6 prev[6]=4 prev[4]=2 prev[2]=1 reverse [6,4,2,1] ↪ 1,2,4,6

BACKWARD

	x	y	queue	visited	dist dictionary	next dictionary
			6	{6}	0 1 2 3 4 5 6 | | | | | | 0	0 1 2 3 4 5 6 | | | | | | |
	6	4	4	{4,6}	0 1 2 3 4 5 6 | | | | 1 | | 0	0 1 2 3 4 5 6 | | | | 6 | |
		5	4,5	{4,5,6}	0 1 2 3 4 5 6 | | | | 1 | 1 | 0	0 1 2 3 4 5 6 | | | | 6 | 6 |
	4	2	5	{2,4,5,6}	0 1 2 3 4 5 6 | 1 | 2 | | 1 | 1 | 0	0 1 2 3 4 5 6 | 1 | 4 | | 6 | 6 |
		3	5,2 5,2,3	{2,3,4,5,6}	0 1 2 3 4 5 6 | 1 | 2 | 2 | 1 | 1 | 0	0 1 2 3 4 5 6 | 1 | 4 | 4 | 6 | 6 |
	5	3	2,3			
	2	1	3 3,1	{1,2,3,4,5,6}	0 1 2 3 4 5 6 | 3 | 2 | 2 | 1 | 1 | 0	0 1 2 3 4 5 6 | 2 | 4 | 4 | 6 | 6 |

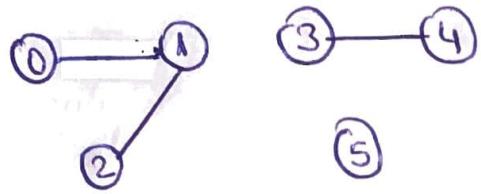
Nin - dictionary

0	- []
1	- []
2	- [1]
3	- [2]
4	- [2,3]
5	- [3]
6	- [4,5]

y = s = 1 → stop path [1,2,4,6] dist[1] = 3

s=1 next[1]=2 next[2]=4 next[4]=6=t length = 3

Find the connected components of an undirected graph using BFS



Nout

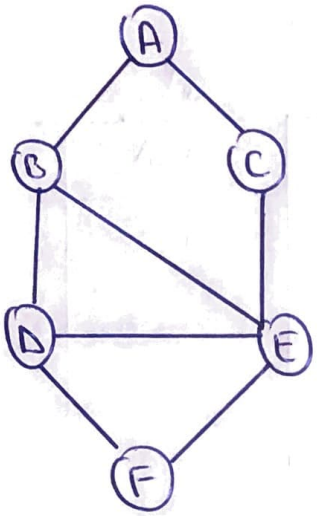
0	-[1]
1	-[0,2]
2	-[1]
3	-[4]
4	-[3]
5	-[]

	x	y	queue	acc (set)	visited	
					{}	
call accessible (0)	0	1	0	{0}		
	1	2	1	{0,1}		
	2		2	{0,1,2}		connected comp.
					{0,1,2}	0—1—2
call accessible (3)	3		3	{3}		
	4		4	{3,4}		
					{3,4}	connected comp.
					{0,1,2,3,4}	3—4
call accessible (5)	5		5	{5}		connected comp.
					{0,1,2,3,4,5}	5

```
def accessible(g, s):        # g: graph, s: vertex
    acc = set()
    acc.add(s)
    list = [s]               (queue)
    while len(list) > 0:
        x = list[0]
        list = list[1:]
        for y in g.parseNout(x):
            if y not in acc:
                acc.add(y)
                list.append(y)
    return acc
```

Depth - first search (DFS)

$O(V+E)$
- E → edges
- V → vertex



Print: A B D E F C

Visited: A, B, D, E, F, C

When stack is empty → STOP