Гимназија Пирот
Српских владара 128, Пирот

Матурски рад из Програмирања

ТЕОРИЈА ГРАФОВА

Ментор:
Ненад Костић

Ученик:
Александар Госпавић IV5

Пирот, јун 2022.

САДРЖАЈ

Увод

Теорија графова има велику примену у различитим областима, многи инжењерски, биолошки, физички и социолошки проблеми могу да се моделују помоћу графова. Да би све то имало смисла, математичке апстракције које представљају графове су морале наћи свој пут до рачунара како би целокупан допринос који је теорија дала могао имати и практичну вредност, једноставније речено – да би нешто могло да се израчуна.

Многе математичке апстракције имају своје место у рачунарским наукама. Захваљујући њиховом развоју данас је могуће једноставно решавати комплексне проблеме, и што је можда још значајније – могуће их је решавати брзо.

У овом раду биће представљено:

- Појам графа и основне дефиниције везане за графове
- Ојлеров граф, као зачетак теорије графова
- Начини представљања
- Претрага графа
- Тополошко сортирање
- Алгоритми најкраћег пута
- Минимално разапињуће стабло
- Задаци

Сви алгоритми и задаци биће писани у програмском језику C++. Због лакшег записивања променљивих у коду ће бите коришћени изрази и имена на енглеском језику.

Појам графа

Шта су структуре података?

Дефиниција 1 Структура података (енгл. Data Structure) је специјалан формат за организацију података и за његово смештање у рачунарску меморију.

Најбитније структуре података су: низови, листе, редови, стабла, графови... Дакле, ево те магичне везе између теорије графова и рачунарства.

Граф је апстрактни математички објекат, а цртеж који се састоји од тачака и линија је само геометријска представа графа. Међутим, уобичајено је да се таква слика назива графом. Пошто је граф састављен из тачака и линија, које спајају по две тачке, он је одатле могуће извести и формалну дефиницију графа. Оваква уопштена дефиниција омогућује да се граф примењује не само у математици, већ и у информатици, електротехници и техници уопште.

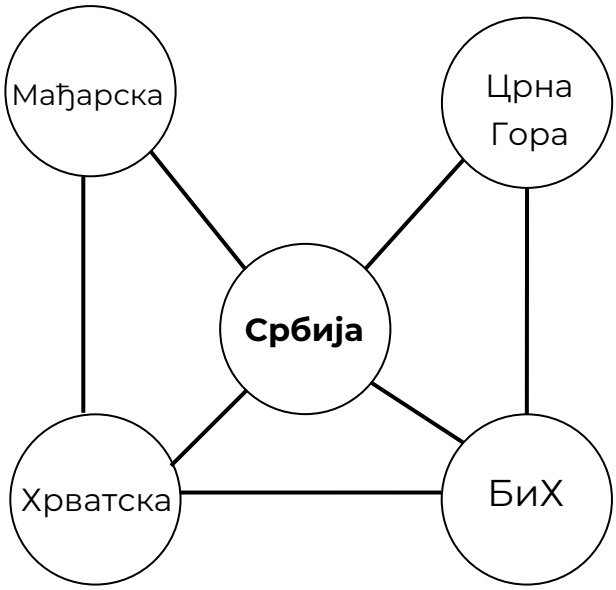
Дефиниција 2 Граф је уређени пар $\boldsymbol{G = (V, E)}$ где је $\boldsymbol{V}$ непразан скуп врхова (енгл. Vertices) или чворова (енгл. Nodes), а $\boldsymbol{E}$ скуп ивица (енгл. Edges) или грана.

Напомена: Скуп чворова V графа G може бити бесконачан. Такав граф са бесконачним бројем чворова или грана зове се бесконачан граф. Насупрот њиме графови за коначним бројем чворова и грана су коначни графови. У овом раду сматраће се да је граф коначан.

Једноставније, граф је скуп чворова који могу бити спојени гранама.

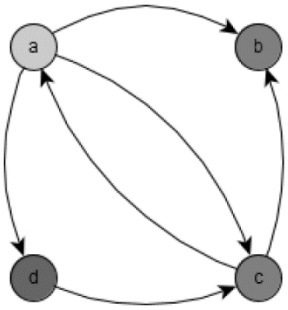
На пример, у друштвеним мрежама сваки појединац може се представити као чвор. Ако су два појединца пријатељи, њихово пријатељство може се представити граном.

Још један пример могу бити градови у држави. Сваки град се може представити као чвор, а путеви између њих као гране. Или, државе и њихове границе као на слици 1.



Слика 1 Граф на коме државе представљају чворове, а гране означавају да те две државе деле границу.

Математички, то ће углавном бити овако:
Граф $\mathrm{G} = (\mathrm{V}, \mathrm{E}); \mathrm{V} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}; E = \{(a,b),(a,c),(a,d),(c,a),(c,b),(d,c)\}$



Слика 2 Усмерени граф са 4 чвора

Као што се види, на овој слици гране су представљене стрелицама. Дакле постоје различите врсте грана, с тога и различите врсте графова. Ове, али и остале дефиниције су обухваћене у наставку.

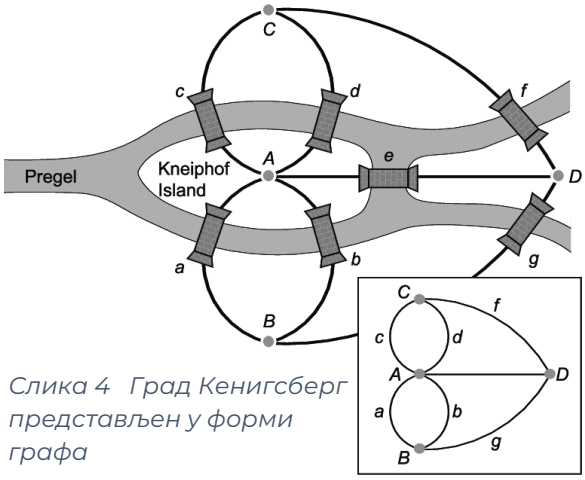
Леонард Ојлер

Пре дефиниција, кратак осврт на зачетак теорије графова као посебне области математике. Један од значајнијих људи у овој области јесте швајцарски математичар Леонард Ојлер (енгл. *Leonhard Euler*).

Слика 3 Леонард Ојлер (1707-1783)

Ојлеру су током боравка у Кенигсбергу (нем. *Konigsberg*; данашњи Калињинград) мештани поставили проблем да пређе преко свих 7 мостова (који спајају 2 обале реке Прегел међусобно и са 2 острва) тако да преко сваког пређе тачно једанпут. Ојлер је дао одречан одговор.

Августа 26., 1735. године, Ојлер је презентовао свој рад на овом проблему Сант Петерсбургшкој академији наука доказујући да је такав обилазак мостова немогућ уз напомену да се његов метод може проширити и на произвољан распоред острва и мостова. Тачније, Ојлер је само формулисао потребне и довољне услове да такав обилазак постоји, али није сматрао да је потребно да покаже довољне услове у општем случају. Први потпуно коректан доказ овог тврђења је дао Хирхолцер (нем. *Hierholzer*).



Слика 4 Град Кенигсберг представљен у форми графа

На слици 4 је представљена мапа Кенигсберга (из времена Ојлера) са његовим мостовима. Ојлер је свакој обали и острву придружио чворове графа, а гране између њих су били мостови. Тако је он добио један мултиграф.

Године 1766., у својој 59. години, Ојлер је остао скоро слеп након неке болести. Захваљујући невероватној меморији, последњих 17 година живота могао је да настави са радом у оптици, алгебри и проучавању лунарног кретања написавши скоро половину свих својих радова. Радио је у скоро свим областима математике, али је повремено такође био окупиран проблемима рекреативне математике. Неки од његових радова у овој области довела су до развоја нових метода, па чак и нових грана математике.

Ојлер је чланак о Проблему Кенигсбергшких мостова написао 1736. године (и стога се та година узима за оснивање *Теорије графова*) и он је први пут објављен 1741. године, али је тада пробудио мало интереса међу осталим математичарима.

Овај проблем и резултат су остали мало познати до краја 19. века када су га енглески математичари Џорџ Лукас (енгл. *George Lucas*, 1882.) и Раус Бол (енгл. *Rouse Ball*, 1892.) укључили у своје књиге о рекреативној математици. Појам Ојлеровог графа за граф који се може нацртати не подижући оловку са папира се одомаћио захваљујући Кенигу, који га је искористио у својој пионирској књизи о Теорији графова 1936. године.

Тражење Ојлеровог пута налази примену у још неким проблемима Комбинаторне оптимизација, попут проблема кинеског поштара, али и у раду са ласерима, где нам је циљ да оптимално користимо ласер и самим тим појефтинимо цену финалног производа (методама Комбинаторне оптимизације постигнута је уштеда времена рада и до 90%).

Теорема 1 Ојлерова теорема: Граф се може нацртати без подизања оловке, прелазећи преко сваке ивице тачно једанпут, ако и само ако граф има два или ниједан чвор непарног степена[1].

