Київський національний університет імені Тараса Шевченка

Кафедра математичної інформатики

Тарануха В.Ю.

Задачі на графах для курсу Алгоритміка

Навчальний посібник

Київ-2021

УДК 004.021

Тарануха В.Ю. Задачі на графах для курсу Алгориміка: Навчальний посібник для студентів факультету комп'ютерних наук та кібернетики / В. Ю.Тарануха. –Київ: електронна публікація на сайті факультету, 2021. – 69 с.

Рецензенти:

Марченко О.О., доктор фіз.-мат. наук, професор, професор кафедри математичної інформатики Київського національного університету імені Тараса Шевченка

Глибовець М.М., доктор фіз.-мат. наук, професор, професор кафедри інформатики факультету інформатики Національного університету «Києво-Могилянська академія» .

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
12 квітня 2021 року
протокол № 13

Автор:

Тарануха В.Ю. кандидат кандидат фіз.-мат. наук, асистент кафедри математичної інформатики факультету комп'ютерних наук та кібернетики

У навчальному посібнику розглянуто задачі на графах для курсу Алгориміка. Розглянуто способи представлення графів, задачі на графах, алгоритми розв'язання задач та їх складність.

Призначений для студентів фізико-математичних та технічних спеціальностей вищих навчальних закладів.

Передмова

Алгоритміка це дисципліна, що вивчає алгоритми, та їх застосування до різних задач. Даний посібник орієнтований на групу алгоритмів що працюють з графами.

Представити граф, знайти в ньому шляхи шлях, розрахувати оптимальний маршрут – ці всі задачі є вкрай важливими з технічної точки зору, наприклад, для маршрутизації інтернет-трафіку. Більш того, штучний інтелект неможливий без засобів обробки графів, хоч би для знаходження шляху чи реалізації обходу дерева гри.

В даному посібнику розібрано основні задачі, наведено алгоритми для задач, їх програмний код, та оцінку складності там де вона необхідна.

Задачі розділені на блоки що відповідають певній подібності умов та засобів розв’язання.

Відповідно, Розділ 1 присвячено базовим задачам, як то пошук в ширину та глибину. Розділ 2 присвячено питанням зв’язності графів. Розділ 3 присвячено віднаходженню найкоротших шляхів. Розділ 4 присвячено циклам. Розділ 5 присвячено кістяковим деревам. Розділ 6 присвячено потокам в мережах.

Розділ 1. БАЗОВІ ЗАДАЧІ

1.1 Основні визначення

Означення. Графом називається структура G = (V, E), де V – скінченна множина вершин або вузлів , E – множина пар (ребер), яка задається як бінарне відношення на V: $E \subseteq V \times V$.

Вершини *u* та *v* називаються суміжними, якщо вони з'єднані ребром *e*. В такому разі кажуть, що вершини *u* та *v* інцидентні ребру *e*, також ребро *e* інцидентне вершинам *u* та *v*. Про ребро (u, v) говорять, що воно виходить з вершини u і входить в вершину v. При цьому порядок вершин у парі (*u*,*v*) не має значення.

Означення. дуга це – направлене ребро.

Означення. Орієнтованим графом (орграфом) називається структура G = (V, E), де V – скінченна множина вершин, E – множина дуг, яка задається як бінарне відношення на V: $E \subseteq V \times V$.

Про дугу (u, v) говорять, що вона виходить з вершини u і входить в вершину v. При цьому порядок вершин у парі (*u*,*v*) має значення.

Граф, що має як ребра так і дуги, називається мішаним. Дуга чи ребро, що сполучає вершину саму зі собою називається петлею. Ребра чи дуги одного напряму, які з'єднують ту саму пару вершин, називаються кратними (паралельними) ребрами.

Означення. Неорієнтовані граф називається простим, якщо він не має петель і довільна пара вершин з'єднана не більше ніж одним ребром

Означення. Простий граф називається повним, якщо кожна пара вершин з'єднана ребром. Такий граф містить C_n^2 ребер.

Означення. Ступенем вершини в неорієнтованому графі називається кількість інцидентних їй ребер. Для орієнтованих графів розрізняють вхідну і вихідну вершини; сума вхідних і вихідних ступенів називається ступенем вершини.

Граф заданий як : V = {1, 2, 3, 4, 5, 6}, E={{1, 3},{3, 2},{3, 2},{4, 4}, {2, 4}, {5, 6},{4, 5}} зображений на Рисунку 1.

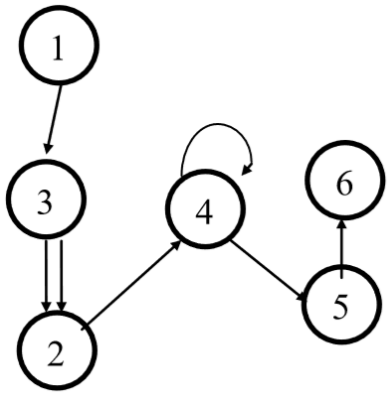


Рисунок 1. Орієнтований граф з петлею та кратними дугами

Означення. Якщо кожному ребру графа поставлено у відповідність число, то такий граф називається зваженим.

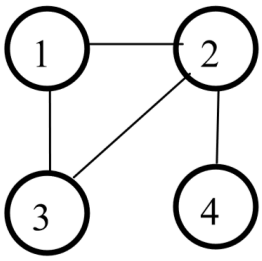
Означення. Матрицею суміжності графа G (V, E), |V|=N, називається така булева матриця A розміру $n \times n$, що A [i, j] = 1 тоді і тільки тоді, коли між вершинами i та j існує ребро. У разі зваженого графа матриця суміжності представляється двовимірної числовий матрицею, в якій A [i, j] дорівнює вазі ребра, якщо між вершинами i та j воно існує, і A [i, j] = 0 інакше.

Матриця суміжності для графа, зображеного на Рисунку 1, має вигляд:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Властивість. Матриця суміжності неорієнтованого графа симетрична.

Приклад: матриця суміжності і неорієнтовані граф, що їй відповідає.



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Алгоритмічні властивості: Перевірка присутності ребра (v_i, v_j) за допомогою матриці суміжності займає час O (1). Знаходження всіх вершин, суміжних з v_i, вимагає O(n) часу (для цього достатньо переглянути *i*-й стовпчик матриці суміжності).

Означення. Шлях довжини k з вершини *u* в вершину *v* визначається як послідовність вершин (v_0, v_1, ...,v_k), в якій $v_0 = u$, $v_k = v$ і (v_{i-1}, v_i) є E для всіх

i = 1, ... , k. Шлях довжини k складається з k ребер. Вершину v_0 називають початком шляху, а v_k – його кінцем.

Якщо для заданих вершин u і v існує шлях з *u* в *v*, то кажуть, що вершина v досяжна з вершини u.

Означення. Шлях називається простим, якщо всі вершини в ньому різні.

Означення. Циклом в орієнтованому графі називається шлях, у якому початкова вершина збігається з кінцевою і який містить хоча б одне ребро.

Петля (ребро, що з'єднує вершину сама з собою) називається циклом довжини 1. Цикл називається простим, якщо в ньому відсутні однакові вершини (крім першої та останньої), то є всі вершини v_0, v_1, ...,v_k різні.

Означення. Граф (неорієнтований), в якому відсутні цикли, називається ациклічним.

Означення. Орієнтований граф, який не містить ребер – циклів, називається простим.

У неорієнтованому графі шлях (v_0, v_1, ...,v_k) називається (простим) циклом, якщо k ≥3, $v_0 = v_k$, і всі вершини v_0, v_1, ...,v_k різні.

Означення. Неорієнтовані граф називається зв'язним, якщо для довільної пари вершин існує шлях з однієї вершини в іншу. Для неорієнтованого графа відношення "бути досяжним з" є відношенням еквівалентності на множині вершин. Класи еквівалентності називаються компонентами зв'язності графа.

1.2 Пошук в глибину

Пошуком в глибину (DFS – depth first search) називається один з методів обходу графа G = (V,E), суть якого полягає в тому, щоб йти "вглиб" поки це можливо. У процесі пошуку в глибину вершинам графа присвоюються номери (мітки). Обхід вершин графа відбувається згідно з принципом: якщо з поточної вершини є ребра, що ведуть у не пройдені вершини, то йдемо туди, інакше повертаємося назад.

Пошук в глибину починається з вибору початкової вершини v графа G, яка відразу ж позначається як пройдена. Потім для кожної невідміченої вершини, суміжної з *v*, рекурсивно викликається пошук в глибину. Коли всі вершини, досяжні з *v*, будуть відмічені, пошук закінчується. Якщо деякі вершини залишаться невідміченими, то вибирається довільна з них і пошук повторюється. Процес пошуку триває до тих пір, поки всі вершини графа G = (V,E).

Складнішим, і більш дієвим варіантом є не випадковий вибір, а використання тернарної розмітки. А саме: «не опрацьовано», «розпочато», «завершено». Зручний спосіб представлення – через колір вершин. Спочатку кольору всіх вершин білі («не опрацьовано»). Коли вершина проходиться перший раз, вона стає сірою («розпочато». Вершина є остаточно обробленою, якщо повніс-

тю переглянуті всі її суміжні вершини. Коли вершина оброблена, вона стає чорною («завершено»).

Для багатьох задач корисно мати можливість оцінити коли саме почалась та завершилась обробка вершин. Для цього можна розставляти мітки часу:

pre(*v*) – час, коли вершина стала сірою;

post(*v*) – час, коли вершина оброблена і стала чорною.

Мітки часу *pre*(*v*) *post*(*v*) є цілими числами від 1 до 2|V|. Для кожної вершини *v* має місце нерівність: *pre*(*v*)<*post*(*v*). Вершина *v* буде білою до моменту часу *pre*(*v*), сірою. з *pre*(*v*) до *post*(*v*) і чорною після *post*(*v*).

Тоді пошук в глибину на зв'язному неорієнтованому графі можна виконати таким алгоритмом:

dfs(u)

1. Вершину *u* фарбуємо в сірий колір;

2. Якщо це шукана вершина – кінець.

