Математичка гимназија

Матурски рад из информатике

АЛГОРИТАМ DSU

(Disjoint Set Union) и његове примене

Ученик

Милан Икодиновић
IVa

Ментор

проф. Снежана Јелић

Београд, мај 2026.

Садржај

1

Увод

У информатици многи алгоритми настају ради решавања конкретних проблема. Један од таквих проблема гласи: имамо велики број елемената подељених у групе. Како да те групе брзо спајамо? Како да у сваком тренутку знамо да ли два елемента припадају истој групи?

Овај проблем се често јавља у графовским задацима, мрежама, обради слика и многим другим областима. Како је временом расла количина података, јављала се све већа потреба за што ефикаснијим решењем.

У почетку су овакви проблеми могли да се решавају једноставнијим, али споријим поступцима. Када је количина података мала, и неефикасна решења могу бити довољна. Међутим, чим порасте количина података или учесталост упита, јавља се потреба за посебном структуром података, познатом као DSU (*Disjoint Set Union*), односно као структура дисјунктних скупова.

Први значајан корак у решавању овог проблема учинили су амерички математичари и информатичари Бернард А. Галер (*Bernard A. Galler*) и Мајкл Џон Фишер (*Michael John Fischer*) 1964. године, када су представили дисјунктне скупове помоћу шуме стабала. Ова идеја била је важна јер је по први пут омогућила да се проблем решава уређеном структуром.

Године 1973. Џон Едвард Хопкрофт (*John Edward Hopcroft*) и Џефри Дејвид Улман (*Jeffrey David Ullman*) дали су бољу анализу временске сложености ове структуре. Неколико година касније, 1975. године, амерички информатичар и математичар Роберт Тарјан (*Robert Endre Tarjan*) дао је дубљу анализу ове структуре и показао да се применом кључних оптимизација временска сложеност може знатно смањити, до скоро константне амортизоване сложености.

Тако се из једноставног проблема развила усавршена структура која је постала један од кључних алата у теорији алгоритама. Њен значај временом се проширио и на многе друге проблеме, па се DSU данас користи у бројним алгоритмима и различитим применама.

Најпознатија примена јавља се у Крускаловом алгоритму за налажење минималног разапињућег стабла, алгоритму названом по америчком математичару и информатичару Џозефу Крускалу (*Joseph Kruskal*).

Поред тога, DSU се користи за праћење повезаних компоненти у графу, проверу постојања циклуса у графу, у алгоритмима за обраду слика као што је Хошен--Копелманов алгоритам (*Hoshen--Kopelman*) обележавања кластера на мрежи, као и у разним задацима где је потребно динамички одржавати групе елемената које се током рада спајају.

У овом раду биће приказано како се дошло до ефикасне верзије DSU структуре, које су оптимизације уведене и које су њене најважније примене.

2

Основни појмови

Да би се структура дисјунктних скупова јасно разумела, потребно је увести неке кључне појмове. Ови појмови представљају математичку основу на којој лежи овај алгоритам.

2.1 Скупови, партиције и релације еквиваленције

Скуп се као појам обично не дефинише, већ се узима као основни појам. За потребе овог рада довољно је навести нека основна својства и ознаке:

- $\varnothing$ - ознака за празан скуп;
- $x \in A$ - елемент x припада скупу A;
- $A = \{x \mid P(x)\}$ - скуп A се састоји од свих елемената x који имају особину P;
- $A \subseteq B$ - скуп A је подскуп скупа B;
- $A \cap B$ - пресек скупова;
- $A \cup B$ - унија скупова.

Један од најважнијих појмова за ову тему јесте **партиција скупа**. Партиција скупа A је фамилија непразних подскупова $\{A_1, A_2, \ldots, A_n\}$ таквих да важи:

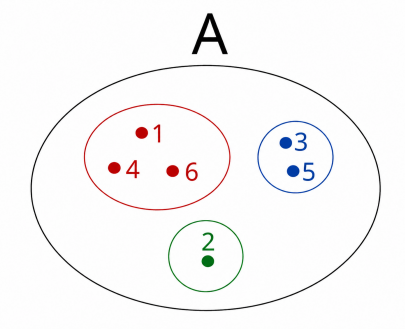
1. $A_i \neq \varnothing$;
2. $A_1 \cup A_2 \cup \cdots \cup A_n = A$;
3. $A_i \cap A_j = \varnothing$ за $i \neq j$.

