Graph algorithms

Graph representation

↳ adjacency matrix : memory $\Theta(n^2)$

↳ list of edges: memory : $\Theta(m)$

↳ list of neighbours (inbound / outbound / both)
: memory $\Theta(n+m)$

Glossary

↳ walk: sequence of 1 or more vertices s.t. each vertex has an edge to the next one

- length of the walk: no. of edges along it
  no. of vertices - 1
- repeating vertices or edges is allowed

↳ path: a walk with no repeating vertices

↳ cost of a walk: the sum of the costs of the edges that form that walk

↳ closed walk : walk that starts and ends in the same vertex

↳ cycle: a closed walk of length at least 1, with no repeating vertices or edges.

Graph traversal

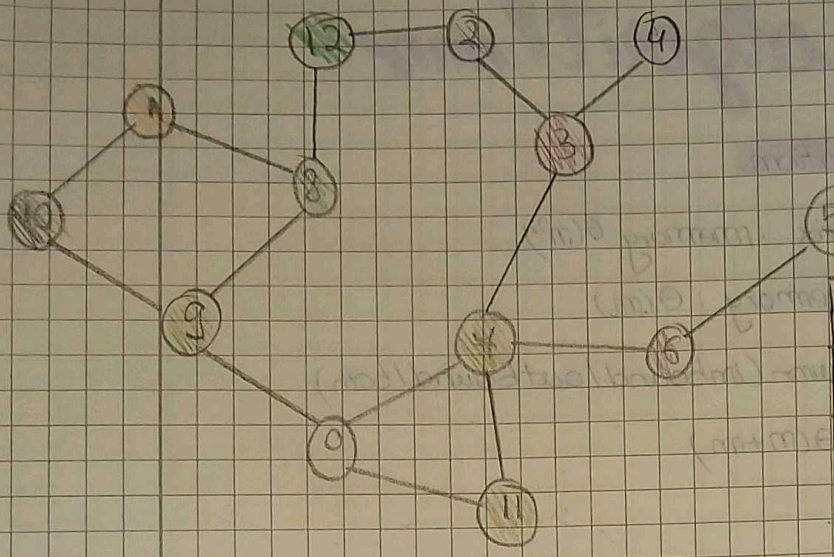
BREADTH-FIRST TRAVERSAL

Time Complexity : $O(V+E)$

Useful for: finding the shortest path on unweighted graphs

↳ fundamental search algorithm

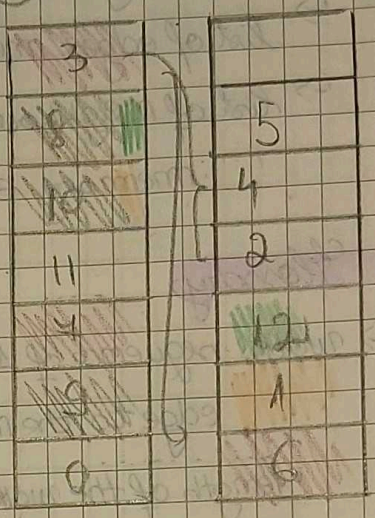
Starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.



Beginning at 0

0's unvisited neighbours:

9, 7, 11



Breadth-first traversal

Input:
G: graph
s: a vertex

Output:
accessible: set of vertices accessible from s
prev: a map that maps each accessible vertex to its predecessor on a path from s to it

Algorithm:

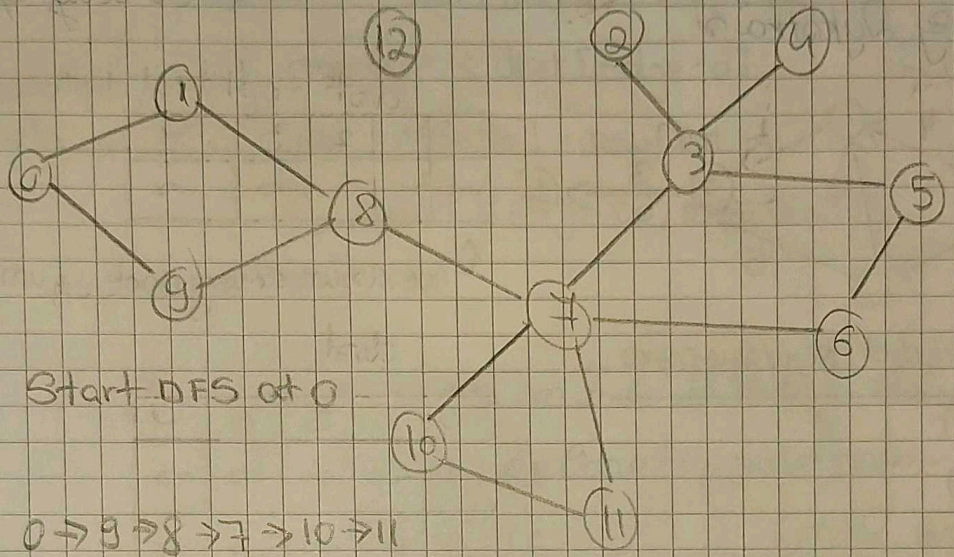
```
Queue Q
Dictionary prev
Dictionary dist
Set visited
Q.enqueue(s)  //add the starting vertex to the queue
visited.add(s)
dist[s]=0
while not Q.isEmpty() do
    x = Q.dequeue()
    for y in Nout(x) do
        if y not in visited then
            Q.enqueue(y)
            visited.add(y)
            dist[y] = dist[x]+1
            prev[y]=x
```

accessible = visited

DEPTH-FIRST SEARCH TRAVERSAL

Complexity: O(V+E)

⇒ DFS plunges depth-first into a graph without regard for which edge it takes next, until it cannot go any further at which point it backtracks and continues



Start DFS at 0

0 → 9 → 8 → 7 → 10 → 11

↳ we do not want to revisit 7 so we backtrack

7 → 3 → 2 → backtrack

3 → 4

3 → 5 → 6

8 → 1

Backtrack all the way to 0

Pseudocode

```
n = no. of nodes in the graph
g = adjacency list
visited = [false, ..., false] # size n
function dfs(at):
    if visited[at]: return
    visited[at] = true
    neighbours = graph[at]
    for next in neighbours:
        dfs(next)
```

Connected components using DFS

⇒ start DFS at every unvisited node and mark all reachable nodes as being part of the same component

```
function findComponents():
    for (i = 0; i < n; i++):
        if ! visited[i]:
            count++
            dfs(i)
    return count, components
```

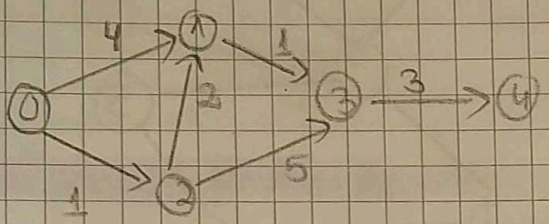
DIJKSTRA'S ALGORITHM

Complexity: $O(E * \log(V))$

2) Single Source Shortest Path (SSSP) algorithm for graphs with non-negative edge weights

3) once a node has been visited its optimal distance cannot be improved (Greedy approach)

Lazy Dijkstra's



dist - optimal distance from the start node

0	1	2	3	4
∞	∞	∞	∞	∞

↑ we assume every node is unreachable

(index, dist) key value pairs

1) (0,0)
2) (1,4)
3) (2,1) *
4) (1,3)
5) (3,6) ⇒ Pruned with 2 ⇒ we switch to 1
6) (3,4) ⇒ Pruned with 1
7) (4,7)

dist

	0	1	2	3	4
1)	0	∞	∞	∞	∞
2)	0	4	∞	∞	∞
3)	0	4	1	∞	∞
	0	3	1	∞	
	0	3	1	6	∞
	0	3	1	4	∞
	0	3	1	4	7

Code sequence

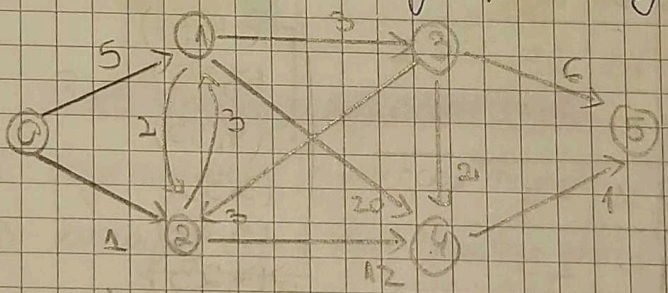
g - adjacency list
m - no. of nodes
s - index of start node

```
function dijkstra(g, m, s):
    vis = [false, false, ..., false] # size m
    dist = [∞, ∞, ..., ∞] # size m
    pq = empty priority queue
    pq.insert((s, 0))
```

```
while pq.size() != 0:
    index, minValue = pq.poll() // remove the next pair
    vis[index] = true
    if (dist[index] < minValue): continue
    for (edge : g[index]):
        if vis[edge.to]: continue #skip visited neighbours
        newDist = dist[index] + edge.cost
        if newDist < dist[edge.to]:
            dist[edge.to] = newDist
            pq.insert ((edge.to, newDist))
    return Dist
```

