Drum elementar = nu se repeta noduri
Drum simplu = nu se repeta muchii
Graf partial = putem elimina muchii
Subgraf = putem elimina noduri
Graf biconex -> graf care nu are noduri critice
Graf tare conex -> graf in care exista drum de la x la y, oricare ar fi x si y varfuri
Nr max muchii graf bipartit: par(n^2 / 4) impar(n + 1)(n - 1) / 4
Graf transpus (doar pe graf orientat) = inversam sensul muchiilor
Graf complementar = daca avem muchie -> eliminam
= daca nu aveam original -> add
V = vertices;
E = edges;
Complexitate BFS and DFS -> O(V + E)
BFS -> arborele de inaltime minima

Muchie critica -> niv_min[j] > niv[i], unde j este fiu a lui i.
Nod critic -> radacina -> are cel putin 2 fii in arborele DF
-> alt nod -> are cel putin un fiu j cu niv_min[j] >= niv[i]
Complexitate algoritm muchii/noduri critice: O(V + E)
Complexitate Sortare Topologica -> O(V + E)
Complexitate Kruskal -> O(E log V)
Complexitate Prim -> O(V^2) or O(E log V)
Complexitate Dijkstra -> O(V^2) or O(E log V)
Complexitate Bellman-Ford -> O(E * V)
Complexitate Distante minime in DAG -> O(V+E)
Complexitate Floyd-Warshall -> O(V^3)
Complexitate Edmonds-Karp -> O(V*E^2)
Complexitate algoritmul lui Mickey Mouse (Hierholzer) -> O(E)
Complexitate algoritmul lui Tarjan -> O(V+E)
Taietura in retea -> este o bipartitie a multimii varfurilor a.i. Sursa si destinatia sunt in partitii diferite
Daca graful nu este biconex => nu este hamiltonian
Graf eulerian -> are toate nodurile de grad par
Algoritm verificare graf hamiltonian -> O(n^2 * 2^n)

$$\sum_{f \in F} d_M(f) = 2|E|$$

|V| - |E| + |F| = 2

Fie G=(V, E) un graf planar conex **bipartit** cu n=|V|>2 și m=|E|. Atunci:

a) $m \leq 2n - 4$

b) $\exists\ x \in V$ cu $d(x) \leq 3$.

Orice graf planar conex este 6 -colorabil.

Fie G=(V, E) un graf planar conex cu n=|V|>2 și m=|E|.

Atunci:

a) m ≤ 3n - 6

b) ∃ x ∈ V cu d(x) ≤ 5.

Demonstrație

$$\sum_{f \in F} d_M(f) = 2|E|$$

$$d_M(f) \geq 3$$

$$|V| - |E| + |F| = 2$$

|V| - |E| + |2E|/3 <= 2

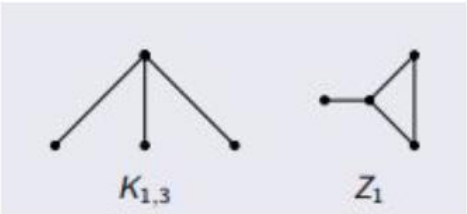
3|V| - |E| <= 6 -> a

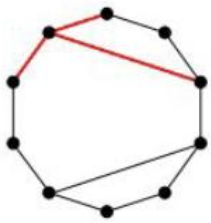
Dirac's Theorem (1952) — A simple graph with n vertices ($n \geq 3$) is Hamiltonian if every vertex has degree $\frac{n}{2}$ or greater.

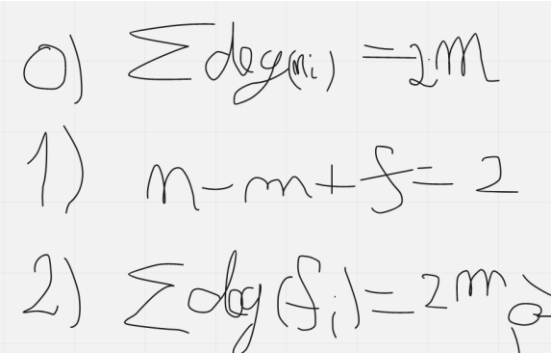
Ore's Theorem (1960) — A simple graph with n vertices ($n \geq 3$) is Hamiltonian if, for every pair of non-adjacent vertices, the sum of their degrees is n or greater.