Граф који може бити нацртан под наведеним условима зове се **Ојлеров граф**, а одговарајући пут **Ојлеров пут**.

И чувена Ојлерова формула која важи за раванске графове:

Ако је v број чворова датог графа, e број грана, и f број страна (затворених или отворених области оивичених гранама) тада важи следећа релација.

$$f + v - e = 2$$

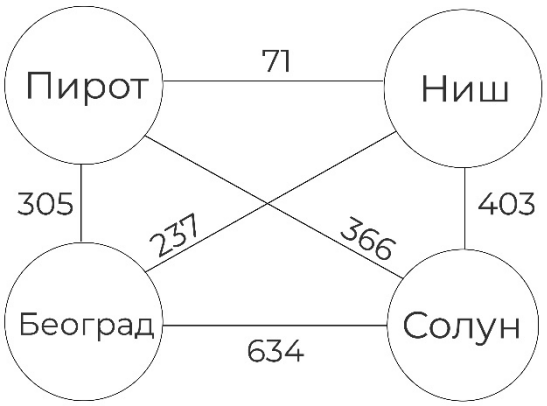
[1] Степен чвора објашњен у Терминологија графова, Дефиниција 6

Терминологија графова

Током овог рада биће приказани различити алгоритми и специфични проблеми везани за графове. Зато ево неких основних дефиниција.

Врсте графова

- **Неусмерени граф** *(енгл. Undirected или Bidirected)*
  У овим графовима једна грана која повезује чвор a са чвором b, такође повезује чвор b са чвором a. Дакле може се ићи из a у b и из b у a.
  Пример: граф на слици 1 је неусмерени.
- **Усмерени граф** *(енгл. Directed)*
  Овде гране имају смер. Ако једна грана повезује чвор a са чвором b, онда се може ићи из a у b, али не и супротно. Гране код ових графова су приказане стрелицама које приказују смер.
  Пример: граф на слици 2 је усмерени.
- **Тежински граф** *(енгл. Weighted)*
  Код ове врсте графова, грани је придружен неки реалан број, односно вредност. Обично су приказани као дужина ивице. На пример, растојања између градова:



Слика 5 Тежински граф, где градови представљају чворове, а њихова растојања (у км) гране.

У неким случајевима се могу користити и комбинације горе наведених графова. На пример усмерени тежински граф.

Даље у овом раду, број чворова биће представљен словом $n(N)$, а број грана словом $m(M)$.

Важније дефиниције

Дефиниција 3 Пут (енгл. Path) је низ ивица или чворова који воде од почетног до одредишног чвора.

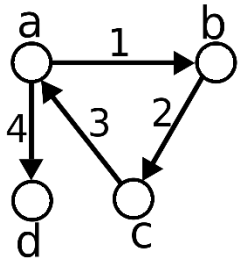
На слици 6 један од путева је: $a \to b \to c$

Дефиниција 4 Циклус (енгл. Cycle) је пут где је одредишни чвор исти као почетни.

На слици 6 циклус је: $a \to b \to c \to a$

Дефиниција 5 Хамилтонов пут је пут који пролази кроз све чворове у графу само једном.

На слици 6 Хамилтонов пут је: $b \to c \to a \to d$



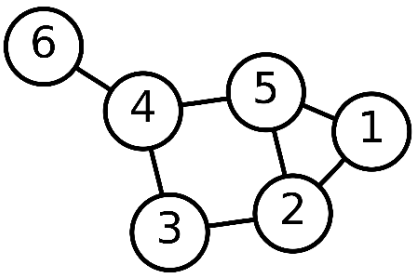
Слика 6 Усмерени граф

Дефиниција 6 Степен чвора (енгл. Node degree) је број веза или суседа том чвору. Степен чвора $\boldsymbol{v}$ означава се са $\boldsymbol{deg(v)}$. Два чвора су суседна ако су повезана граном.

Граф на слици 7 има следеће степене:

Чвор	$\boldsymbol{deg(v)}$
1	2
2	3
3	2
4	3
5	3
6	1

Табела 1 Приказ степена сваког чвора на слици 7



Слика 7 Неусмерени граф са 6 чвора и 7 грана

Напомена: Код усмерених графова, постоји улазни и излазни степен. Улазни степен означава број грана које се завршавају у том чвору, док излазни степен означава број грана који почињу из тог чвора.

Дефиниција 7 Компонента неусмереног графа је подграф у ком су било која два чвора међусобно повезана путевима, и који није повезан са додатним чворовима у суперграфу.

Граф на слици 8 има 3 компоненте.

Дефиниција 8 Петља (енгл. Loop) је ивица која повезује чвор са самим собом, тј. Ивица којој је почетни и крајњи чвор исти.

Дефиниција 9 Прост граф је граф без петљи.

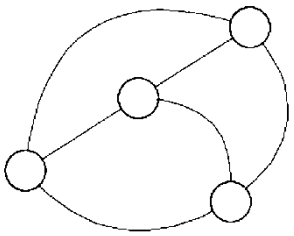
Дефиниција 10 Повезан граф (енгл. Connected) је граф где за свака два чвора постоји пут који их повезује.

Дефиниција 11 Комплетан или потпун граф са N чворова је неусмерени прост граф који има N чворова и свака два су повезана.

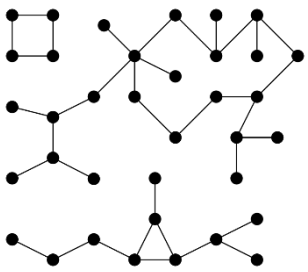
Граф на слици 9 је комплетан.

Дефиниција 12 Стабло је повезани граф без циклуса. Стабло са N чворова има тачно $N-1$ грану.

Дефиниција 13 Шума је граф са више стабала.



Слика 9 Комплетан или потпун граф



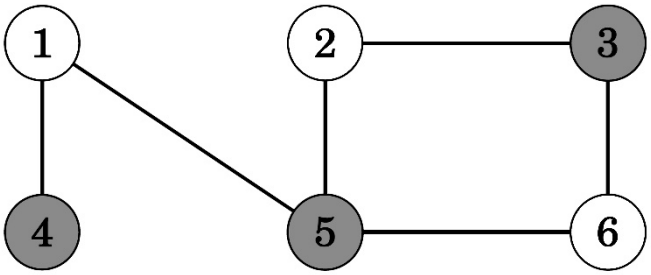
Слика 8 Неусмерени граф са 3 компоненте

Бојење графа

Код бојења графа, сваком чвору се додељује боја тако да ни један суседни чвор нема исту боју.

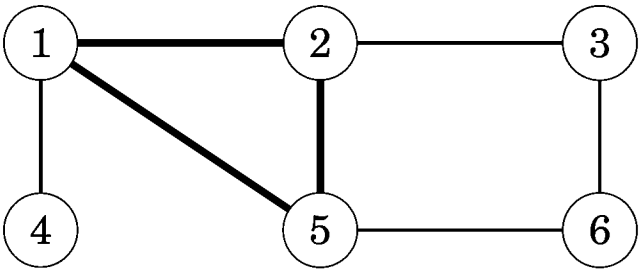
Дефиниција 14 Бипартитни граф је граф којег је могуће обојити користећи две боје.

Такође, испада да је граф бипартитни онда када не садржи циклусе са непарним бројем грана. Граф на слици 10 је бипартитни.



Слика 10 Бипартитни граф

Насупрот њему, граф на слици 11 није бипартитни јер постоји циклус са непарним бројем грана (циклус 1-2-5-1).



Слика 11 Граф који није бипартитни

Представљање графова

Постоје више начина да се граф представи у алгоритмима. Избор структуре података зависи од величине графа и начина на који се решава задатак.