3. Для кожної вершини *v*, суміжної з *u*, що має білий колір, викликаємо **dfs**(*v*);

4. Фарбуємо вершину u в чорний колір;

Увага:Якщо граф незв’язний, то даний алгоритм доповнюється зовнішнім циклом:

1. Для кожної вершини *v* у графі *G*

2. якщо *v* має білий колір, викликаємо **dfs**(*v*);

Наявність міток дозволяє виконати класифікацію ребер графа, що буде описано нижче.

Приклад. Послідовність вершин графа при пошуку в глибину

Вхід. Перший рядок містить кількість вершин n в зв'язному графі. Кожна з наступних рядків містить пару вершин, з'єднаних ребром графа. Вершини графа нумеруються з 1 до n.

Вихід. Послідовність вершин, відвідуваних при пошуку в глибину, починаючи з вершини 1 як показано в прикладі.

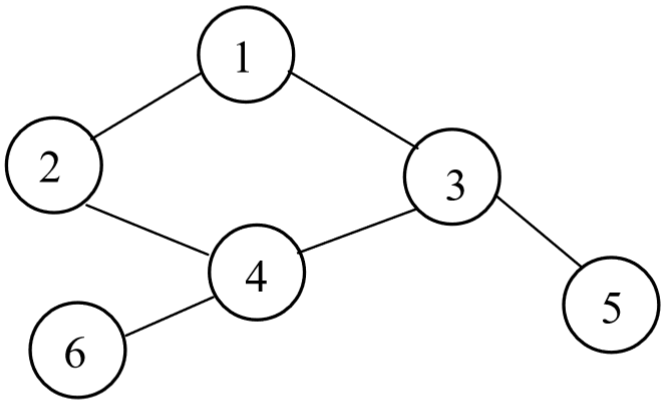


Рисунок 2. Граф для обходу в глибину.

Вхідні дані	***Вихідні дані***
6	Вершина 1 відвідана
1 2	Вершина 2 відвідана
1 3	Вершина 4 відвідана
2 4	Вершина 3 відвідана
3 4	Вершина 5 відвідана
3 5	Вершина 6 відвідана
4 6	

Код алгоритму DFS реалізований мовою C++ наведено нижче. При цьому діють такі позначення: Нехай *m* – матриця суміжності графа, *used* – булевий масив, в якому *used*[i]=1, якщо вершина і позначена (пройдена) і used[i]=0 інакше.

```
#include <stdio.h>
#include <memory.h>
#define MAX 100

int m[MAX][MAX],used[MAX];
int i,n,a,b,ptr;

void dfs(int v)
{
  int i;
  used[v] = 1;
  printf("Вершина %d відвідана\n",v);
  for(i=1;i<=n;i++)
```

```
    if (m[v][i] && !used[i]) dfs(i);
}

void main(void){
  freopen("dfs.in","r",stdin);
  memset(m,0,sizeof(m)); memset(used,0,sizeof(used));
  scanf("%d",&n);
  while(scanf("%d %d",&a,&b) == 2)
    m[a][b] = m[b][a] = 1;
  dfs(1);
}
```

Приклад . Розстановка міток на графі при пошуку в глибину.

Використовується граф з Рисунку 2.

Вхід. Перший рядок містить кількість вершин n в зв'язному графі. Кожен з наступних рядків містить пару вершин, з'єднаних ребром графа. Вершини графа нумеруються з 1 до n.

Вихід. В *i*-му рядку вивести інформацію про *i*-ї вершині в форматі, наведеному в прикладі. Разом з номером вершини і слід вивести значення *pre*[*i*] і *post*[*i*].

Вхідні дані	***Вихідні дані***
6	Вершина: 1, Сіра: 1, Чорна 12
1 4	Вершина: 2, Сіра: 2, Чорна 11
1 6	Вершина: 3, Сіра: 4, Чорна 7
2 4	Вершина: 4, Сіра: 3, Чорна 10
2 5	Вершина: 5, Сіра: 5, Чорна 6
2 6	Вершина: 6, Сіра: 8, Чорна 9
3 4	

Код алгоритму DFS реалізований мовою C++ наведено нижче. При цьому діють такі позначення: Нехай *m* – матриця суміжності графа, *used* – масив цілих чисел, в якому *used*[*i*]=0, якщо вершина біла, *used*[*i*]=1, якщо вершина сіра і *used*[*i*]=2 якщо чорна.

```
#include <stdio.h>
#include <memory.h>
```

```
#define MAX 100

int m[MAX][MAX],pre[MAX],post[MAX],used[MAX];
int i,n,a,b,time;

void dfs(int v){
  int i;
  used[v] = 1;
  pre[v] = time++;
  for(i=1;i<=n;i++)
    if (m[v][i] && !used[i]) dfs(i);
  used[v] = 2;
  post[v] = time++;
}

void main(void){
  memset(m,0,sizeof(m)); memset(used,0,sizeof(used));
  scanf("%d",&n);
  while(scanf("%d %d",&a,&b) == 2)
    m[a][b] = m[b][a] = 1;
  time = 1; dfs(1);

  for(i=1;i<=n;i++) printf("Вершина: %d, Сіра: %d, Чорна
%d\n",i,d[i],f[i]);
}
```

Приклад. Розглянемо орієнтований граф, зображений на Рисунку 3. На Рисунку 4 показаний обхід графа в глибину з розстановкою міток, починаючи з вершини 4. Біля кожної вершини v запишемо її номер в послідовності обходу в глибину, а в фігурних дужках – мітки часу *pre*(*v*) і *post*(*v*).

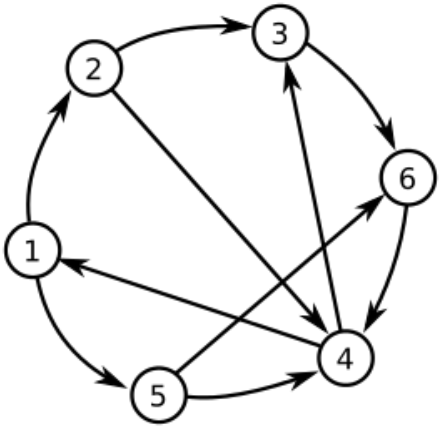


Рисунок 3. Орієнтований граф.

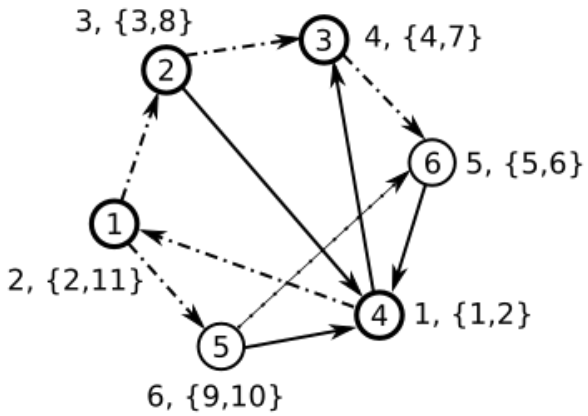


Рисунок 4. Пошук в глибину.

В результаті обходу графа в глибину отримуємо підграф, що описує порядок передування. Він являє собою дерево (ліс) пошуку в глибину. Ліс виникає коли граф не зв'язний. На Рисунку 5 представлений відповідний підграф передування.

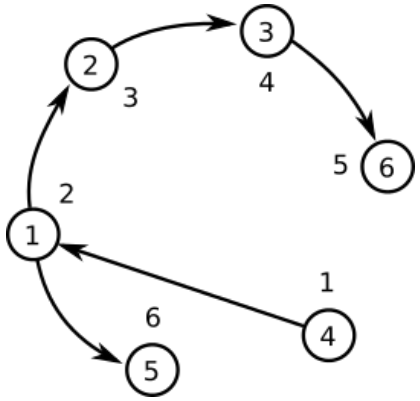


Рисунок 5. Підграф передування для графу з Рисунку 3.

Теорема про білий шлях. Вершина *v* є нащадком вершини *u* в лісі пошуку в глибину тоді і тільки тоді, коли в момент *pre*(*u*) виявлення вершини *u* існує шлях з *u* в *v*, що складається тільки з білих вершин.

1.2.1 Класифікація ребер графу

Пошук в глибину використовується для класифікації ребер орієнтованого графа. Є 4 типи ребер:

1. *Дугами* дерева є ребра, що ведуть до вершин, які раніше не відвідувалися. Вони формують для заданого графа глибинний кістяковий ліс.

2. *Зворотними* називаються дуги, що йдуть в кістяковому лісі від нащадків до предків. Дуга, що йде з вершини в саму себе також вважається зворотною.

3. *Прямими* називаються дуги, що йдуть від предків до власних нащадків, але які не є дугами дерева.

4. *Поперечними* дугами є ребра, що з'єднують вершини, які не є ні предками, ні нащадками.

Для неорієнтованих графів, без урахування кратності кількість класів менша:

1. *Дугами* дерева є ребра, що ведуть до вершин, які раніше не відвідувалися. Вони формують для заданого графа глибинний кістяковий ліс.

2. *Зворотними* називаються дуги, що йдуть в кістяковому лісі від нащадків до предків. Дуга, що йде з вершини в саму себе також вважається зворотною.

Це пов’язано в тому числі з тим, що по одному ребру можна пройти лише 1 раз.

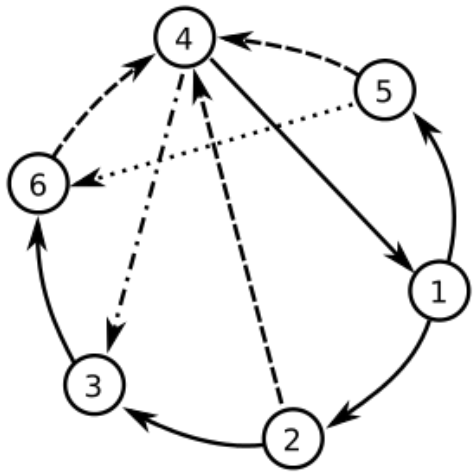


Рисунок 6. Глибинний кістяковий ліс.

На Рисунку 6 зображений глибинний ліс: дуги дерева – суцільні; зворотні дуги – штрихом; прямі дуги – штрих-пунктиром; поперечні дуги – пунктиром.