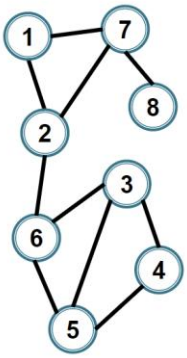
Fie G un graf conectat cu ordinal n ≥ 3, conectivitatea κ(G), și numărul de independență α(G). Daca κ(G) ≥ α(G), atunci G este hamiltonian

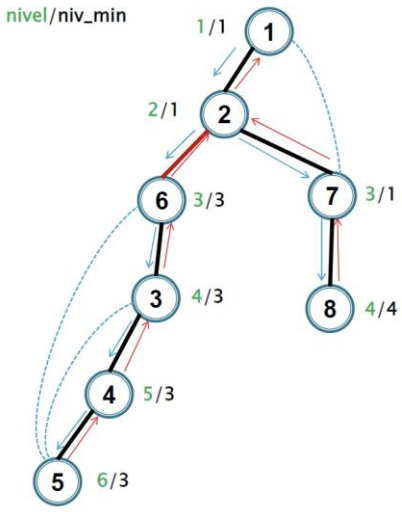
Daca G este un graf 2-conectat ¸si liber de {K1,3, Z1} atunci G este hamiltonian.





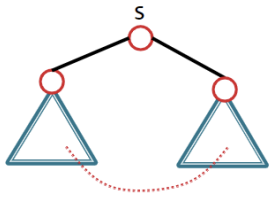






- Arborele DF
  - rădăcina s este punct critic ⇔

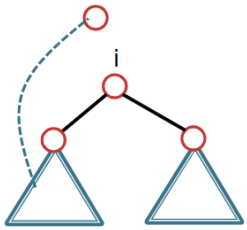
    are cel puţin 2 fii în arborele DF



nu există muchii între subarbori
(de traversare)

  - un alt vârf i din arbore este critic ⇔

    are cel puțin un fiu j cu
    niv_min[j] ≥ nivel[i]



Componente tare conexe: algoritm Kosaraju

- Următorul algoritm de timp liniar (adică Θ(V +E)) determină componentele tare conexe ale unui graf orientat G = (V, E) folosind două căutări în adâncime, una în G şi una în GT.
- Componente-Tare-Conexe(G)
- 1: apelează CA(G) pentru a calcula timpii de terminare f[u] pentru fiecare vârf u
- 2: calculează GT
- 3: apelează CA(GT), dar în bucla principală a lui CA, consideră vârfurile în ordinea descrescătoare a timpilor f[u] (calculaţi în linia 1)
- 4: afişează vârfurile fiecărui arbore în pădurea de adâncime din pasul 3 că o componentă tare conexă separată

Sortare topologică - Algoritm

```
coada C ← ∅;
adauga in C toate vârfurile v cu d⁻[v]=0

cat timp C ≠ ∅ executa
   i ←  extrage(C);
   adauga i in sortare
   pentru ij ∈ E executa
      d⁻[j] = d⁻[j] - 1
      daca d⁻[j]=0 atunci
          adauga(j, C)
```

Kruskal

Complexitate

Varianta 2 – dacă folosim arbori Union/Find

- Sortare -> O(m log m) = O(m log n)
- n * Initializare -> O(n)
- 2m * Reprez -> O(m log n)
- (n-1) * Reuneste -> O(n log n)

O(m log n)

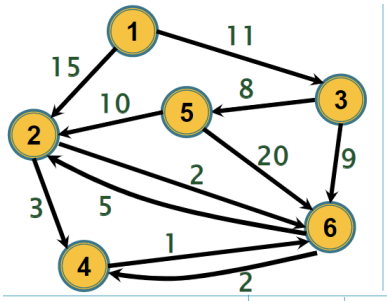
Prim

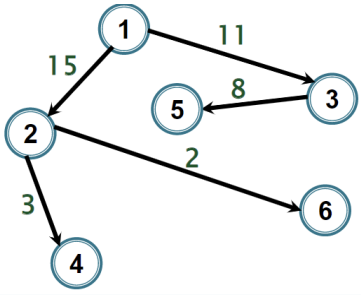
Varianta 2 – memorarea vârfurilor din într-un min-heap Q (min-ansamblu)