Eager Dijkstra's using an Indexed Priority Queue

2> avoid inserting duplicate key-value pairs



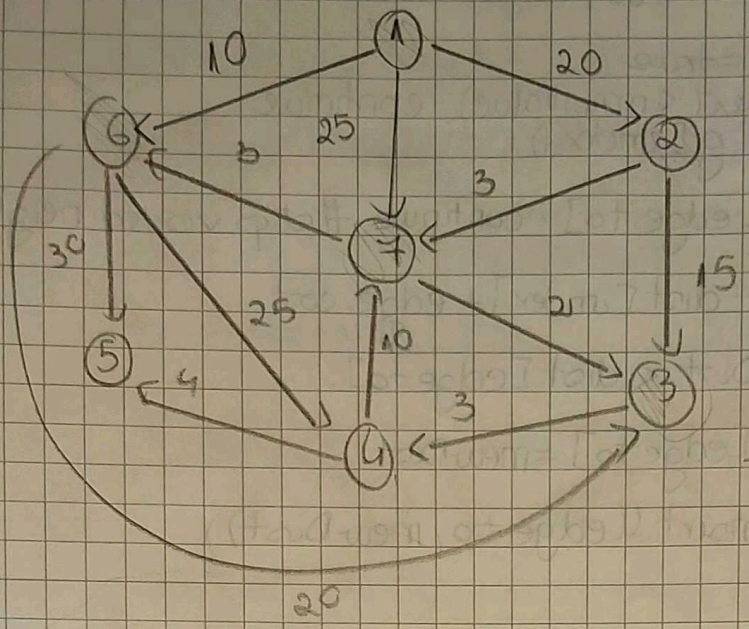
0	1	2	3	4	5
0	∞	∞	∞	∞	∞
0	5	∞	∞	∞	∞
0	5	1	∞	∞	∞
0	5	1	∞	13	∞
0	4	1	∞	13	∞
0	4	1	7	13	∞
0	4	1	7	9	∞
0	4	1	7	9	13
0	4	1	7	9	10

for 2.
dist[1] + edge.cost = 2 < 4

for 4
dist[1] + edge.cost = 24 < 13

(index, dist)
→ (0,0)
(1,5) → (1,4)
(2,1)
(4,13) → (4,9)
(3,7)
(5,13) → (5,10)

Practice problem



prev dictionary

	1	2	3	4	5	6	7
0	1	6	6	6	1	1	
0	1	6	6	6	1	2	
0	1	7	6	6	1	2	
0	1	4	3	6	1	2	
0	1	7	3	4	1	2	

(index, dist)

(1,0)
(2,20)
(7,10)
(7,25)
(4,35)
(3,30)
(3,40)
(4,33)
(3,28)
(4,30)
(4,34)

Start vertex: 1

dist

	1	2	3	4	5	6	7
	0	∞	∞	∞	∞	∞	∞
V = 1	0	20	∞	∞	∞	10	25
V = 6	0	20	30	35	40	10	25
V = 2	0	20	30	35	40	10	23
V = 7	0	20	25	35	40	10	23
V = 3	0	20	25	28	40	10	23
V = 4	0	20	26	30	32	10	23

Path 0 1 7 3 4 1 2

Shortest path from 1 to 5 : 1 → 2 → 7 → 3 → 4 → 5

20 + 3 + 2 + 3 + 4 = 32

path[0] = 1

path[1] = 2

path[2] = 7

path[7] = 3

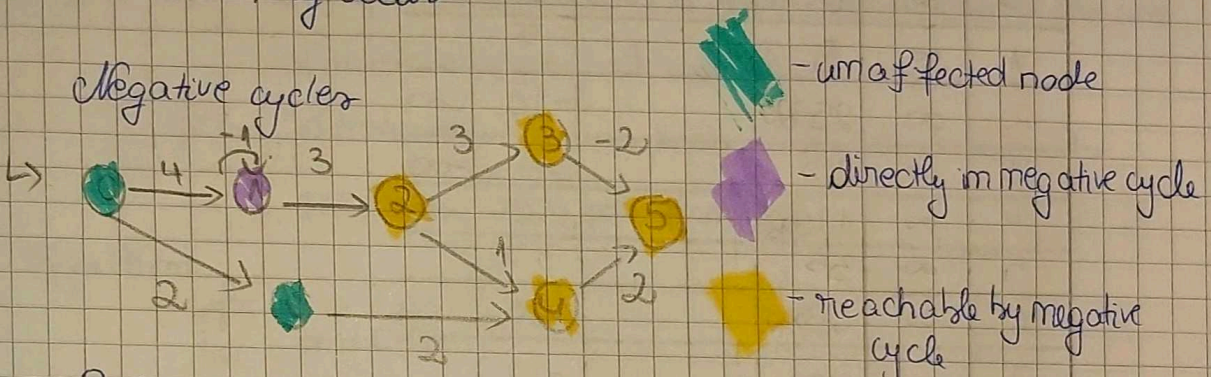
path[3] = 4

path(4)

BELLMAN-FORD ALGORITHM

- Single Source Shortest Path algorithm
- Complexity: $O(\epsilon V)$
  $O((\epsilon+V)\log(V))$ when using a binary heap priority queue
  ↳ Dijkstra
- useful when the graph has negative edge weights
- can be used to detect negative cycles and determine where they occur.

Negative cycles



Bellman-Ford Algorithm Steps

ϵ - edge number, V - vertices number, S - id of the starting node

D - array of size V that tracks the best distance from S to each node

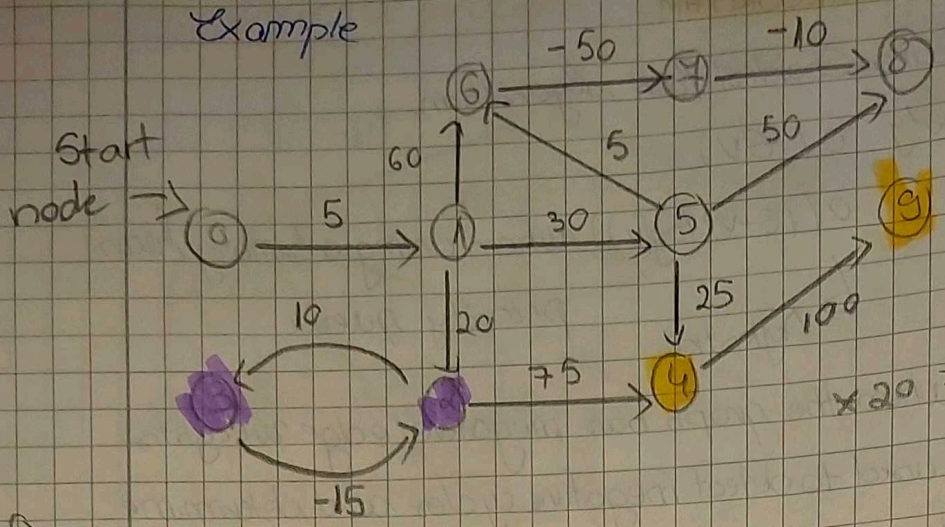
1.) Set every entry in D to $+\infty$

2.) D[S] = 0

3.) Relax each edge V-1 times

```
for (i = 0; i < V-1; i++):
    for edge in graph.edges: //Relax Edge
        if (D.[edge.from] + edge.cost < D[edge.to])
            D[edge.to] = D[edge.from] + edge.cost
// Repeat to find nodes caught in a negative cycle
    for (i = 0; i < V-1; i++):
        for edge in graph.edges:
            if (D[edge.from] + edge.cost < D[edge.to])
                D[edge.to] = -∞
```

Example



0	∞	0	...	0	
1	∞	5		5	
2	∞	25 ~~20~~		-20	-∞
3	∞	35		-5	-∞
4	∞	100 ~~60~~		60	-∞
5	∞	35		35	
6	∞	65 ~~40~~		40	
7	∞	-10		-10	
8	∞	~~-30~~		-20	
9	∞	200		160	-∞
		↑ iteration to relax edges		↑ how detecting any negative cycles	

function Bellman_Ford (list vertices, list edges, vertex source)

distance[], predecessor[]

// Step 1. Initialize the graph

for each vertex v in vertices:

if v is source then distance[v] := 0

else distance[v]: inf

predecessor[v] = null

// Step 2 Relax edges repeatedly

for i from 1 to size(vertices)-1:

for each edge (u,v) with weight w in edges:

if distance[u]+w < distance[v]:

distance[v] := distance[u]+w

predecessor[v] := u

// Step 3 : check for negative weight cycles
for each edge (u,v) with weight w in edges:
if distance[u]+w < distance[v]:
error "Graph contains a negative weight cycle"
return distance[], predecessor[]

Improvement

Input: s,t - 2 vertices

Output: dist: a map that associates to each accesible vertex the cost of the minimum cost walk from s to t

prev: a map that maps each accesible vertex to its predecessor on a path from s to t

Algorithm:

```
for x in V do:
    dist[x] = ∞
dist[s] = 0
changed = true; i = 0
while changed && i < V-1 do:
    changed = false
    for (x,y) in E do
        if dist[y] > dist[x] + cost(x,y) then
            dist[y] = dist[x] + cost(x,y)
            prev[y] = x
            changed = true
    i = i + 1
```

