Laborator 12 - Multimi disjuncte - Union-Find

Responsabili:

- Andrei Pârvu (2014)
- Mugurel Ionut Andreica (2013)
- Eliana Tîrșa (2013)

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil să:

- înțeleagă structura de date "disjoint sets";
- înțeleagă operațiile Union și Find ce pot fi aplicate asupra structurii
- folosească mulțimile disjuncte pentru a rezolva o serie de aplicații

Noțiuni teoretice

Structura de date "mulțimi disjuncte" consideră că, inițial, există `N` elemente distincte (numerotate de la `1` la `N`), fiecare făcând parte dintr-o mulțime separată. Structura suportă două operații:

- `Union(x,y)`: unește mulțimea din care face parte elementul `x` cu mulțimea din care face parte elementul `y`. În urma unirii, elementele din cele două mulțimi vor face parte din aceeași mulțime.
- `Find(x)`: întoarce un identificator al mulțimii din care face parte elementul `x`. Operația `Find` are urmatoarele proprietăți:
  - dacă `x` și `y` sunt elemente din aceeași mulțime, atunci `Find(x)=Find(y)`.
  - dacă `x` și `y` sunt elemente din mulțimi diferite, atunci `Find(x)!=Find(y)`.

Una dintre implementările posibile pentru structura de mulțimi disjuncte este cea de a reprezenta fiecare mulțime sub forma unui arbore (un arbore general, nu binar). Pentru fiecare element `x` se va pastra o valoare `parent[x]` (inițial, `parent[x]` este `0` pentru toate elementele `x`).

Operația `Find(x)` pornește de la elementul `x` și urcă în sus în arbore, folosind legăturile `element - parent[element]`, până când ajunge la un element `rx` pentru care `parent[rx]=0`. `Find(x)` va returna acest element `rx`. `rx` este, practic, rădăcina arborelui ce reprezintă mulțimea respectivă. Să observăm că dacă două elemente `x` și `y` sunt în aceeași mulțime, atunci atât `Find(x)`, cât și `Find(y)`, vor returna aceeași valoare (întrucât ambele elemente fac parte din același arbore, se va ajunge la același element radacină).

Operația `Union(x,y)` începe prin a calcula elementele `rx=Find(x)` și `ry=Find(y)`. Dacă `rx=ry`, atunci elementele `x` și `y` sunt deja în aceeași mulțime și nu mai este necesar să efectuăm alte operații. Dacă `rx!=ry` atunci este suficient să setăm părintele unuia dintre reprezentanți ca fiind celălalt reprezentant: de ex., setăm `parent[rx]=ry`.

Este evident că eficiența acestei implementări a structurii de date "mulțimi disjuncte" depinde de înalțimea arborilor formați. Cu cât un arbore are o adâncime mai mare, cu atât operația `Find` va avea un timp de execuție mai mare. Intrucât operația `Union` utilizează intern operația `Find`, înalțimea arborilor afectează și eficiența operației `Union`.

Varianta prezentată mai sus reprezintă versiunea "de baza" a structurii de date "mulțimi disjuncte" și există scenarii de utilizare în care această implementare are o eficiență scăzută (se formează arbori cu înalțimi foarte mari). De exemplu, dacă efectuăm, în ordine, operațiile `Union(1,2)`, `Union(1,3)`, …, `Union(1,N)`, se obține un "arbore-linie": `parent[1]=2`,

- Reguli generale și de notare
- Catalog
- Concursuri
- Calendar

Laboratoare

- Laborator 1 - Introducere in C++
- Laborator 2 - Noțiuni de C++
- Laborator 3 - Stive
- Laborator 4 - Cozi
- Laborator 5 - Liste generice
- Laborator 6 - HashTable
- Laborator 7 - Grafuri
- Laborator 8 - Arbori Binari
- Laborator 9 - Arbori Binari de Căutare
- Laborator 10 - Heap-uri
- Laborator 11 - Treap-uri
- Laborator 12 - Mulțimi Disjuncte

Teme

- Tema 1
- Tema 2
- Tema 3
- Tema 4

Resurse

- Debugging
- Data Structure Visualization

Table of Contents

`parent[2]=3, …, parent[N-1]=N`.

Pentru a evita astfel de situații, în practica, există două tipuri de euristici ce pot fi utilizate (independent sau împreună).