- **Iniţializare Q** -> O(n)
- **n * extragere vârf minim** -> O(n log n)
- **actualizare etichete vecini** -> O(m log n)

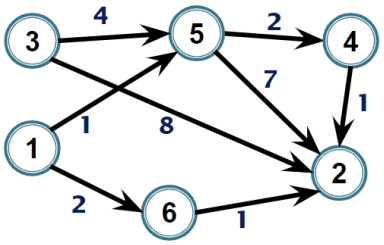
O(m log n)

Dijkstra





d/tata	1	2	3	4	5	6
	[0/0,	∞/0,	∞/0,	∞/0,	∞/0,	∞/0]
Sel. 1:	[– ,	15/1,	11/1,	∞/0,	∞/0,	∞/0]
Sel. 3:	[– ,	15/1,	– ,	∞/0,	19/3,	20/3]
Sel. 2:	[– ,	– ,	– ,	18/2,	19/3,	17/2]
Sel. 6:	[– ,	– ,	– ,	18/2,	19/3,	–]
Sel. 4:	[– ,	– ,	– ,	– ,	19/3,	–]
Sel. 5:	[– ,	– ,	– ,	– ,	– ,	–]



Sortare topologică 1, 3, 6, 5, 4, 2

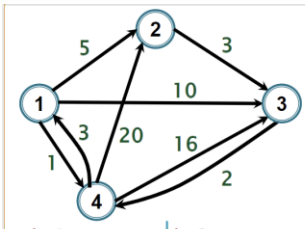
s=3 – vârf de start

Ordine de calcul distanțe: 1, 3, 6, 5, 4, 2

d/tata	1	2	3	4	5	6
	[∞/0,	∞/0,	0/0,	∞/0,	∞/0,	∞/0]
u = 1:	[∞/0,	∞/0,	0/0,	∞/0,	∞/0,	∞/0]
u = 3:	[∞/0,	8/3,	0/0,	∞/0,	4/3,	∞/0]
u = 6:	[∞/0,	8/3,	0/0,	∞/0,	4/3,	∞/0]
u = 5:	[∞/0,	8/3,	0/0,	6/5,	4/3,	∞/0]
u = 4:	[∞/0,	7/4,	0/0,	6/5,	4/3,	∞/0]
u = 2:	[∞/0,	7/4,	0/0,	6/5,	4/3,	∞/0]

Algoritmi – G=(V, E) graf orientat

G – neponderat Parcurgere lățime BF	G – ponderat, ponderi >0 Algoritmul lui Dijkstra	G – ponderat fără circuite
BF(s) coada C ← ∅; adauga(s, C)	Dijkstra(s) (min-heap) Q ← V {se putea incepe doar cu Q ← {s} +vector viz; v∈Q ⇔ v nevizitat}	DAGS(s) SortTop ← sortare_topologica(G)
pentru fiecare u∈V d[u]=∞;tata[u]=viz[u]=0 viz[s]← 1; d[s] ← 0	pentru fiecare u∈V d[u] = ∞; tata[u]=0 d[s] = 0	pentru fiecare u∈V d[u] = ∞; tata[u]=0 d[s] = 0
cat timp C ≠ ∅ u ← extrage(C); pentru fiecare uv∈E daca viz[v]=0 d[v] ← d[v]+1 tata[v] ← u adauga(v, C) viz[v] ← 1	cat timp Q ≠ ∅ u = extrage(Q) vârf cu eticheta d minimă pentru fiecare uv∈E daca v∈Q si d[u]+w(u,v)<d[v] d[v] = d[u]+w(u,v) tata[v] = u repara(v,Q)	pentru fiecare u ∈ SortTop pentru fiecare uv∈E daca d[u]+w(u,v)<d[v] d[v] = d[u]+w(u,v) tata[v] = u
scrie d, tata	scrie d, tata	scrie d, tata
O(n+m)	O(m log(n)) / O(n^2)	O(n+m)



