Arbori de acoperire minimă

grupa CR 1.2B, an I, al 2-lea semestru, Calculatoare Română

Niculescu Marius Andrei

May 21, 2020

Contents

1 Enunţul problemei

Se consideră un graf neorientat conex G. Se numeşte arbore de acoperire a lui G un arbore liber T astfel încât:

i) T include fiecare vârf a lui G;

ii) T este subgraf a lui G, adică orice muchie a lui T este şi muchie a lui G.

Dacă graful G are costuri pozitive asociate muchiilor, se poate asocia un cost unui arbore de acoperire, egal cu suma costurilor muchiilor sale. Să se implementeze doi algoritmi diferiţi care determină arborele de acoperire minimă al unui graf neorientat conex dat (sugestii: doi dintre algoritmii Boruvka, Prim şi Kruskal).

2 Algoritmi

ARBORE_MINIM_DE_ACOPERIRE_PRIM($numar\_noduri$,$graf$)
1. **pentru** $i = 0$,$numar\_noduri - 1$ **execută**
2. $valori\_noduri[i] = 10000$
3. $noduri\_vizitate[i] = 0$
4. $valori\_noduri[0] = 0$
5. $nod\_parinte[0] = -1$
6. **pentru** $i = 0$,$numar\_noduri - 1$ **execută**
7. $u = determinare\_valoare\_minima(numar\_noduri,valori\_noduri,noduri\_vizitate)$
8. $noduri\_vizitate[u] = 1$
9. **pentru** $v = 0$,$numar\_noduri - 1$ **execută**
10. **dacă** $graf[u][v] != 0$ and $noduri\_vizitate[v] == 0$ and $graf[u][v] < valori\_noduri[v]$
11. $nod\_parinte[v] = u$
12. $valori\_noduri[v] = graf[u][v]$
13. $afisare\_arbore\_minim(numar\_noduri,nod\_parinte,graf)$

Complexitatea de timp şi de memorie a acestui algoritm este O($numar\_noduri^2$), deoarece se utilizează o matrice de adiacenţă pentru a reprezenta graful neorientat conex, cu costuri pozitive asociate muchiilor.

În acest algoritm am folosit 3 vectori:

- nod_părinte[număr_noduri] este utilizat pentru a stoca "părintele" fiecărui nod vizitat ce urmează a fi folosit în afişarea laturilor care formează arborele.
- valori_noduri[număr_noduri] este folosit pentru a asocia nodurilor adiacente nodului curent valorile muchiilor de care sunt legate atâta timp cât aceste noduri nu au fost încă vizitate. Dacă în urma repetării acestui pas se găseşte o muchie cu o valoare mai mică decât cea asociată deja nodului corespunzător, această nouă valoare va fi asociată nodului dacă acesta nu a fost deja vizitat.

- noduri_vizitate[număr_noduri] ţine cont de nodurile care au fost vizitate sau nu. Astfel, 0 va însemna nod nevizitat, iar 1 nod vizitat.

La începutul algoritmului iniţializăm valorile din noduri cu numere cât mai mari, deoarece mai târziu va trebui să extragem minimul din acest vector. De asemenea, marcăm faptul că nici un nod nu a fost vizitat la început. Apoi, dând valoarea 0 primului nod asigurăm faptul că acesta va fi ales primul din vectorul valori_noduri[număr_noduri] cu ajutorul funcţiei determinare_valoare_minimă.

Funcţia determinare_valoare_minimă găseşte nodul cu cea mai mică valoare asociată şi care nu a fost încă vizitat. Odată ce se găseşte acest nod, acesta va fi marcat ca şi vizitat. În următorul for (linia 9) se actualizează mereu valorile şi părinţii nodurilor adiacente nodului selectat prin intermediul variabilei u de mai sus. De asemenea, se iau în considerare doar acele noduri care sunt nevizitate.

