Grafuri

Banu Denis, Ioniță Alexandru

Faculty of Computer Science,
Al. I. Cuza University of Iași

Overview i

Overview ii

Componente tare conexe

Drumuri minime in graf

Arbore partial de cost minim - APM

Probleme

Graf

Definiție

Un graf este o pereche ordonată de mulțimi $G = \{V, E\}$ unde V este o mulțime de elemente numite noduri(vârfuri) și E este o multime de elemente numite muchii(un element din mulțimea E fiind o pereche(ordonată sau neordonată) de elemente din V.

Graf

Definiție

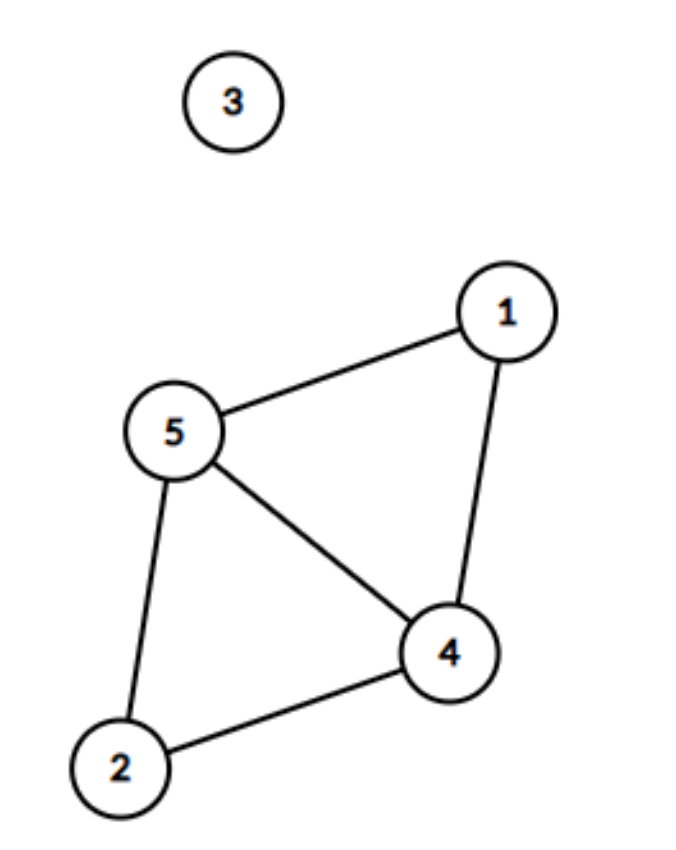
Un graf este o pereche ordonată de mulțimi $G = \{V, E\}$ unde V este o mulțime de elemente numite noduri(vârfuri) și E este o multime de elemente numite muchii(un element din mulțimea E fiind o pereche(ordonată sau neordonată) de elemente din V.

Informal

Un graf este alcătuit din noduri și muchii. Fiecare muchie unește două noduri.

Matrice de adiacență

Se construiește o matrice A care poate avea două valori (0 sau 1).
Daca A[x][y] este 1 atunci există muchie între nodurile x și y.



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

Matrice de adiacență

```
int A[NMAX][NMAX], n, m, x, y;
cin >> n >> m;
for (int i = 1; i <= m; i++){
    cin >> x >> y;
    A[x][y] = A[y][x] = 1;
}
```

Complexitate memorie: $O(N^2)$

Observații

- Matricea de adiacență se poate folosi atunci când numărul de noduri este mic ($\leq$1000).
- Este utilă atunci când trebuie să determinăm rapid dacă există muchie între două noduri.

Liste de adiacență

Pentru fiecare nod se construiește o listă în care sunt introduse toate nodurile cu care acesta are o muchie comună.