Као пример, нека је

$$A = \{1, 2, 3, 4, 5, 6\}.$$

Једна могућа партиција скупа A је

$$\big\{\{1, 4, 6\}, \{3, 5\}, \{2\}\big\}.$$



Слика 1. Пример партиције скупа

Са појмом партиције уско је повезан и појам **релације еквиваленције**. Релација ρ је:

1. **рефлексивна** ако важи $x \,\rho\, x$;
2. **симетрична** ако из $x \,\rho\, y$ следи $y \,\rho\, x$;
3. **транзитивна** ако из $x \,\rho\, y$ и $y \,\rho\, z$ следи $x \,\rho\, z$.

Релација је **релација еквиваленције** ако је рефлексивна, симетрична и транзитивна. Релације еквиваленције деле скуп на класе еквиваленције, а оне чине партицију скупа. Управо је ова веза суштина структуре дисјунктних скупова.

2.2 Графови и повезане компоненте

Граф је апстрактни математички објекат који се састоји од скупа чворова и скупа грана које повезују неке парове чворова. Граф се означава са

$$G = (V, E),$$

где је V скуп чворова, а E скуп грана.

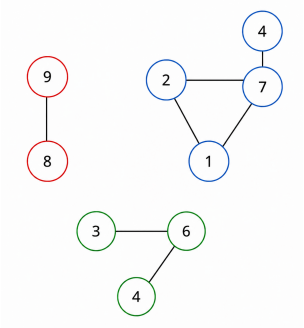
У овом раду посматрају се неусмерени графови. За два чвора кажемо да су повезани ако постоји пут који спаја та два чвора. Повезана компонента неусмереног графа представља максималан скуп чворова тако да је унутар тог скупа могуће доћи од једног до свих осталих чворова.

Компоненте графа чине партицију скупа његових чворова. Сваки чвор припада тачно једној компоненти, док различите компоненте не деле ниједан чвор. Због тога је DSU структура природан алат за овакве проблеме.

Када се у граф дода нова грана, постоје две могућности:

- два чвора већ припадају истој компоненти;
- два чвора припадају различитим компонентама, па се те компоненте спајају.

Управо на том принципу функционише структура дисјунктних скупова.



Слика 2. Пример повезаних компоненти у неусмереном графу

3

Структура дисјунктних скупова

3.1 Основне операције

У претходном поглављу описан је проблем спајања компоненти. Структура дисјунктних скупова служи за одржавање поделе неког скупа елемената на више међусобно дисјунктних подскупова. Основна идеја ове структуре је да сваки подскуп има свог представника. Такав приступ олакшава и спајање скупова и проверу припадности два елемента истом скупу, јер се све своди на рад са представницима. Уводимо три основне операције структуре дисјунктних скупова: прављење једночланог скупа, проналажење представника и спајање два скупа.

`napravi_skup(x)`

Прави нови скуп који садржи само елемент x.

`spoji_skupove(x, y)`

Спаја скупове у којима се налазе елементи x и y.

`nadji_predstavnika(x)`

Враћа представника скупа у коме се налази елемент x.

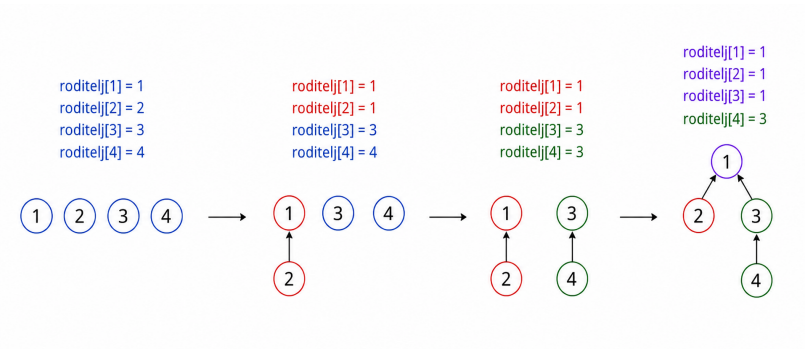
3.2 Репрезентација помоћу шуме стабала

Скупови ће бити представљени у облику стабала, при чему свако стабло представља један скуп. Корен стабла представља представника тог скупа.