1.2.2 Властивості пошуку в глибину

Час початку і закінчення обробки вершини при пошуку в глибину утворюють правильну дужкову структуру, якщо появу вершини *u* позначати через

«(*u*», а закінчення обробки вершини u позначати через «*u*)». Пошук в глибину на графі, зображеному на Рисунку 3, утворює наступну дужкову структуру:

(4 (1 (2 (3 (6 6) 3) 2) (5 5) 1) 4),

яку можна відслідкувати на Рисунку 4.

Розглянемо відрізки на осі розміченій натуральними числами.

Теорема про дужкову структуру. При пошуку в глибину для довільних двох вершин *u* і *v* виконується одна і тільки одна з наступних властивостей:

1. Відрізки (*pre*(*u*), *post*(*u*)) і (*pre*(*v*), *post*(*v*)) не перетинаються;

2. Відрізок (*pre*(*u*), *post*(*u*)) повністю міститься в відрізку (*pre*(*v*), *post*(*v*)). При цьому *u* є нащадком *v* при пошуку в глибину.

3. Відрізок (*pre*(*v*), *post*(*v*)) повністю міститься в відрізку (*pre*(*u*), *post*(*u*)). При цьому *v* є нащадком *u* при пошуку в глибину.

З властивості 3 можна записати зручний ***критерій*** для перевірки того, чи є вершина *v* нащадком вершини *u* при пошуку в глибину:

pre(*u*) <*pre*(*v*)<*post*(*v*)<*post*(*u*).

Властивості ребер.

Ребро (*u*, *v*) є

а) ребром дерева або прямим ребром тоді і тільки тоді, коли

pre(*u*) <*pre*(*v*)<*post*(*v*)<*post*(*u*).

б) зворотним ребром тоді і тільки тоді, коли

pre(*v*) <*pre*(*u*)<*post*(*u*)<*post*(*v*).

в) поперечним ребром тоді і тільки тоді, коли

pre(*v*) <*post*(*v*) <*pre*(*u*)<*post*(*u*).

Алгоритмічні властивості: алгоритм пошуку в глибину є P-повним, тобто поліноміальним за часом відносно кількості вершин та ребер O(|V|+|E|), але дуже погано розпаралелюється, навіть теоретично. Крім того, при безпосередній реалізації стек викликів дуже швидко заповнюється, тому для великих

графів рекомендується емулювати стек шляхом підтримки вказівників на наступні вершини для всіх сірих вершин.

1.3 Пошук в ширину

Якщо задано граф $G=(V,E)$ та початкову вершину s, алгоритм пошуку в ширину систематично обходить всі досяжні із s вершини. На першому кроці вершина s позначається, як пройдена, а в список додаються всі вершини, досяжні з s (без відвідування проміжних вершин). На кожному наступному кроці всі поточні вершини списку відмічаються, як пройдені, а новий список формується із вершин, котрі є ще не пройденими сусідами поточних вершин списку. Для реалізації списку вершин найчастіше використовується структура даних «черга». Виконання алгоритму продовжується до досягнення шуканої вершини або до того кроку, коли на певній ітерації в список не включається жодна вершина. Другий випадок означає, що всі вершини, доступні з початкової, уже відмічені, як пройдені, а шлях до цільової вершини не знайдений.

Пошук в ширину на зв'язному неорієнтованому графі можна виконати таким алгоритмом:

bfs(u)

1. Помістити вершину u, з якої починається пошук, в порожню чергу.
2. Витягти з початку черги вершину u і позначити її як опрацьовану.
3. Якщо вершина u є цільовою, то завершити пошук.
4. В іншому випадку, в кінець черги додаються всі наступники вершини u, які ще не опрацьовані і не перебувають в черзі.
5. Якщо черга порожня, то всі вершини зв'язного графа були переглянуті, отже, цільова вершина недосяжна з початкової. Завершити пошук.
6. Повернутися до п. 2.

Приклад. Послідовність вершин графа при пошуку в ширину

Вхід. Перший рядок містить кількість вершин *n* в зв'язному графі. Дуги рядок містить *m* кількість ребер. Кожна з наступних рядків містить пару вершин, з'єднаних ребром графа. Вершини графа нумеруються з 1 до n.

Вихід. Послідовність вершин, відвідуваних при пошуку в глибину, починаючи з вершини 1 як показано в прикладі.

Використовується граф з Рисунку 2.

Вхідні дані	***Вихідні дані***
6	Вершина 1 відвідана
6	Вершина 2 відвідана
1 4	Вершина 3 відвідана
1 6	Вершина 4 відвідана
2 4	Вершина 5 відвідана
2 5	Вершина 6 відвідана
2 6	
3 4	

Код алгоритму BFS реалізований мовою C++ наведено нижче. При цьому діють такі позначення: *status* – масив, в якому зберігається статус вершини; змінна adj відповідає за розміщення матриці суміжності. Використовуються наступні службові функції:

`q_insert()` – вставляє елемент в чергу;

`q_delete()` – вилучає елемент з черги;

`is_q_empty()` – перевіряє чи черга порожня.

```
#include <iostream.h>
#include <conio.h>
#define MAX_NODE 50

struct node{
    int vertex;
    node *next;
};
```

```
node *adj[MAX_NODE];
int totNodes;
int queue[MAX_NODE],f=-1,r=-1;

void q_insert(int item){
    r = r+1;
    queue[r]=item;
    if(f==-1) f=0;
}

int q_delete(){
    int delitem=queue[f];
    if(f==r) f=r=-1;
    else
       f=f+1;
    return(delitem);
}

int is_q_empty(){
    if(f==-1)return(1);
    else return(0);
}

void createGraph(){
    node *newl,*last;
    int neighbours,neighbour_value;
    cout<<\"Введіть кількість вершин у графі: \";
    cin>>totNodes;
    for(int i=1;i<=totNodes;i++){
        last=NULL;
        cout<<\"\\nВведіть кількість вершин суміжних з
вершиною \"<<i<<\"\\n\";
        cin>>neighbours;
        for(int j=1;j<=neighbours;j++){
            cout<<\"Введіть сусіда #\"<<j<<\" : \";
            cin>>neighbour_value;
            newl=new node;
            newl->vertex=neighbour_value;
            newl->next=NULL;
            if(adj[i]==NULL) adj[i]=last=newl;
            else
            {last->next = newl;
                last = newl;
            }
        }
```

```
    }
}

void BFS_traversal()
{
    node *tmp;
    int N,v,start_node,status[MAX_NODE];
    const int ready=1,wait=2,processed=3;
    cout<<\"Введіть початкову вершину : \";
    cin>>start_node;

    for(int i=1;i<=totNodes;i++)
        status[i]=ready;

    q_insert(start_node);
    status[start_node]=wait;

    while(is_q_empty()!=1){
        N = q_delete();
        status[N]=processed;
        cout<<\"   \"<<N;.
        tmp = adj[N];
        while(tmp!=NULL){
            v = tmp->vertex;
            if(status[v]==ready){
                q_insert(v);
                status[v]=wait;
            }
            tmp=tmp->next;
        }
    }
}

void main()
{
    createGraph();
    BFS_traversal();
}
```

Алгоритмічні властивості:

1. Так як в пам'яті зберігаються всі опрацьовані вершини, складність алгоритму *за пам'яттю* становить O(|V|) у гіршому випадку.

2. Так як в гіршому випадку алгоритм відвідує всі вузли графа, при зберіганні графа у вигляді списків суміжності, *часова складність* алгоритму становить O(|V|+|E|).

3. Якщо довжини ребер графа рівні між собою, пошук в ширину є оптимальним, тобто завжди знаходить найкоротший шлях. У разі зваженого графа пошук в ширину знаходить шлях, що містить мінімальну кількість ребер, але не обов'язково найкоротший.

1.3.1 Пошук в ширину з двох та більше джерел

Оскільки у поганих випадках оцінка за пам’яттю складає O(|V|), а для пошуку найкоротших шляхів пошук у глибину має надто високу складність за часом, то було розроблено алгоритм, що дозволяє прискорити пошук, на основі алгоритму пошуку в ширину.

Для цього здійснюється пошук в ширину одночасно з двох джерел, і на кожній ітерації виконується порівняння множин опрацьованих вершин. Для ряду видів графів такий пошук діє набагато швидше а ніж звичайний пошук в ширину.

Питання для самоперевірки:

1. Яка основна структура пам’яті використовується у алгоритмі пошуку в глибину?
2. Яка основна структура пам’яті використовується у алгоритмі пошуку в ширину?
3. Яка складність перевірки наявності ребра у графі з використанням матриці суміжності?
4. Дуги, прямі, зворотні, поперечні – в чому між ними різниця?

Розділ 2 ЗВ'ЯЗНІСТЬ

Означення. Неорієнтований граф називається *зв'язним*, якщо для довільної пари вершин існує шлях з однієї в іншу.

Означення. Максимальний зв'язний підграф графа G називається *зв'язною компонентою*. *Зв'язні компоненти* утворюють класи еквівалентності по відношенню досяжності вершин.

Властивість. Неорієнтований граф є зв'язним тоді і тільки тоді, коли він складається з єдиної зв'язної компоненти.

Означення. Орієнтований граф називається *зв'язним*, якщо неорієнтований граф, який отримано з орієнтованого шляхом видалення орієнтації ребер, є *зв'язним*.

Означення. Орієнтований граф називається *сильно зв'язним*, якщо з довільної вершини досягається довільна інша (по орієнтованим дугам).

Властивість. Довільний орієнтований граф можна розбити на сильно зв'язні компоненти, які визначаються як класи еквівалентності "*u* досягається з *v* і *v* досягається з *u*".

Означення. Сильно зв'язною компонентою орієнтованого графа G(V, E) називається така максимальна множин вершин $C \subseteq V$, що для кожної пари вершин *u* і *v* з C вершини *u* і *v* досяжні одна з одної.

Означення. Транспонуванням графу G(V, E) називається граф $G^T(V, E^T)$, в якому $E^T = \{(u, v): (v, u) \in E\}$.

Властивість: Графи G і G^T мають одні і ті ж сильно зв'язні компоненти, так як *u* і *v* досяжні один з одного в G тоді і тільки тоді, коли вони досяжні один з одного в G^T.

2.1 Побудова та підрахунок зв'язних компонент

Застосуємо систему множин, що не перетинаються до задачі про знаходження зв'язних компонент неорієнтованого графа. Алгоритм розбиває множину вершин графа на множини, що не перетинаються, які відповідають зв'язним компонентам.