Liste de adiacență

```
vector<int> v[NMAX];
cin >> n >> m;
for (int i = 1; i <= m; i++){
    cin >> x >> y;
    v[x].push_back(y);
    v[y].push_back(x);
}
```

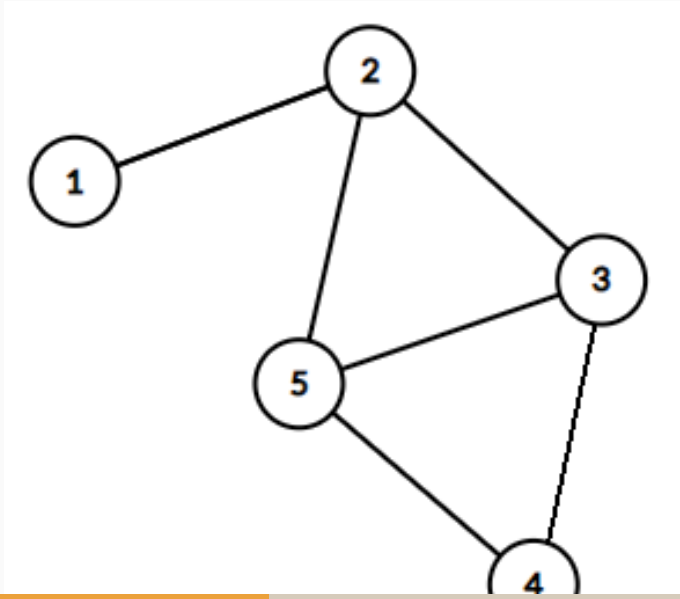
Complexitate memorie: O(m+n)

BFS - parcurgere în lățime

- Nodurile sunt parcurse în ordine în funcție de distanța față de nodul de început.
- Se utilizează o coadă.
- La început se introduce nodul de plecare în coadă.
- În fiecare etapă se scoate câte un nod din coadă, se parcurg toți vecinii acestuia și se introduc în coadă nodurile care nu au mai fost vizitate.
- Algoritmul se termină atunci când nu mai sunt elemente în coadă.

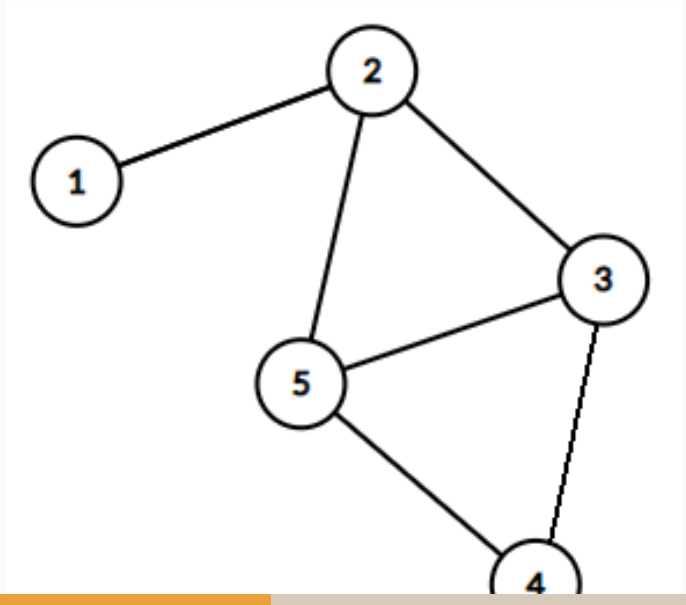
BFS - parcurgere în lățime

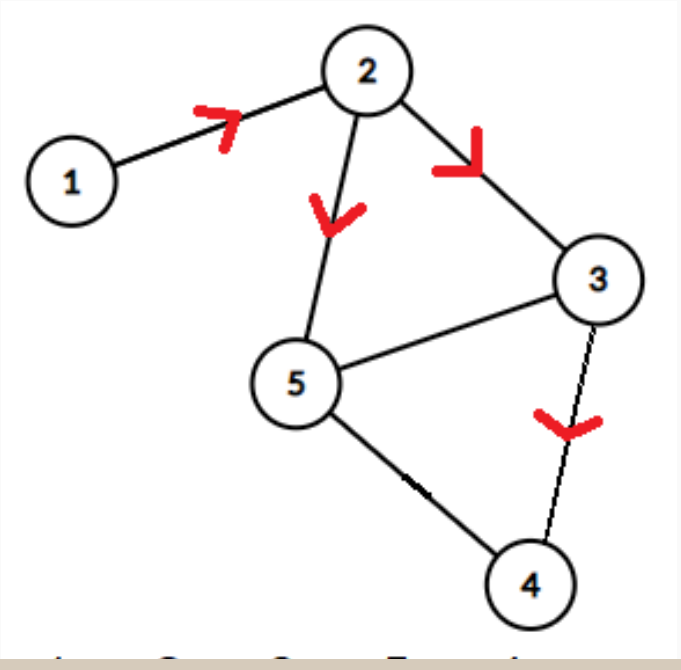
- Nodurile sunt parcurse în ordine în funcție de distanța față de nodul de început.
- Se utilizează o coadă.
- La început se introduce nodul de plecare în coadă.
- În fiecare etapă se scoate câte un nod din coadă, se parcurg toți vecinii acestuia și se introduc în coadă nodurile care nu au mai fost vizitate.
- Algoritmul se termină atunci când nu mai sunt elemente în coadă.