За имплементацију је важно да се одржава низ `roditelj` који за сваки чвор чува елемент који се у стаблу налази непосредно изнад њега.

Представник скупа добија се тако што се, почев од датог чвора, прати низ родитеља све док се не дође до корена стабла.

На слици је приказан пример извршавања операција спајања, као и промена низа `roditelj`.



Слика 3. Пример извршавања операција спајања и одговарајућих промена низа `roditelj`

3.3 Наивна имплементација

На основу претходно описаних идеја, наивна имплементација наведених функција у програмском језику C++ може изгледати овако:

```
1  void napravi_skup(int x)
2  {
3      roditelj[x] = x;
4  }
5
6  int nadji_predstavnika(int x)
7  {
8      if(x == roditelj[x])
9      {
10         return x;
11     }
12     return nadji_predstavnika(roditelj[x]);
13 }
14
15 void spoji_skupove(int x, int y)
16 {
17     x = nadji_predstavnika(x);
18     y = nadji_predstavnika(y);
19     if (x != y)
20     {
21         roditelj[y] = x;
22     }
23 }
```

Недостатак ове имплементације је то што процес налажења представника може бити веома спор. У најгорем случају стабло може бити ланац од n чворова. Тада би временска сложеност тражења представника била $O(n)$, јер би било потребно проћи кроз скоро све чворове. Пошто је сложеност сваког позива $O(n)$, укупна сложеност за q упита износи $O(nq)$. У следећем поглављу биће приказане оптимизације које отклањају овај недостатак.

4

Оптимизације DSU структуре

У претходним поглављима видели смо да наивна имплементација може бити спора у неповољним случајевима. У овом поглављу биће уведене три најважније оптимизације:

- Union by size (унија према величини),
- Union by rank (унија по ранку),
- Path compression (компресија путање).

4.1 Union by size (Унија према величини)

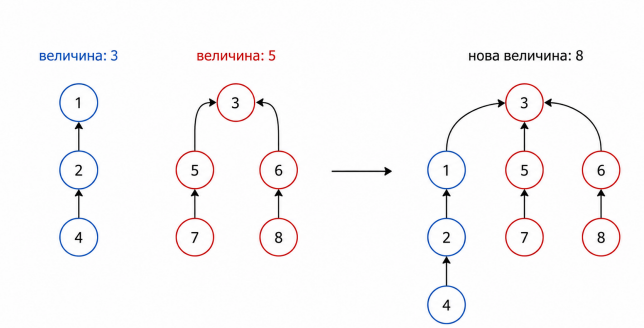
У овој оптимизацији мења се функција `spoji_skupove`. Прецизније, мења се правило по коме се одређује које се стабло придружује другом. У наивној имплементацији друго стабло се увек придруживало првом, што у пракси може довести до стабала која су ланци дужине n. Овом оптимизацијом то се избегава систематским избором стабла које ће бити придружено другом.

Увек се скуп са мањим бројем чворова придружује скупу са већим бројем чворова. За сваки скуп памти се број чворова у њему помоћу низа `velicina`. У функцији `napravi_skup` сваком елементу се додељује вредност 1, јер у том тренутку чини једночлани скуп. При спајању се одређује који од елемената x и y припада мањем скупу, па се тај скуп придружује већем.