W=d=

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	20	16	0

p=

0	1	1	1
0	0	2	0
0	0	0	3
4	4	4	0

d=

k=1

0	5	10	1
∞	0	3	∞
∞	∞	0	2
3	8	13	0

k=2

0	5	8	1
∞	0	3	∞
∞	∞	0	2
3	8	11	0

k=3

0	5	8	1
∞	0	3	5
∞	∞	0	2
3	8	11	0

k=4

0	5	8	1
8	0	3	5
5	10	0	2
3	8	11	0

p=

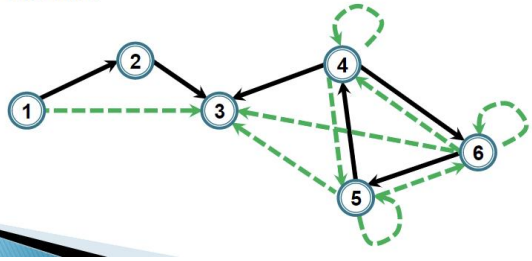
0	1	1	1
0	0	2	0
0	0	0	3
4	1	1	0

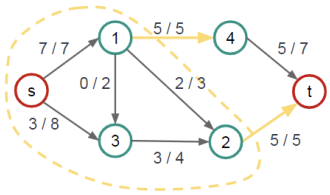
0	1	2	1
0	0	2	0
0	0	0	3
4	1	2	0

0	1	2	1
0	0	2	3
0	0	0	3
4	1	2	0

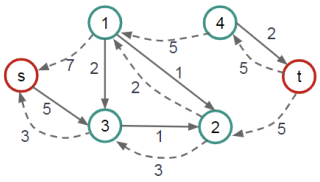
0	1	2	1
4	0	2	3
4	1	0	3
4	1	2	0

Închiderea tranzitivă

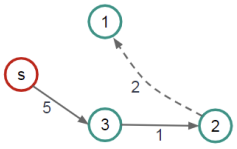




Graful rezidual G_f



BF(s) - în graful rezidual



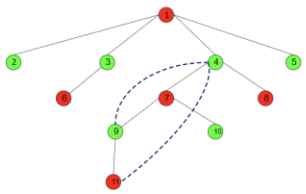
Nu mai există drum de creștere ⇒ s-t flux maxim (valoare 10) + s-t tăietură minimă (de capacitate tot 10, determinată de vârfurile accesibile din s în G_f: S = { s, 1, 3, 2 })

- **Teorema König – Caracterizarea grafurilor bipartite**

Demonstrație ⇐ Presupunem G conex.

Colorăm propriu cu 2 culori un arbore parțial T al său.

Orice altă muchie uv din graf are extremitățile colorate diferit deoarece formează un ciclu elementar cu lanțul de la u la v din arbore și acest ciclul are lungime pară, deci u și v se află pe niveluri de paritate diferită în T



```python
from typing import List

class Solution:   #its basically a modified bfs that can visit a vertex multiple times if the mask asociated
                  #to that node has changed since the last appearance
    def shortestPathLength(graph: List[List[int]]) -> int:
        if len(graph) == 1:
            return 0

        noOfVertices = len(graph)
        allNodesVisited = (1 << noOfVertices) - 1  # <- all bits set to 1
        que = []
        visited = set() # to keep track of visited nodes and the bits that were set the last time a node was visited

        for i in range(noOfVertices):
            mask = 1 << i
            que.append((i,(0, mask)))  # vertex, (distance, mask)
            visited.add((i, mask))

        while que:
            front = que.pop(0)
            vrtx = front[0]
            dist = front[1][0]
            mask = front[1][1]
            for nvertex in graph[vrtx]:
                newMask = (mask | (1 << nvertex)) # will set the bit of the neighbour to 1
                if newMask == allNodesVisited:
                    return dist + 1
                if (nvertex, newMask) in visited: # if already in visited, we dont add it to que
                    continue
                visited.add((nvertex, newMask))
                que.append((nvertex, (dist+1, newMask)))

print(Solution.shortestPathLength([[1, 2, 3], [0], [0], [0]]))
```