- graf[u][v] va avea valoare nenulă numai în cazul în care există o muchie între u si v (nodurile sunt adiacente).
- noduri_vizitate[v] va avea valoarea 0 numai dacă nodul v este nevizitat.
- valoarea corespunzătoare nodului v va fi actualizată numai dacă valoarea muchiei este mai mică decât valoarea deja atribuită nodului.

La final, funcţia afişare_arbore_minim va afişa muchiile arborelui minim de acoperire găsit şi valorile acestora cu ajutorul vectorului nod_părinte[număr_noduri] în care se află nodul "părinte" al fiecărui nod vizitat pentru a putea determina muchiile din care este compus arborele.

```
ARBORE_MINIM_DE_ACOPERIRE_KRUSKAL(numar_noduri,graf)
1. pentru i = 0,numar_noduri − 1 execută
2.         nod_parinte[i] = −1
3.         pentru j = 0,numar_noduri − 1 execută
4.                 dacă graf[i][j] == 0 execută
5.                         graf[i][j] = 10000
6. cât timp numar_muchii_arbore < numar_noduri
7.         minim = 10000
8.         pentru i = 0,numar_noduri − 1 execută
9.                 pentru j = 0,numar_noduri − 1 execută
10.                        dacă graf[i][j] < minim execută
11.                                minim = graf[i][j]
12.                                a = u = i
13.                                b = v = j
14.        u = gaseste_parinte(numar_noduri,u,nod_parinte)
15.        v = gaseste_parinte(numar_noduri,v,nod_parinte)
16.        dacă reuniune_noduri(numar_noduri,u,v,nod_parinte) ! = 0 execută
17.                afişează numar_muchii_arbore. muchia (a,b) = minim
18                 numar_muchii_arbore + +
19.                cost_arbore_minim+ = minin
20.        graf[a][b] = graf[b][a] = 10000
21. afişează cost_arbore_minim
```

Complexitatea de timp şi de memorie a acestui algoritm este O($numar\_noduri^2$), deoarece se utilizează o matrice de adiacenţă pentru a reprezenta graful neorientat conex, cu costuri pozitive asociate muchiilor.

În acest algoritm am folosit vectorul nod_părinte[număr_noduri] pentru a ţine cont de părintele fiecărui nod din arborele minim de acoperire format. La început, vectorul are pe toate poziţiile valoarea -1, deoarece toate nodurile sunt "libere" (nu au nici un părinte). Parcurgem apoi matricea de adiacenţă pentru a înlocui 0-urile (ceea ce înseamnă că nu există o legatură între nodurile respective) cu o valoare cât mai mare, cum ar fi 10000, pentru a elimina muchia din viitoarele căutări ale valorii minime din matrice.

Arborele minim de acoperire va avea mereu $numar\_noduri - 1$ muchii. În variabila $numar\_muchii\_arbore$ vom ţine cont de numărul de muchii din arborele care este format. Căutăm apoi valoarea minimă din matrice (muchia cu cea mai mică valoare asociată) şi o salvăm în variabila $minim$. Copiem numărul fiecărui nod care formează această muchie în câte două variabile diferite care vor fi utilizate mai târziu. Cu ajutorul variabilelor u şi v vom verifica prin funcţia găseşte_părinte dacă aceste noduri au acelaşi părinte. În caz afirmativ, muchia va fi înlăturată, deoarece va forma un ciclu. În caz contrar, muchia va fi ataşată la arbore, iar numărul de muchii din arbore va creşte cu o unitate şi la costul arborelui minim de acoperire se va adăuga valoarea acestei muchii. De asemenea, muchia va fi afişată cu ajutorul variabilelor a şi b în care au fost salvate nodurile care delimitează această muchie şi cu ajutorul variabilei $minim$ în care se află valoarea asociată muchiei. Chiar dacă muchia a fost ataşată sau nu la arbore, valoarea acesteia va fi actualizată la 10000 pentru a o înlătura din viitoarele căutări ale valorii minime din matricea de adiacenţă. Aceşti paşi (liniile 7-20) se vor repeta atâta timp cât arborele minim de acoperire format are mai puţin de număr_noduri - 1 muchii. În final se va afişa "costul" arborelui.