Овом оптимизацијом временска сложеност по упиту постаје $O(\log n)$, што је знатно боље од наивне имплементације.

```
void napravi_skup(int x)
{
    roditelj[x] = x;
    velicina[x] = 1;
}

void spoji_skupove(int x, int y)
{
    x = nadji_predstavnika(x);
    y = nadji_predstavnika(y);
    if(x != y)
    {
        if(velicina[x] < velicina[y])
        {
            swap(x, y);
        }
        roditelj[y] = x;
        velicina[x] += velicina[y];
    }
}
```



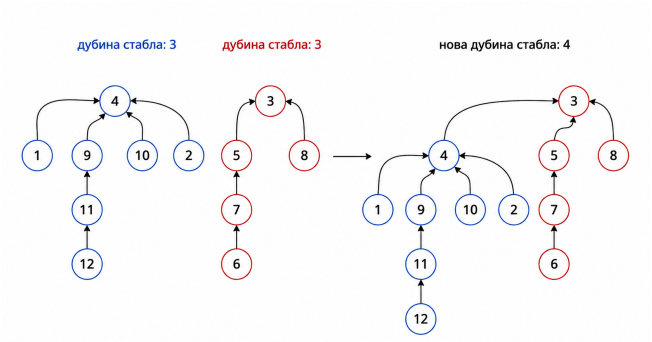
Слика 4. Пример оптимизације union by size. Стабло са мањим бројем чворова придружује се стаблу са већим бројем чворова. Након спајања, величина новог скупа једнака је збиру величина претходна два скупа.

4.2 Union by rank (Унија по ранку)

Ова оптимизација ради по сличном принципу као претходна, с тим што се уместо величине скупа памти ранг. Ранг не мора увек да буде стварна дубина стабла; он служи као горња граница висине стабла и као помоћна вредност при спајању. При спајању се корен стабла мањег ранга качи на корен стабла већег ранга. Ранг новодобијеног корена мења се само ако су два спојена стабла имала исти ранг; у том случају ранг новог корена повећава се за 1. У супротном, ранг остаје једнак већем од два претходна ранга.

```
void napravi_skup(int x)
{
    roditelj[x] = x;
    rang[x] = 0;
}

void spoji_skupove(int x, int y)
{
    x = nadji_predstavnika(x);
    y = nadji_predstavnika(y);
    if(x != y)
    {
        if(rang[x] < rang[y])
        {
            swap(x, y);
        }
        roditelj[y] = x;
        if(rang[x] == rang[y])
        {
            rang[x]++;
        }
    }
}
```



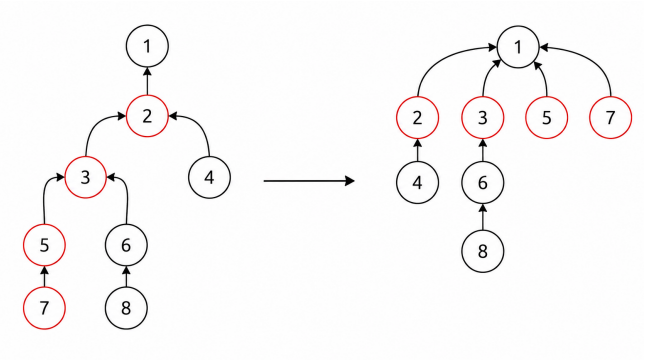
Слика 5. Пример оптимизације union by rank у случају када два стабла имају исти ранг. Један корен се поставља као родитељ другом корену, а ранг новог корена повећава се за 1.

4.3 Path compression (Компресија путање)

Идеја ове оптимизације је да убрза процес налажења представника скупа. Када се позива функција `nadji_predstavnika` за чвор x, заправо се проналази представник за све чворове на путу од чвора x до корена. Основна идеја је да се за сваки од тих чворова скрати путања тако што се њихов родитељ поставља директно на представника.

```
int nadji_predstavnika(int x)
{
    if(x == roditelj[x])
    {
        return x;
    }
    return roditelj[x] = nadji_predstavnika(roditelj[x]);
}
```

Комбинацијом ових оптимизација DSU структура постаје изузетно ефикасна. Union by size (или union by rank) спречавају настанак дубоких стабала приликом спајања, док path compression додатно скраћује путању приликом проналажења представника. Због тога се ове оптимизације у пракси користе заједно. Тада временска сложеност по упиту постаје скоро константна, тачније, амортизована временска сложеност износи $O(\alpha(n))$, где је $\alpha(n)$ инверз Акерманове функције.



Слика 6. Пример компресије путање након позива функције `nadji_predstavnika(7)`. Чворови на путу од чвора 7 до представника скупа директно се повезују са представником, чиме се скраћују будуће претраге.