+ identify the negative cycles

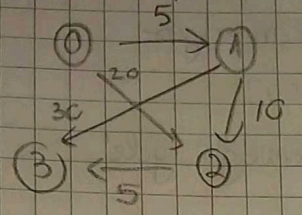
BELLMAN'S OPTIMALITY PRINCIPLE

$d(x,z) <= d(x,y) + d(y,z)$

if $x \ldots, y \ldots, z$ is an optimal walk from x to z, then

$x, \ldots, y$ is an optimal part from x to y

Example of manual execution



D=0, k=3

	changed	edge (x,y)	dist dict	prev dict
init	true		0 1 2 3 0 ∞ ∞ ∞	
iteration 1	false		0 1 2 3	0 1 2 3
	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	
		(0,2)	0 5 15 20	
		(1,2)	0 5 15 20	
		(1,3)	0 5 15 20	
		(2,3)	0 5 15 20	
	stop			

The minimum cost walk from 0 to 3 is:
$0 \to 1 \to 2 \to 3$ and has the cost of 20

SHORTEST PATHS AND MATRIX MULTIPLICATION

Assumption: negative edge weights may be present, but no negative weight cycles

Shortest path from v_i to v_j: p_{ij}^m s.t. $|p_{ij}^m| \leq m$

i.e. path p_{ij}^m has at most m edges

$i = j \Rightarrow |p_{ii}| = 0$ & $w(p_{ii}) = 0$

$i \neq j \Rightarrow$ decompose path p_{ij}^m into p_{ik}^{m-1} & $v_k \to v_j$

$\Rightarrow p_{ik}^{m-1}$ - shortest path from v_i to v_k

$\delta(v_i, v_j) = \delta(v_i, v_k) + w_{kj}$

d_{ij}^{m} = minimum weight of any path from v_i to v_j that contains at most m edges

$m = 0$

↳ $d_{ij}^{0} = \begin{cases} 0 & \text{, if } i = j \\ \infty & \text{, if } i \neq j \end{cases}$

$m \geq 1$

$$d_{ij}^{m} = \min\left\{ d_{ij}^{m-1}, \min_{1 \leq k \leq m \wedge k \neq j} \{ d_{ik}^{m-1} + w_{kj} \} \right\}$$

Computing the shortest path weights bottom up

$W = D^1$, compute a series of matrices $D^2, D^3, \ldots, D^{m-1}$ where

$D^m = (d_{ij}^{m})$, $m = 1, 2, \ldots, m-1$

↳ final matrix D^{m-1} contains actual shortest path weights

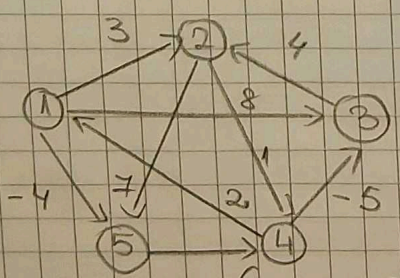
i.e. $d_{ij}^{m-1} = \delta(v_i, v_j)$

```
D^1 <- W
for m <- 2 to m-1 do
    D^m <- EXTEND(D^(m-1), W)
return D^(m-1)
```

```
EXTEND(D, W)
▲ D = (dij) - n×n matrix
for i <- 1 to n do
  for j <- 1 to n do
    dij <- ∞
  for k <- 1 to n do
    dij <- min{dij, dik + wkj}
return D  {cij = aik × bkj}
```

Running time: $\Theta(n^4) = \Theta(V^4)$

Example.



	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$D1 = W$

D^0 - identity matrix

$D^1 = D^0 \times W = W$

$D^2 = D^1 \times W = W^2$

$D^{m-1} = W^{m-1}$

$D2 = EXTEND(D1, W) = W^2$

	1	2	3	4	5
1	0	3	8	2	-4
2	3	0	-4	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	8	∞	1	6	0

1st line {0, ∞, ∞, ∞, -4}

1 → 0

2 → {3, 32, ∞, ∞, 0}

2 → 3

3 → {8, ∞, 8, 1, ∞, ∞} ✓ $8 = d_{11} + w_{11}, d_{13} + w_{13}$

$D3 \to W^3$

$$\begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^4 \quad \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

FLOYD-WARSHALL ALGORITHM

- APSP (all pairs shortest path) algorithm
- it can find the shortest path btw. all pairs of nodes
- Complexity: $O(V^3)$
- can detect negative cycles
- optimal way to represent our graph: 2D adjacency matrix m
  - m[i][j] - edge weight btw. i and j
  - m[i][j] = ∞ - no edge from i to j

Main idea: compute all intermediate routers btw. i and j to find the optimal path

↳ m[a][b] = min(m[a][b], m[a][c] + m[c][b])

dp[k][i][j] - shortest path from i to j routing through nodes {0, 1, ..., k-1, k}

dp[n-1] is the 2D matrix solution we're after

dp[k][i][j] = m[i][j] if k = 0

otherwise

dp[k][i][j] = min(dp[k-1][i][j], dp[k-1][i][k] + dp[k-1][k][j])

dp[k-1][i][k] + dp[k-1][k][j]: find best distance from i to j through node k reusing best solution from 0 - k-1

↓

It is possible to compute the solution for k in place → $O(V^2)$

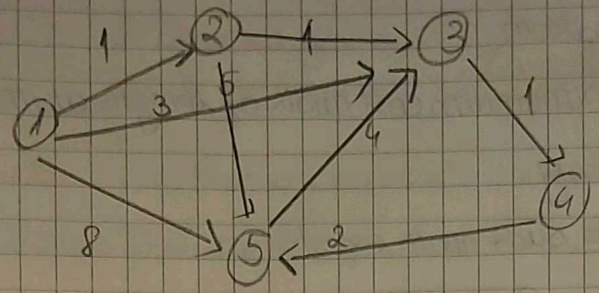
dp[i][j] = m[i][j] if k = 0

otherwise

dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j])

```
for (k=0; k<n; k++)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (dp[i][k] + dp[k][j] < dp[i][j])
                dp[i][j] = dp[i][k] + dp[k][j]
                next[i][j] = next[i][k]
```

Example.



$$D_0 = \begin{pmatrix} 0 & 1 & 3 & \infty & 8 \\ \infty & 0 & 1 & \infty & 5 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & 4 & \infty & 0 \end{pmatrix} \qquad P = \begin{pmatrix} 0 & 1 & 2 & 0 & 4 \\ 0 & 1 & 2 & 0 & 4 \\ 0 & 0 & 2 & 3 & 0 \\ 0 & 0 & 0 & 3 & 4 \\ 0 & 0 & 2 & 0 & 4 \end{pmatrix}$$

Using $k=1$ as intermediate vertex

$$D_1 = \begin{pmatrix} 0 & 1 & 3 & \infty & 8 \\ \infty & 0 & 1 & \infty & 5 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & 4 & \infty & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 1 & 2 & \infty & 6 \\ \infty & 0 & 1 & \infty & 5 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & 4 & \infty & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 6 \\ \infty & 0 & 1 & 2 & 5 \\ \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & 6 & 5 & 0 \end{pmatrix} \qquad D_5 = \begin{pmatrix} 0 & 1 & 2 & 3 & 5 \\ \infty & 0 & 1 & 2 & 4 \\ \infty & \infty & 0 & 1 & 3 \\ \infty & \infty & 6 & 0 & 2 \\ \infty & \infty & 4 & 5 & 0 \end{pmatrix}$$

MINIMUM COST WALK BY DYNAMIC PROGRAMMING

$w_{k,x}$ = cost of minimum cost walk of length at most k from r to x, or ∞ if no such walk exists

Recurrence relation

$$w_{0,r} = 0$$

$$w_{0,x} = \infty, \text{ for } x \neq r$$

$$w_{k+1,x} = \min\left(w_{k,x}, \min_{y \in IN(x)}\left(w_{k,y} + c(y,x)\right)\right)$$

We compute w row by row

To retrieve the path, we go back from t

LECTURE - MINIMUM COST WALK BTW ALL PAIRS OF VERTICES

Matrix multiplication

$w_{k,x,y}$ = cost of a minimum cost walk of length at most k from x to y

∞ if no such walk exists

Recurrence relation. Best case

$$\begin{cases} w_{0,x,x} = 0 \\ w_{1,x,y} = cost(x,y), \text{ if } (x,y) \text{ - edge of the graph} \\ w_{1,x,y} = \infty \text{ if } (x,y) \text{ - not an edge, } x \neq y \end{cases}$$

Actual recurrence:

$$w_{k+\ell,x,y} = \min_{z} (w_{k,x,z} + w_{\ell,z,y})$$

of length at most k; of length at most ℓ; of length at most k+ℓ

main idea: compute $w_{k,x,y}$ for a value of k based upon the already computed values for $k/2$: $k = k/2 + k/2$

Floyd-Warshall

```
for i=0 to n-1 do
    for j=0 to n-1 do
        if i==j then
            w[i,j]=0
        else if (i,j) is edge in G then
            w[i,j] = cost(i,j)
            next[i][j] = j
        else
            w[i,j] = infinity

for k=0 to n-1 do                    computing the minimum
    for i=0 to n-1 do
        for j=0 to n-1 do
            if w[i,j] > w[i,k] + w[k,j] then
                w[i,j] = w[i,k] + w[k,j]
                f[i,j] = f[i,k]
```