Спочатку кожна вершина розглядається як одноелементна підмножина. Далі для кожного ребра графа об’єднуємо підмножини, в які попали кінці цього ребра. Коли всі ребра будуть оброблені, множина вершин розбивається на зв’язні компоненти.

Побудову зв’язних компонент можна виконати наведеним нижче алгоритмом, де використовуються такі функції:

Make_Set(*v*) – створення множини, що складається з єдиною вершини *v*;

Find_Set(*u*) – знаходження представника множини, що містить вершину *u*;

Union(*u*,*v*) – об'єднання множин, що містять вершини *u* і *v*.

ConnComp(G);

1. для кожної вершини $v \in V$:
2. Make-Set(*v*);
3. для кожного ребра $(u,v) \in E$:
4. якщо Find-Set(u) ≠ Find-Set(v)
5. то Union(u, v);

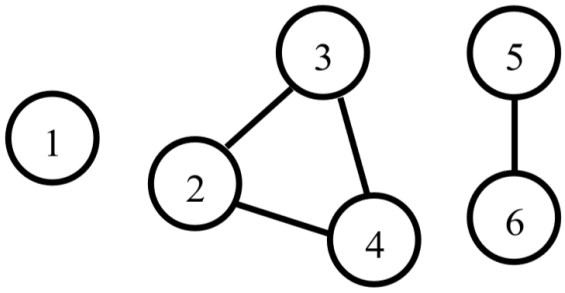


Рисунок 7. Граф з трьома зв’язними компонентами {1},{2,3,4},{5,6}

Приклад. Знаходження зв’язних компонент для графа, зображеного на Рисунку 7.

ребро	система множин, що не перетинаються
початок	{1} {2} {3} {4} {5} {6}
(2, 3)	{1}, {2,3} {4} {5} {6}
(2, 4)	{1}, {2,3,4} {5} {6}
(3,4)	{1}, {2,3,4} {5} {6}
(5, 6)	{1}, {2,3,4} {5,6}

Код алгоритму ConnComp реалізований мовою C++ наведено нижче. Використовуються наступні службові функції та змінні:

Змінна MAX містить максимальну кількість вершин у графі. У *mas*[*i*] міститься номер вершини, на яку вказує вершина *i*. Спочатку кожна вершина вказує сама на себе: *mas*[*i*] = *i*. Для кожного ребра (*a*,*b*) робимо об'єднання множин, які містять вершини *a* та *b*.

Функція *Find_Set*(*n*) повертає номер вершини – представника множини, що містить вершину *n*. Рухаємося по вказівнику на наступний елемент доки не зустрінемо представника множини (його вказівник вказує на нього самого).

Функція *Union*(*x*, *y*) об'єднує дві множини, які містять вершини *x* та *y*. Шукаємо представників множин, що містять *x* та *y*. Нехай цими представниками будуть x_1 та y_1. Якщо $x_1 = y_1$, то *x* та *y* містяться в одній множині і в такому разі нічого не робимо. Інакше вказівник представника x_1 буде вказувати на y_1.

```
#include <cstdio>
#include <algorithm>
#include <vector>
#define MAX 100
using namespace std;

int mas[MAX];
vector<vector<int> > e;
vector<vector<int> >::iterator iter;
vector<int> temp(2,0);

int Find_Set(int n){
    while (n!=mas[n]) n=mas[n];
    return n;
}

int Union(int x,int y){
    int x1 = Find_Set(x),y1 = Find_Set(y);
    mas[x1] = y1;
    return (x1 != y1);
}
```

```
void main(){
    int i,j,n;
    freopen("conn_compon.in","r",stdin);

    scanf("%d",&n);
    for(i=1;i<=n;i++) mas[i] = i;

    while(scanf("%d %d",&temp[0],&temp[1])==2)
e.push_back(temp);

    for(iter=e.begin();iter!=e.end();iter++)
        Union((*iter)[0],(*iter)[1]);
    for(i=1;i<=n;i++)
        printf("Вершина: %d, звязана з: %d, \n",mas[i]);
}
```

Підрахунок кількості зв'язних компонент виконується схожим алгоритмом.

Приклад. Обчислити кількість зв'язних компонент у графі, зображеному на Рисунку 7.

Вхід. Перший рядок містить кількість вершин графу n ($n < 100$). Кожен наступний рядок містить пару вершин, що сполучені ребром.

Вхід. Вивести кількість компонент зв'язності графу.

Приклад входу	Приклад виходу
6	3
2 3	
2 4	
3 4	
5 6	

Код для цієї задачі аналогічний попередньому, проте є невелика різниця в головній функції.

```
void main(void)
{
  int i,a,b,n,count=0;
```

```
  freopen("connect.in","r",stdin);
  scanf("%d",&n);
  for(i=1;i<=n;i++) mas[i] = i;
  while(scanf("%d %d",&a,&b) == 2) Union(a,b);
  for(i=1;i<=n;i++)
    if (mas[i] == i) count++;
  printf("%d\n",count);
}
```

Алгоритмічні властивості: оскільки компоненти зв'язності зберігаються у вигляді системи непересічних множин, то при правильній реалізації песимістична часова складність алгоритму складе O(|E|*log(|E|)).

2.2 Сильно зв’язні компоненти

Наведений нижче у вигляді псевдокоду алгоритм Косарайю за лінійний час O (|V|+|E|) знаходить сильно зв'язні компоненти орієнтованого графа G(V,E).

StronglyConnComp(G)

1. Для кожної верші-ни v є V:

2. Викликаємо пошук в глибину DFS(G), обчислюємо мітки завершення *post*[*v*].

3. Будуємо транспонований граф G^T.

3. Викликаємо пошук в глибину DFS(G^T), але в головному циклі процедури DFS розглядаємо вершини в порядку спадання значень *post*[*v*].

4. Дерева лісу пошуку в глибину, отриманого в пункті 3, представляють собою сильно зв’язні компоненти.

На основі цього алгоритму визначається граф компонент G^{SCC} (V^{SCC}, E^{SCC}) наступним чином. Нехай граф G має сильно зв'язані компоненти C_1, C_2, ..., C_k. Множина вершин $V^{SCC} = \{v_1, v_2, ..., v_k\}$ складається з вершин v_i для кожної сильно зв'язної компоненти C_i графа G.

Приклад. Розглянемо виконання алгоритму обчислення сильно зв'язкових компонент на графі що зображений на Рисунку 8.

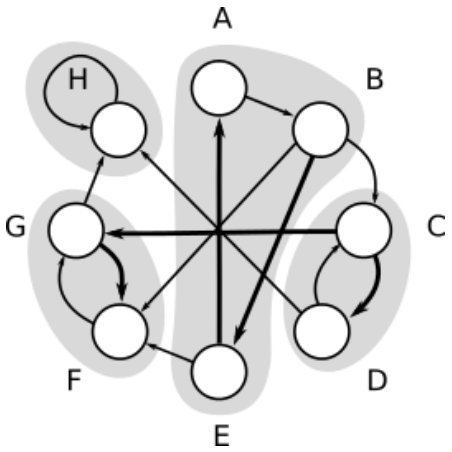


Рисунок 8. Граф для пошуку сильно зв'язаних компонент.

1. Викликаємо DFS (G), біля кожної вершини v розставляємо мітки *pre*(*v*) та *post*(*v*), як показано на Рисунку 9.:

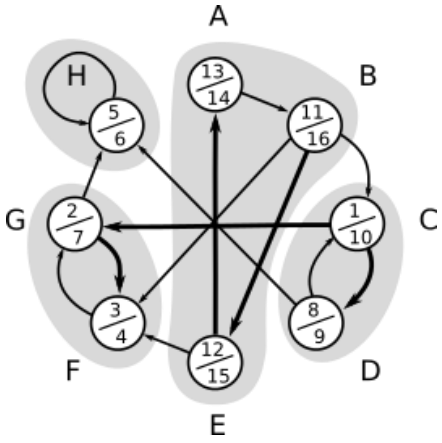


Рисунок 9.

2. Обчислюємо G^T, як показано на Рисунку 10:

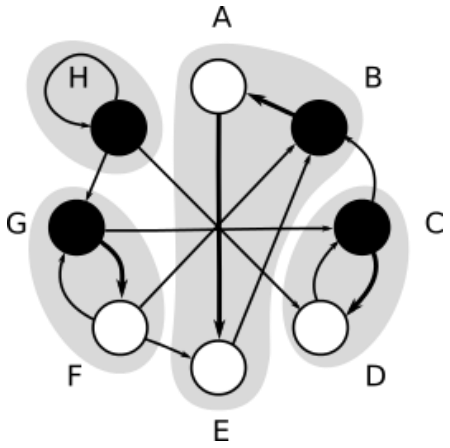


Рисунок 10.

3. Викликаємо DFS(G^T), розглядаючи вершини в порядку спадання значень *post*(*v*). Отримуємо граф компонентів, як показано на Рисунку 11:

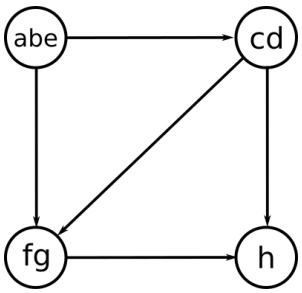


Рисунок 11.

4. Сильно зв'язними компонентами в графі будуть: {*a*, *b*, *e*}, {*c*, *d*}, {*f*, *g*} и {*h*}.