BFS - parcurgere în lățime

- Nodurile sunt parcurse în ordine în funcție de distanța față de nodul de început.
- Se utilizează o coadă.
- La început se introduce nodul de plecare în coadă.
- În fiecare etapă se scoate câte un nod din coadă, se parcurg toți vecinii acestuia și se introduc în coadă nodurile care nu au mai fost vizitate.
- Algoritmul se termină atunci când nu mai sunt elemente în coadă.





BFS

```
bool viz[NMAX];
vector<int> v[NMAX];
queue<int> Q;
Q.push(1);
while (!Q.empty()){
    int nod = Q.front();
    Q.pop();
    for (auto it : v[nod]){
        if (!viz[it]){
            viz[it] = 1;
            Q.push(it);
        }
    }
}
```

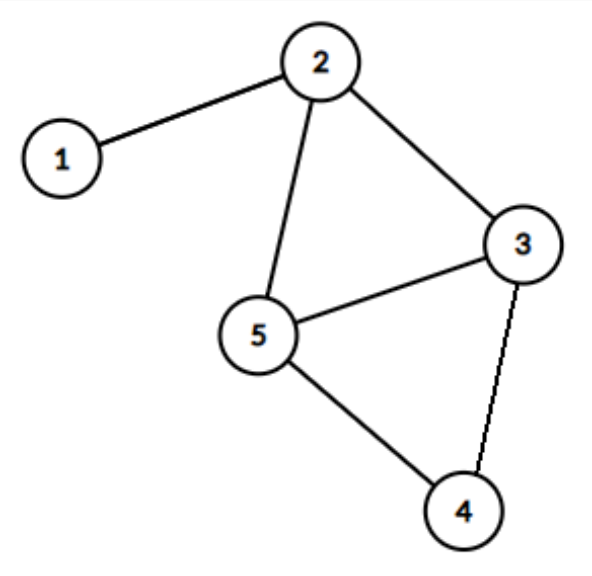
Complexitate timp: O(m)

DFS - parcurgere în adâncime

- În general se implementează recursiv.
- La început se apeleaza funcția DFS pentru nodul 1.
- În interiorul funcției se parcurg toți vecinii nodului curent și se apelează recursiv aceeași funcție pentru cei nevizitați.
- Algoritum se termină când nu mai există noduri nevizitate.