3 Date experimentale

INIŢIALIZARE_MATRICE_ADIACENTĂ($numar\_noduri$,$graf$)
1. **pentru** $i = 0$,$numar\_noduri - 1$ **execută**
2. **pentru** $j = 0$,$numar\_noduri - 1$ **execută**
3. $graf[i][j] = -1$
4. **pentru** $i = 0$,$numar\_noduri - 1$ **execută**
5. **pentru** $j = 0$,$numar\_noduri - 1$ **execută**
6. **dacă** $i == j$ **execută**
7. $graf[i][j] = 0$
8. **altfel dacă** $graf[i][j] == -1$ and $graf[j][i] == -1$ **execută**
9. $graf[i][j] = rand()\%10000$
10. **altfel**
11. $graf[i][j] = graf[j][i]$
12. **afişează** $graf[i][j]$

Prima dată am iniţializat elementele matricei cu valoarea -1. Deoarece nu există o muchie între un nod şi el însuşi am pus valoarea 0 atunci când $i = j$.

După aceea dacă atât pe poziţia $graf[i][j]$ cât şi $graf[j][i]$ se află valoarea -1, atunci se generează aleatoriu o valoare. Dacă pe una dintre cele două poziţii se află o valoarea, atunci pe cealaltă poziţie unde se află -1 se va copia acea valoare, deoarece graful este neorientat şi $graf[i][j] = graf[j][i]$ (nu contează sensul de parcurgere a muchiei, deoarece graful este neorientat).

În concluzie, în funcţie de numărul de noduri din graf, acest algortim generează automat datele de intrare necesare rezolvării problemei. Datele generate de acest algoritm sunt semnificative deoarece reprezintă costurile pozitive asociate muchiilor grafului stocate într-o matrice de adiacenţă.

4 Proiectarea aplicaţiei experimentale

4.1 Structura de nivel înalt a aplicaţiei

În C: main.c —> graf.c —> graf.h
În Python: main.py –> header.py şi generator.py

4.2 Descrierea mulţimii datelor de intrare

Se citeşte dintr-un fişier numărul de noduri din graful neorientat conex, iar apoi pe baza algoritmului de mai sus se generează aleatoriu preţurile pozitive asociate muchiilor grafului, stocate într-o matrice.

4.3 Descrierea ieşirilor / rezultatelor

În cazul ambilor algoritmi, datele de ieşire constau în afişarea pe câte o linie a muchiilor (precizând nodurile care le delimitează), care alcătuiesc arborele minim de acoperire al grafului dat şi a valorilor asociate acestor muchii. La final este afişat şi "costul" total, al arborelui format, prin însumarea valorilor asociate muchiilor găsite. De asemenea, este calculat şi timpul de execuţie al funcţiei principale (arbore_minim_de_acoperire_prim şi kruskal). Ambii algoritmi au fost testaţi atât în C cât şi în Python pe acelaşi set de date de intrare (50, 100, 150, 200, 250, 300, 400, 500, 600, 700 noduri în graf).

4.4 Modulele aplicaţiei

Aplicaţia are următoarele module:

- 10 fişiere de tip text, din care se citeşte numărul de noduri ale grafului. Fiecare fişier contribuie la obţinerea unui test unic.
- 10 fişiere de tip text, în care sunt scrise datele de ieşire menţionate în subsecţiunea de mai sus.
- main.c în care sunt deschise fişierele de mai sus în modul "citire" şi "scriere". De asemenea, din acest modul se citeşte numărul de noduri din fişier şi se calculează timpul de execuţie. Tot aici este apelată funcţia

de generare a matricei de adiacenţă a grafului, care conţine costurile pozitive asociate muchiilor, generate în mod aleatoriu şi funcţia principală arbore_minim_de_acoperire_prim sau kruskal (depinde de algoritmul folosit).