Два најчешћа представљања графа су:

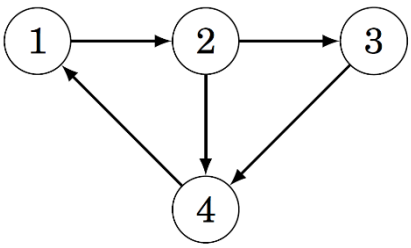
- **Матрица повезаности** (енгл. *Adjacency Matrix*)
- **Листа повезаности** (енгл. *Adjacency List*)

Матрица повезаности (енгл. *Adjacency Matrix*)

Матрица повезаности је дводимензионални низ који показује које ивице граф садржи. Из овакве матрице се може ефикасно проверити да ли постоји ивица између два чвора. Матрица је квадратна, са $\boldsymbol{N}$ колона и $\boldsymbol{N}$ редова, где је $\boldsymbol{N}$ број чворова у графу.

```
int N = 4;
int adj[N][N];
```

Свака вредност $\boldsymbol{adj[a][b]}$ показује да ли граф садржи ивицу која повезује чвор $\boldsymbol{a}$ са чвором $\boldsymbol{b}$. Ако је ивица садржана у графу, онда је $\boldsymbol{adj[a][b] = 1}$, иначе је $\boldsymbol{adj[a][b] = 0}$. На пример, граф



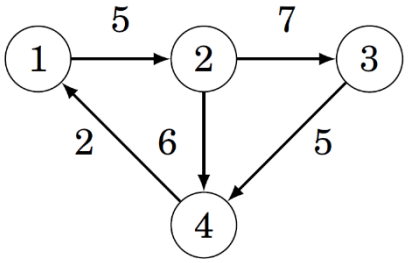
Слика 12 Усмерени граф са 4 чвора, $\boldsymbol{N = 4}$

може бити представљен следећом матрицом:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Табела 2 Матрица повезаности за граф са слици 12

Може се десити да је граф притом и тежински. Матрица повезаности може бити проширена тако да, ако постоји ивица, вредност на том пољу биће њена тежина, а иначе нула. На пример, ако се графу на слици 12 ивицама додају тежине, он постаје овакав:



Слика 13 Усмерени тежински граф са 4 чвора, $\boldsymbol{N = 4}$

Његова матрица повезаности је сада:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Табела 3 Матрица повезаности за граф са слици 13

Предност оваквог представљања је лако додавање, брисање и приступ ивицама у временској сложености $\boldsymbol{O(1)}$. Међутим, матрица повезаности може да прође ако је граф (скоро) потпун (као на слици 8), а иначе захтева $\boldsymbol{O(N^2)}$ простора да прикаже свега пар ивица.

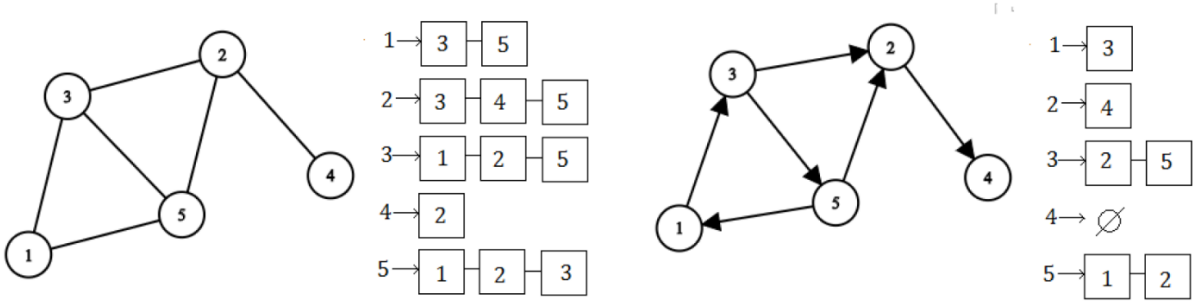
Ово се може побољшати коришћењем $bool$ матрице која ће користити само $1b$ уместо $4B$ меморије по пољу.

```
bool adj[N][N];
```

Међутим, таква матрица се може користити само код нетежинских графова. Због тога, бољи избор би била листа повезаности (суседства).

Листа повезаности (енгл. *Adjacency List*)

Граф се може представити набрајањем суседа сваког чвора, односно за сваки чвор се чува списак његових суседа. На слици 14 приказане су листе повезаности за неусмерени и усмерени граф.



Слика 14 Листе повезаности неусмереног и усмереног графа

Најлакше се имплементира коришћењем структуре вектор (*vector STL C++)*. Пошто нам за сваки чвор треба вектор користићемо низ вектора.

```
int N = 6;
vector<int> adj[N];
```

Листа повезаности за усмерени граф на слици 14 би била:

```
adj[1].push_back(3);
adj[2].push_back(4);
adj[3].push_back(2);
adj[3].push_back(5);
adj[5].push_back(1);
adj[5].push_back(2);
```

Ако је граф неусмерени, може се имплементирати тако што се свака грана дода у оба смера.

За тежинске графове листа повезаности чвора $\boldsymbol{a}$ поред информације суседног чвора, садржи и вредност гране. За овакво представљање потребна је структура пар (*pair STL C++)*.

```
vector<pair<int, int>> adj[N];
```

Предност листе повезаности графа је мала меморијска сложеност $\boldsymbol{O(N+M)}$, док је мана што у сложености $\boldsymbol{O(deg(a))}$ добијамо информацију о постојању гране између два чвора.

Алгоритми претраге графа

Претрага графа представља обилазак свих чворова графа. Чворови се обилазе по одређеном правилу у зависности од ког разликујемо више типова претраге. Два основна алгоритма за обилазак графа су:

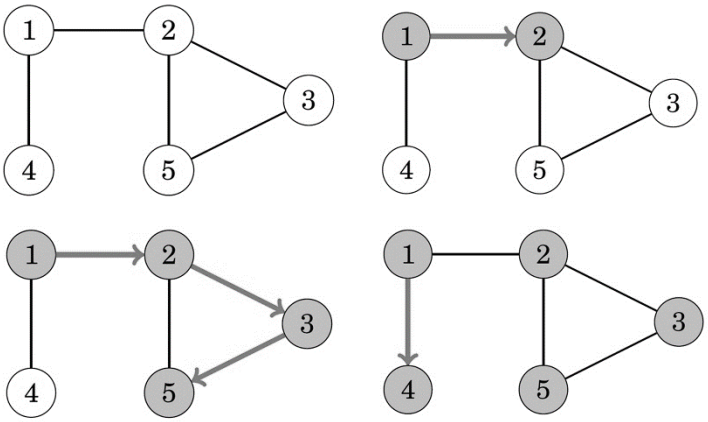
- **претрага у дубину** (ДФС, енгл. *DFS*; *Depth-First-Search*)
- **претрага у ширину** (БФС, енгл. *BFS*; *Breadth-First-Search*)

Једина разлика између ова два алгоритма јесте редослед посећивања чворова.

Претрага у дубину (*Depth-First-Search*)

Као скица за разумевање овог алгоритма претраге може послужити тражење излаза из лавиринта. Напредује се даље остављајући траг све док има где да се креће, а онда се враћа назад.

Алгоритам креће од полазног чвора $\boldsymbol{v}$, обележава да је посећен и за њега позива функцију за обилазак. У функцији за обилазак произвољно се бира један од чворова суседних са $\boldsymbol{v}$ који још није посећен, рецимо $\boldsymbol{u}$. Чвор $\boldsymbol{u}$ се даље маркира као посећен и за њега се рекурзивно позива функција за обилазак. Овај поступак се понавља све док сви суседи чвора $\boldsymbol{v}$ нису посећени.



Слика 15 Корак по корак напредовање ДФС алгоритма

Временска сложеност ДФС алгоритма је $\boldsymbol{O(N+M)}$, зато што је сваки чвор и свака грана обиђена тачно једном.

Имплементација:

Рекурзија:

По својој природи ДФС је рекурзиван. Ово је једноставнији начин. Како ДФС прати посећеност чворова, поред листе повезаности, требаће нам и низ ***visited***, који је иницијално на ***false***:

```
bool visited[N];
```

Претпостављајући да су листа повезаности, и низ посећених чворова глобалне променљиве, ДФС функција изгледа овако:

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // obrada cvora s, npr. ispis
    cout << s << endl;
    for (auto u : adj[s]) {
        dfs(u);
    }
}
```

Стек:

Друга имплементација је итеративна. Зато се уводи структура стек (*stack STL C++).* Алгоритам функционише тако што се почетни чвор „баци“ на врх стека, а потом се обрађује чвор са врха, упоредо додајући његове суседе на врх. Овај процес се понавља док се стек не испразни. Низ посећених чворова је исти као у прошлој имплементацији, декларисан као глобална променљива.

```
void dfsStack(int s) {
    stack<int> stack;
    stack.push(s);
    while (!stack.empty()) {
        int s = stack.top();
        stack.pop();
        if (!visited[s]) {
            // obrada cvora s
            cout << s << " ";
            visited[s] = true;
        }
        for (auto u : adj[s])
            if (!visited[u])
                stack.push(u);
    } }
```

Задатак 1 Избројити компоненте графа

Задатак се решава вишеструким позивањем ДФС функције. Пролази се кроз све чворове и сваку пут када се наиђе на чвор који није посећен, започиње се ДФС обилазак. После сваке обраде ДФС-а, бројач се повећа за један.

За представљање графа употребљена је листа повезаности и рекурзивна имплементација ДФС алгоритма. Чворови су нумерисани од 0.

Временска сложеност је $\boldsymbol{O(N+M)}$.