STRONGLY-CONNECTED COMPONENTS. KOSARAJU ALGORITHM

Input: G: directed graph

Complexity: $O(V+E)$

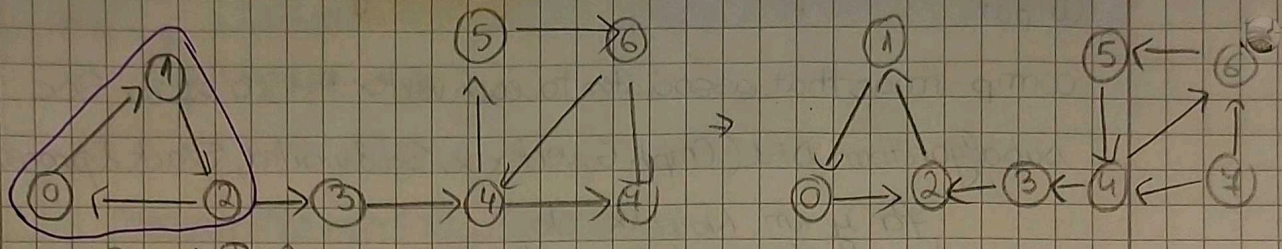
Output:

comp: map that associates to each vertex the ID of its Scc

```
subalgorithm DF1(Graph G, vertex x, Set& visited, Stack& processed)
    for y in Nout(x) do
        if y not in visited then
            visited.add(y)
            DF1(y)
    processed.push(x)

#Algorithm:
    Stack processed
    Set visited
    for x in X do
        if x not in visited then
            visited.add(x)
            DF1(G,x, visited, processed)
    visited.clear()
    Queue q
    int c=0
    while not processed.isEmpty() do
        x=processed.pop()
        if x not in visited then
            c=c+1 // new component
            comp[x]=c
            q.enqueue(x)
            visited.add(x)
            while not q.isEmpty() do // simulating DFS
                x=q.dequeue()
                for y in Nin(x) do
                    if y not in visited then
                        visited.add(y)
                        q.enqueue(y)
                        comp[y]=c
```

∴ On reversing all edges of the graph, the type of the graph won't change. (SCC will remain SCC).



Step 1 Perform DFS

visited
0 1 2 3 4 5 6 7

$\Delta = 0 \Rightarrow$ DFS (G, 0, visited, processed)

Nout(0) 1
DFS (G, 0, vis)
Nout(1) 2
DFS(2)
Nout(2) 3
DFS(3)
Nout(3) 4
DFS(4), Nout(4) → 5, 7
DFS(5) → 6
DFS(6) → 4 ×
→ 7

processed
0
1
2
3
4
5
6
7

visited clear()
C = 0

comp 0 1 2 3 4 5 6 7
1 1 1 2 3 3 3 4

Processed ∅ empty
$\Delta = 0$ ⇒ not in visited ⇒ C = 1

visited 0 2 1

G ∅ 2 1
3 ∅ is empty

3

4 → 6 → 5

x = 0
Nim(x) 2
x = 2
Nim(2) = 1 x = 1
Nim(1) = 0

Reflexive-transitive closure and the reduced graph

Reflexive-transitive closure → accessibility relation in of a graph that graph

Define the relation $x \sim y$ if x is accessible from y and y is accessible from x.

$\sim$ - equivalence relation, defines a partitioning of the set of vertices in the graph. The parts are the strongly connected components

Reduced graph: graph in which the vertices are the SCC of the original graph, where we put an edge between two vertices if there is an edge btw. a vertex of the 1st component and a vertex of the 2nd one

↳ acyclic graph

Directed Acyclic graphs

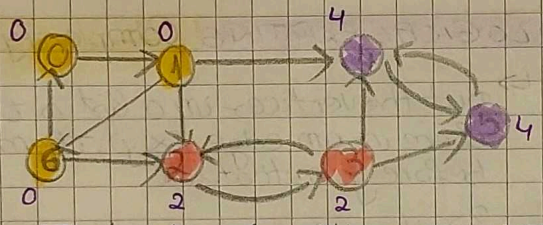
↳ directed graph having no cycle

↳ often used for dependency relations

ex. Vertices are topics in a book and an edge (x,y) means that a topic y cannot be understood before starting a topic x.

TARJAN'S SCC ALGORITHM. Complexity: O(V+E)

Low-link value of the node is the smallest node id reachable from that node when doing DFS (including itself).



ex. Low link of node 1 is 0

3 is 2

Tarjan's algorithm maintains a set of valid nodes from which we can update low-link values. Nodes are added to the stack as they are explored from for the 1st time. Nodes are removed from the stack each time a complete SCC is found.

Overview

↳ mark the id of each node as unvisited

↳ start DFS. Upon visiting a NODE assign it a low link value. Also mark current nodes as visited and add them to the stack.

↳ on DFS callback, if the previous node is on the stack then min the current node's low link value with the last low link value

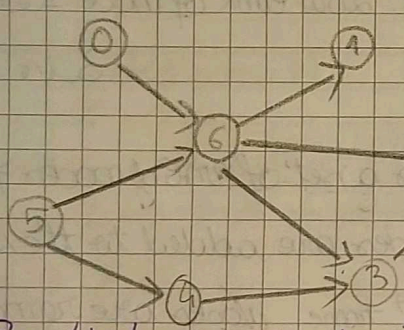
```
for (to : g[at]) : //visit outbound neighbours
  if (ids[to] == UNVISITED) : dfs(to)
  if (onStack[to]) :
    low[at] = min(low[at], low[to])
```

After visiting all neighbours, if the current node started a connected component then pop nodes off the stack until current node is reached

```
if idx[node_id] == low[node_id]:
    i = len(stack) - 1
    while i >= 0:
        node = stack[i]
        stack.pop()
        onStack[node] = False
        low[node] = idx[node_id]
        if node == node_id:
            break
        i -= 1
```

DAG

- **TOPOLOGICAL SORTING** Complexity: $O(V+E)$
  - ↳ put the vertices in a list s.t. whenever there is an edge (x,y), then x comes before y in that list
  - ↳ the solution is not generally unique



Possible topological sortings:

5 4 0 6 1 3 2
0 5 4 6 1 3 2

Property Topological sorting is possible, for a directed graph, if and only if there are no cycles in the graph.

PREDECESSOR COUNTING ALGORITHM

↳ we take a vertex with no predecessors, we put it on the sorted list and we eliminate it from the graph. Then, we take a vertex with no predecessors from the remaining graph and continue in the same way.

Finally, we either process all vertices and end up with the topologically sorted list, or we cannot get a vertex with no predecessors, which means we have a cycle.

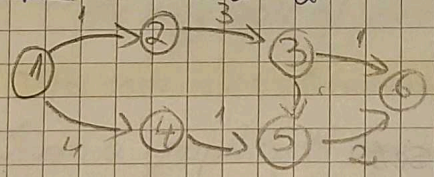
Main idea: don't actually remove vertices, but keep for each vertex, a counter of predecessors still in the graph

Input:
G: directed graph
Output:
sorted: a list of vertices in topological sorting order, or null if G has cycles

Algorithm

```
sorted = emptyList
Queue q
Dictionary count
for x in X do
    count[x] = inDegree(x) // how many predecessors it has
    if count[x] == 0 then // node with no predecessor
        q.enqueue(x)
while not q.isEmpty() do
    x = q.dequeue()
    sorted.append(x)
    for y in Nout(x) do
        count[y] = count[y] - 1
        if count[y] == 0 then
            q.enqueue(y)
if sorted.size() < X.size() then
    sorted = null
else
    for x in X
        v = q.dequeue
```

Example



Topological sorting orders:

1 2 3 4 5 6
1 2 4 3 5 6
1 4 2 3 5 6

	x,y	count: dict	q: queue	sorted: list
init		1 2 3 4 5 6 0 1 1 1 2 2	← 1 ←	[]
it 1	x=1 y=2 y=4	0 0 1 1 2 2 0 0 1 0 2 2	← 2 ← ← 4 ←	[1]
it 2	x=2 y=3	0 0 0 0 2 2	← 4 3 ←	[1,2]
it 3	x=4 y=5	0 0 0 0 1 2	← 3 ←	[1,2,4]
it 4	x=3 y=5 y=6	0 0 0 0 0 2 0 0 0 0 0 1	← 5 ← ← 5 ←	[1,2,4,3]
it 5	x=5 y=6	0 0 0 0 0 0	← 6 ←	[1,2,4,3,5]
it 6	x=6	0 0 0 0 0 0	← ←	[1,2,4,3,5,6]

sizeof(sorted) = 6

DEPTH-FIRST SEARCH-BASED ALGORITHM

→ based on Murphy's law: whatever you're starting to do, you realize something else

Input:
G - directed graph
Output:
sorted: a list of vertices in topological sorting order, or NULL, if G has cycles

```
Subalgorithm TOPOSORTDFS(Graph G, Vertex x, List sorted,
                         set FullyProcessed, set inProcess)
    inProcess.add(x)
    for y in Nin(x)
        if y in inProcess then //there is a cycle
            return false
        else if y not in FullyProcessed then
            ok = topoSortDFS(G, y, sorted, fullyProcessed,
                             inProcess)
            if not ok then
                return false
    inProcess.remove(x)
    sorted.append(x)
    fullyProcessed.add(x)
    return true

Algorithm:
    sorted = emptyList
    fullyProcessed = emptySet
    inProcess = emptySet
    for x in X do
        if x not in fullyProcessed then
            ok = topoSortDFS(G, x, sorted, fullyProcessed, inProcess)
            if not ok then
                sorted = null
                return
```