Код алгоритму реалізований на C++ наведено нижче.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_DEGREE 5
#define MAX_NUM_VERTICES 20

struct vertices_s {
    int visited;
    int deg;
    int adj[MAX_DEGREE]; /* < 0 для віхдних ребер */
} vertices[] = {
    {0, 3, {2, -3, 4}},
    {0, 2, {-1, 3}},
    {0, 3, {1, -2, 7}},
    {0, 3, {-1, -5, 6}},
    {0, 2, {4, -7}},
    {0, 3, {-4, 7, -8}},
    {0, 4, {-3, 5, -6, -12}},
    {0, 3, {6, -9, 11}},
    {0, 2, {8, -10}},
    {0, 3, {9, -11, -12}},
    {0, 3, {-8, 10, 12}},
    {0, 3, {7, 10, -11}}
};
int num_vertices= sizeof(vertices)/sizeof(vertices[0]);

struct stack_s {
    int top;
```

```
    int items[MAX_NUM_VERTICES];
} stack = {-1, {}};

void stack_push(int v) {
    stack.top++;
    if (stack.top < MAX_NUM_VERTICES)
    stack.items[stack.top] = v;
    else {
    printf("Stack is full!\n");
    exit(1);
    }
}

int stack_pop() {
    return stack.top < 0 ? -1 : stack.items[stack.top--];
}

void dfs(int v, int transpose) {
    int i, c, n;
    vertices[v].visited = 1;
    for (i = 0, c = vertices[v].deg; i < c; ++i) {
    n = vertices[v].adj[i] * transpose;
    if (n > 0)
        if (!vertices[n - 1].visited)
        dfs(n-1, transpose);
    }
    if (transpose < 0)
    stack_push(v);
    else
    printf("%d ", v + 1);
}

void reset_visited() {
    int i;
    for (i = 0; i < num_vertices; ++i)
    vertices[i].visited = 0;
}

void order_pass() {
    int i;
    for (i = 0; i < num_vertices; ++i)
    if (!vertices[i].visited)
        dfs(i, -1);
}
```

```
void scc_pass() {
    int i = 0, v;
    while((v = stack_pop()) != -1) {
    if (!vertices[v].visited) {
        printf("scc %d: ", ++i);
        dfs(v, 1);
        printf("\n");
    }
    }
}

int main(void) {
    order_pass();
    reset_visited();
    scc_pass();
    return 0;
}
```

Алгоритмічні властивості: якщо граф реалізований як перелік ребер, то часова складність буде O(|E|+|V|)). У випадку, якщо граф задано матрицею суміжності, то складність буде $O(|V|^2)$)

На основі наведеного алгоритму можна сформулювати наступні теоретичні твердження.

Лема. Нехай C_1 і C_2 - різні сильно зв'язні компоненти в орієнтованому графі G (V,E). Нехай u_1, $v_1 \in C_1$, u_2, $v_2 \in C_2$, а також в G є шлях з u_1 в u_2. Тоді в G не може бути шляху з v_2 в v_1.

Доведення. Якщо припустити зворотне, то вершини u_1 і u_2 будуть досяжні одна з іншої ($u_2 \to v_2 \to v_1 \to u_1$). Тобто ці вершини повинні входити в одну сильно зв'язну компоненту.

Нехай $U \subseteq V$. Визначимо d[u]=*pre*(u) – час початку роботи над вершиною пошуком в глибину. Визначимо f[u]=*post*(u) – час завершення роботи над вершиною пошуком в глибину. Визначимо $d(U) = \min_{u \in U}\{d[u]\}$, $f(U) = \max_{u \in U}\{f[u]\}$. Тобто d(U) і f(U) представляють собою найраніше час відкриття і найпізніше час завершення для всіх вершин з U.

Лема. Нехай C_1 і C_2 - різні сильно зв'язні компоненти в орієнтованому графі G(V, E). Нехай є ребро (u, v) є E, де u є C_1, v є C_2. Тоді f (C_1)> f (C_2).

Питання для самоперевірки:

1. Чим відрізняється сильно зв’язний орієнтований граф від зв’язного?
2. Яка алгоритмічна складність підрахунку зв’язних компонент?
3. Яка алгоритмічна складність підрахунку сильно зв’язних компонент?

Розділ 3. НАЙКОРОТШІ ШЛЯХИ

Є три основні задачі пошуку найкоротших шляхів: пошук шляхів між заданою парою вершин, пошук шляхів між кожною парою вершин у графі; пошук шляхів від заданої вершини до всіх інших вершин. Досі було розглянуто лише першу задачу. В цьому розділі буде розглянуто наступні дві.

3.1 Алгоритм Дейкстри для пошуку шляхів у графі

Нехай G = (V, E) – орієнтований граф, кожне ребро якого позначено невід'ємним числом (вага ребра). Відмітимо деяку вершину *s* і назвемо її витоком (джерелом). Треба знайти найкоротші шляхи від витоку *s* до всіх інших вершин графа G.

Приклад графу наведено та шляхів на ньому наведено на Рисунку 12

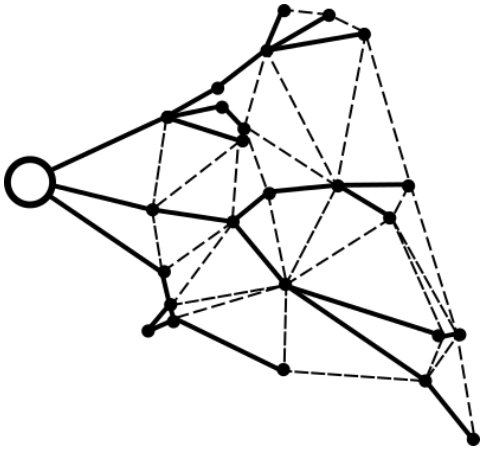


Рисунок 12.Шляхи знайдені алгоритмом Дейкстри.

Будь-яка частина найкоротшого шляху сама є найкоротший шлях. Це дозволяє для вирішення завдання застосувати динамічне програмування.

Метод динамічного програмування застосовується до завдань, які лежать в множині однотипних завдань, і деякі з завдань цієї множини прості і мають очевидні рішення, і крім того, є можливість знаходити рішення нових завдань ґрунтуючись на наявних рішеннях. Так в нашому випадку завдання «знайти найкоротший шлях з A в B» є одним із завдань виду «знайти найкоротший шлях з A в X». Одна з цих завдань має очевидне рішення «знайти найкоротший

шлях з А в А» - нікуди йти не потрібно, довжина найкоротшого шляху дорівнює 0. Залишилося описати спосіб покроково розширювати множину вирішених завдань зазначеного виду.

Необхідно, щоб задача мала оптимальну структуру і розпадалась на множину підзадач, що перекриваються. Оптимальність підструктури означає, що оптимальне рішення підзадачі меншого розміру може бути використано для вирішення всієї задачі.

Наше завдання про найкоротших шляхах володіє необхідною властивістю оптимальності для підзадач:

Лема. Нехай G=(V,E) – зважений орієнтований граф з ваговою функцією w:E$\rightarrow$R$^+$. Якщо p = (v_1, v_2, …, v_k) – найкоротший шлях из v_1 в v_k і $1 \le i \le j \le k$, то p_{ij} = (v_i, v_{i+1}, …, v_j) є найкоротший шлях из v_i в v_j.

Позначимо через $\delta(u,v)$ величину найкоротшого шляху між вершинами u и v. Вага ребра між вершинами x и y будемо позначати w(x, y).

Наслідок. Нехай p найкоротший шлях з s в v. Нехай $u \rightarrow v$ – останнє ребро цього шляху. Тоді $\delta(s, v) = \delta(s, u) + w(u, v)$.

Лема. Нехай G=(V,E) – зважений орієнтований граф з ваговою функцією w: E $\rightarrow$ R$^+$. Нехай $s \in$ V. Тоді для будь-якого ребра $(u,v) \in$ E має місце нерівність:

$$\delta(s, v) \le \delta(s, u) + w(u, v)$$

Для вирішення задачі віднаходження найкоротшого шляху з одним джерелом скористаємося жадібним алгоритмом Дейкстри. Принцип жадібного вибору тут виправданий, так як послідовність локально оптимальних виборів дає глобально оптимальне рішення.

Алгоритм Дейкстри будує множину S вершин, для яких найкоротші шляхи від витоку вже відомі. На кожному кроці до множини S додається та з вершин v, відстань до якої від витоку не більша, ніж до інших вершин. Таке додавання вершини v якраз і характеризує принцип жадібного вибору. Після додавання v в S найкоротша відстань від витоку до v вже ніколи не покращиться,

встановимо *used*[*v*]=1. Оскільки ваги дуг невід'ємні, то найкоротший шлях від витоку до конкретної вершини з множини S буде проходити тільки через вершини з S. Такий шлях будемо називати особливим. На кожному кроці алгоритму використовується масив *d*, в який записуються довжини найкоротших особливих шляхів для кожної вершини. Коли множина S буде містити всі вершини графа (для всіх вершин будуть знайдені особливі шляхи), тоді масив *d* буде містити довжини найкоротших шляхів від джерела до кожної вершини. Алгоритм може закінчити роботу не опрацювавши вершину лише якщо граф має більше однієї компоненти зв'язності.

Псевдокод алгоритму наведено нижче, при цьому діють домовленості; нехай *u*— вершина, від якої шукаються відстані, V – множина вершин графу, d_i— відстань від вершини *u* до вершини *i*, , $w_{(i, j)}$ – вага ребра (*i*, *j*).

Dijkstra(*u*,G):

1. Множина вершин U, до яких відстань відома, встановлюється рівною {*u*}.
2. Якщо U=V, алгоритм завершено.
3. Потенційні відстані D_i до вершин з V\U встановлюються на «нескінченність».
4. Для всіх ребер (*i*, *j*), де *i*єU та *j*єV\U:
5. якщо $D_j > d_i + w_{(i, j)}$, то $D_j := d_i + w_{(i, j)}$.
6. Шукається *i*єV\U, при якому D_i мінімальне.
7. Якщо D_i = «нескінченність», то алгоритм завершено.
8. Інакше $d_i := D_i$,
9. U := U∪{i}
10. Перехід до кроку 2.

Розглянемо реалізацію алгоритму Дейкстри мовою C++ за допомогою масивів. Вважаємо, що вершини графа G позначені цілими числами, серед яких виділено джерело *source*. Елемент матриці *m*[*i*][*j*] містить вага ребра (*i*, *j*). Якщо ребра (*i*, *j*) не існує, то *m*[*i*][*j*] покладемо рівним максимальному числу. Покладемо *used*[*i*]=1, якщо вершина і вже увійшла в множину S.

Функція `Relax(int i, int j)` відповідає за релаксацію (зміни мітки) шляху(ребра).

Функція `Find_Min(void)` відповідає за пошук номера ще невикористаної вершини з найменшою відстанню від витоку (значенням $d[i]$).

Функція `Init(void)` відповідає за ініціалізацію масивів.