```
1   #include <iostream>
2   #include <vector>
3   using namespace std;
4   #define N 4
5
6   vector<int> adj[N];
7   bool visited[N];
8
9   void dfs(int s) {                // rekurzivna
10      if (visited[s]) return;      // implementacija dfs-a
11      visited[s] = true;
12      for (auto u : adj[s])
13          dfs(u); }
14
15  int count_components() {         // funkcija za brojanje
16      int num_of_comp = 0;         // vraca celobrojnu vrednost
17      for (int i = 0; i < N; ++i)
18          if (!visited[i]) {
19              dfs(i);
20              ++num_of_comp;
21          }
22      return num_of_comp; }
23
24  int main() {
25      adj[0].push_back(1);
26      adj[1].push_back(0);
27      adj[2].push_back(3);
28      adj[3].push_back(2);
29      cout << count_components();
30      return 0;
31  }
```

Задатак 2 Пронаћи циклус у графу

Ако у графу постоји циклус он ће бити откривен када се у ДФС обиласку наиђе на грану која води од чвора $\boldsymbol{v}$ који се тренутно обрађује до неког већ посећеног чвора $\boldsymbol{u}$.

Дакле, функција $\boldsymbol{find\_cycles}$ је мало измењени ДФС алгоритам, који враћа тачну вредност када наиђе на неки већ посећени чвор, или у супротном наставља претрагу. У $\boldsymbol{main}$ функцији је једноставна провера помоћу $\boldsymbol{if}$ услова и испис резултата.

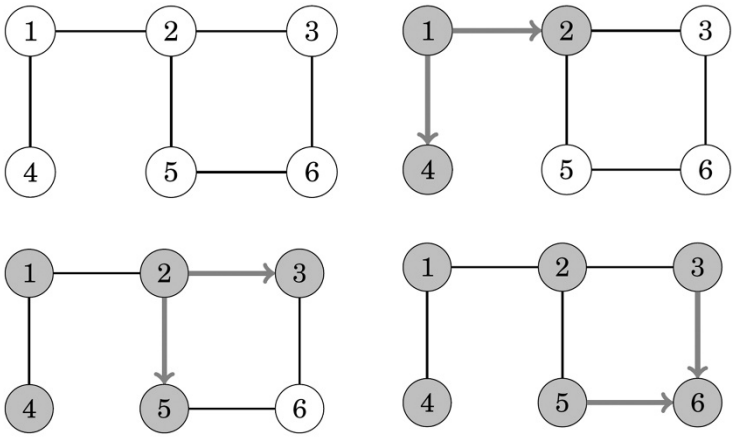
Као и у претходном задатку, граф је представљен листом повезаности и чворови су нумерисани од 0.

Временска сложеност је $\boldsymbol{O(N+M)}$

```cpp
#include <iostream>
#include <vector>
using namespace std;
#define N 5

vector<int> adj[N];
bool visited[N];

bool find_cycles(int s) {
    visited[s] = true;
    for (auto u : adj[s])
        if (visited[u]) return true;
        else return find_cycles(u);
    return false; }

int main() {
    adj[0] = {1};
    adj[1] = {2};
    adj[2] = {3, 4};
    adj[3] = {0};
    adj[4] = {};
    if (find_cycles(0)) cout << "Ciklus pronadjen.";
    else cout << "Nema ciklusa.";
    return 0; }
```

Претрага у ширину (*Breadth-First-Search*)

Претрагом у ширину граф се обилази ниво по ниво, то јест прво се обилазе чворови који су најближи полазном чвору, затим њима најближи... Полазећи од једног чвора најпре обилазимо све његове директне суседе који још нису посећени, а након тога настављамо од тих суседа обилазак на исти начин.



Слика 16 Корак по корак напредовање БФС алгоритма

Временска сложеност БФС алгоритма је такође $\boldsymbol{O(N+M)}$.

Имплементација:

За разлику од ДФС-а, БФС алгоритам није по природи рекурзиван. Зато се уводи структура ред (*queue STL C++).* Претрага започиње из полазног чвора који се обележава као посећен и ставља у ред. Даље се избацује из реда, обрађује се и пролази кроз све његове суседе који се додају у ред. Овај процес се понавља све док се ред не испразни.

```
void bfs(int s) {
    list<int> queue;
    visited[s] = true;
    queue.push_back(s);
    while(!queue.empty()) {
        s = queue.front();
        // obrada cvora s
        cout << s << " ";
        queue.pop_front();
        for (auto adjecent: adj[s])
            if (!visited[adjecent]) {
                visited[adjecent] = true;
                queue.push_back(adjecent);
            } } }
```

Задатак 3 Испитати да ли је граф бипартитни (за повезани граф)

На почетку, сви чворови се означе као необојени (у коду са -1), почетном се додели прва боја (1), а свим његовим суседима који немају боју додели се друга (0). Ако боја тренутног чвора и боја његовог суседа буду исте, програм се прекида и враћа нетачно. Ако овај измењени БФС алгоритам прође кроз све чворове и свима додели одговарајућу боју, значи да је граф бипартитни.

Временска сложеност је $\boldsymbol{O(N+M)}$.

```
#include <bits/stdc++.h>
using namespace std;
#define N 5
vector<int> adj[N];

bool is_bipartite(int src) {
    int colorArr[N];
    for (int i = 0; i < N; ++i) colorArr[i] = -1;
    colorArr[src] = 1; // početni čvor - prva boja

    queue<int> q;
    q.push(src);
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        // za sve susede trenutnog cvora dodeliti
        // suprotnu boju ili prekinuti pretragu
        for (auto v : adj[u])
            if (colorArr[v] == -1) {
                colorArr[v] = 1 - colorArr[u];
                q.push(v); }
            else if (colorArr[v] == colorArr[u])
                return false; }
    return true; }
int main() {
    adj[0] = {1, 3}; adj[1] = {2};
    adj[2] = {3, 4}; adj[3] = {0}; adj[4] = {};
    if (is_bipartite(0)) cout << "Bipartitni je.";
    else cout << "Nije bipartitni";
    return 0; }
```

Тополошко сортирање

Тополошко сортирање је још једна употреба ДФС алгоритма.

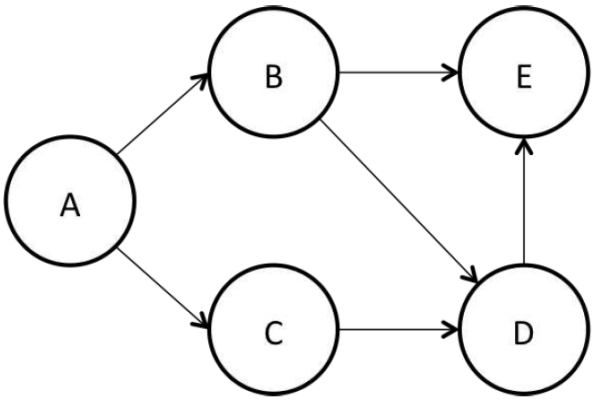
Нека постоји неки скуп послова које треба завршити у одређеном редоследу, односно да неки послови могу да се започну тек када се заврше неки други. На пример, четири посла: $\boldsymbol{A, B, C, D}$ тако да посао $\boldsymbol{A}$ мора бити завршен пре послова $\boldsymbol{C}$ и $\boldsymbol{D}$, и посао $\boldsymbol{D}$ мора бити завршен пре посла $\boldsymbol{B}$. Проблем је пронаћи редослед извршавања послова тако да важе ова правила. Једно решење је: $\boldsymbol{ADBC}$**.** Ово значи да ће се прво завршити посао $\boldsymbol{A}$, онда $\boldsymbol{D}$, затим $\boldsymbol{B}$, и на крају посао $\boldsymbol{C}$.

Послови се могу представити усмереним графом без циклуса. У случају да посао $\boldsymbol{A}$ долази пре посла $\boldsymbol{B}$, постоји ивица која иде из $\boldsymbol{A}$ у $\boldsymbol{B}$.

Алгоритам заснован на ДФС-у

Најједноставнија имплементација би био алгоритам ДФС-а. Решење је добијено додавањем чворова на стек по њиховом завршетку. Како је граф усмерени, да неки чвор не би промакао, ДФС се мора позвати за сваки непосећени чвор. Када се сви чворови додају на стек, једноставним читањем чворова са врха стека добијамо тражени редослед.

На слици 17 један од могућих ДФС обиласка је: $\boldsymbol{ACDEB}$, али ако се сваки чвор дода на стек онда када су сви његови суседи посећени, обилазак би изгледао: $\boldsymbol{ABCDE}$, што је и решење Тополошког сортирања.



*Слика 17 Усмерени граф који представља редослед извршавања послова. Посао **A** пре послова **B** и **C**. Посао **D** после **B** и **C**, и посао **E** после **B** и **D**.*

Због једноставности, чворови $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{E}$ су замењени класичним нумерисањем од 0.

Граф је представљен листом повезаности. ДФС алгоритам је имплементиран рекурзивно, а временска сложеност алгоритма је $\boldsymbol{O(N+M)}$.