Highest cost path between two vertices in $O(V+E)$

Generalized: longest path cost in graph G

Topologically sort G
for each vertex $v \in V$ in linearized order
do $dist(v) = \max_{(u,v) \in E} \{dist(u) + w(u,v)\}$
return $\max_{v \in V} \{dist\ v\}$

```
dist[] = {-INF, -INF, ...}
dist[c] = 0 ; // source vertex
  for every vertex u in topological order
      if (u == z) break; // destination vertex
      for every adjacent vertex v of u
          if ((dist[u] + weight(u,v)) > dist[v])
              dist[v] = dist[u] + weight(u,v)
return dist[z]
```

TREES

Tree: undirected graph that is connected and has no cycles

↳ the graph is a tree

↳ the graph is connected and has at most n-1 edges

↳ the graph has no cycles and at least n-1 edges

↳ the graph is connected and minimal (if we remove any edge it becomes unconnected)

↳ the graph has no cycles and is maximal (if we add any edge, it closes a cycle)

↳ for any pair of vertices, there is a unique path connecting them

THE MINIMUM SPANNING TREE PROBLEM

Given a graph with non-negative costs, find a tree with the same vertices and a subset of the edges of the original graph (a spanning tree) of minimum total cost.

Algorithms for solving this: → Kruskal
→ Prim

KRUSKAL ALGORITHM

→ greedy algorithm

processing vertices

Complexity: $O(E \log E + E \log V)$

sorting edges

Steps:

1.) Sort all edges in ascending order of their weight

2.) i) Pick the smallest edge
ii) Check if the new edge forms a cycle in our spanning tree being formed
iii) If cycle is not formed → include the edge
else → discard the edge

Repeat step 2 until V-1 edges are included in the MST

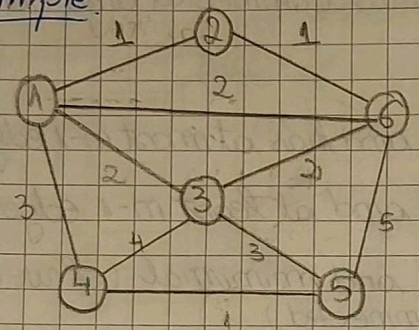
The basic algorithm

Input
G - undirected graph with costs

Output:
edges a collection of edges forming a MST

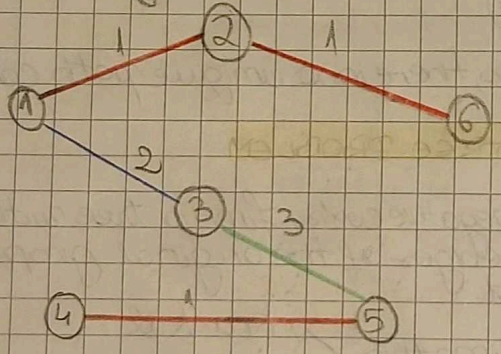
Algorithm:
$e_0, \ldots, e_{m-1}$ = list of edges sorted increasing by cost
edges = $\emptyset$
for i from 0 to m-1 do
    if edges $\cup \{e_i\}$ has no cycles then
        edges.add(e_i)

Example.



Edges	Cost
(1,2)	1
(2,6)	1
(4,5)	1
(1,6)	2
(1,3)	2
(3,6)	3
(3,4)	3
(3,5)	3
(4,3)	4
(5,6)	5

Building the MST



An MST of cost 8

1 edges with cost 1
(1,6) creates a cycle
(3,6) creates a cycle
2 edges with cost 2
3 edges with cost 3
(4,3) creates a cycle

How to test the existence of cycles?

Alternative: Keep track of the connected components of edges and notice that a cycle is formed adding a new edge $\Leftrightarrow$ the endpoints of the edge are in the same component

PRIM'S MINIMUM SPANNING TREE ALGORITHM

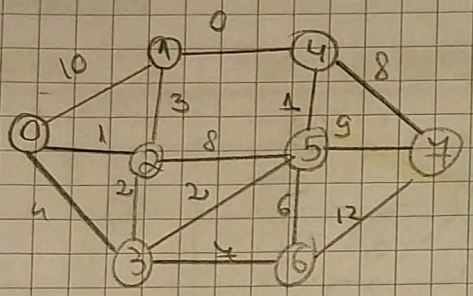
Lazy version $O(E * \log(E))$ ↳ works well on dense graphs

Eager version: $O(E * \log(V))$

Lazy Prim Overview

↳ PQ (priority queue) that sorts edges based on min edge cost (used to determine the next node to visit and the edge to get there)

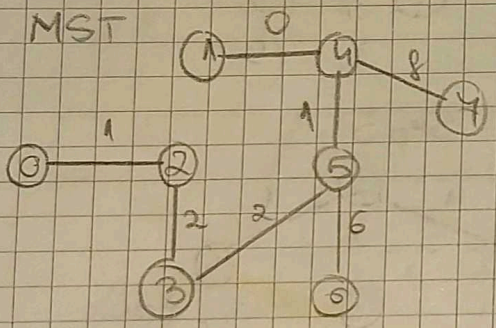
Start the algorithm on any node r. Mark r as visited and iterate over all edges of r, adding them to the PQ. While the PQ is not empty and a MST has not been formed, dequeue the next cheapest edge from the PQ. If the dequeued edge is outdated (the node it edge from the PQ points to has already been visited) then skip it and poll again. Else, mark the current node as visited and add the selected edge to the MST



Edges in PQ
start, end, cost

(0,1,10)
(0,2,1)
(0,3,4)
(2,1,3)
(2,5,8)
(2,3,2)
(2,0,1) → 0 was visited already
(3,5,2)
(3,6,7)
(5,4,1)
(5,6,6)
(5,7,9)
(4,1,0)
(4,7,8)
(6,7,12)

MST



The algorithm

Input:
G: directed graph with costs

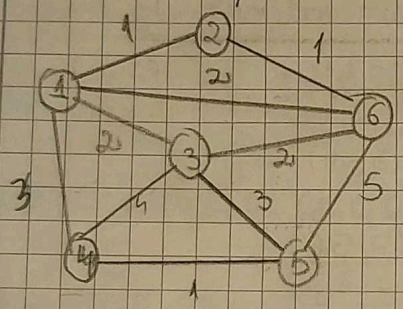
Output:
edges: a collection of edges forming a minimum cost spanning tree

Algorithm
PriorityQueue Q
Dictionary prev, dist
edges = ∅

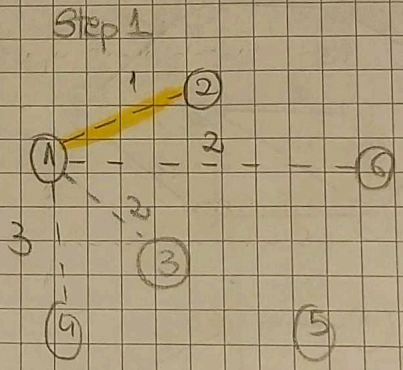
choose in x arbitrarily:
vertices = {r}
for x in N(r) do
dist[x] = cost(x,r)
prev[x] = r
Q.enqueue(x, d[x]) //2nd argument is priority

```
while not Q.isEmpty() do
    x = Q.dequeue()   //dequeue the element with minimum
                      //value of priority
    if x ∉ vertices then // not a part of MST yet
        edges.add({x, prev[x]})
        vertices.add(x)
        for y in N(x) do
            if y not in dist.keys() or cost(x,y) < dist[y] then
                dist[y] = cost(x,y)
                Q.enqueue(y, dist[y])
                prev[y] = x
```

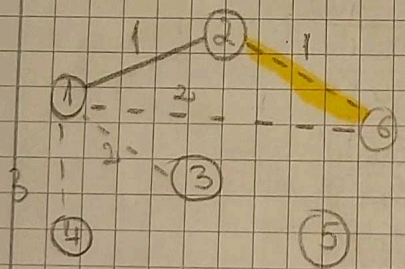
Example



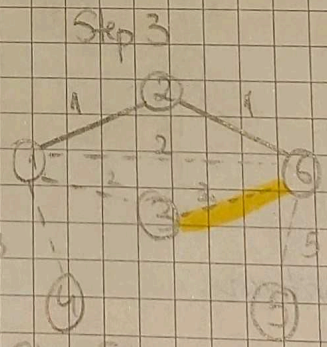
Step 1



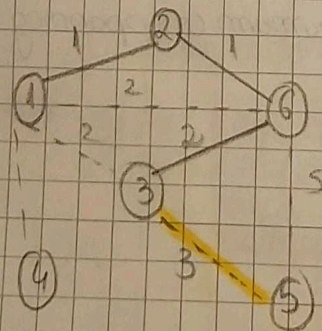
Step 2



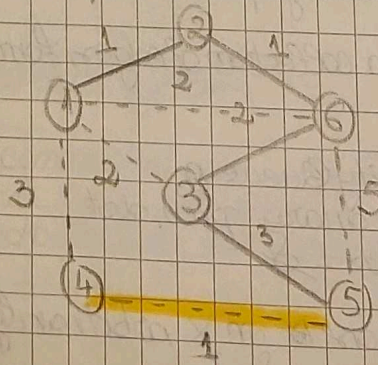
Step 3



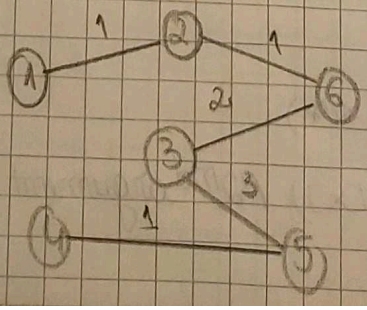
Step 4



Step 5



MST



ACTIVITY SCHEDULING PROBLEM

2) you are given a list of activities to be done for a project and each activity has a list of prerequisites (activities) and a duration