- graf.c în care se află definiţiile tuturor funcţiilor folosite în aplicaţie.
- graf.h în care se află declaraţiile tuturor funcţiilor, variabila număr_noduri şi variabilele f şi e utilizate în lucrul cu fişiere, toate acestea fiind folosite în aplicaţie.

4.5 Funcţiile aplicaţiei

Funcţiile ce urmează a fi prezentate sunt utilizate atât în C cât şi în Python. Prima dată voi prezenta funcţiile corespunzătoare algoritmului Prim.

- iniţializare_matrice_adiacentă este utilizată pentru a construi matricea de adiacenţă a grafului. Funcţia primeşte ca parametri numărul de noduri ale grafului şi o matrice pătratică cu număr_noduri coloane şi număr_noduri linii. Elementele din matrice sunt generate în mod aleatoriu şi reprezintă costurile pozitive asociate muchiilor grafului. Pe diagonala principală a matricii se pun numai 0, deoarece în acest graf nu pot exista muchii între un nod şi el însuşi. Având în vedere faptul că graful este neorientat şi conex, atunci $graf[i][j] = graf[j][i]$, adică nu contează sensul de parcurgere a unei muchii, iar valoarea rămâne aceeaşi. La final, matricea primită ca parametru va fi modificată astfel încât să constituie reprezentarea grafului sub forma unei matrici de adiacenţă, în care costurile asociate muchiilor sunt generate aleatoriu.
- determinare_valoare_minimă este folosită pentru a găsi nodul cu cea mai mică valoare asociată şi care nu a fost încă vizitat. Nodurile "capătă valori" în felul următor: Pe măsură ce parcurgem graful asociem nodurilor adiacente nodului curent valoarea muchiei prin care sunt legate de nodul curent, până când nodurile devin vizitate în cele din urmă. În cazul în care, repetând pasul de mai sus, se găseşte o muchie cu o valoare mai mică decât cea care este deja ataşată în nodul corespunzător, iar nodul nu este încă vizitat, se va actualiza cu această valoare, iar dacă muchia are o valoare mai mare, atunci nu se va face nici o schimbare. Funcţia primeşte ca parametri numărul de noduri, vectorul în care se ţine cont de valorile asociate nodurilor şi vectorul în care se ţine cont dacă un nod a fost vizitat sau nu. Se parcurge vectorul în care se află valorile nodurilor şi funcţia returnează numărul nodului care are valoarea minimă şi care nu a fost încă vizitat.
- afişare_arbore_minim este utilizată pentru a afişa muchiile arborelui minim de acoperire găsit şi valorile acestora cu ajutorul vectorului nod_părinte în care se află nodul "părinte" al fiecărui nod vizitat pentru a putea determina muchiile din care este compus arborele. De asemenea, în această funcţie este calculată şi suma acestor valori pentru a determina "costul"

arborelui construit. Funcţia primeşte ca parametri numărul de noduri, vectorul nod_părinte şi matricea de adiacenţă a grafului.

- arbore_minim_de_acoperire_prim reprezintă funcţia principală care primeşte ca parametri numărul de noduri şi matricea de adiacenţă a grafului. Prima dată vectorul noduri_vizitate va avea valoarea 0 pe toate poziţiile deoarece nici un nod nu este vizitat. Vectorul valori_noduri va avea valoarea 0 pe poziţia 0, ceea ce asigură faptul că primul nod va fi rădăcina arborelui ce urmează a fi construit. Apoi într-o iteraţie de număr_noduri de ori, cu ajutorul funcţiei determinare_valoare_minimă se găseşte valoarea minimă dintre nodurile care nu au fost încă vizitate, iar în vectorul noduri_vizitate se va pune valoarea 1 pe poziţia găsită de funcţie. Apoi într-o altă iteraţie de număr_noduri de ori se actualizează valorile şi părinţii nodurilor adiacente nodului selectat prin intermediul funcţiei folosite anterior. De asemenea, se iau în considerare doar acele noduri care nu au fost deja vizitate. În final se apelează funcţia afişare_arbore_minim pentru a oferi datele de ieşire discutate mai sus.