Euristica "Union-by-Size"

Pentru fiecare element `x`, se mai pastrează o valoare `size[x]`, ce reprezintă numarul de noduri din subarborele a cărui rădăcină este nodul `x`. Aceasta valoare va fi păstrată actualizată doar pentru acele elemente `x` care sunt rădăcini de arbore (adică au `parent[x]=0`). Inițial, vom avea `size[x]=1` pentru fiecare element `x`.

In cazul operației `Union`, după ce determinăm reprezentanții `rx` și `ry` (și aceștia sunt diferiți), avem de ales între a seta `parent[rx]=ry` și `parent[ry]=rx`. Vom alege noua rădăcină a arborelui mulțimii "unite", ca fiind acel nod care are valoarea `size` mai mare. De exemplu, dacă `size[rx]>size[ry]` atunci vom seta `parent[ry]=rx` și actualizăm `size[rx]=size[rx]+size[ry]` (deoarece doar `rx` a mai rămas rădăcina de arbore, dintre `rx` și `ry`).

In felul acesta, se garantează că înălțimea oricărui arbore cu `M` elemente în el este de ordinul `log(M)` (logaritm în baza `2` din `M`).

Euristica "Compresia drumului"

Să considerăm o operație `Find(x)`. Se pornește de la elementul `x` și se parcurg elementele `x, parent[x], parent[parent[x]], …, rx` (`rx` este rădăcina arborelui din care face parte nodul `x`). Să presupunem că elementele parcurse sunt `a1, a2, …, ak` (unde `a1=x` și `ak=rx`). După determinarea lui `rx` putem seta `parent[a1]=rx, parent[a2]=rx, …, parent[ak-1]=rx`. Mai exact, putem lega toate elementele prin care am trecut direct de rădăcina arborelui. In felul acesta, dacă apelăm `Find(y)` în viitor, unde `y` este un element din aceeași mulțime ca și `x`, care se află într-unul din subarborii elementelor `a1, …, ak-1`, drumul de la `y` la rădăcină va fi mai scurt (și, deci, operația `Find(y)` va fi mai rapidă).

Aplicatii

Algoritmul lui Kruskal

Considerăm un graf cu `N` noduri și `M` muchii. Fiecare muchie `(i,j)` (între nodurile `i` și `j`) are un cost `c(i,j)`. Se dorește determinarea unui `arbore parțial de cost minim`. Un arbore parțial de cost minim constă dintr-o submulțime de `N-1` muchii care leagă toate cele `N` noduri ale grafului și al caror cost total este minim.

Algoritmul lui Kruskal funcționează în felul următor. Se sortează crescător după cost cele `M` muchii ale grafului (în cazul în care există mai multe muchii de cost egal, acestea pot fi considerate în orice ordine). Apoi se inițializează o structură de tip "mulțimi disjuncte" cu `N` elemente. In continuare se parcurg cele `M` muchii ale grafului în ordinea crescătoare a costului.

Să presupunem că am ajuns la muchia `(i,j)`. Dacă `Find(i)!=Find(j)` atunci vom adauga muchia `(i,j)` la arborele parțial de cost minim și vom apela `Union(i,j)`. Dacă `Find(i)=Find(j)` atunci mergem mai departe (înseamnă că nodurile `i` și `j` sunt deja legate între ele prin niște muchii ale arborelui parțial de cost minim și putem ignora muchia `(i,j)`).

La final, dacă graful este conex, muchiile selectate formează un arbore parțial de cost minim. Dacă graful nu este conex, muchiile selectate de algoritm formează câte un arbore parțial de cost minim în fiecare componentă conexă a grafului.

Determinarea componentelor conexe ale unui graf

Se consideră un graf neorientat cu `N` noduri și `M` muchii. Dorim să determinăm componentele conexe ale acestui graf.

Se inițializează o structură de date de tip "mulțimi disjuncte" cu `N` elemente. Vom considera muchiile grafului în orice ordine. Pentru orice muchie `(i,j)` vom apela `Union(i,j)`. La final, dacă avem `Find(i)=Find(j)` pentru două noduri `i` și `j`, atunci aceste două noduri fac parte din aceeași componentă conexă a grafului.