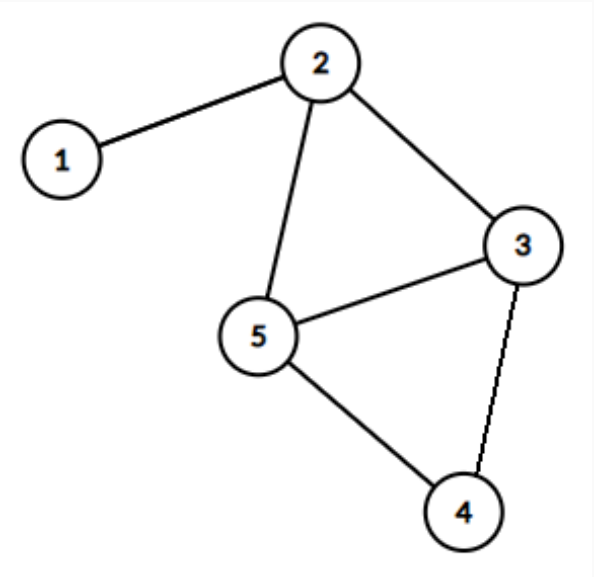
DFS - parcurgere în adâncime

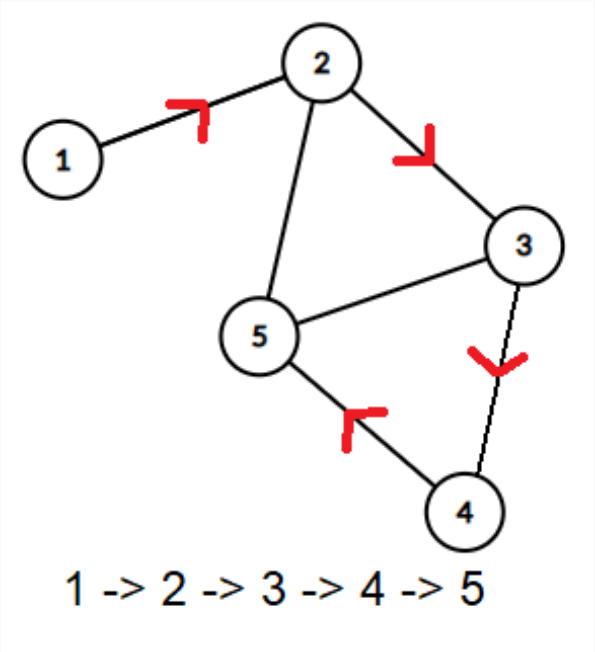
- În general se implementează recursiv.
- La început se apeleaza funcția DFS pentru nodul 1.
- În interiorul funcției se parcurg toți vecinii nodului curent și se apelează recursiv aceeași funcție pentru cei nevizitați.
- Algoritum se termină când nu mai există noduri nevizitate.



DFS - parcurgere în adâncime

- În general se implementează recursiv.
- La început se apeleaza funcția DFS pentru nodul 1.
- În interiorul funcției se parcurg toți vecinii nodului curent și se apelează recursiv aceeași funcție pentru cei nevizitați.
- Algoritum se termină când nu mai există noduri nevizitate.





DFS

```
bool viz[NMAX];
vector<int> v[NMAX];
void DFS(int nod){
    for (auto it : v[nod])
        if (!viz[it]){
            viz[it] = 1;
            DFS(it);
        }
}
```

Complexitate timp: O(m)

Conexitate

Un graf este **conex** dacă se poate ajunge de la orice nod al său în oricare alt nod.

Conexitate

Un graf este **conex** dacă se poate ajunge de la orice nod al său în oricare alt nod.

Ciclu

Un **ciclu** este un drum care trece prin muchii diferite și care începe și se termină în același nod.

Arbori

Definiție

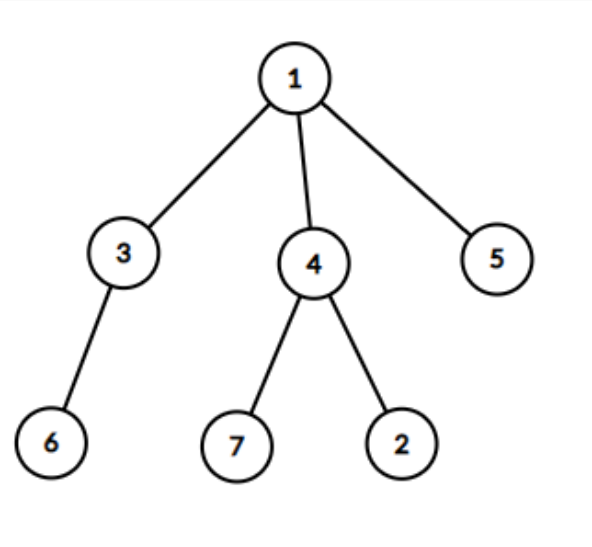
Un **arbore** este un graf conex aciclic.