```
#include <stdio.h>
#include <memory.h>
#define MAX 100
#define INF 2100000000

int i, j, n, source;
int b, e, dist, v;
int m [MAX] [MAX], used [MAX], d [MAX];

void Init(void){
  memset (m, 63, sizeof (m));
  memset (used, 0, sizeof (used));
  memset (d, 63, sizeof (d)); d [source] = 0;
}

void Relax (int i, int j){
  if (d[j]> d[i] + m[i][j]) d[j] = d[i] + m[i][j];
}

int Find_Min(void){
  int i, v, min = INF / 2;
  for (i = 1; i <= n; i ++)
    if (! used [i] && d [i] <min) min = d [i], v = i;
  return v;
}


int main (void){
    scanf ( "% d% d", & n, & source);
    Init ();
    while (scanf ( "% d% d% d", & b, & e, & dist) == 3)
m[b][e]=dist;

for (i = 1; i <n; i ++){
    v = Find_Min ();
    for (j = 1; j <= n; j ++)
      if (! used [j]) Relax (v, j);
    used [v] = 1;
```

```
  }

for (i = 1; i <= n; i ++)
      printf ( "From source% d to destination% d distance
is% d \ n",source, i, d[i]);
return 0;
}
```

Алгоритмічні властивості: В найпростішому випадку часова складність сягає $O(|V|^2)$. Для розріджених графів (тобто таких, для яких |E| багато менше $|V|^2$) не відвідані вершини можна зберігати в двійковій купі, а в якості ключа використовувати значення d[i], час роботи такої реалізації – O((|V|+|E|)*log(|V|)).

Приклад. На Рисунку 13 зображено граф для алгоритму Дейкстри.

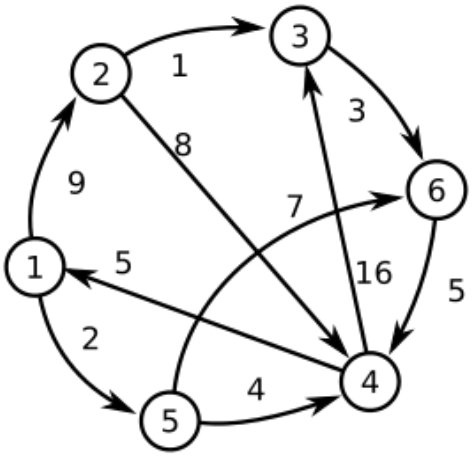


Рисунок 13. Граф для алгоритму Дейкстри

Вхід. Перший рядок містить кількість вершин n в графі і номер вершини - початку *s*. Кожний наступний рядок описує орієнтоване ребро графу і містить номери вершин, які воно з'єднує, а також його вагу. Вершини графа нумеруються числами від 1 до n.

Вихід. Для кожної вершини v графа вивести рядок: «Від джерела *s* до мети *v* відстань *dist»*

Тут *dist* – найкоротша відстань від витоку *s* до вершини *v*. Рядки виводити в порядку зростання номерів вершин *v*.

Приклад входу	***Приклад виходу***
6 4	Від джерела *4* до мети *1* відстатнь *5*
1 2 9	Від джерела *4* до мети *2* відстатнь 14
1 5 2	Від джерела *4* до мети *3* відстатнь 15

2 3 1	Від джерела *4* до мети *4* відстатнь 0
2 4 8	Від джерела *4* до мети *5* відстатнь 7
3 6 3	Від джерела *4* до мети *6* відстатнь 14
4 1 5	
4 3 16	
5 4 4	
5 6 7	
6 4 5	

Розглянемо граф, наведений в тестовому прикладі. Вершина 4 є джерелом. Ініціалізуємо S ={} Для кожного значення *k* покладемо *d*[*k*] рівним максимальному натуральному числу. При цьому *d*[4]=0, оскільки відстань від вершини до її самої дорівнює 0.

На першій ітерації знаходимо найменше значення серед *d*[*i*], де *i* вершина, ще не увійшла в S. min{*d*[1], *d*[2], *d*[3], *d*[4], *d*[5], *d*[6]}=*d*[4]=0. Першою в множину S буде включена вершина 4. Перераховуємо значення *d*[*i*] для всіх i є V\S = {1, 2, 3, 5, 6}:

d[1] = min (*d*[1], *d*[4] + *m*[4][1]) = min (+ ∞, 0 + 5) = 5;

d[2] = min (*d*[2], *d*[4] + *m*[4][2]) = min (+∞, 0 + ∞) = + ∞;

d[3] = min(*d*[3], *d*[4] + *m*[4][3]) = min (+∞, 0 + 16) = 16;

d[5] = min(*d*[5], *d*[4] + *m*[4][5]) = min (+∞, 0 + ∞) = + ∞;

d[6] = min(*d*[6], *d*[4] + *m*[4][6]) = min (+∞, 0 + ∞) = + ∞;

Розглянемо другу ітерацію. Шукаємо мінімум серед *d*[*i*], де *i* є S: min{*d*[1], *d*[2], *d*[3], *d*[4], *d*[5], *d*[6]}=*d*[1]=5. Другою у множину S буде включена вершина 1, тобто S = {1, 4}. Перераховуємо значення *d*[i] для всіх *i* є V\S = {2, 3, 5, 6}:

d[2] = min(*d*[2], *d*[1] + *m*[1][2]) = min (+∞, 5 + 9) = 14;

d[3] = min(*d*[3], *d*[1] + *m*[1][3]) = min (16, 5 + ∞) = 16;

d[5] = min(*d*[5], *d*[1] + *m*[1][5]) = min (+∞, 5 + 2) = 7;

d[6] = min(*d*[6], *d*[1] + *m*[1][6]) = min (+∞, 5 + ∞) = + ∞;

Результат виконання всіх ітерацій наведено в таблиці нижче.

итерація	S	*d*[1]	*d*[2]	*d*[3]	*d*[4]	*d*[5]	*d*[6]
початок	{}	+ ∞	+ ∞	+ ∞	**0**	+ ∞	+ ∞
1	{4}	**5**	+ ∞	16	*0*	+ ∞	+ ∞
2	{1, 4}	*5*	14	16	*0*	**7**	+ ∞
3	{1, 4, 5}	*5*	14	16	*0*	*7*	**14**
4	{1, 4, 5, 6}	*5*	**14**	16	*0*	*7*	*14*
5	{1, 2, 4, 5, 6}	*5*	*14*	**15**	*0*	*7*	*14*

Вершина *v*, яка вибирається і додається до S на кожному кроці, виділена і підкреслена.

3.2 Пошук найкоротших шляхів між парами вершин

Є орієнтований граф G = (V, E), кожній дузі (*i, j*) якого поставлено у відповідність невід'ємне число *g*[*i*][*j*]. Необхідно знайти довжини найкоротших шляхів між усіма парами вершин.

Завдання можна вирішувати, послідовно використовуючи алгоритм Дейкстри для кожної вершини, оголошуючи її в якості джерела. Однак існує інше рішення поставленого завдання, відоме як алгоритм Флойда-Уоршела.

Нехай вершини графа послідовно пронумеровані числами від 1 до n. Якщо дуга (*i, j*) в графі відсутня, то покладемо *g*[*i*][*j*]=+∞. Кожному діагонального елементу матриці відстаней *g* дамо нульове значення (довжина найкоротшого шляху від вершини до її самої дорівнює 0).

Над матрицею *g* здійснюємо *n* ітерацій. Після *k*-ої ітерації g[*i*][*j*] містить довжину найкоротшого шляху з *i* в *j*, який не проходить через вершини з номерами, більшими *k*. На *k*-ій ітерації для перерахунку матриці *g* використовується формула:

$g_k[i][j] = \min (g_{k-1}[i][j], g_{k-1}[i][k] + g_{k-1}[k][j])$.

Псевдокод алгоритму наведено нижче, при цьому *dist* – масив ініційований відстанями, *next* – масив ініційований null, *w*(*u, v*) – вага ребра:

FloydWarshall()

1.Для кожного ребра (*u, v*):

2. dist[u][v] ← w(u, v)

3. next[u][v] ← v

4. Для кожної вершини *v*:

5. dist[v][v] ← 0

6. next[v][v] ← v

7.Для цілого *k* від 1 to |V|:

8. Для цілого *i* від 1 to |V|:

9. Для цілого *j* від 1 to |V|:

10. Якщо dist[i][j] > dist[i][k] + dist[k][j] то

11. dist[i][j] ← dist[i][k] + dist[k][j]

12. next[i][j] ← next[i][k]

Код алгоритму реалізований мовою C++, без можливості відновити шляхи наведено нижче:

```
#include <stdio.h>
#include <memory.h>
int n,a,b,dist;
int i,j;
int g[10][10];

void floyd(void){
     int i,j,k;
     for(k=1;k<=n;k++)
          for(i=1;i<=n;i++)
               for(j=1;j<=n;j++)
                    if(g[i][k]+g[k][j]<g[i][j])
               g[i][j]=g[i][k]+g[k][j];
}

void main(void){
     scanf("%d",&n);
     memset(g,0x3F,sizeof(g));
     for(i=1;i<=n;i++) g[i][i] = 0;
     while(scanf("%d %d %d",&a, &b, &dist)==3)
g[a][b]=dist;

     floyd();

     for(i=1;i<=n;i++){
          for(j=1;j<=n;j++)
```

```
            printf("%2d ",g[i][j]);
            printf("\n");
    }
}
```

Алгоритмічні властивості: Часова складність алгоритму дорівнює $O(|V|^3)$.

Приклад. Для графу наведеного на Рисунку 14 оцінити відстані за алгоритмом Флойда-Уоршела.

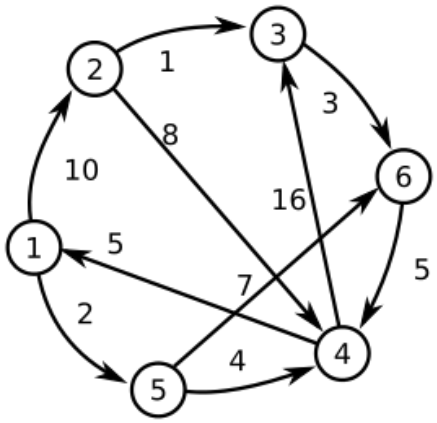


Рисунок 14. Граф для алгоритму Флойда-Уоршела.

Вхід. Перший рядок містить кількість вершин *n* в графі. Кожний наступний рядок описує орієнтоване ребро графа *i,* вершини, які воно з'єднує, а також його вагу. Вершини графа нумеруються числами від 1 до *n*.

Вихід. Вивести таблицю *res* розміром *n* на *n*, де *res*[*i*][*j*] містить довжину найкоротшго шляху між вершинами *i* та *j*.

Приклад входу	***Приклад виходу***
6	0 10 11 6 2 9
1 2 10	13 0 1 8 15 4
1 5 2	13 23 0 8 15 3
2 3 1	5 15 16 0 7 14
2 4 8	9 19 20 4 0 7
3 6 3	10 20 21 5 12 0
4 1 5	

4 3 16	
5 4 4	
5 6 7	
6 4 5	

Вивід найкоротших шляхів. Для можливості виводу найкоротшого шляху між двома вершинами треба реалізувати псевдокод повністю. Матриця *next* відповідає за такі шляхи. Елемент *next*[*i*][*j*] містить вершину *k*, отриману при обчисленні найменшого значення g [*i*] [*j*].

Для виведення на екран послідовності вершин, що представляють собою найкоротший шлях від *i* до *j*, викликається процедура *PrintPath*, наведена нижче.