```
#include <bits/stdc++.h>
using namespace std;
#define N 5
stack<int> S;
vector<int> adj[N];
bool visited[N];

void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    for (auto u : adj[s]) dfs(u);
    S.push(s); }

void topological_sort() {
    for (int i = 0; i < N; ++i)
        if (!visited[i]) dfs(i); }

int main() {
    adj[0] = {1, 2};
    adj[1] = {3, 4};
    adj[2] = {3};
    adj[3] = {4};
    adj[4] = {};
    topological_sort(); // poziv funkcije
    while (!S.empty()) { // ispis redosleda
        cout << S.top() << " ";
        S.pop(); }
    return 0; }
```

Канов алгоритам

Још једно занимљиво решење представио је Кан које ради са улазним степенима свих чворова (види Дефиницију 6). Посматрајмо опет граф на слици 17.

Прво, у сложености $\boldsymbol{O(N+M)}$ се израчунају улазни степени свих чворова. Низ $\boldsymbol{in\_degree}$ би после ове операције изгледао:

Чворови	A(0)	B(1)	C(2)	D(3)	E(4)
Улазни степен	0	1	1	2	2

Табела 4 Приказ улазних степена чворова на графу са слике 17

Затим, чворови са улазним степеном 0, додају се у ред. Чворови са почетка реда се додају низу $\boldsymbol{top\_order}$, где се акумулира решење, а улазне степене суседима тренутног чвора смањујемо за 1. Када они постану 0, они се додају на почетак реда и процес се понавља.

Овај алгоритам може се представити следећом функцијом:

```
void topological_sort_khan() {
    vector<int> in_degree(N, 0);
    // ulazni stepen cvorova u O(N+M)
    for (int u = 0; u < N; ++u)
        for (auto v : adj[u])
            in_degree[v]++;

    // dodavanje cvorova sa ulaznim stepenom 0
    queue<int> q;
    for (int u = 0; u < N; ++u)
        if (in_degree[u] == 0)
            q.push(u);

    // resenje se redja u nizu top_order
    vector<int> top_order;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        top_order.push_back(u);
        for (auto v : adj[u])
            if (--in_degree[v] == 0)
                q.push(v); }

    for (auto x : top_order) cout << x << " "; }
```

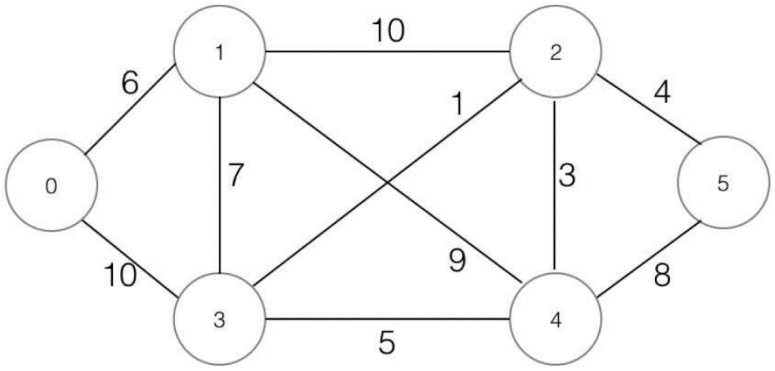
Граф је представљен листом суседства, а чворови су нумерисани од 0.

Алгоритми најкраћег пута

Проналажење најкраћег пута између два чвора графа је веома важан проблем који има пуно практичних употреба. На пример, проналажење најкраћег пута између два града ако су познате дужине путева.

У нетежинском графу, дужина пута једнака је броју грана које морају да се пређу. Овде се проблем решава једноставном БФС претрагом. Међутим, занимљивији су тежински графови. Алгоритми у наставку управо су везани за проналажење најкраћег пута у тежинском графу.

Дакле, пут је скуп ивица од чвора $\boldsymbol{v}$ до чвора $\boldsymbol{u}$, и вредност (дужина) пута је сума вредности свих ивица. Најкраћи пут је зато пут са најмањом сумом који води од једног до другог чвора. На слици 18 најкраћи пут од чвора $\mathbf{0}$ до чвора $\mathbf{5}$ је 15, и путања је $\mathbf{0-3-2-5}$.



Слика 18 Тежински граф над којим је објашњен Дајкстрин алгоритам најкраћег пута

Дајкстрин алгоритам

Овај алгоритам проналази најкраћи пут између почетног чвора и свих осталих чворова. Због свог похлепног (енгл. *Greedy*) начина, ради само **за тежинске графове са позитивним вредностима (тежинама)**.

Први пут је представљен од стране холандског научника Едсхера Дајкстре (енгл. *Edsger W. Dijkstra*), 1959. године.

Нека је дат граф $\boldsymbol{G}$ са $\boldsymbol{n}$ чворова нумерисаних од $\mathbf{0}$ до $\boldsymbol{n}-\mathbf{1}$ где је чвор $\boldsymbol{x}$ почетни чвор.

Први корак је иницијализација низа $\boldsymbol{D}$, који чува растојања свих чворова од почетног и низа $\boldsymbol{V}$ који прати посећеност чворова:

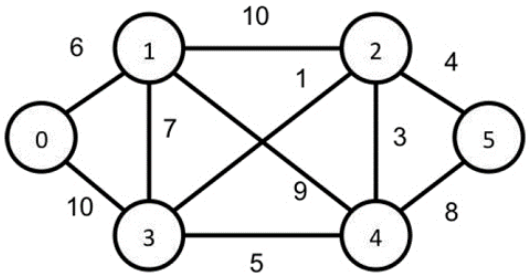
$$V_i = false, D_i = \begin{cases} \infty & i \neq x \\ 0 & i = x \end{cases}$$

за свако $\boldsymbol{i}$ од $\mathbf{0}$ до $\boldsymbol{n} - \mathbf{1}$.

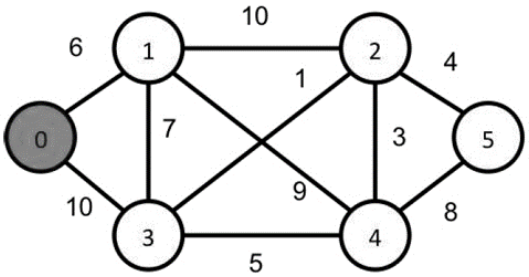
Када је $\boldsymbol{V_i} = \boldsymbol{true}$, вредност смештена у $\boldsymbol{D_i}$ садржи минималну вредност за пут од чвора $\boldsymbol{x}$ до чвора $\boldsymbol{i}$. Алгоритам даље проналази чвор $\boldsymbol{a}$, где је $\boldsymbol{D_a}$ минимум низа и $\boldsymbol{V_a} = \boldsymbol{false}$ што значи да чвор још није посећен. Означава га као посећен и ажурира сва растојања до његових суседа:

$$D_b = \min(D_b,\ D_a + w_b) \text{ где је } V_b = false$$

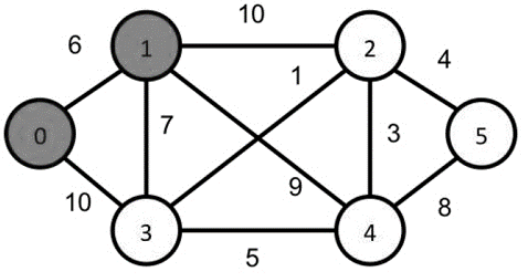
Овај процес се понавља све док сви чворови не буду посећени или следећи чвор није могуће пронаћи (граф није повезани, *Дефиниција 10*). Следећа илустрација показује итерације алгоритма упоредо са вредностима низова $\boldsymbol{D}$ и $\boldsymbol{V}$.



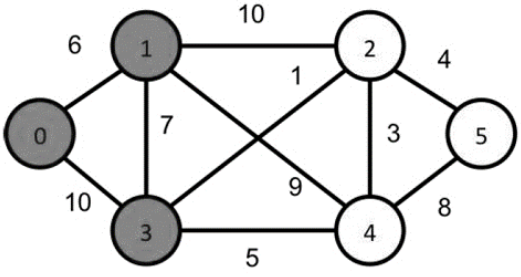
	0	1	2	3	4	5
D =	0	INF	INF	INF	INF	INF
V =	0	0	0	0	0	0



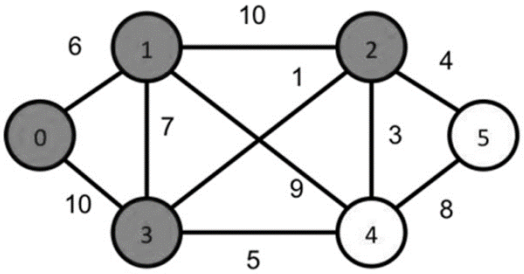
	0	1	2	3	4	5
D =	0	6	INF	10	INF	INF
V =	1	0	0	0	0	0



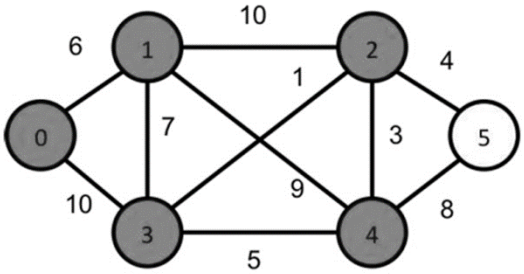
	0	1	2	3	4	5
D =	0	6	16	10	15	INF
V =	1	1	0	0	0	0



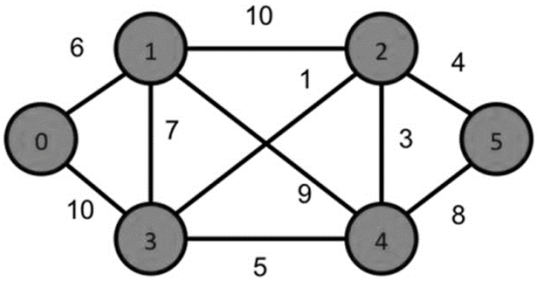
	0	1	2	3	4	5
D =	0	6	11	10	15	INF
V =	1	1	0	1	0	0



	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	0	0



	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	1	0



	0	1	2	3	4	5
D =	0	6	11	10	14	15
V =	1	1	1	1	1	1

Слика 19 Илустроване итерације Дајкстриног алгоритма са чвором $\mathbf{0}$ *као почетним*

Имплементација:

Први начин:

Овај начин је познатији као „Дајкстра без хипа“. Једноставнији је за имплементацију, међутим његова временска сложеност је релативно велика. Свако бирање следећег чвора за обраду захтева пролазак кроз све чворове и проналазак оног са најмањим растојањем од почетног. Његова временска сложеност је $\boldsymbol{O(N^2)}$.

Други начин:

Овај начин је пак познатији као „Дајкстра са хипом“ и захтева коришћење неке напредније структуре података. Ефикасност алгоритма лежи у ефикасном проналажењу чвора са најмањим растојањем од почетног који још није посећен. Адекватна структура за то је ред са

приоритетом (*структура queue STL C++*). Овај ред би садржао чворове сортиране по њиховим растојањима од почетног чвора. Оваква структура омогућава да се следећи чвор за обраду пронађе у логаритамском времену. Следећа функција на крају алгоритма исписује растојања свих чворова од почетног:

```
void dijkstra(int x) {
    vector<int> D(N, INF);
    vector<bool> V(N, false);
    D[x] = 0;

    queue<pair<int, int>> q;
    q.push({0, x});
    while (!q.empty()) {
        int a = q.front().second; q.pop();
        if (V[a]) continue;
        V[a] = true;

        // prolazak kroz sve susede
        // i azuriranje rastojanja
        for (auto u : adj[a]) {
            int b = u.first, w = u.second;
            if (D[a]+w < D[b]) {
                D[b] = D[a]+w;
                q.push({-D[b], b});
            }
        }
    }
    for (auto distances : D)
        cout << distances << " "; }
```

Ред садржи пар у форми $(-\boldsymbol{d}, \boldsymbol{x})$, који означава да је тренутно растојање до чвора $\boldsymbol{x}$ једнако са $\boldsymbol{d}$. Такође, важно је приметити да се у ред додају негативне вредности за растојање. Разлог за то је природа структуре реда у програмском језику *C++*, где највеће вредности одлазе на почетак реда, а нама требају најмање.

Временска сложеност алгоритма је $\boldsymbol{O(N + MlogM)}$.

Примена Дајкстирног алгоритма је велика. Од Гугл мапа (енгл. *Google Maps)*, за проналажење најкраћег пута између два места, преко друштвених мрежа где служи за предлагање пријатељстава, затим у телефонским мрежама, где се помоћу овог алгоритма одређује најкраћи пут протока информације, до прављења распореда летова на аеродромима.

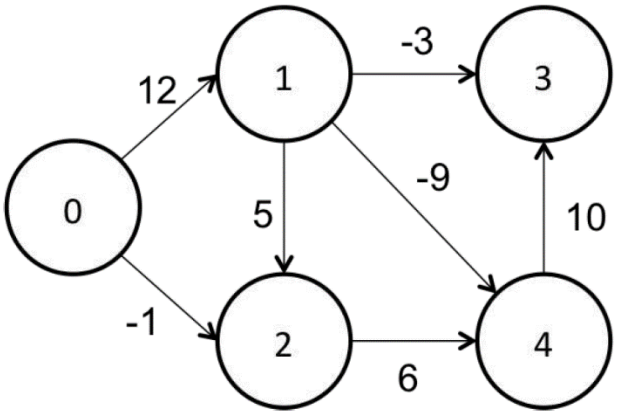
Белман-Фордов алгоритам

Проблем Дајкстре јавља се код тежинских графова са негативним вредностима (тежинама). Белман-Фордов алгоритам, представљен у одвојеним радовима Ричарда Белмана (енгл. *Richard Bellman*) и Лестера Форда (енгл. *Lester Ford*), **способан је да ради исправно и са негативним вредностима**.

Користи се низ $\boldsymbol{D}$ где $\boldsymbol{D_i}$ представља најкраћи пут, то јест код овог алгоритма је правилније рећи - најмању вредност за пут од почетног чвора до чвора $\boldsymbol{i}$, јер пут може бити негативан. Алгоритам иде кроз све ивице графа и ако постоји ивица која повезује чвор $\boldsymbol{a}$ са чвором $\boldsymbol{b}$ и има вредност $\boldsymbol{w}$, низ $\boldsymbol{D}$ се ажурира на следећи начин:

$$D_b = \min\left(D_b, D_a + w\right)$$

Ова једначина је познатија као **„скраћивање ивица“**. Процес се понавља $\boldsymbol{n}$ пута за сваку ивицу.



Слика 20 Усмерени тежински граф са негативним вредностима (тежинама)

Најпре, иницијализација низа $\boldsymbol{D}$:

$$D_i = \begin{cases} \infty & i \neq 0 \\ 0 & i = 0 \end{cases}$$

Следећи корак је „скраћивање ивица“, који за граф на слици 20 изгледа овако:

$$\begin{aligned}
1 \to 4 \quad & D_4 = \min(\infty, \infty - 9) = \infty - 9 = \infty \\
0 \to 1 \quad & D_1 = \min(\infty, 0 + 12) = 12 \\
1 \to 3 \quad & D_3 = \min(\infty, 12 - 3) = 9 \\
2 \to 4 \quad & D_4 = \min(\infty, \infty + 6) = \infty \\
0 \to 2 \quad & D_2 = \min(\infty, 0 - 1) = -1 \\
4 \to 3 \quad & D_3 = \min(9, \infty + 10) = 9 \\
1 \to 2 \quad & D_2 = \min(-1, 12 + 5) = -1
\end{aligned}$$

Низ $\boldsymbol{D}$ сада изгледа овако:

0	12	-1	9	∞

Да би сви чворови били обрађени, овај поступак се понавља бар $\boldsymbol{n}$ пута, па је крајњи изглед низа $\boldsymbol{D}$:

0	12	-1	9	3

Имплементација:

За имплементацију овог алгоритма биће потребан један начин представљања графа који није употребљаван у досадашњем раду – **низ ивица.** Низ се садржи из торки $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{w})$, што означава да постоји ивица између чворова $\boldsymbol{a}$ и $\boldsymbol{b}$, и њена вредност је $\boldsymbol{w}$. У програмском језику *C++* ово се може представити структуром тројки (структура *tuple STL C++).*

```
vector<tuple<int, int, int>> edges;
```

```
void bellman_ford(int x) {
    vector<int> D(N, INF);
    D[x] = 0;
    for (int i = 0; i < N; ++i) {
        for (auto e : edges) {
            int a, b, w;
            tie(a, b, w) = e;
            D[b] = min(D[b], D[a]+w);
        }
    }
    for (auto distances : D)
        cout << distances << " "; }
```

Временска сложеност алгоритма је $\boldsymbol{O(NM)}$.

Флојд-Воршалов алгоритам

Представљен од стране Роберта Флојда (енгл. *Robert Floyd*), базиран на раду Стивена Воршала (енгл. *Stephen Warshall*), овај алгоритам се користи за **проналазак најкраћег пута између свих ивица**, чије су вредности представљене матрицом повезаности. Алгоритам ради и за позитивне и за негативне вредности (тежина).

Идеја иза Флојд-Воршаловог алгоритма је да се константно ажурирају најмање вредности такве да се из чвора $\boldsymbol{i}$ оде у чвор $\boldsymbol{j}$ преко чвора $\boldsymbol{k}$.