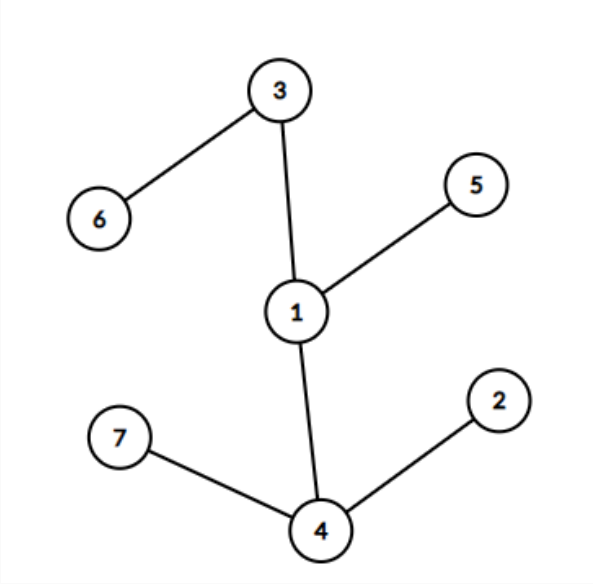
Arbori

Definiție

Un **arbore** este un graf conex aciclic.

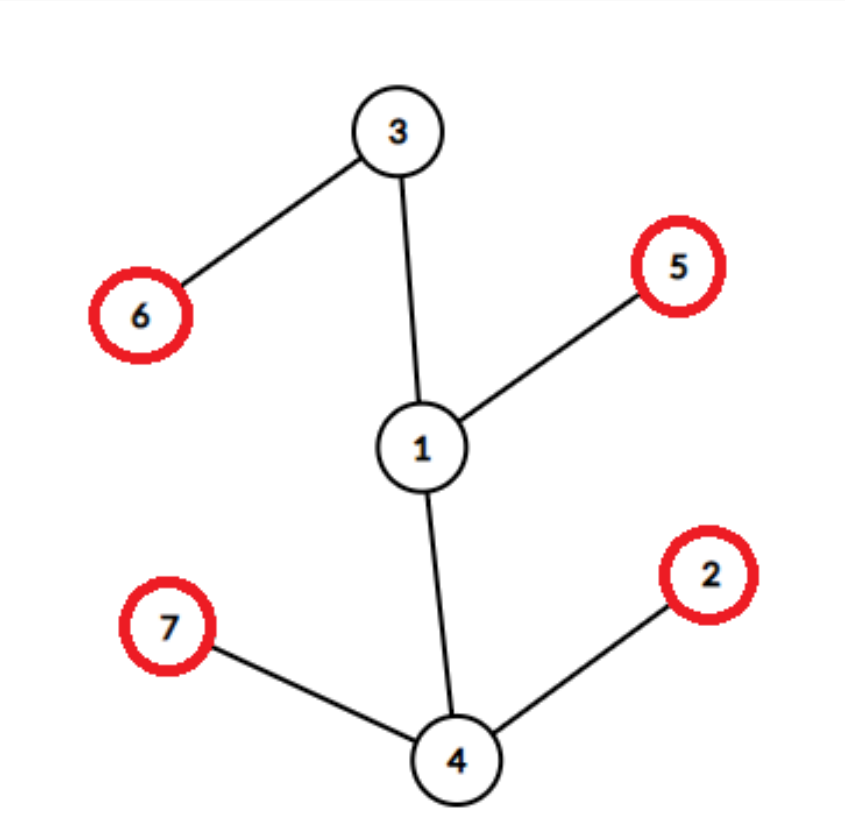
Acesta poate să aibă sau să nu aibă o radacină.





Frunze

Nodurile care au un singur vecin se numesc **frunze**.



Diametrul unui arbore

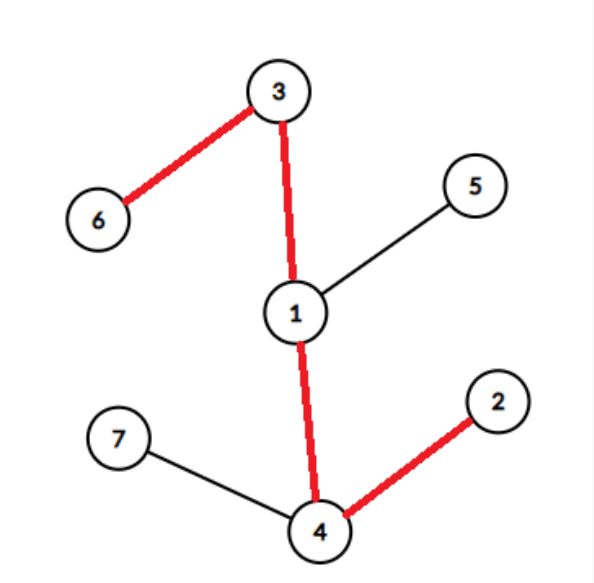
Diametrul unui arbore este cel mai lung drum care unește două frunze.