Output: a scheduling of activities (the starting and the ending time for each activity). If activity B depends on A, then B must start when or after A ends, however, two activities that do not depend on each other can be executed in parallel

Goal: execute the project as quickly as possible

Example:

Activities	Prerequisites	Duration	Earliest	Latest
0	-	1	0-1	1-2
5	-	2	0-2	0-2
6	0,5	5	2-7	2-7
4	5	1	2-3	6-7
1	6	2	7-9	8-10
3	4,6	2	7-9	7-9
2	3,6	1	9-10	9-10

Lab 4. Drum critic.

→ fiecare activitate este indivizibilă
→ fiecare activitate are o durată cunoscută
→ odată începută nu mai poate fi întreruptă

Relație de precedență "terminare - început"

↳ activitatea B nu poate începe decât după un interval de timp t_{AB} de la terminarea activității A

Durata totală de execuție → intervalul de timp în care se efectuează toate activitățile acestuia, respectând toate interdependențele dintre activități.

Activitățile sunt reprezentate în moduri, arcele prezintă relațiile de precedență dintre activități

Se mai adaugă grafului 2 moduri fictive:

X - activitate unică de început cu durata de execuție 0, care precede toate activitățile reale fără activități precedente

Y - activitate unică de sfârșit cu durata de execuție 0 care urmează activităților fără activități succesor reale

Structură nod graf

earliest → $t_m(v_i)$	v_i	$t^*_m(v_i)$ ← earliest finishing time
latest ← $t_M(v_i)$	$d(v_i)$	$t^*_M(v_i)$ → latest

beginning ~~finishing~~ time

v_i - activitatea

$d(v_i)$ - durata activității

Graful asociat proiectului este aciclic, deci va sorta topologic

n activități reale ⇒ graf cu n+2 moduri

v_0 - activitatea fictivă de început

v_{n+1} - activitatea fictivă de sfârșit

$v_{1,\dots,n}$ - activitățile reale ale proiectului

Topologic order: $v_0, v_1, \dots, v_n, v_{n+1}$

Earliest starting and finishing time:

$t_m(v_0) = t^*_m(v_0)$

for $i = 1, \dots, n+1$ do:

$t_m(v_i) = \max\{t^*_m(v_j) \mid v_j$ - predecesor al nodului $v_i\}$

$t^*_m(v_i) = t_m(v_i) + d(v_i)$

Latest starting and finishing time

$t^*_M(v_{n+1}) = t^*_m(v_{n+1})$

$t_M(v_i) = t^*_M(v_{n+1}) - d(v_{n+1})$

for $i = n, \dots, 0$ do

$t^*_M(v_i) = \min\{t_M(v_j) \mid v_j$ - succesor al nodului $v_i\}$

$t_M(v_i) = t^*_M(v_i) - d(v_i)$

Activitatea v este critică dacă

↳ $t_m(v) = t_M(v)$

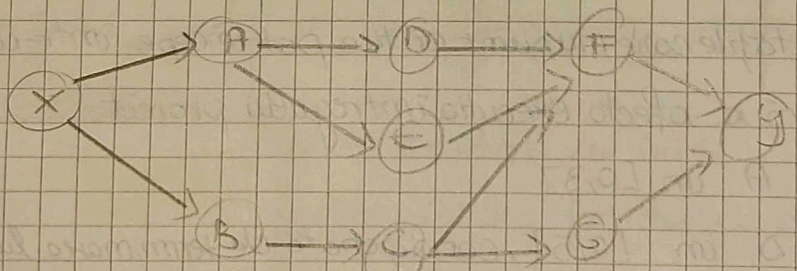
- v trebuie să înceapă la un timp exact pt. a nu afecta execuția întregului proiect

Pentru o activitate care nu este critică există un interval în care poate începe $[t_m(v), t_M(v)]$ fără a afecta execuția întregului proiect

Un drum critic - drum de la nodul fictiv de început la nodul fictiv de sfârșit ce conține succesiunea tuturor activităților critice

Exemplu

Activitate	Durată	Precedențe
A	2	-
B	3	-
C	5	B
D	3	A
E	3	A
F	3	C, D, E
G	2	C



Topological order: X A B D E C F G Y

Computing earliest starting and finishing time: t_m, t^*_m

$t_m(X) = t^*_m(X) = 0$

$t_m(A) = \max\{t^*_m(X)\} = 0$ $\quad t^*_m(A) = t_m(A) + 2 = 2$

$t_m(B) = \max\{t^*_m(X)\} = 0$ $\quad t^*_m(B) = 3$

$t_m(C) = \max\{t^*_m(X), t^*_m(B)\} = 3$; $t^*_m(C) = 8$

$t_m(D) = \max\{t^*_m(A)\} = 2$; $t^*_m(D) = 5$

$t_m(E) = \max\{t^*_m(A)\} = 2$; $t^*_m(E) = 5$

$t_m(F) = \max\{t^*_m(C), t^*_m(D), t^*_m(E)\} = 8$; $t^*_m(F) = 11$

$t_m(G) = 8$

$t^*_m(G) = 10$

$t_m y =$

$t_m(y) = \max\{t^*_m(G), t^*_m(F)\} = 11$; $t^*_m(y) = 11$

Latest starting and finishing time: t_M, t^*_M

$t_M(y) = t^*_M(y) = t^*_m(y) = t_m(y) = 11$

$t^*_M(G) = \min\{t_M(y)\} = 11$; $t_M(G) = t^*_M(G) - 2 = 9$

$t^*_M(F) = \min\{t_M(y)\} = 11$; $t_M(F) = t^*_M(F) - 3 = 8$

$t^*_M(C) = \min\{t_M(F), t_M(G)\} = 8$; $t_M(C) = t^*_M(C) - 5 = 3$

$t^*_M(E) = \min\{t_M(F)\} = 8$; $t_M(E) = t^*_M(E) - 3 = 5$

$t^*_M(D) = \min\{t_M(F)\} = 8$; $t_M(D) = t^*_M(D) - 3 = 5$

$t^*_M(B) = \min\{t_M(C)\} = 3$; $t_M(B) = t^*_M(B) - 3 = 0$

$t^*_M(A) = \min\{t_M(D), t_M(E)\} = 5$; $t_M(A) = t^*_M(A) - 2 = 3$

$t^*_M(X) = \min\{t_M(A), t_M(B)\} = 0 = t_M(X)$

Activități critice care încep la timpi bine stabiliți ($t_m = t_M$)

X: 0, B: 0, C: 3, F: 8, Y: 11

Drumul critic: X, B, C, F, Y

Timpul minim de execuție este 11

Activitățile care nu sunt critice pot începe într-un interval de

Critical activities: X:0, D:0, E:0, B:3; F:2; C:7; Y:8

NP-Hard problems

P, NP, NP-complete problems

Complexity classes

P class of problems

Execution time: no. of steps until reaching the final state

↳ compared to the size of input data (the no. of symbols used for encoding the input data on the tape)

Problems are classified according to the complexity of the best algorithm for solving them. (the best Turing machine that solves them)

Class P: consists in all problems for which there is a Turing machine and a polynomial such that the machine solves the problem and the no. of steps is bounded by the polynomial applied to the size of input data.

Non-deterministic Turing machine

↳ for it, there are several next actions for a single current state. Thus, there are multiple executions possible

For yes/no problems, the link between the executions and the answers:

-> the machine has 2 final states: ok / fail

-> if at least one execution finished in ok, the answer to the problem is yes

-> if all executions finish in fail, the answer is no

The class NP contains all problems for which there is a non-deterministic Turing machine that solves the problem in polynomial time.

$P \subseteq NP$

Generally, a NP problem is a problem for which a 'yes' answer means there is a ~~vector~~ solution that is a vector of polynomial size and whose correctness can be checked in polynomial time.