```
void floyd_warshall() {
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++) {
                D[i][j] = min(D[i][j],
                D[i][k]+D[k][j]);
            } }
```

Временска сложеност алгоритма је $\boldsymbol{O(N^3)}$.

Следећа илустрација показује како се матрица $\boldsymbol{D}$ у коду, мења за сваку вредност индекса $\boldsymbol{k}$, користећи граф на слици 20.

0	12	-1		
	0	5	-3	-9
		0		6
			0	
			10	0

Почетно стање и $k = 0$

0	12	-1	**9**	**3**
	0	5	-3	-9
		0		6
			0	
			10	0

$k = 1, k = 2, k = 3$

0	12	-1	9	3
	0	5	-3	-9
		0	**16**	6
			0	
			10	0

$k = 4$

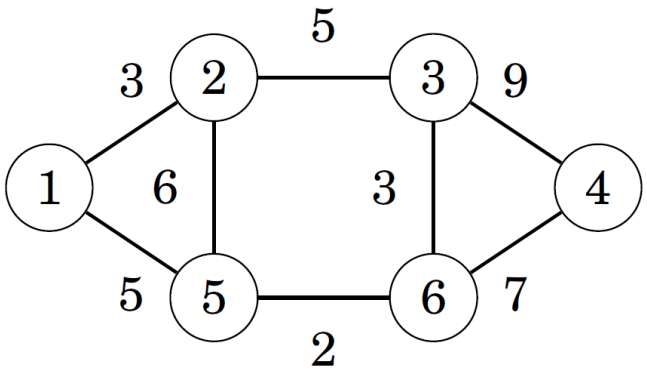
Празна поља означавају ∞ вредност.

Предност овог алгоритма је лака имплементација, али због своје временске сложености, **препоручује се да се овај алгоритам користи код графова са веома малим бројем ивица.**

Минимално разапињуће стабло

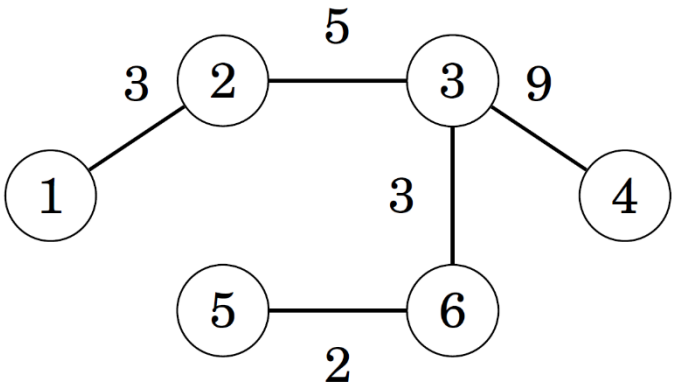
Разапињуће стабло (енгл. *Spanning Tree*) графа се састоји од свих чворова и неких (или можда чак и свих) грана тако да постоји пут између било која два чвора. За овакво стабло важе све особине као и за обично стабло (види *Дефиницију 12*).

На пример, за следећи граф:



Слика 21 Неусмерени тежински граф

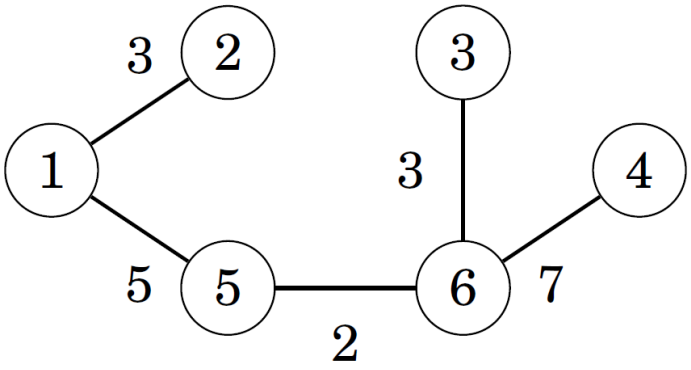
једно разапињуће стабло је:



Слика 22 Разапињуће стабло графа на слици 21

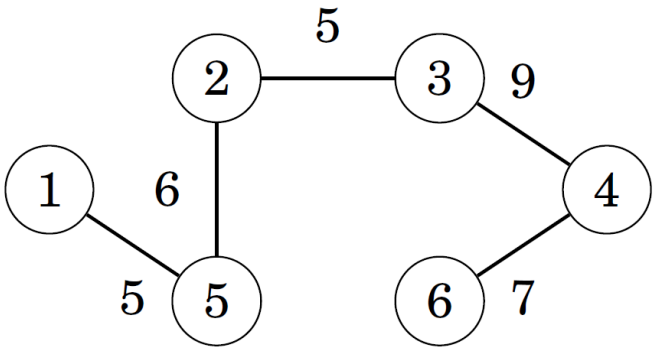
Вредност разапињућег стабла је збир свих његових грана. За стабло горе ова вредност је $\mathbf{3 + 5 + 9 + 3 + 2 = 22}$.

Минимално разапињуће стабло (енгл. *Minimum Spanning Tree; MST*) је оно разапињуће стабло чија је вредност најмања могућа. Вредност таквог стабла за граф на слици 21 је **20** и изгледа овако:



Слика 23 Минимално разапињуће стабло графа на слици 21

На сличан начин, **максимално разапињуће стабло (енгл. *Maximum Spanning Tree*) је оно разапињуће стабло чије је вредност највећа могућа.** За граф на слици 21, његова вредност је **32** и изгледа овако:



Слика 24 Максимално разапињуће стабло графа на слици 21

За проналазак оваквих стабала могу се користити неколико похлепних (енгл. *Greedy)* метода, које обрађују ивице графа, сортиране по њиховим вредностима (тежинама).

Два најпознатија алгоритма су **Примов** (енгл. *Robert Clay Prim)* и **Краскалов** (енгл. *Joseph Bernard Kruskal*) алгоритам.

Примов алгоритам

Чешки математичар Војтех Јарник (*Vojtěch Jarník*) је 1930. године смислио овај алгоритам, а касније су до њега независно дошли Роберт Прим и Едсхер Дајкстра. Најчешће се назива Примов алгоритам.

Следећим поступком добија се минимално разапињуће стабло повезаног тежинског графа. Важно је напоменути да се на сличан начин долази до минималне разапињуће шуме тако што се описани алгоритам понови за сваку компоненту повезаности.

Опис:

1. Бира се произвољан чвор $\boldsymbol{v_1}$.

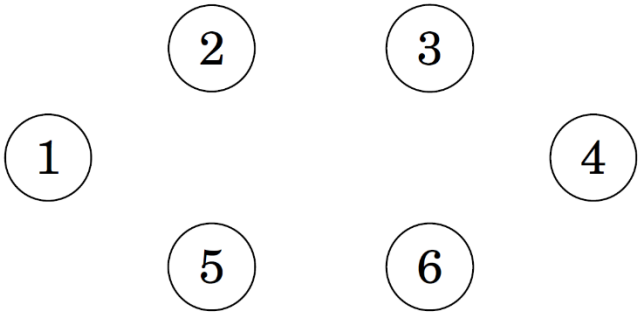
2. Од свих грана које полазе из чвора $\boldsymbol{v_1}$ бира се она која има најмању тежину.

3. Ако су досад изабрани чворови $\boldsymbol{v_1, v_2, \dots, v_{k+1}}$ и гране $\boldsymbol{e_1, e_2, \dots, e_k}$ онда се нова грана $\boldsymbol{e_{k+1}}$ која спаја чворове $\boldsymbol{v_i}$ и $\boldsymbol{v_j}$ бира на следећи начин: $\boldsymbol{v_i}$ припада скупу $\boldsymbol{\{v_1, v_2, \dots, v_{k+1}\}}$, а $\boldsymbol{v_j}$ не припада скупу $\boldsymbol{\{v_1, v_2, \dots, v_{k+1}\}}$ и $\boldsymbol{e_{k+1}}$ је минималне тежине од свих таквих грана.

4. Уколико је изабрано $\boldsymbol{n-1}$ грана алгоритам је завршен, у супротном понавља се корак 3.

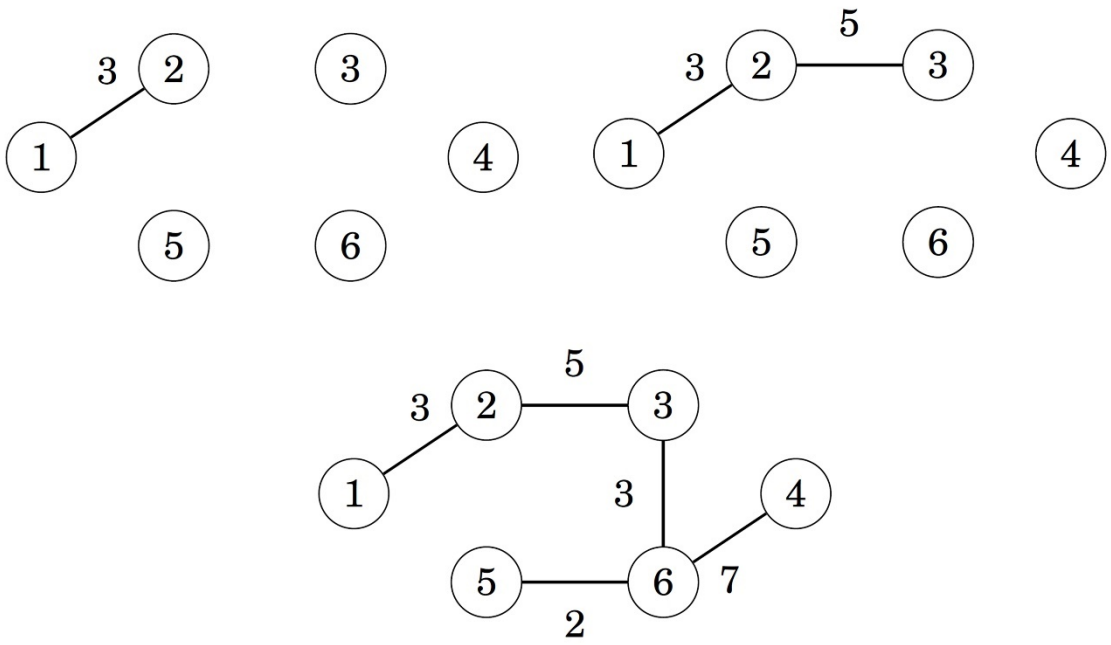
Примов алгоритам личи на Дајкстрин алгоритам. Разлика између њих је та да Дајкстрин алгоритам увек бира ивицу чије је растојање најмање од почетног чвора, а Примов алгоритам једноставно узима ивицу са најмањом вредношћу која додаје нов чвор стаблу.

На пример, како Примов алгоритам функционише на графу на слици 21. Иницијално, нема ивица између чворова:



Слика 25 Почетно стање Примовог алгоритма за граф на слици 21

Нека је почетни чвор **1**, итерације даље изгледају овако:



Слика 26 Итерације Примовог алгоритма

Имплементација:

Као и код Дајкстриног алгоритма, Примов алгоритам се може ефикасно имплементирати коришћењем реда са приоритетом. Такав ред би требало да одржава све чворове који се могу повезати са тренутном компонентом, користећи једну грану, у растућем редоследу по вредностима (тежинама) ивица.