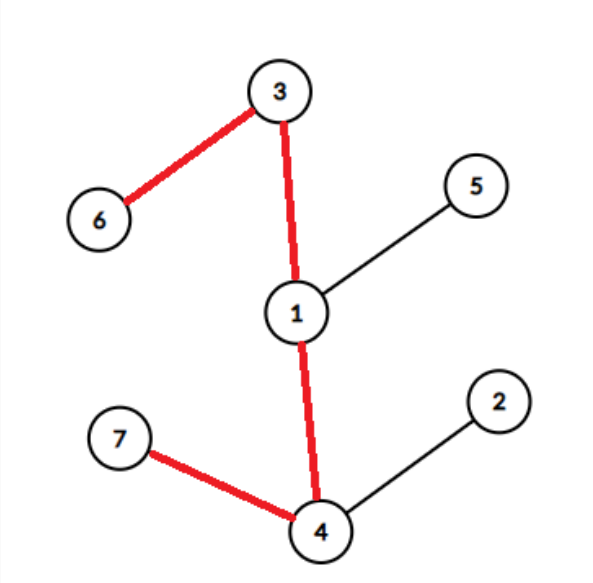
Diametrul unui arbore

Diametrul unui arbore este cel mai lung drum care unește două frunze.

- Pentru a găsi diametrul arborelui putem pornim un DFS din orice nod și să căutam cel mai depărtat nod de acesta. Nodul găsit este o frunză ce reprezintă unul din capetele diametrului.
- Apelam aceeași funcție DFS pentru nodul găsit pentru a determina celălalt capăt al diametrului.
- Complexitate: O(n)

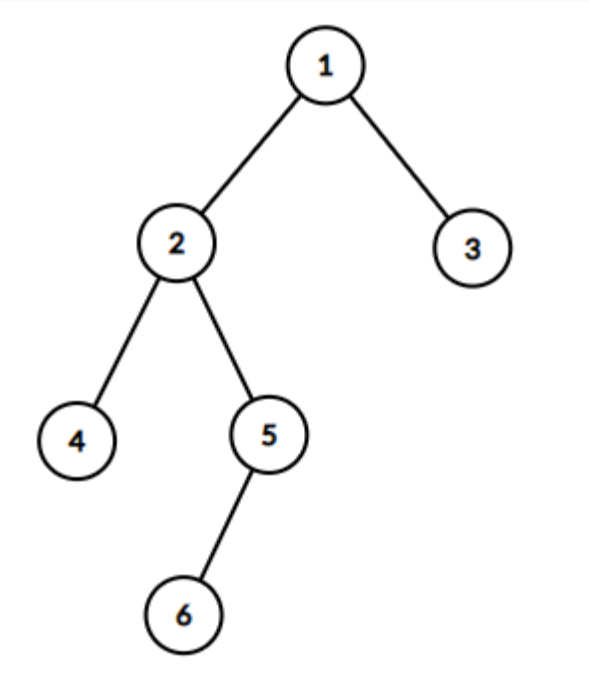
Diametrul unui arbore nu este unic.