Важно. Комбинацијом оптимизација *union by rank* и *path compression* добија се амортизована временска сложеност $O(\alpha(n))$.

4.4 Коначна имплементација DSU структуре

Након увођења оптимизација *union by rank* и *path compression*, добија се коначна и ефикасна имплементација DSU структуре. У њој сваки елемент има свог родитеља, док се низ `rang` користи за одржавање ранга корена. Комбинацијом ове две оптимизације добија се веома ефикасна структура података погодна за практичну примену.

```
void napravi_skup(int x)
{
    roditelj[x] = x;
    rang[x] = 0;
}

int nadji_predstavnika(int x)
{
    if(x == roditelj[x])
    {
        return x;
    }
    return roditelj[x] = nadji_predstavnika(roditelj[x]);
}

void spoji_skupove(int x, int y)
{
    x = nadji_predstavnika(x);
    y = nadji_predstavnika(y);
    if(x != y)
    {
        if(rang[x] < rang[y])
        {
            swap(x, y);
        }
        roditelj[y] = x;
        if(rang[x] == rang[y])
        {
            rang[x]++;
        }
    }
}
```

Функција `napravi_skup` прави нови једночлани скуп. Функција `nadji_predstavnika` проналази представника скупа уз компресију путање, док функција `spoji_skupove` спаја два скупа водећи рачуна о рангу стабала.

5

Временска сложеност DSU структуре

Нека је n број елемената, а q укупан број упита над структуром.

5.1 Наивна имплементација

Као што је раније напоменуто, наивна имплементација приликом спајања два скупа не води рачуна о висини стабала. Због тога се у најгорем случају може добити стабло које је ланац дужине n.

Ако је елемент x на дну таквог ланца, операција `nadji_predstavnika(x)` мора да прође кроз све претке елемента x док не стигне до корена. У најгорем случају број пређених чворова пропорционалан је броју елемената, па је временска сложеност ове операције $O(n)$.

Пошто операција `spoji_skupove(x,y)` најпре позива две операције `nadji_predstavnika`, и она у најгорем случају има сложеност $O(n)$.

Дакле, наивна имплементација DSU структуре има линеарну временску сложеност по операцији у најгорем случају.

Закључак. Наивна имплементација DSU структуре има временску сложеност $O(n)$ по операцији у најгорем случају.

5.2 Union by size

Код оптимизације *union by size*, приликом спајања два стабла, корен мањег стабла постаје дете корена већег стабла.

Доказаћемо да је дубина сваког чвора највише $\log_2 n$.

Посматрајмо произвољан чвор x. Дубина чвора x може да се повећа само када се стабло које садржи x качи за друго, веће, стабло. Пошто се увек мање стабло везује за веће, након таквог спајања величина стабла у коме се налази x барем се удвостручи.

Ако се дубина чвора x повећала d пута, онда се величина стабла које садржи x барем d пута удвостручила. Зато важи:

$$2^d \leq n.$$

Логаритмовањем добијамо:

$$d \leq \log_2 n.$$

Према томе, дубина сваког чвора је највише $\log_2 n$. Операција `nadji_predstavnika` прати пут од чвора до корена, па је њена сложеност $O(\log n)$.

Пошто операција `spoji_skupove` користи две операције `nadji_predstavnika`, и њена сложеност је $O(\log n)$.

Дакле, DSU са оптимизацијом *union by size*, али без компресије путање, има сложеност

$$O(\log n)$$

по операцији у најгорем случају.

5.3 Union by rank

Код оптимизације *union by rank*, сваком корену придружује се ранг. Ранг представља горњу границу висине стабла. При спајању два стабла, корен стабла мањег ранга постаје дете корена стабла већег ранга. Ако су рангови једнаки, један корен постаје дете другог, а ранг новог корена повећава се за 1.

Доказаћемо да корен ранга r има барем 2^r чворова у свом стаблу.

Доказ се изводи индукцијом.

За $r = 0$, стабло има бар један чвор, па важи:

$$1 = 2^0.$$

Претпоставимо да свако стабло ранга r има барем 2^r чворова. Ранг стабла се повећава са r на $r + 1$ само када се споје два стабла истог ранга r. По индуктивној претпоставци, свако од та два стабла има барем 2^r чворова. Зато ново стабло има барем

$$2^r + 2^r = 2^{r+1}$$

чворова.