```
void prim(int x) {
    vector<int> parent(N, -1);
    vector<int> key(N, INF);
    vector<bool> in_mst(N, false);
    queue<pair<int, int>> q;

    q.push({0, x});
    key[x] = 0;
    while (!q.empty()) {
        int a = q.front().second;
        int a_weight = q.front().first; q.pop();
        if (in_mst[a]) continue;
        in_mst[a] = true;
        for (auto u : adj[a]) {
            int b = u.first, w = u.second;
            if (in_mst[b] == false && w <= key[b]) {
                q.push({-w, b});
                parent[b] = a;
                key[b] = w; } }
    }
    for (int i = 1; i < N; ++i)
        cout << parent[i] << " - " << i << endl; }
```

На почетку се почетни чвор дода у ред и означи се да је у стаблу. Даље се разматрају све гране и редом додају у ред ако већ нису. Важно је приметити да као код Дајкстре, у ред стављамо негативне вредности. Пратећи родитеље чворова на крају је могућ испис свих грана која су у стаблу.

Временска сложеност алгоритма је као и код Дајкстре, $\boldsymbol{O(N + MlogM)}$.

Краскалов алгоритам

Творац овог алгоритма је Џозеф Краскал који је 1956. године објављен у једном његовом раду. За разлику од Примовог алгоритма, Краскалов алгоритам налази минималну разапињућу шуму графа уколико је он неповезан, односно стабло у случају повезаног графа.

Овај алгоритам сврстава се у похлепне алгоритме, јер у сваком кораку додаје грану минималне тежине, али такву да добијени граф нема циклуса.

Опис:

1. Нека сваки чвор у почетку представља засебно стабло. Сваки чвор се означи као представник стабла које садржи само њега. Скуп грана $\boldsymbol{E_1}$ је у почетку празан скуп.

2. Гране се сортирају неопадајуће по вредности (тежини).

3. Крећући се кроз сортирани низ, ако грана која се тренутно посматра (између чворова $\boldsymbol{v}$ и $\boldsymbol{u}$) повезује два различита подстабла, додаје се у скуп грана $\boldsymbol{E_1}$, а подстабла одређена чворовима $\boldsymbol{v}$ и $\boldsymbol{u}$ се спајају. У супротном се наставља даље.

4. Корак 3 се понавља док се не изабере $\boldsymbol{n-1}$ грана.

Скуп грана $\boldsymbol{E_1}$ и свих чворова у графу представља тражену минималну разапињућу шуму.

Имплементација:
На почетку сваки чвор представља корен свог стабла, односно важи даје за сваки чвор графа $\boldsymbol{parent}[\boldsymbol{v}] = [\boldsymbol{v}]$. Даље, проласком кроз све гране за два чвора проверава се да ли су у истом подстаблу. Ако нису, споје се, иначе се прелази на следећу грану. Ово се може имплементирати коришћењем листе ивица, у форми торки $(\boldsymbol{w, a, b})$.

```
void kruskal() {
    for (int i = 0; i < N; ++i) parent[i] = i;
    sort(edges.begin(), edges.end());
    int ans = 0;
    for (auto e : edges) {
        int a, b, w;
        tie(w, a, b) = e;
        if (find(a) != find(b)) {
            unite(a, b);
            ans += w;
            cout << a << " - " << b
            << "    " << w << endl; } }
    cout << "Cost is: " << ans;}
```

Алгоритам исписује редом гране које се садрже у минималном разапињућем стаблу и на крају исписује његову вредност.

Временска сложеност алгортима је $\boldsymbol{O(MlogM)}$.

Структура дисјунктних скупова (енгл. *Disjoint Sets*)

Како се у Краскаловом алгоритму користе структуре дисјунктних скупова, потребно их је објаснити понаособ.

Проблем који се поставља је следећи:

Посматра се скуп који се састоји од више дисјунктних делова односно подскупова таквих да је пресек свака два подскупа празан скуп.

Потребно је обезбедити ефикасно извршавање два типа операција:

1. $\boldsymbol{find(x)}$ - проналази подскуп којем припада конкретни елемент $\boldsymbol{x}$.
2. $\boldsymbol{unite(x,y)}$ – обједињује два подскупа задата са по једним елементом, први са $\boldsymbol{x}$, а други са $\boldsymbol{y}$.

$\boldsymbol{find(x)}$ - када је потребно наћи подскуп којем припада конкретан елемент иде се уназад преко родитеља све до корена стабла односно представника траженог подскупа. Имплементација може бити рекурзивна или итеративна.

$\boldsymbol{unite(x,y)}$ - ако је потребно спојити два подскупа на почетку се морају наћи њихови представници, а затим се са родитеља једног од два представника поставља други који постаје корен новонасталог стабла у коме су сада сви чворови из оба подскупа.

У Краскаловом алгоритму, наведене функције се могу имплементирати на следећи начин:

```
int find(int i) {
    while(parent[i] != i) {
        parent[i] = parent[parent[i]];
        i = parent[i]; }
    return i; }

void unite(int a, int b) {
    int p1 = find(a);
    int p2 = find(b);
    parent[p1] = parent[p2]; }
```

Или у случају рекурзивне имплементације функције $\boldsymbol{find(x)}$:

```
int find(int i) {
    if (parent[i] == -1)
        return i;
    return parent[i] = find(parent[i]);
}
```

Примене алгоритама за налажење минималног разапињућег стабла:

Дизајн мрежа – овај алгоритам налази примену у разним проблемима где је потребно наћи најоптималнију мрежу некаквих веза било да су у питању телефонске линије међу градовима, повезивање рачунара или чак мрежа путева у некој области.

На пример, ако је у некој компанији потребно повезати рачунаре кабловима и зна се цена повезивања за сваки пар рачунара онда се наведеним алгоритмом добија најисплативија мрежа каблова.

Закључак

Граф у математици, без рачунарства и програмирања не би имао никакву практичну вредност. Због тога су област теорије графова и алгоритми за рад са графовима веома битни у развоју технологије и науке уопште.

Гугл мапе, авионски летови, дизајн рачунарских и телефонских линија, друштвене мреже и многе друге ствари које свакодневно користимо су имплементиране и оптимизоване управо користећи ове алгоритме.

Циљ овог рада био је опис и имплементација основних и напреднијих алгоритама за рад са графовима.

Између осталог, рад може служити и као приручник такмичарима из програмирања. Ови алгоритми су срж решења тежих задатака, зато је битно знати их и имати их на једном месту у сваком тренутку.

Велико хвала мом ментору, професору Ненаду Костићу, због кога сам пре свега заволео програмирање, научио много тога и добио мотивацију за даљи рад.

Литература

1. David Esparza Alba, Juan Antonio Ruiz Leal: Algorithms for Competitive Programming, 2021.
2. Mark Allen Weiss: Data Structures and Algorithm Analysis in C++ Fourth Edition, 2014.
3. Antti Laaksonen: Competitive Programmer's Handbook, 2018.
4. Kenneth Rosen: Discrete Mathematics and Its Applications Seventh Edition, 2011.
5. Geek for geeks, https://www.geeksforgeeks.org/
6. cplusplus, https://www.cplusplus.com/
7. Stack Overflow, https://stackoverflow.com/
8. Petlja, https://petlja.org/

Датум предаје: _____

Чланови испитне комисије:

Председник _____

Испитивач _____

Члан _____

Коментар:

Оцена вредности матурског рада _____

Оцена одбране матурског рада _____

Закључна оцена _____

Датум одбране: _____