Exerciții

Porniți exercițiile de la header-ul următor

```
#ifndef __DSUF_H
#define __DSUF_H

class DisjointSetsUnionFind {
  public:
    int N;

    DisjointSetsUnionFind(int N) {
      this->N = N;
    }

    virtual void Union(int x, int y) = 0;
    virtual int Find(int x) = 0;
};

#endif
```

1. [1p] Definiți o clasă ce extinde clasa `DisjointSetsUnionFind` și care implementează funcțiile virtuale `Union` și `Find`:

- [0.5p] implementați euristica `union-by-size` în funcția `Union`
- [0.5p] implementați euristica `compresia drumului` în funcția `Find`

Puteți porni de la codul de mai jos, unde operațiile `Union` și `Find` sunt implementate în varianta de bază (fără niciuna din cele două euristici):

```
#ifndef __BASIC_DSUF_H
#define __BASIC_DSUF_H

#include <stdlib.h>

class BasicDisjointSetsUnionFind: public DisjointSetsUnionFind {
  public:
    int *parent;

    BasicDisjointSetsUnionFind(int N): DisjointSetsUnionFind(N)
      parent = new int[N + 1];
      for (int i = 1; i <= N; i++) {
        parent[i] = 0;
      }
    }

    void Union(int x, int y) {
      if (x <= 0 || x > N || y <= 0 || y > N)
        return;
      int rx = Find(x), ry = Find(y);
      if (rx != ry) {
        parent[rx] = ry;
      }
    }

    int Find(int x) {
      if (x <= 0 || x > N)
        return 0;
      while (parent[x] > 0)
        x = parent[x];
      return x;
    }

    ~BasicDisjointSetsUnionFind() {
```

```
        delete[] parent;
    }
};

#endif
```

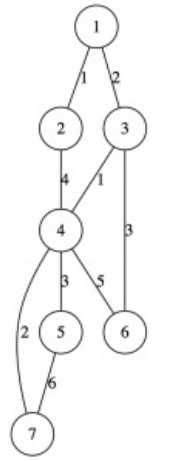
2. [4p] Implementaţi algoritmul lui Kruskal utilizând clasa pentru mulțimi disjuncte definită la punctul anterior. Din fișierul de intrare `kruskal.in` se citesc următoarele date, în ordine:

- `N` = numarul de noduri ale grafului
- `M` = numarul de muchii ale grafului
- cele `M` muchii, sub forma `a b c`, având semnificația că există muchie între nodurile `a` și `b`, având costul `c`

Afisați pe ecran costul arborelui parțial de cost minim și muchiile ce il compun.

kruskal.in

```
7 9
1 2 1
1 3 2
2 4 4
3 4 1
4 5 3
5 7 6
4 7 2
4 6 5
3 6 3
```



3. [4p] Determinați componentele conexe ale unui graf utilizând clasa pentru mulțimi disjuncte definită la punctul `1`. Din fișierul de intrare `componente.in` se citesc următoarele date, în ordine:

- `N` = numarul de noduri ale grafului
- `M` = numarul de muchii ale grafului
- cele `M` muchii, sub forma `a b`, având semnificația că există muchie între nodurile `a` și `b`

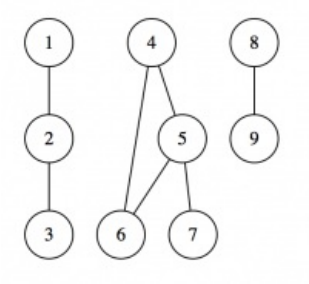
Afisați pe ecran numarul `C` de componente conexe. Pe următoarele `C` linii veți afisa nodurile grafului ce fac parte din fiecare componentă conexă (câte o componentă conexă pe fiecare linie).

`Hint`: După parcurgerea tuturor muchiilor și efectuarea operațiilor `Union` corespunzătoare fiecarei muchii a grafului, grupați nodurile grafului în funcție de rezultatul funcției `Find`. Utilizați un `Hashtable` pentru a realiza această grupare. Considerați că funcție de hash chiar funcția `Find`. Toate nodurile introduse în `Hashtable` care au aceeași valoare a funcției `Find` sunt în aceeași componentă conexă.

componente.in

```
9 7
```

```
1 2
2 3
4 5
4 6
5 6
5 7
8 9
```



Resurse

[1] Disjoint-set data structure

[2] Algoritmul lui Kruskal