Notite examen - Algoritmi Fundamentali

Dinu Florin-Silviu
grupa 231

Contents

Notiuni fundamentale

Grafuri izomorfe

Daca 2 grafuri:

- Au acelasi numar de noduri
- Au acelasi numar de muchii
- Nodurile formeaza o secventa cu acelasi grad
- Daca un graf are un ciclu de lungime k, si celalalt graf are la fel

Grafuri bipartite

Daca un graf:

- Neorientat
- Putem imparti nodurile in 2 multimi $V = V_1 \cup V_2$ cu $V_1 \cap V_2 = \varnothing$
- Fiecare muchie are o extremitate intr-o parte si cealalta in cealalta parte, adica $|e \cap V_1| = |e \cap V_2| = 1$

Construire graf cu secventa gradelor (Havel-Hakimi)

Ordonam descrescator gradele si incepem sa trasam muchii catre urmatoarele, scazand din gradele lor pana cand ajungem sa avem doar 0. Daca avem $d_i < 0$ sau $d_i > n - 1$ sau $(\sum_{i=0}^{n} d_i)\%2 = 1$ atunci nu are solutie.

Parcurgeri

BFS

Se iau toti vecinii nevizitati, se pun in coada, se continua cu urmatorul din coada. Complexitate $O(V + E)$. Se foloseste pentru iesirea din labirint cu drum minim si calcul nivel.

```
void bfs(vector<vector<int>> &lista, vector<int> &pbfs, queue<int> &q)
    {
    int s = q.front();
    q.pop();

    int nivel = pbfs[s] + 1;

    for (auto x: lista[s]) {
        if (pbfs[x] == -1) {
            pbfs[x] = nivel;
            q.push(x);
        }
    }

    if (q.empty()) return;

    bfs(lista, pbfs, q);
}
```

- Muchiile de arbore sunt muchii din pădurea de adâncime G_π. Muchia (u, v) este o muchie de arbore dacă v a fost descoperit explorând muchia (u, v).

- Muchiile înapoi sunt acele muchii (u, v) care unesc un vârf u cu un strămoş v într-un arbore de adâncime. Buclele (muchii de la un vârf la el însuşi) care pot apărea într-un graf orientat sunt considerate muchii înapoi.
- Muchiile înainte sunt acele muchii (u, v) ce nu sunt muchii de arbore şi conectează un vârf u cu un descendent v într-un arbore de adâncime.
- Muchiile transversale sunt toate celelalte muchii. Ele pot uni vârfuri din acelaşi arbore de adâncime, cu condiţia ca unul să nu fie strămoşul celuilalt, sau pot uni vârfuri din arbori, de adâncime, diferiţi.

DFS

Complexitate $O(V + E)$.

```
void Graph::DFS(int v)
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
```

Sortare topologica

Se face pe DAG (directed acyclic graph)

Cu BFS

Incepe de la toate nodurile care au $indegree == 0$ (DAG garanteaza cel putin 1 astfel de nod). Dupa ce scoatem nodul din coada, il punem in vector.

```
vector<int> topo(int N, vector<int> adj[]) {
    queue<int> q;
    vector<int> indegree(N, 0);
    for(int i = 0;i<N;i++) {
        for(auto it: adj[i]) {
            indegree[it]++;
        }
    }

    for(int i = 0;i<N;i++) {
        if(indegree[i] == 0) {
            q.push(i);
        }
    }
    vector<int> topo;
    while(!q.empty()) {
        int node = q.front();
        q.pop();
```

```
        topo.push_back(node);
        for(auto it : adj[node]) {
            indegree[it]--;
            if(indegree[it] == 0) {
                q.push(it);
            }
        }
    }
    return topo;
}
```

Cu DFS

Diferentea fata de DFS e ca pun la sfarsit nodul pe stack. Diferentele fata de anterior sunt ca iau tot ce nu e vizitat si ca e mai eficient dpdv al memoriei, cat timp lantul cel mai lung nu da stackoverflow. Complexitate $O(V + E)$

```
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int>& Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack
    // which stores result
    Stack.push(v);
}
```

Muchii critice

Acele muchii care nu fac parte dintr-un ciclu. Facem un DFS si tinem 2 valori: low si disc. Pentru fiecare nod, mai intai punem disc ca fiind $disc_anterior + 1$, trecem la copil, pentru el memoram parintele, apoi facem DFS, apoi actualizam low ca fiind $min(low[copil], low[parinte])$. Daca este critica, atunci $low[copil] > disc[parinte]$.

```
void Graph::bridgeUtil(int u, bool visited[], int disc[],
                                    int low[], int parent[])
{
    // A static variable is used for simplicity, we can
    // avoid use of static variable by passing a pointer.
    static int time = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;
```

```
    // Go through all vertices adjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;  // v is current adjacent of u

        // If v is not visited yet, then recur for it
        if (!visited[v])
        {
            parent[v] = u;
            bridgeUtil(v, visited, disc, low, parent);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u]  = min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is  below u in DFS tree, then u-v
            // is a bridge
            if (low[v] > disc[u])
              cout << u <<" " << v << endl;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = min(low[u], disc[v]);
    }
}
```