```
void Path(int i, int j){
int k = next[i][j];
if (!k) return;
Path(i, k); printf("%d ",k); Path(k, j);
}

void PrintPath(int i, int j){
printf("%d ",i); Path(i, j); printf("%d\n",j);
}
```

При виклику процедури PrintPath (3, 2) для наведеного вище графа буде виведена наступна послідовність вершин: 3, 6, 4, 1, 2.

3.3 Транзитивне замикання

Нехай G(V, E) – орієнтований граф, A – його матриця суміжності.

Означення. Булева матриця C, в якій C[*i*, *j*] = 1 тоді і тільки тоді коли існує шлях з вершини *i* до вершини *j*, називається ***транзитивним замиканням*** матриці суміжності A.

Транзитивне замикання можна обчислити за допомогою алгоритма Флойда-Уоршела, застосувавши на *k*-ій ітерації наступну формулу до булевої матриці C:

$C_k[i, j] = C_{k-1}[i, j]$ or ($C_{k-1}[i, k]$ and $C_{k-1}[k, j]$)

Ця формула встановлює існування шляху від *i* до *j*, який проходить через вершини з номерами не більшими за *k*, лише в наступних випадках:

1) Вже існує шлях від *i* до *j*, який проходить через вершини з номерами не більшими за $k - 1$.

2) Існує шлях від *i* до *k*, який проходить через вершини з номерами не більшими за $k - 1$ та шлях від *k* до *j*, який також проходить через вершини з номерами не більшими за $k - 1$.

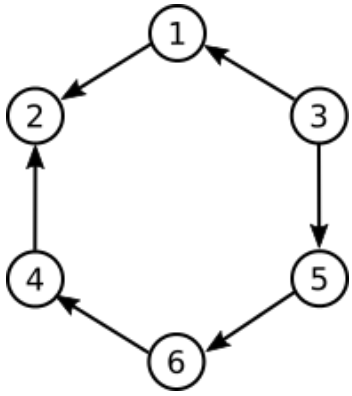


Рисунок. 15. Орієнтований граф

Приклад. Запустимо алгоритм Уоршелла на графі, поданому на Рисунку 15. Матриця суміжності A та матриця транзитивного замикання C мають вид:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, C = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Розглянемо матрицю суміжності A графу G як булеву матрицю, в якій $A[i, j] = 1$ якщо існує ребро, яке сполучає вершини *i* та *j*, та $A[i, j] = 0$ інакше. Вважаємо, що довжини усіх ребер дорівнюють 1. Матриця суміжності містить інформацію про шляхи довжини 1 між вершинами графу. Матриця $A^2 = A * A$ містить інформацію про наявність шляхів довжини 2. І взагалі, $A^n[i, j] = 1$ якщо існує шлях між вершинами *i* та *j* довжини *n*. Якщо такого шляху не існує, то $A^n[i, j] = 0$. Транзитивне замикання матриці суміжності A визначається як логічна операція **or** матриць A^i, $i = 1, n$, де *n* – розмір матриці A:

Closure(A) = A^1 **or** A^2 **or** A^3 **or** ... **or** A^n

Матриці суміжності A та транзитивного замикання C із прикладу пов'язані рівністю:

C = A^1 **or** A^2 **or** A^3 **or** A^4 **or** A^5 **or** A^6

Питання для самоперевірки:

1. Чим відрізняється результат роботи алгоритму Дейкстри від результату алгоритму Флойда-Уоршела?
2. Яка алгоритмічна складність алгоритму Дейкстри?
3. Яка алгоритмічна складність алгоритму Флойда-Уоршела?

Розділ 4. ЦИКЛИ

Базові означення для циклів були дані у Розділі 1. В цьому розділі увага буде зосереджена на двох спеціальних випадках циклів, а саме на ейлерових циклах та на гамільтонових циклах.

4.1 Ейлерів цикл

Вперше поняття ейлерового циклу з'явилося в 1736, коли Леонард Ейлер вирішував задачу про сім кенігсберських мостів: «Чи можна обійти всі сім мостів, заданих на рисунку нижче, побувавши на кожному рівно по одному разу?»

Граф, що відповідає задачі зображено на Рисунку 16.

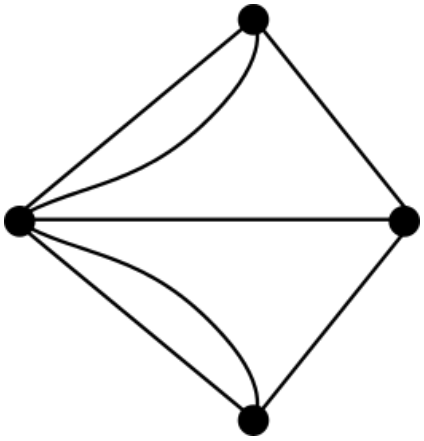


Рисунок 16. Граф, що відповідає семи кенігсбергським мостам.

Означення. Ейлеровим шляхом називається такий шлях, який проходить по кожному ребру графа рівно один раз.

Означення. Ейлеровим циклом називається такий Ейлерів шлях, який починається і закінчується в одній і тій же вершині.

Означення. Граф, що містить Ейлером цикл, називається ейлеровим графом.

Теорема. Зв'язний неорієнтований граф містить Ейлерів цикл тоді і тільки тоді, коли кількість вершин непарного ступеня дорівнює нулю.

Для пошуку ейлерового циклу в неорієнтованому графі використовується пошук в глибину з вилученням ребер. Порядок перегляду вершин запам'ятовується. Якщо знайдена вершина, з якої не виходять ребра (ми їх знищили процесі пошуку в глибину), її номер заноситься в стек і пошук починається з попередньої вершини. Алгоритм триває до тих пір, поки є не пройдені ребра. У стеці будуть записані номери вершин графа в порядку, який відповідає ейлеровому циклу.

Код алгоритму реалізований мовою C++ наведено нижче. При цьому діють такі позначення: *m* – матриця суміжності графа, *cycle* – стек, *n* – кількість вершин в графі.

```
void euler(int i)
{
  int j;
  for(j=1;j<=n;j++)
  if (m[i][j])
  {
    m[i][j] = m[j][i] = 0;
    euler(j);
  }
  cycle[counter++] = i;
}
```

Приклад. Граф задано на Рисунку 17. Чи є в ньому ейлерів цикл та який він?

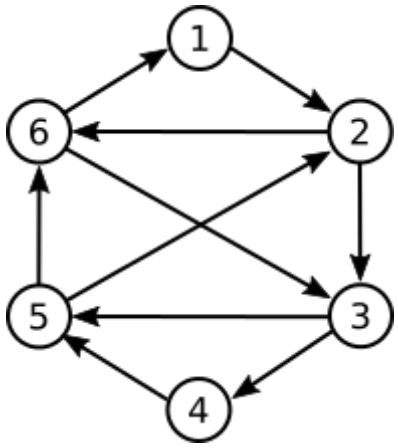


Рисунок 17. Приклад для ейлерового циклу.

Відповідно до теореми в цьому графі є ейлерів цикл. З алгоритмом можна визначити, що він має таку структуру: 1 2 3 4 5 2 6 3 5 6 1.

Алгоритмічні властивості простого алгоритму: пошук циклу безпосередньо є неефективним, має часову складність $O(|E|^2)$.

Через це для великих графів краще застосовувати інші алгоритми, що використовують додаткові знання.

Властивість: ейлерів цикл є об'єднанням всіх простих циклів.

Псевдокод процедури для пошуку всіх циклів наведено нижче, *cycles* – масив:

find_all_cycles(*v*)

1. поки є цикл, що проходить через *v*:
2. знаходимо його,
3. додаємо все вершини знайденого циклу в *cycles* (зберігаючи порядок обходу)
4. вилучаємо цикл з графа
5. ідемо по елементам *cycles:*
6. кожен элемент *cycles*[*i*] додаєм до відповіді
7. з кожного елемента рекурсивно викликаємо *find_all_cycles*(*cycles*[*i*])

Алгоритмічні властивості:Часова складність такого алгориму O(|E|).

4.2 Гамільтонів цикл

Означення. Гамільтоновим називається шлях, який проходить через кожну першину один раз.

Означення. Гамільтоновим називається цикл, є гамільтоновим шляхом.

Означення. Граф називається гамільтоновим, якщо він містить гамільтонів цикл.

Приклад. Гіперкуб порядку n (n>1) має гамільтонів цикл. Його описує код Грея.

Приклад. Повний граф з n (n> 2) вершинами має гамільтонів цикл. Оскільки між двома будь-якими вершинами існує ребро, то існує ребро з v_i в v_{i+1}. Існує також ребро з останньої вершини v_n в першу v_1.

Теорема. Для будь-якої вершини з гамільтонового циклу існує рівно два ребра з цього циклу, інцідентні даної вершині.

Наслідок. Граф, що містить вершину ступеня 1, не може бути гамільтоновим.

Теорема. Якщо граф має міст (розрізне ребро), то він не може бути гамільтоновим. Якщо компоненти графа, отриманий шляхом видалення розроізного ребра, мають гамильтонов цикл, то вихідний граф має гамільтонів шлях.

Для знаходження всіх можливих гамільтонових шляхів будемо використовувати перебір з поверненням (бектрекінг). Оскільки в гамільтонів цикл входять всі вершини графа, то не має значення з якої вершини починати пошук такого циклу (для визначеності будемо починати пошук гамільтонового циклу з першої вершини). Припустимо, що вже знайдено *k* вершин циклу. Якщо *k* дорівнює кількості вершин у графі і існує ребро з *k*-ої вершини в першу (з якої починали пошук), то цикл знайдений і його необхідно вивести на друк. Якщо існують ребра, що виходять з останньої (*k*-ої) вершини до ще непереглянутих вершин, то включаємо одну з них в рішення, помічаємо її як переглянуту і продовжуємо з неї пошук. Якщо таких ребер не існує, то повертаємося на крок назад до *k*-1 вершині і продовжуємо пошук. Таким чином будуть знайдені всі вершини гамільтонового шляху.

Код алгоритму реалізований мовою C++ наведено нижче. При цьому діють такі позначення: *n* – кількість вершин в графі, *cycle* – массив вершин гамільтонового циклу, *used* –масив, в якому *used*[i]=1, якщо вершина i позначена (пройдена) і *used*[i]=0 інакше.