Examples:

↳ given a graph with 2 vertices, is there a path from 1st to 2nd?
is there a cycle in it?
is there a Hamiltonian cycle in it?
and an integer k, is there a Hamiltonian cycle of cost at most k^3

Problem A reduces to problem B if there is a way of solving A, as follows:

↳ the input for A is transformed, through a polynomial-time algorithm, into a valid input for B

↳ a solution for B is applied

↳ the output from B is transformed, through a polynomial-time algorithm, into an output for A

↳ the final result is the correct answer to the original problem A

NP hard and NP complete problems

NP-hard: all NP problems reduce to it

NP-complete: NP & NP-hard

Known NP-Complete problems

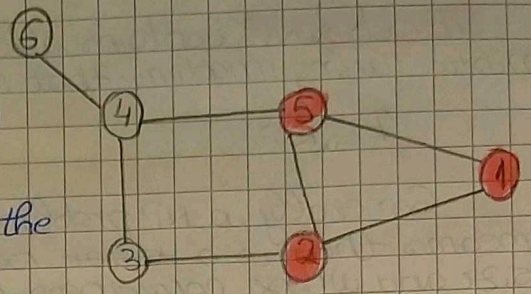
2) Hamiltonian cycle (TSP problem)
2) Clique;
2) vertex cover
2) vertex coloring

CLIQUE

↳ given an undirected graph, find a clique of maximum size

Clique in an undirected graph: subset of vertices of a graph such that the induced subgraph is complete (for every pair of vertices in the clique, there is an edge between them)

Example

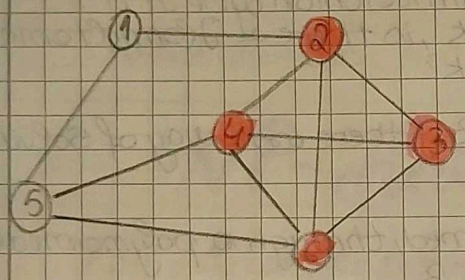


The graph has one maximum clique → the Δ $\{1,2,5\}$

and four more maximal cliques, the pairs $\{2,3\}, \{3,4\}, \{4,5\}, \{4,6\}$

Maximal clique - clique to which no more vertices can be added

Maximum clique - clique that includes the largest possible no. of vertices



The sub graph containing vertices 2, 3, 4, 6 forms a complete graph.

↳ 4-clique

Max-clique problem is a non-deterministic algorithm. We first try to determine a set of k distinct vertices and then we try to test whether these vertices form a complete graph.

Since there is no polynomial time deterministic algorithm to solve this problem, it is NP-complete.

Algorithm: Max-Clique (G, n, k)

```
S := ∅
for i = 1 to k do
    t := choice(1, ..., n)
    if t ∈ S then
        return failure
    S := S ∪ t
for all pairs (i, j) s.t. i ∈ S and j ∈ S and i ≠ j do
    if (i, j) is not an edge of the graph then
        return failure
return success
```

VERTEX COVER

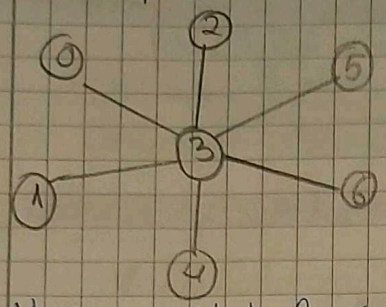
↳ given an undirected graph, find a vertex cover of minimum size

↳ vertex cover of a graph - subset of vertices which covers every edge

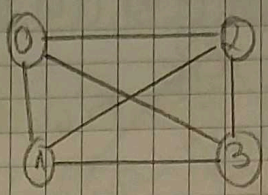
↳ an edge is covered of one of its endpoints is chosen

↳ NP Complete Problem

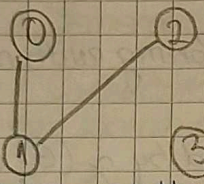
Examples



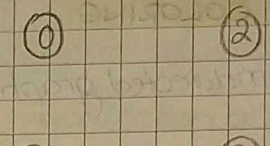
Minimum Vertex Cover is {3}



Minimum Vertex Cover
{0,1,2}, {0,1,3}, {1,2,3}



Minimum Vertex Cover is {1}



Minimum Vertex Cover is empty {}

Greedy Algorithm - doesn't always work Complexity O(log V)

Idea: Keep finding a vertex which covers the maximum number of edges

Step 1. Find a vertex v with maximum degree

Step 2. Add v to the solution and remove v and all its incident edges from the graph

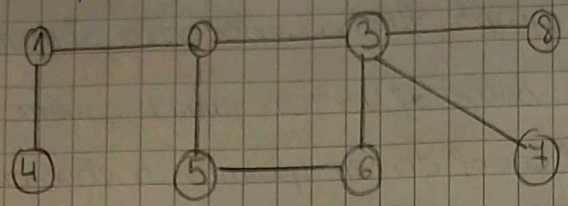
Step 3. Repeat until all the edges are covered

Algorithm (2) Complexity (O(V+E))

↳ Approximation Vertex Cover

```
C <- ∅
E' <- E
while E' ≠ ∅ do
    let (u,v) be an arbitrary edge of E'
    C <- C ∪ {(u,v)}
    remove from E' all edges incident on either u or v
```

Example



Initially $C = \emptyset$

$E' = \{(1,2), (2,3), (1,4), (2,5), (3,6), (5,6), (3,7), (3,8)\}$

C = 1 2 3 6 ⇒ (3,6) (5,6)

(1,4) ↓(1,2)

(2,5) (2,3) (3,7) (3,8)

OR 1,5,3 form a vertex cover of size 3

VERTEX COLORING

↳ given an undirected graph, find a vertex coloring with minimum no. of colors

Notation: The chromatic number of G, denoted by $\chi(G)$ is the minimum no. of colors required to color all vertices of G

↳ NP-complete problem

Basic Greedy Algorithm (does not guarantee use of minimum colors)

1. Color first vertex with first color.
2. Do following for remaining V-1 vertices

↳ consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

HAMILTONIAN CYCLE

↳ Hamiltonian Path in an undirected graph: path that visits each vertex exactly once

Backtracking algorithm

↳ create an empty path array and add 0 to it
↳ add other vertices starting from vertex 1
↳ before adding a vertex, check whether it is adjacent to the previously added vertex and not already added.

If we find such a vertex, we add it as a part of our solution.
else return false.

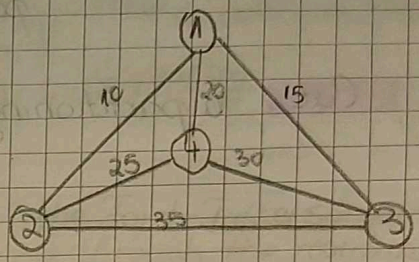
TRAVELLING SALESMAN PROBLEM. (TSP)

↳ given a digraph with costs, find a minimum cost Hamiltonian cycle

↳ given a set of cities and distance btw. every pair of cities, find the shortest possible route that visits every city exactly once and returns to the starting point.

Example A TSP 1-2-4-3-1

Cost 10+25+30+15=80



Complexity NP-hard problem

Naive solution

1.) Consider city 1 as the starting and ending point

2.) Generate all $(n-1)!$ permutations of cities

3.) Calculate the cost of every permutation and keep track of minimum cost permutation

4.) Return the permutation with the minimum cost.

Flows

Transport graph and maximum flow

In a transport graph, we have a source vertex, where a producer of some commodity is located, a destination vertex, and all the other vertices act as intermediates.

Each edge represents the possibility to transport a certain amount of that commodity. The amount is the capacity of that edge.

Goal: plan how to transport a maximum amount of that commodity from the source to the destination

We have

↳ a directed graph $G=(X,E)$

↳ a source vertex s and a destination vertex t

↳ each edge (x,y) has a positive capacity $cap(x,y)$

Flow - assignment of a flow value to each edge s.t.

↳ $0 \leq flow(x,y) \leq cap(x,y)$

↳ for each edge, except for the source or the destination, the inbound flow is equal to the outbound flow.

$\forall x \in X \quad \sum_{y \in in(x)} flow(y,x) = \sum_{y \in out(x)} flow(x,y)$

2> for the source vertex, there is a positive net outbound flow
⇒ total value of the flow

2> for the destination vertex, there is a positive net inbound flow

Classical problem: set a maximum flow in the transport graph to maximize the total flow value among all possible flows

Cut: a partitioning of the vertices into two sets
↳ one containing the source
↳ the other containing the destination

Net flow across the cut: "left to right" flow (the total flow along the edges leading from the set containing the source to the set containing the destination) minus the "right to left" flow

Assumming the cut is $(A, X \setminus A)$, $s \in A$, $t \in X \setminus A$

$$\text{flow}(A, X \setminus A) = \sum_{(x,y) \in E,\ x \in A, y \in X \setminus A} \text{flow}(x,y) - \sum_{\substack{(x,y) \in E \\ x \in X \setminus A \\ y \in A}} \text{flow}(x,y)$$

Capacity of the cut: 'left-to right' capacity

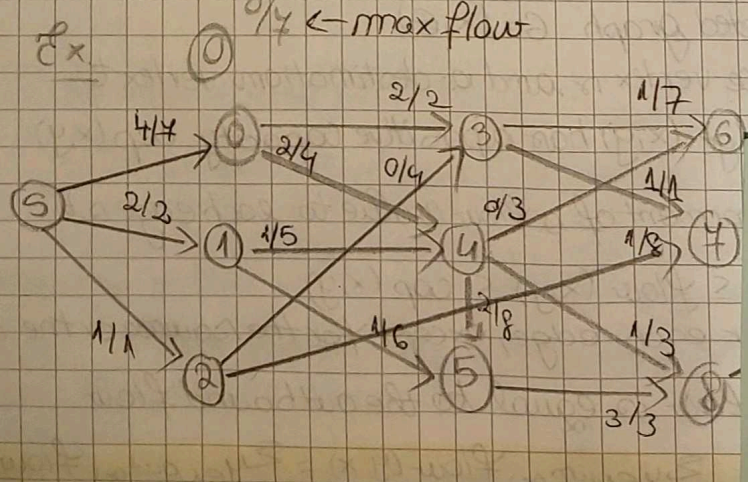
$$\text{cap}(A, X \setminus A) = \sum_{\substack{(x,y) \in E \\ x \in A \\ y \in X \setminus A}} \text{cap}(x,y)$$

Maximum flow is obtained when: all "right to left" edges are saturated and all "left to right" edges hole zero flow

MAX FLOW: FORD-FULKERSON METHOD

Q: With an infinite input source, how much "flow" can we push through the network given that each edge has a capacity?