Puncte critice

V este punct critic, daca exista 2 noduri x si y, cu $x, y \neq v$, pentru care v apartine oricarui $x, y-lant$

```
void APUtil(vector<int> adj[], int u, bool visited[],
            int disc[], int low[], int& time, int parent,
            bool isAP[])
{
    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    for (auto v : adj[u]) {
        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v]) {
            children++;
            APUtil(adj, v, visited, disc, low, time, u, isAP);

            // Check if the subtree rooted with v has
```

```
            // a connection to one of the ancestors of u
            low[u] = min(low[u], low[v]);

            // If u is not root and low value of one of
            // its child is more than discovery value of u.
            if (parent != -1 && low[v] >= disc[u])
                isAP[u] = true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent)
            low[u] = min(low[u], disc[v]);
    }

    // If u is root of DFS tree and has two or more children.
    if (parent == -1 && children > 1)
        isAP[u] = true;
}
```

Conexitate in graf orientat

Exista 2 tipuri de conexitati intr-un graf orientat:

- Slab conex: Drum de intre $\forall$ 2 noduri daca consideram graful neorientat
- Tare conex: Drum intre $\forall$ 2 noduri

Kosaraju

Fac un DFS in care adaug pe stack nodurile. Transpun graful. Popuiesc stackul, fac DFS pe acel nod in care marchez elementele ca vizitate, apoi popuiesc urmatorul element si fac DFS daca e nevizitat. Complexitate $O(V + E)$.

```
#include <bits/stdc++.h>

#define MAX_N 20001
#define ll long long int
using namespace std;
int n, m;

struct Node {
  vector < int > adj;
  vector < int > rev_adj;
};

Node g[MAX_N];

stack < int > S;
bool visited[MAX_N];

int component[MAX_N];
vector < int > components[MAX_N];
int numComponents;

void dfs_1(int x) {
  visited[x] = true;
```

```
  for (int i = 0; i < g[x].adj.size(); i++) {
    if (!visited[g[x].adj[i]]) dfs_1(g[x].adj[i]);
  }
  S.push(x);
}

void dfs_2(int x) {
  printf("%d ", x);
  component[x] = numComponents;
  components[numComponents].push_back(x);
  visited[x] = true;
  for (int i = 0; i < g[x].rev_adj.size(); i++) {
    if (!visited[g[x].rev_adj[i]]) dfs_2(g[x].rev_adj[i]);
  }
}

void Kosaraju() {
  for (int i = 0; i < n; i++)
    if (!visited[i]) dfs_1(i);

  for (int i = 0; i < n; i++)
    visited[i] = false;

  while (!S.empty()) {
    int v = S.top();
    S.pop();
    if (!visited[v]) {
      printf("Component %d: ", numComponents);
      dfs_2(v);
      numComponents++;
      printf("\n");
    }
  }
}

int main() {

  cin >> n >> m;
  int a, b;
  while (m--) {
    cin >> a >> b;
    g[a].adj.push_back(b);
    g[b].rev_adj.push_back(a);
  }

  Kosaraju();
  printf("Total number of components: %d\n", numComponents);

  return 0;
}
```

Lema

Dacă două vârfuri se află în aceeaşi componentă tare conexă, atunci nici un drum între ele nu părăseşte, vreodată, componentă tare conexă.

Demonstrație: Fie u și v două noduri din componenta tare conexă. Presupunem ca exista w în afara componentei și există drum u - > v prin w. Atunci avem drum de la u la w dar avem și drumul w->v->u deci și drum de la w la u deci w este în componenta tare conexă.

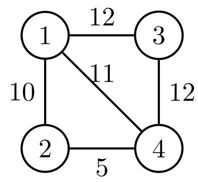
APM - arbori partiali de cost minim

Kruskal

$O(E * logE + E * logV)$ Algoritmul are 3 pasi:

- Sorteaza toate muchiile in oridinea crescatoare a greutatii
- Alege muchia cea mai mica, daca formeaza un ciclu, treci la urmatoarea, daca nu, adaug-o.
- Repeta pasul anterior pana ai $V - 1$ muchii.

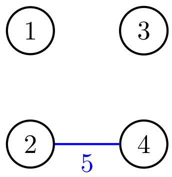
Exemplu Fie graful



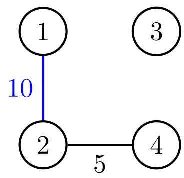
Pasul 1. Sortam ASC muchiile: $\{(2, 4, 5), (1, 2, 10), (1, 4, 11), (1, 3, 12), (3, 4, 12)\}$. Am folosit notatia $(nod1, nod2, greutate)$.

Pasii 2 & 3.

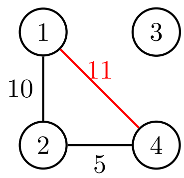
1. Incepem recursia alegand cea mai mica muchie $(2, 4, 5)$.



