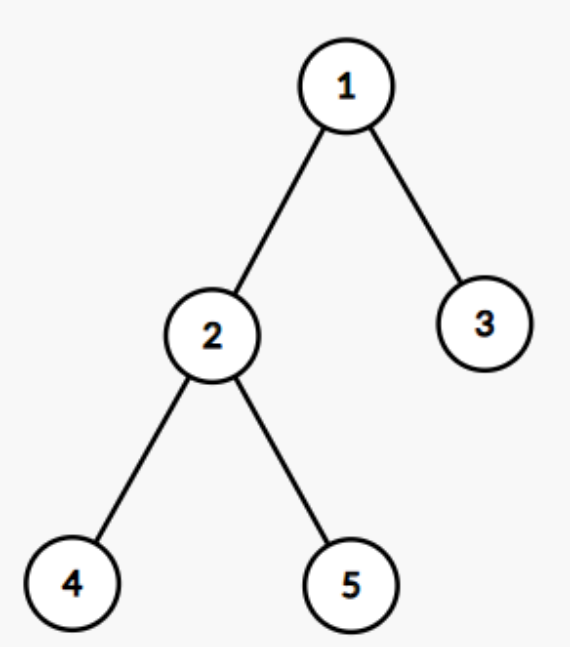
Arbori. DSU. Kruskal

BUZATU GIULIAN & NIȚĂ ALEXANDROS

Arbori

Un arbore este un graf conex și aciclic, cu n noduri și n-1 muchii. Fiecare arbore are o singură rădăcină (un nod fără părinte), dar poate avea mai multe frunze (noduri fără fii).

Într-un arbore definim noțiunea de strămoș, adică un nod care se află deasupra nodului curent în drumul său spre rădăcina arborelui, cât și pe cea de descendent, adică nodurile pentru care nodul curent este strămoș.



Disjoint Set Union (DSU)

Structura de date Disjoint Set Union (Păduri de mulțimi disjuncte), numită și Union-Find, este des folosită și întâlnită în problemele de algoritmică. Aceasta ne ajută în situații precum următoarea:

Avem mai multe elemente, iar fiecare se află într-o mulțime separată și fiecare mulțime are un reprezentant. Această structură de date ne ajută să aflăm reprezentantul mulțimii în care se află un anume element, cât și să unim două mulțimi într-una singură.

Disjoint Set Union (DSU)

DSU suportă următoarele 2 operații:

-find_set(v) – returnează reprezentantul mulțimii în care se află v; acest reprezentant este și el un element al mulțimii și este ales de către structura de date prin intermediul operației union_sets(a,b) și se poate schimba pe parcursul programului;

-union_sets(a,b) – realizează reuniunea celor două mulțimi în care se află elementele a și b. Dacă cele două elemente sunt în aceeași mulțime, lucru pe care îl putem afla cu ajutorul operației find_set, atunci operația nu modifică nimic în cadrul structurii de date.

Disjoint Set Union (DSU)

Pentru o interpretare mai ușoară a acestei structuri de date, ne putem gândi că fiecare mulțime este de fapt un arbore. Astfel, în prima operație căutăm, de fapt, rădăcina arborelui în care se află nodul v, iar pentru cea de a doua, ne dorim să unim doi arbori într-unul.

Disjoint Set Union (DSU)

Totuși, dacă pur și simplu implementăm cele două operații, complexitatea acestei structuri de date poate degenera foarte ușor în $O(n)$ pentru fiecare operație find_set.

Disjoint Set Union (DSU)

Totuși, dacă pur și simplu implementăm cele două operații, complexitatea acestei structuri de date poate degenera foarte ușor în $O(n)$ pentru fiecare operație find_set.

Prin urmare, s-au descoperit următoarele optimizări:

1. Compresia drumurilor
2.1. Reuniune după rang
2.2. Reuniune după mărime

Optimizările 2.1. și 2.2. sunt echivalente. În urma aplicării optimizărilor, complexitatea structurii de date devine aproape $O(1)$ în medie.