Arbore binar

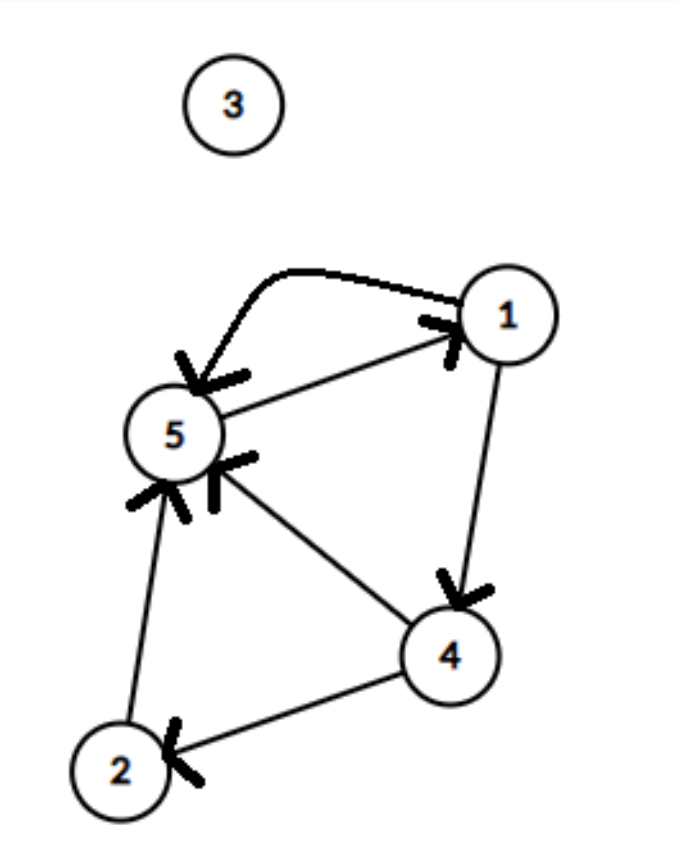
Un arbore binar este un arbore în care orice nod are maxim doi fii.



Grafuri orientate

Definiție

Grafurile orientate sunt asemănătoare grafurilor neorientate cu excepția că muchiile dintre două noduri nu pot fi parcurse decât într-un sens.



Sortare topologică

Descriere

O **sortare topologică** a vârfurilor unui graf orientat aciclic este o operație de ordonare liniară a vârfurilor, astfel încat, dacă există un arc (i, j), atunci i apare înaintea lui j în această ordonare.

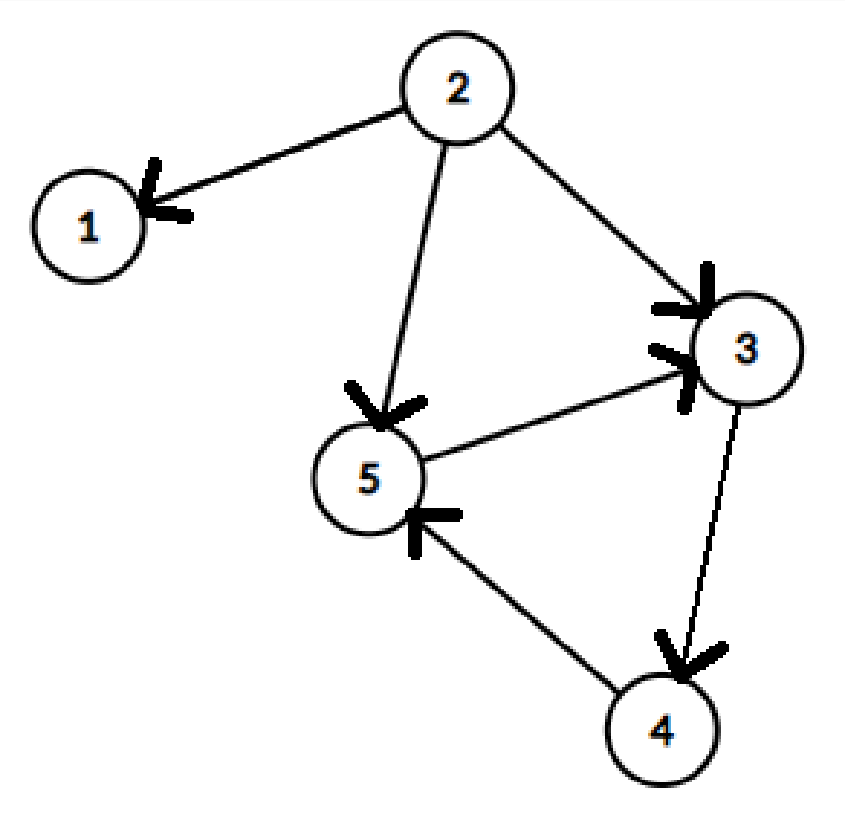
Soluție

- Folosim o listă în care introducem pe rând nodurle grafului.
- La început introducem în listă toate nodurile care au gradul interior zero.
- Atunci când este introdus un nod în listă, scoatem nodul respectiv din graf.
- După fiecare introducere a unui nod in listă, verificăm dacă prin ștergerea sa apar noduri noi cu gradul interior zero și le introducem în continuare în listă.
- Ordinea nodurilor din listă este ordinea din sortarea topologică a grafului.