Дакле, тврђење важи за сваки ранг r. Ако је укупан број чворова n, онда за сваки корен ранга r важи:

$$2^r \leq n.$$

Одатле следи:

$$r \leq \log_2 n.$$

Пошто је ранг горња граница висине стабла, висина сваког стабла је највише $O(\log n)$. Зато је временска сложеност операције `nadji_predstavnika` једнака $O(\log n)$, а самим тим је и временска сложеност операције `spoji_skupove` једнака $O(\log n)$.

Дакле, DSU са оптимизацијом *union by rank*, али без компресије путање, има сложеност

$$O(\log n)$$

по операцији у најгорем случају.

5.4 Компресија путање уз union by rank или union by size

Код компресије путање, приликом извршавања операције `nadji_predstavnika`, сви чворови на путу од посматраног чвора до корена директно се повезују са кореном. На тај начин структура стабла се мења тако да будуће операције `nadji_predstavnika` постају брже.

Ако се компресија путање користи заједно са оптимизацијом *union by rank* или *union by size*, добија се много боља амортизована сложеност $O(\alpha(n))$, где је $\alpha(n)$ инверзна Акерманова функција.

Функција $\alpha(n)$ расте изузетно споро. За све практичне вредности броја n, њена вредност је мала константа. Зато се у пракси DSU са обе оптимизације понаша скоро као структура са константном сложеношћу по операцији.

Овај резултат је знатно тежи за доказивање од претходних оцена. Формални доказ користи амортизовану анализу и инверзну Акерманову функцију, а класичан резултат потиче од Тарјана. Зато је у овом раду наведена идеја резултата и његова последица, без потпуног формалног доказа.

Табела 1. Поређење временских сложености различитих имплементација DSU структуре

Имплементација	**Временска сложеност по операцији**
Наивна имплементација	$O(n)$
Union by size	$O(\log n)$
Union by rank	$O(\log n)$
Union by rank + path compression	$O(\alpha(n))$, амортизовано
Union by size + path compression	$O(\alpha(n))$, амортизовано

Из Табеле 1 јасно се види колики напредак се постиже увођењем оптимизација у структуру дисјунктних скупова.

6

Примене DSU структуре

Структура дисјунктних скупова има значајну примену у различитим областима информатике, јер омогућава ефикасно одржавање група елемената које се током рада спајају. Посебно је корисна у проблемима у којима је потребно брзо одредити да ли два елемента припадају истој целини, као и у ситуацијама када се те целине постепено мењају.

У овом поглављу биће приказане неке од најважнијих примена DSU структуре. Посебна пажња биће посвећена Крускаловом алгоритму, јер је то најпознатија и најзначајнија примена ове структуре у теорији графова.

6.1 Праћење повезаних компоненти у графу

Једна од основних примена DSU структуре јавља се при праћењу повезаних компоненти у неусмереном графу. Као што је већ објашњено, повезане компоненте графа чине партицију скупа чворова. Сваки чвор припада тачно једној компоненти, а додавањем нове гране могуће је да се две различите компоненте споје у једну.

Управо у таквим ситуацијама DSU представља природно и ефикасно решење. На почетку се сваки чвор налази у засебном скупу. Када се обради грана која спаја чворове u и v, најпре се проверава да ли та два чвора већ припадају истој компоненти. Ако припадају, структура се не мења. У супротном, њихове компоненте се спајају операцијом `spoji_skupove`.

На тај начин могуће је динамички одржавати информацију о повезаности графа, без потребе да се после сваке нове гране поново претражује цео граф. Ово је нарочито важно код великих графова, где би наивни поступци били превише спори.

6.2 Провера постојања циклуса у неусмереном графу

DSU структура се може користити и за проверу да ли у неусмереном графу настаје циклус приликом додавања нових грана.