DSU – Optimizări

1. **Compresia drumurilor**:

Am văzut cum putem să deducem un reprezentant al unui nod parcurgând arborele din care face parte nodul până la rădăcina sa. Însă acest arbore poate deveni adânc, făcând parcurgerea sa lentă. Trebuie să găsim o metodă de a scurta cumva drumul. Aceasta se poate realiza prin mutarea fiecărui nod din lanțul de la nodul căutat la rădăcină direct la nodul rădăcină. Astfel, la următoarea apelare a vreunui astfel de nod, în loc să trecem prin tot lanțul, vom trece doar printr-o muchie.

DSU – Optimizări

2.1. Reuniune dupa rang

La reuniunea arborilor, vom ține minte adâncimea fiecăruia. Astfel, atunci când vom uni doi arbori, vom alege să-l unim pe cel mai puțin adânc la cel mai adânc. Dacă am face invers, nodurile din arborele mai adânc devin și mai adânci cu 1 (muchia cu care unim rădăcinile).

DSU – Optimizări

2.2. Reuniune dupa mărime

În mod asemănător, putem să ținem un vector în care memorăm numărul de noduri din fiecare arbore. Vom uni arborele cu număr mai mic de noduri la cel cu număr mai mare pentru a avea mai puține noduri pentru care să schimbăm părintele.

Cele două metode au aceeași complexitate și se comportă similar ca performanță. De notat este că nu vom actualiza rangul arborilor după ce folosim compresia de drumuri, acesta rămânând o limită superioară.

DSU – Complexitate

Așa cum am menționat mai sus, combinarea celor două optimizări – compresia drumurilor și reuniunea după rang/mărime – ajungem la un timp aproape constant pentru query-uri.

Complexitatea acestei structuri de date este greu de demonstrat, aceasta fiind $O(\text{inversa funcției lui Ackermann})$, care crește foarte încet. Creșterea este atât de înceată încât funcția nu depășește valoarea 4 pentru nicio valoare rezonabilă a lui n (aproximativ $n < 10^{600}$)

DSU – Implementare

Problemă: https://infoarena.ro/problema/disjoint

Implementare: https://infoarena.ro/job_detail/3170001?action=view-source

Problemă – DSU

Să rezolvăm problema: https://infoarena.ro/problema/bile

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/bile

Hint: Este mult mai ușor să adăugăm elemente în arbori decât să le ștergem.

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/bile

Soluție: Parcurgem vectorul de coordonate de la final și adăugăm fiecare nod într-unul din arborii vecini, dacă nu face deja parte din acesta. Folosim reuniunea după mărime, deoarece ne ajută pentru aflarea maximului. După ce verificăm pentru toți vecinii elementului curent, vom actualiza maximul. La final afișăm vectorul în care avem stocate valorile.

Implementare: https://infoarena.ro/job_detail/3170179?action=view-source

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/curcubeu

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/curcubeu

Hint 1: Putem să aplicăm operațiile de la ultima la prima.

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/curcubeu

Hint 1: Putem să aplicăm operațiile de la ultima la prima.
Hint 2: Ce optimizare specifică DSU putem folosi la această problemă?

Problemă - DSU

Să rezolvăm problema: https://infoarena.ro/problema/curcubeu

Soluție: Parcurgem operațiile de la ultima la prima, deoarece știm că, în acest mod, odată actualizată o poziție, aceasta nu își va mai schimba culoarea. Putem să ne folosim de compresia drumurilor pentru a reține pentru fiecare element care este primul element necolorat de la dreapta sa. Astfel, pentru fiecare operație colorăm un element și după continuăm să colorăm de la părintele următorului element. Totodată actualizăm și părintele nodurilor pe care le colorăm.

Implementare: https://infoarena.ro/job_detail/3169973?action=view-source

Algoritmul lui Kruskal

Problemă: Dându-se un graf ponderat, să se afle un arbore parțial cu suma costurilor muchiilor minimă.