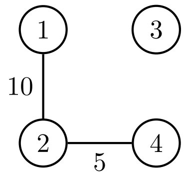
2. Continuam cu $(1, 2, 10)$.



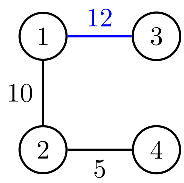
3. Continuam cu $(1, 4, 11)$.



Observam ca face ciclu, deci o ignoram



4. Continuam cu $(1, 3, 12)$.



Observam ca avem $V - 1$ muchii, deci ramanem cu acest graf

K-clustering (aplicatie) Imparte in k clustere (are $n - k$ pasi)

Pseudocod:	**Pseudocod – modelare cu graf complet G:**
	V= {o_1,..., o_n}, w($o_i o_j$) = d(o_i,o_j)
Inițial fiecare obiect (cuvânt) formează o clasă	Inițial fiecare vârf formează o componentă conexă (clasă): **T'=(V, ∅)**
pentru i = 1, n-k	pentru i = 1, n-k
• alege două obiecte o_r, o_t din clase diferite cu d(o_r, o_t) minimă	• alege o muchie e_i=uv de **cost minim din G astfel încât u și v sunt în componente conexe diferite** ale lui T'
• reunește clasa lui o_r și clasa lui o_t	• reunește componenta lui u și componenta lui v: **E(T')=E(T') ∪ {uv}**
returneză cele k clase obținute	returnează cele k mulțimi formate cu vârfurile celor k componente conexe ale lui T'

Prim

Complexitate $O(V^2)$ sau $O(V * logV + E * logV)$ daca folosim minheap. Algoritmul are 6 pasi:

- Alegem un nod n aleatoriu si il scoatem din lista nodurilor de ales
- Adaugam in structura de date toti vecinii sai impreuna cu greutatea muchiilor
- Actualizam eticheta pentru fiecare varf cu distanta minima fata de cele incluse in graf
- Adaugam nodul cu distanta minima cea mai mica, il scaotem din lista nodurilor de ales si ii facem parintele nodul de care se leaga
- Adaugam in structura de date toti vecinii sai impreuna cu greutatea muchiilor
- Repetam al 4-lea pas.

Prim

- s– vârful de start
- `inițializează Q cu V`
- `pentru fiecare u∈V executa`

```
      d[u] = ∞; tata[u]=0
 d[s] = 0
```

- `cat timp Q ≠ ∅ executa`

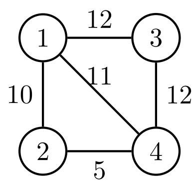
    extrage un vârf u∈Q cu eticheta d[u] minimă

```
    pentru fiecare uv∈E executa
            daca v∈Q si w(u,v)<d[v] atunci
                d[v] = w(u,v)
                tata[v] = u
```

- scrie (u, tata[u]), pentru u≠ s

↑ Prim prin vector vizitat ↑

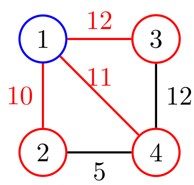
Exemplu Fie graful



Pasul 1. Plecam de la nodul 1 si facem urmatoarea structura

$$[(d/tata)] = \{(0,0),(\infty,0),(\infty,0),(\infty,0)\}$$

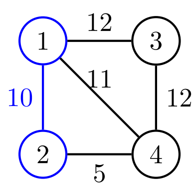
Pasul 2.



Pasul 3.

$$[(d/tata)] = \{(0,0),(10,0),(12,0),(11,0)\}$$

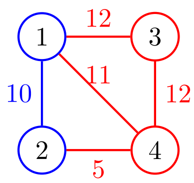
Pasul 4.



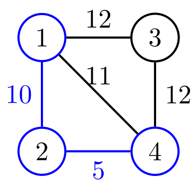
Pasul 5.

$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,0)\}$$

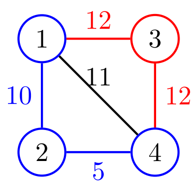
Pasul 6. Repetam



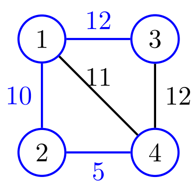
$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,0)\}$$



$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,2)\}$$

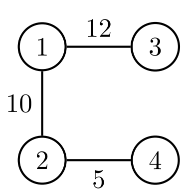


$$[(d/tata)] = \{(0,0),(10,1),(12,0),(5,2)\}$$



$$[(d/tata)] = \{(0,0),(10,1),(12,1),(5,2)\}$$

APM final



Prim(G, w, s)

```
pentru fiecare u∈V executa
     d[u] = ∞; tata[u]=0
d[s] = 0
inițializează Q cu V
cat timp Q ≠ ∅ executa
      u=extrage vârf cu eticheta d minimă din Q
      pentru fiecare v adiacent cu u executa
             daca v∈Q si w(u,v)<d[v] atunci
                 d[v] = w(u,v)
                 tata[v] = u
                 //actualizeaza Q - pentru Q heap
scrie (u, tata[u]), pentru u≠ s
```

↑ **Prim prin minheap** ↑