Посматрајмо обраду грана једну по једну. За сваку грану која спаја чворове u и v, најпре се одређују представници скупова којима та два чвора припадају. Ако је

$$\texttt{nadji\_predstavnika}(u) = \texttt{nadji\_predstavnika}(v),$$

то значи да су чворови u и v већ повезани неким путем. Додавањем нове гране између њих тада се затвара циклус.

Ако чворови u и v припадају различитим скуповима, онда се њихове компоненте спајају и циклус се не формира.

Ова примена је веома значајна јер омогућава једноставну и ефикасну проверу циклуса током постепеног конструисања графа. Управо се ова идеја користи и у Крускаловом алгоритму, где је потребно додавати гране тако да не настане циклус.

6.3 Крускалов алгоритам

Најпознатија примена DSU структуре јавља се у Крускаловом алгоритму за налажење минималног разапињућег стабла тежинског графа.

Нека је дат повезан неусмерен тежински граф $G = (V, E)$. Разапињуће стабло тог графа јесте подграф који садржи све чворове графа, повезан је и нема ци-

клуса. Ако је свакој грани придружена тежина, онда је минимално разапињуће стабло оно разапињуће стабло чији је збир тежина грана најмањи.

Крускалов алгоритам ради на следећи начин:

1. све гране графа сортирају се по растућој тежини;
2. гране се затим разматрају тим редом;
3. ако грана спаја две различите компоненте, она се додаје у решење;
4. ако би додавање гране створило циклус, та грана се прескаче.

Да би се брзо одредило да ли два краја гране припадају истој компоненти, користи се управо DSU структура. На почетку је сваки чвор у посебном скупу. Када се нека грана прихвати, скупови који садрже њене крајеве се спајају.

Захваљујући томе, провера да ли би нека грана формирала циклус постаје веома ефикасна. Без DSU структуре, овај алгоритам би био знатно спорији.

Суштина примене. У Крускаловом алгоритму DSU структура омогућава ефикасну проверу да ли два чвора већ припадају истој повезаној компоненти, чиме се брзо одлучује да ли грана сме да се дода у разапињуће стабло.

Укупна временска сложеност Крускаловог алгоритма доминантно зависи од сортирања грана, што даје сложеност $O(m \log m)$, где је m број грана. Операције над DSU структуром не повећавају значајно ову сложеност, јер су веома ефикасне.

6.3.1 Имплементација Крускаловог алгоритма

У наставку је приказан поједностављен пример Крускаловог алгоритма у програмском језику C++, уз употребу DSU структуре. Претпоставља се да су гране графа већ учитане у низ `grane`, као и да свака грана има облик `{u, v, tezina}`.

```
struct Grana
{
    int u, v, tezina;
};

bool uporedi(Grana a, Grana b)
{
    return a.tezina < b.tezina;
}

int kruskal(vector<Grana>& grane, int broj_cvorova)
{
    sort(grane.begin(), grane.end(), uporedi);
    for(int i = 1; i <= broj_cvorova; i++)
    {
```

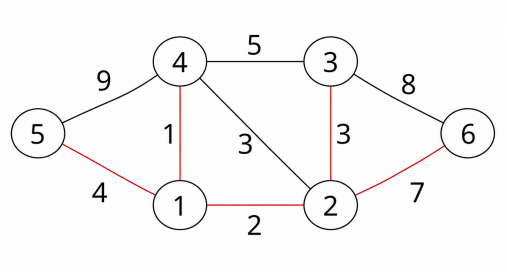
```
16          napravi_skup(i);
17      }
18      int ukupna_tezina = 0;
19      for(Grana g : grane)
20      {
21          if(nadji_predstavnika(g.u) != nadji_predstavnika(g.v))
22          {
23              spoji_skupove(g.u, g.v);
24              ukupna_tezina += g.tezina;
25          }
26      }
27      return ukupna_tezina;
28  }
```

Након сортирања грана по тежини, алгоритам их разматра редом. Ако крајеви неке гране припадају различитим скуповима, грана се додаје у разапињуће стабло и скупови се спајају. У супротном, грана се прескаче јер би њено додавање створило циклус.