Algoritmul lui Kruskal

Vrem să avem suma minimă, deci, intuitiv, vom folosi cele mai mici muchii. Prin urmare, sortăm muchiile după cost. Acum va trebui, pentru fiecare muchie, să hotărâm dacă o adăugăm în arbore sau nu. Mai exact, dacă muchia duce la pierderea proprietăților de arbore, evident nu o vom alege. Dacă cele două noduri sunt deja în arbore, atunci nu adăugăm muchia. Folosim pentru aflarea acestei informații DSU.

Algoritmul lui Kruskal

Descriere algoritm:

1. Sortăm muchiile după cost.
2. Parcurgem vectorul cu muchii și, pentru fiecare muchie, decidem dacă o păstrăm sau nu.
3. Verificăm dacă cele două vârfuri ale muchiei sunt deja în arbore. Dacă nu sunt, adăugăm muchia. Daca sunt, trecem la următoarea.

Complexitatea depinde de algoritmul de sortare folosit, deoarece mereu vom implementa DSU cu cele două optimizări. Sortarea prin inserție va produce o complexitate de $O(n^2)$, în timp ce cea prin interclasare obține $O(n \log n)$. Recomandăm funcția $std :: sort$ din STL, ce va duce la complexitatea $O(n \log n)$.

Algoritmul lui Kruskal

Problemă: https://infoarena.ro/problema/apm
Implementare : https://infoarena.ro/job_detail/3170211?action=view-source

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/rusuoaica

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/rusuoaica

Hint: Cum putem aplica cele trei operații eficient pe graful nostru?

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/rusuoaica

Soluție: Aplicăm algoritmul lui Kruskal, dar folosindu-ne de operațiile suplimentare pe care ni le oferă problema. Dacă putem lua o muchie în arbore, avem două cazuri: aceasta are costul strict mai mare decât A, atunci putem să o înlocuim cu o muchie de cost A, altfel, pur și simplu luăm muchia în arbore. În ambele cazuri, actualizăm corespunzător răspunsul. Dacă o muchie nu este aleasă, atunci scădem costul ei din răspuns. La final, este posibil să nu avem doar un APM, ci o pădure de „APM-uri”. Deci conectăm arborii între ei cu muchii de cost A.

Implementare: https://infoarena.ro/job_detail/3170000?action=view-source

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/ninjago

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/ninjago

Hint: Încercați să vedeți care muchii sunt prioritare.

Problemă – Algoritmul lui Kruskal

Să rezolvăm problema: https://infoarena.ro/problema/ninjago

Soluție: Pentru început, să constatăm că cerința 1 nu cere nimic legat de distanță. Deci vom parcurge graful printr-un DFS/BFS prin muchiile în care nu avem E. Pentru cerințele 2 și 3 vom folosi Algoritmul lui Kruskal, dar trebuie să vedem cum să facem sortarea. Vom sorta în funcție de numărul de E-uri din cod, iar, dacă acestea sunt egale, în funcție de costul muchiei. Astfel, minimizăm numărul de E-uri (cel mai important) și distanța parcursă (mai puțin important).

Implementare: https://infoarena.ro/job_detail/3170250?action=view-source

Temă

- https://infoarena.ro/problema/mexc
- https://codeforces.com/edu/course/2/lesson/7/2/practice/contest/289391/problem/A
- https://infoarena.ro/problema/desen
- https://codeforces.com/contest/25/problem/D

Probleme suplimentare

- https://codeforces.com/problemset/problem/1468/J
- https://infoarena.ro/problema/autobuze
- https://infoarena.ro/problema/oracol
- https://infoarena.ro/problema/autobuze3
- https://codeforces.com/problemset/problem/1810/E

Lectură suplimentară

- https://cp-algorithms.com/data_structures/disjoint_set_union.html
- https://codeforces.com/edu/course/2/lesson/7
- https://wikious.com/en/Disjoint-set_data_structure
- https://wikious.com/en/Ackermann_function#Inverse