A doua oară voi prezenta funcţiile corespunzătoare algoritmului Kruskal.

- iniţializre_matrice_adiacentă care face acelaşi lucru ca în algoritmul precedent, deoarece este funcţia de generare aleatorie a datelor de intrare pentru ambii algoritmi.

- găseşte_părinte al cărei scop este de a evita formarea ciclurilor. Funcţia primeşte ca parametri numărul de noduri, nodul al cărui părinte trebuie găsit şi vectorul nod_părinte în care se ţine cont de părintele fiecărui nod din arborele minim de acoperire. De fiecare dată când se alege o muchie se verifică care este nodul părinte al fiecărui nod din care este formată muchia. În acest fel putem distinge trei cazuri:

  i) Dacă cele două noduri nu au fost ataşate la arborele minim de acoperire (valoarea lor din nod_părinte este -1), atunci muchia se ataşează la arbore, iar primul nod devine părintele celui de-al doilea ($nod\_parinte[v] = u$), unde u si v reprezintă nodurile care delimitează muchia.

  ii) Dacă muchia selectată are un nod ataşat la arborele minim de acoperire, iar celălalt nod nu este inclus în acesta (nod_părinte este -1), atunci acesta va fi adăugat la arbore, iar primul nod devine părintele celui de-al doilea ca în primul caz.

  iii) Dacă muchia selectată are ambele noduri în arbore, atunci se verifică dacă nu cumva adăugarea acestei muchii va forma un ciclu. Pentru fiecare nod se traversează din "părinte în părinte" până se ajunge la ultimul nod părinte (rădăcina). Dacă pentru ambele noduri această rădăcină coincide, atunci muchia este înlăturată deoarece ar forma un ciclu. Această traversare se efectuează într-o structură repetitivă de tip while cu condiţia $nod\_parinte[nod]! = -1$ (cât timp nodul are părinte) în care se execută comanda $nod = nod\_parinte[nod]$ (se trece la părintele nodului).

Funcţia returnează părintele nodului care este analizat. Dacă nodul face parte dintr-o legătură în care părintele său are la rândul lui un părinte şi aşa mai departe, atunci funcţia returnează mai exact rădăcina acestei legături.

- reuniune_noduri, funcţie prin care se reunesc 2 noduri pentru a face parte din acelaşi arbore minim de acoperire. Cu ajutorul funcţiei de mai sus (găseşte_părinte) cele două noduri se reunesc prin urmare a constatării faptului că fiecare dintre ele au o rădăcină a arborelui diferită (dacă funcţia de mai sus ar returna aceeaşi valoare pentru ambele noduri ar însemna faptul că se va forma un ciclu, iar reuniunea nu va avea loc). Funcţia primeşte ca parametri numărul de noduri din graf, nodurile (u şi v) care delimitează muchia care este analizată şi vectorul nod_părinte. Dacă funcţia adaugă cu succes muchia la arbore fără a forma un ciclu, atunci primul nod devine părintele celui de-al doilea nod($nod\_parinte[v] = u$), iar funcţia va returna 1. În caz contrar, returnează 0.