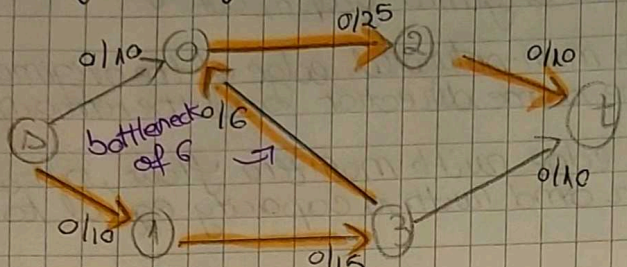
Ex



In this flow graph, the max flow is 7
↳ 1+2+4=7
Sum of the flows entering the sink node.

Flow graph / flow network

↳ directed ~~flow~~ graph where each edge has a certain capacity which can receive a certain amount of flow. The flow running through an edge must be less than or equal to the capacity.



Each edge in the flow graph has a certain flow and a capacity.

To find the maximum flow, the Ford-Fulkerson method repeatedly finds augmenting paths through the residual graph and augments the flow until no more augmenting paths can be found.

Augmenting path → path of edges in the residual graph with unused capacity greater than zero from the source to the sink t.

↳ it can only flow through edges which are not fully saturated yet

In the highlighted augmenting path, the bottleneck is the smallest edge on the path. We can use the bottleneck value to augment the flow along the path.

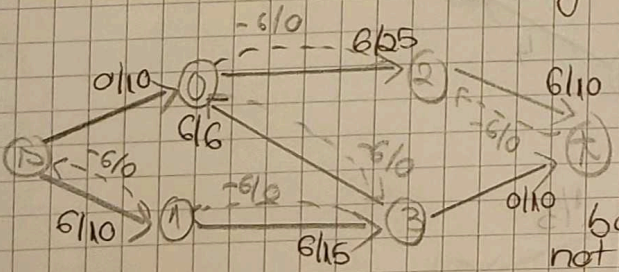
The smallest remaining capacity of any edge along the augmenting path is

$min(10-0, 15-0, 6-0, 25-0, 10-0) = 6$

↳ difference btw. the capacity and the current flow of an edge

Augmenting the flow : updating the flow values of the edges along the augmenting path

↳ also need to decrease the flow along each residual edge by the bottleneck value



Residual edges exist to "undo" bad augmenting paths which do not lead to a maximum flow.

Residual graph - graph which also contains the residual edges

The Ford-Fulkerson method continues finding augmenting paths and augments the flow until no more augmenting paths from s->t exist.

The sum of the bottlenecks found in each augmenting path is equal to the **max flow**.

Ford-Fulkerson algorithm

1.) Start with a zero flow

2.) For the current flow, construct a residual graph containing the same vertices as the original graph, but:

→ for each non-saturated edge of the original graph, put an edge with the same direction and the remaining capacity

→ for each edge with nonzero flow, put an edge in the reverse direction and with a capacity equal to the value of that flow

3.) Find a path from source to destination in the residual path

4.) compute the capacity of the above path

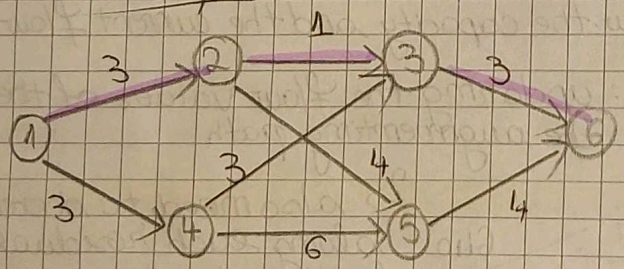
5.) update the flow: For forward edges, increase the flow by a value equal to the capacity of the path

For backward edges, decrease the flow on the corresponding forward edges by the same value

6.) Repeat the steps 2-5 until no path can be found in the residual path from source to destination

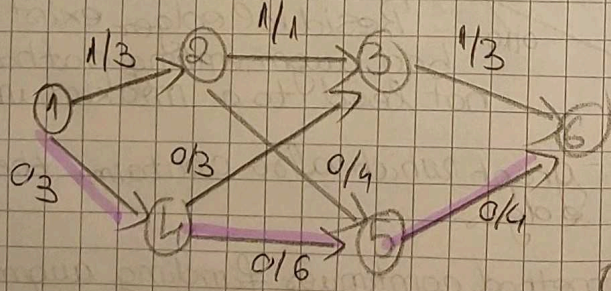
Ford-Fulkerson theorem: The value of the maximum flow is equal to the capacity of the minimum cut

Example.

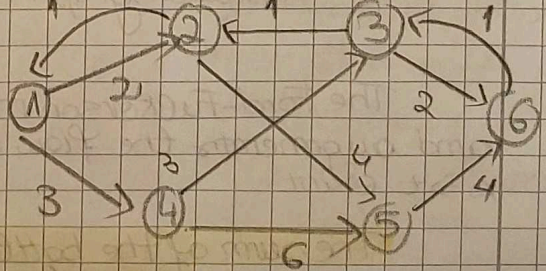


Augmenting path

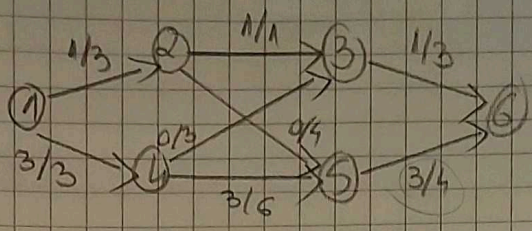
1→2→3→6 : capacity 1

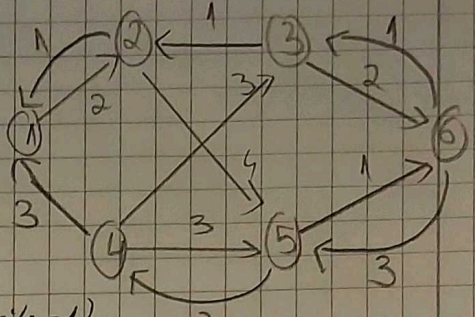


with residual graph

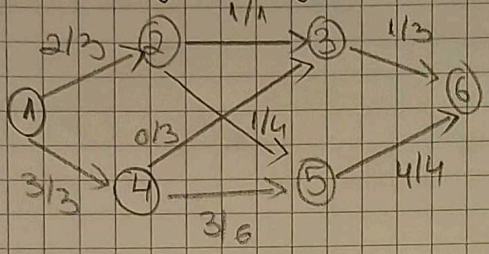


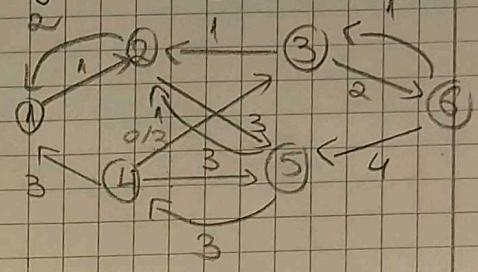
After augmenting path 1,4,5,6 (capacity = 3)





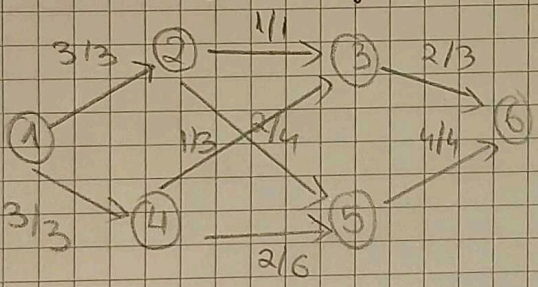
After augmenting path 1,2,5,6 (capacity = 1)

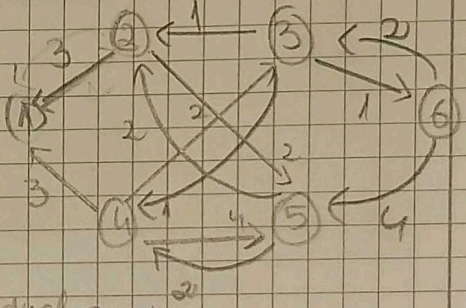




We cannot take 1,4,3,6 bc. 1/4 is saturated

Final augmenting path 1,2,5,4,3,6 (capacity = 1)





☒ The path in found in the residual graph

☒ augmenting path can be found anymore.

The cut {1,3} to {2,3,4,5,6} is saturated. cap. = 6, flow = 6