6.4 Пример примене у Крускаловом алгоритму

На слици је приказан тежински граф над којим се примењује Крускалов алгоритам. Гране се разматрају редом по растућој тежини. У сваком кораку проверава се да ли крајеви посматране гране припадају различитим повезаним компонентама. Ако припадају, грана се додаје у минимално разапињуће стабло; у супротном, прескаче се како не би настао циклус.

У приказаном примеру алгоритам бира најлакше гране које не затварају циклус. DSU структура омогућава да се ефикасно провери да ли су два краја гране већ у истој компоненти, па је на тај начин могуће брзо одредити које гране улазе у минимално разапињуће стабло.



Слика 7. Пример тежинског графа за примену Крускаловог алгоритма

6.5 Обрада слика и груписање пиксела

Још једна важна примена DSU структуре јавља се у обради слика, нарочито у задацима у којима је потребно препознати повезане области пиксела.

Посматрајмо бинарну слику у којој су неки пиксели означени као активни, а остали као неактивни. Циљ може бити да се одреде све повезане групе активних пиксела. Тада се сваки активни пиксел може посматрати као елемент скупа, а суседни активни пиксели се спајају у исту компоненту.

DSU структура омогућава да се ово спајање изводи ефикасно. Када се током анализе слике утврди да су два пиксела суседна и активна, њихови скупови се спајају. На крају обраде свака повезана област одговара једном дисјунктном скупу.

Ова идеја користи се, на пример, у алгоритмима за означавање кластера у мрежи, као што је Хошен--Копелманов алгоритам.

6.6 Динамичко груписање елемената

DSU структура је корисна и у ширем контексту, кад год је потребно одржавати групе елемената које се током времена спајају.

На пример, може се посматрати систем корисника у друштвеној мрежи, где се током времена формирају веће заједнице. Слично томе, у рачунарским мрежама може бити потребно пратити које су машине међусобно повезане, а у разним симулацијама које честице или објекти припадају истој целини.

У свим тим ситуацијама DSU омогућава:

- ефикасно спајање постојећих група,
- брзу проверу да ли два елемента припадају истој групи,
- једноставно одржавање структуре података током великог броја операција.

Због тога је DSU једна од практичних структура података у алгоритамским задацима.

7

Закључак

У овом раду представљена је структура дисјунктних скупова као једна од важних структура података у теорији алгоритама и практичној информатици. Полазећи од основних појмова као што су партиција скупа, релација еквиваленције и повезане компоненте графа, показано је на којој се математичкој основи заснива ова структура.

У наставку су описане основне операције структуре дисјунктних скупова: прављење новог скупа, проналажење представника и спајање два скупа. Затим је приказана репрезентација помоћу шуме стабала, као и наивна имплементација, чији је главни недостатак велика временска сложеност у неповољним случајевима.

Посебна пажња посвећена је оптимизацијама *union by size*, *union by rank* и *path compression*, јер управо оне чине DSU изузетно ефикасном структуром. Показано је да без компресије путање сложеност операција износи $O(\log n)$, док комбинација одговарајућих оптимизација доводи до амортизоване сложености $O(\alpha(n))$, што је у пракси скоро константно време.

На крају су приказане и најважније примене DSU структуре. Посебно је издвојена њена улога у праћењу повезаних компоненти, провери постојања циклуса и Крускаловом алгоритму за налажење минималног разапињућег стабла. Такође је показано да структура дисјунктних скупова има примену и у обради слика, као и у разним проблемима динамичког груписања елемената.

На основу свега изложеног може се закључити да је DSU једноставна, али веома моћна структура података. Њена једноставна идеја, добра ефикасност и широка примена чине је важним алатом у алгоритмима.

Библиографија

[1] B. A. Galler and M. J. Fischer, *An Improved Equivalence Algorithm*, Communications of the ACM, 1964.

[2] J. E. Hopcroft and J. D. Ullman, *Set Merging Algorithms*, SIAM Journal on Computing, 1973.

[3] R. E. Tarjan, *Efficiency of a Good But Not Linear Set Union Algorithm*, Journal of the ACM, 1975.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press, 2009.

[5] J. B. Kruskal, *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*, Proceedings of the American Mathematical Society, 1956.

[6] cp-algorithms, *Disjoint Set Union*, `https://cp-algorithms.com/data_structures/disjoint_set_union.html`