- kruskal reprezintă funcţia principală care primeşte ca parametri numărul de noduri şi matricea de adiacenţă a grafului. Prima dată iniţializăm vectorul nod_părinte cu -1 deoarece toate nodurile sunt "libere"(nu au nici un nod la început). Parcurgem matricea asociată grafului, iar acolo unde nu există legături (acolo unde este valoarea 0) dăm o valoare cât mai mare pentru a elimina elementul din căutarea valorii minime. În variabila număr_muchii_arbore se va ţine cont de numărul de muchii din arborele care este construit pentru a ştii când trebuie să ne oprim din căutare. Prin urmare, într-o iteraţie de tip while cu condiţia $numar\_muchii\_arbore < numar\_noduri$ vom executa următorii paşi. Parcurgem matricea de adiacenţă a grafului şi găsim muchia cu valoarea minimă asociată pe care o salvăm în variabila *minim*. De asemena, nodurile care formează această muchie vor fi salvate în câte două variabile ($a = u = i$ şi $b = v = j$) pentru a putea fi folosite în mai multe funcţii, fără a pierde conţinutul original. Variabilele u şi v se vor folosi în funcţia găseşte_părinte pentru a analiza dacă adăugarea acestei muchii va forma un ciclu sau nu. Dacă reuniune_noduri(număr_noduri,u,v,nod_părinte) returnează 1 atunci muchia va fi ataşată la arborele minim de acoperire, iar numărul de muchii va creşte cu o unitate. De asemenea, într-o variabilă cost_arbore_minim, iniţializată cu valoarea 0, se va adăuga valoarea acestei muchii. Cu ajutorul variabilelor a şi b se pot afişa nodurile care delimitează muchia, iar prin intermediul variabilei *minim* valoarea muchiei. Chiar dacă muchia a fost adăugată sau nu la arbore, valoarea acesteia va deveni 10000 (sau un număr cât mai mare) pentru a o elimina din viitoarele căutări ale valorii minime din matrice. După ce iteraţia termină de executat aceşti paşi (se găsesc $numar\_noduri - 1$ muchii pentru arbore) se va afişa şi costul arborelui minim format.

5 Rezultate şi Concluzii

În tabelul următor este pusă în evidenţă performanţa algoritmului Prim din perspectiva timpului de execuţie pe seturi de date de dimensiuni din ce în ce mai mari. Analiza timpului de execuţie al acestui algoritm a fost realizată atât în limbaj C cât şi în Python.

Nr	Numărul de noduri	C	Python
1	50	0.000000000000	0.002000093460083008
2	100	0.000000000000	0.010000467300415039
3	150	0.000000000000	0.021001100540161133
4	200	0.000000000000	0.04500269889831543
5	250	0.000000000000	0.05800318717956543
6	300	0.000000000000	0.08300495147705078
7	400	0.000000000000	0.1480085849761963
8	500	0.000000000000	0.23101329803466797
9	600	0.000000000000	0.3280186653137207
10	700	0.000000000000	0.4460256099700928

În tabelul următor este pusă în evidenţă performanţa algoritmului Kruskal din perspectiva timpului de execuţie pe seturi de date de dimensiuni din ce în ce mai mari. Analiza timpului de execuţie al acestui algoritm a fost realizată atât în limbaj C cât şi în Python.

Nr	Numărul de noduri	C	Python
1	50	0.000000000000	0.14700865745544434
2	100	0.000000000000	0.9950568675994873
3	150	0.000000000000	4.017229795455933
4	200	0.000000000000	6.58237624168396
5	250	0.000000000000	14.098806619644165
6	300	0.000000000000	27.25555920600891
7	400	0.000000000000	101.21778964996338
8	500	0.000000000000	133.46263337135315
9	600	1.000000000000	312.83489298820496
10	700	3.000000000000	499.54057216644287

În concluzie, testând ambii algoritmi pe aceleaşi seturi de date de intrare putem observa faptul că primul algoritm, Prim, este mai eficient din punct de vedere al timpului de execuţie. În timp ce timpul de execuţie al algoritmului Prim creşte foarte încet, în cazul algoritmului Kruskal, pe măsură ce seturile de date au dimensiuni din ce în ce mai mari, timpul de execuţie creşte în mod considerabil.

References

[1] LaTeX project site, http://latex-project.org/

[2] https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes

[3] https://en.wikipedia.org/wiki/Prim%27s_algorithm

[4] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[5] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*. MIT Press, 3rd Edition, 2009, pp. 624-636.