Componente tare conexe

Definiție

O componentă **tare conexă** este o submulțime de noduri și muchii ale unui graf orientat(subgraf) în care se poate ajunge de la orice nod la orice nod.



CTC

```
int main(){
    for (int i=1;i<=m;i++){
        cin >> a >> b;
        v[a].push_back(b);
        v2[b].push_back(a);
    }
    for (int i=1;i<=n;i++) if (!uz[i]) dfs(i);
    memset(uz,0,sizeof(uz));
    for (int i = H.size() - 1; i >= 0; i--)
            if (!uz[H[i]]){
                ans.push_back(vector<int>());
                dfs2(H[i]);
            }
    }
}
```

CTC

```
void dfs(int nod){
    uz[nod] = 1;
    for (auto it : v[nod]){
        if (uz[it]) continue;
        dfs(it);
    }
    H.push_back(nod);
}

void dfs2(int nod){
    uz[nod] = 1;
    for (auto it : v2[nod]){
        if (uz[it]) continue;
        dfs2(it);
    }
    ans.back().push_back(nod);
}
```

Roy Floyd

Descriere

Gaseste drumul minim de la fiecare nod la fiecare, in timp $O(N^3)$

Roy-Floyd

```
// dmin[i][j] = matricea de adiacenta
for(int k = 1; k <= n; k++)
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
      if(dmin[i][j] > dmin[i][k] + dmin[k][j])
        dmin[i][j] = dmin[i][k] + dmin[k][j];
```

Dijkstra

Descriere

Gaseste drumul minim dintre un nod si toate celelalte noduri in complexitate $O(N^2)$ sau $O(M * log(N))$, daca este implementat cu heap.

Dijkstra

```
void dijkstra(int nod) {

  // initializari
  for(long long i=1;i<=n;i++) d[i] = 1e9;
  set <long long, cmp> s; // custom compare, dupa vectorul d
  d[nod] = 0; s.insert(nod);

  ...
}
```

Dijkstra

```
void dijkstra(int nod) {
  // initializari
  for(long long i=1;i<=n;i++) d[i] = 1e9;
  set <long long, cmp> s; // custom compare, dupa vectorul d
  d[nod] = 0; s.insert(nod);

  while(!s.empty()) {
    int p = *s.begin(); s.erase(p);

    for(auto f : v[p]) {
      if(d[p] + f.second < d[f.first]) {
        s.erase(f.first);          // il sterg ca sa nu
        d[f.first] = f.second + d[p]; // strice set-ul
        s.insert(f.first);  // dupa ce il updatez, il
            readaug
      }
    }
  }
}
```

Arbore partial de cost minim - APM

Descriere

APM - un arbore care include toate vârfurile și o submultime a muchiilor grafului initial, iar costul total al muchiilor alese este minim.

Algoritmi cunoscuti - Prim/Kruskal

Vom studia algoritmul lui Prim (algoritm de tip greedy):

Pornim dintr-un nod de start si punem toate muchiile incidente cu acesta intr-un min-heap.

Alegem muchia cu costul cel mai mic, care are un varf nevizitat - v.

Introducem in min-heap toate muchiile noului nod v.

Repetam algoritmul pana cand vizitam toate nodurile.

Algoritmul lui Prim

```
#define st first
#defint nd second
...
prioriy_queue <pair<int, pair<int, int>> > heap;
//                    -cost,   nod1, nod2

while(!heap.empty()) {
  edge = heap.top(); heap.pop();
  if(use[edge.nd.nd] == 1 && use[edge.nd.st] == 1) continue;

  ans += (-edge.st); // actualizez valoarea
  new_nod = (use[edge.nd.st]==0 ? edge.nd.st : edge.nd.nd);

  for(auto i : new_nod)
    heap.push({-i.st, {new_nod, i.nd}});
}
```

Probleme

- BFS
- DFS
- Diametrul unui arbore
- Componente tare conexe
- Dijkstra
- Roy-Floyd
- APM
- Sortare Topologica
- Rețele
- Graf
- Friend of Friend
- PolandBall and Forest
- Kefa and Park
- Andryusha and Colored Balloons
- Leha and...
- Codeforces 505D
- Codeforces 650C
- arg061_C