```
void gamilt(int v)
{
  int i;
```

```
  if ((cnt == n) && m[cycle[n-1]][1])
  {
    for(i=0;i<n;i++) printf("%d ",cycle[i]);
    printf("\n");
    return;
  }
  for(i=1;i<=n;i++)
    if (m[v][i] && !used[i])
    {
      used[i] = 1;m[v][i] = m[i][v] = 0;
      cycle[cnt++] = i;
      gamilt(i);
      cnt--;
      used[i] = 0;m[v][i] = m[i][v] = 1;
    }
}

void main(void)
{
    memset(m,0,sizeof(m));
    memset(used,0,sizeof(used));

    scanf("%d",&n);
    while(scanf("%d %d",&a,&b) == 2) m[a][b]=m[b][a]=1;
    cnt = 0;
    cycle[cnt++]=1;
    used[1] = 1;gamilt(1);
    for(int i=1;i<=n;i++)
        if used[i] printf("В цикл входить вершина:
    %d,\n",i);
    return;
}
```

Алгоритмічні властивості:Алгоритм входить до класу NP, тому для нього використовують евристики:

- на кожному кроці перебору при деякій побудованої частини ланцюга перевіряти, чи утворюють решта вершини зв'язний граф (якщо не утворюють, то ланцюг не може бути початком гамільтонової ланцюга);
- на кожному кроці перебору при виборі наступної вершини пробувати спочатку вершини з найменшим залишковим ступенем (кількістю ребер, що ведуть в ще не відвідані вершини).

Приклад. На Рисунку 18 задано граф. Чи є в ньому гамільтонові цикли, відмінні від показаного і якщо є то які?

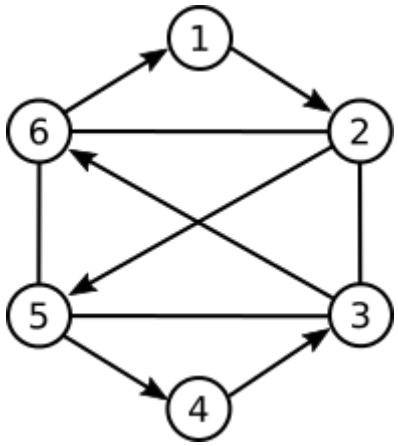


Рисунок 18. Приклад на гамільтонові цикли.

Відповідь: якщо починати з першої вершини, то наведений алгоритм знайде такі гамільтонові цикли: 1,2,3,4,5,6; 1,6,3,4,5,2; 1,6,5,4,3,2 та цикл зображений на рисунку.

Питання для самоперевірки:

1. Що таке ейлерів цикл?
2. Що таке гамільтонів цикл?
3. Яка алгоритмічна складність віднаходження гамільтонового циклу?
4. Яка алгоритмічна складність віднаходження ейлеревого циклу?

Розділ 5. КІСТЯКОВІ ДЕРЕВА

Означення. Для довільного зв'язного неорієнтованого графа G=(V,E) кожне дерево (V, T), де $T \subseteq E$, називається кістяковим деревом (каркасом).

Пошук такого дерева можна робити за допомогою пошуку в глибину і в ширину. При цих пошуках проглядаються всі вершини. Ребро графа буде включено в каркас, якщо відбувається перехід по цьому ребру від переглянутої до ще непереглянутих вершини при відповідному пошуку.

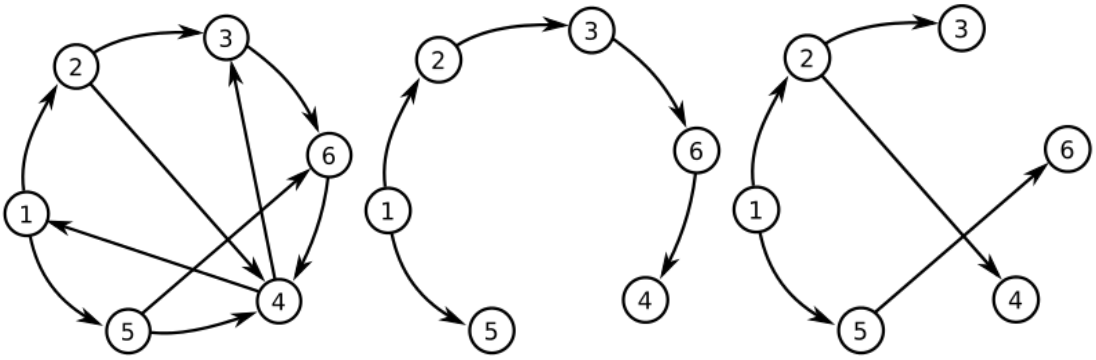


Рисунок 19. Граф і його каркаси, побудовані пошуком в глибину і в ширину.

Нехай G (V, E) – зв'язний граф, в якому кожне ребро (*u*,*v*) позначено числом *w*(*u*,*v*), яке називається вагою (вартістю) ребра. Вартість кістякового дерева обчислюється як сума вартостей всіх ребер, які входять в нього.

Завдання пошуку мінімального кістякового дерева полягає в знаходженні для заданого графа кістякового дерева найменшої вартості.

5.1 Алгоритм Крускала

Розглянемо зв'язний зважений граф G=(V,E), V={1,2,...,n}. Побудова кістякового дерева мінімальної вартості починається з графа T=(V,∅), що складається з вершин графа G і не має ребер. Алгоритм Крускала належить класу жадібних алгоритмів і нагадує обчислення кількості компонент зв'язності. Для зберігання даних він використовує структуру непересічних множин, а для їх обробки – такі функції:

Make_Set(*v*) – створення множини, що складається з єдиною вершини *v*;

Find_Set(*u*) – знаходження представника множини, що містить вершину *u*;

Union(*u*,*v*) – об'єднання множин, що містять вершини *u* і *v*.

Спочатку кожна вершина остова T=(V,∅) є зв'язною (з самою собою) компонентою. В алгоритмі відбувається процес об'єднання зв'язних компонент, результатом якого є формування кістякового дерева.

Множина ребер E проглядається в порядку зростання вартостей ребер. Якщо чергове ребро зв'язує дві вершини з різних компонент, то воно додається в граф T, інакше - відкидається (бо його додавання призведе до появи циклу). Побудова шуканого кістякового дерева закінчується, коли всі вершини графа будуть належати одній компоненті зв'язності, або не існуватиме ребер, що не були опрацьовані (для потенційно незв'язного графу).

Псевдокод до алгоритму виглядає так:

Kruskal(G)

1. T = ∅;
2. для кожної вершини v ∈V(G):
3. викликати Make_Set (*v*);
4. сортуємо ребра з V(E) в неспадному порядку їх ваг *w*;
5. для кожного ребра (*u*,*v*) ∈E (в порядку неспадання ваги):
6. Якщо *Find_Set*(*u*) ≠ *Find_Set*(*v*) тоді
7. T = T U {(*u*,*v*)};
8. *Union*(*u*,*v*);
9. повернути T;

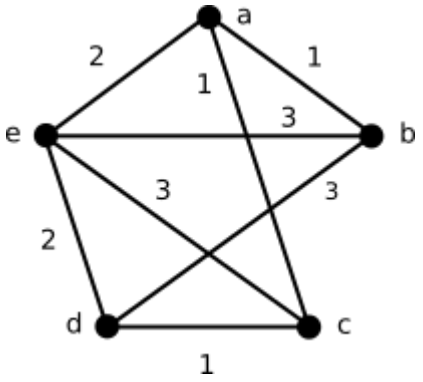


Рисунок 20. Граф для побудови кістякових дерев.

Розглянемо програму на C++, що реалізує алгоритм Крускала. Масив *e* містить ребра графа, кожне з яких містить номери вершин, що з'єднуються цим

ребром і його вагу. Структура непересічних множин моделюється масивом *m*: *m*[*i*] містить номер вершини, на яку вказує вершина *i*. Множина складається з однієї вершини *i*, якщо *m*[*i*]=*i*.

```
#include <cstdio>
#include <algorithm>
#include <vector>
#define MAX 100
using namespace std;

int mas[MAX],res;
vector<vector<int> > e;
vector<vector<int> >::iterator iter;
vector<int> temp(3,0);

int Repr(int n)
{
  while (n != mas[n]) n = mas[n];
  return n;
}

int Union(int x,int y)
{
  int x1 = Repr(x),y1 = Repr(y);
  mas[x1] = y1;
  return (x1 != y1);
}

int lt(vector<int> a, vector<int> b)
{
  return (a[2] < b[2]);
}

void main()
{
  int i,n;
  freopen("kruskal.in","r",stdin);

  scanf("%d",&n);
  for(i=1;i<=n;i++) mas[i] = i;

  while(scanf("%d %d %d",&temp[0],&temp[1],&temp[2]) ==
3)
    e.push_back(temp);
```

```
  sort(e.begin(),e.end(),lt);

  res = 0;
  for(iter=e.begin();iter!=e.end();iter++)
    if (Union((*iter)[0],(*iter)[1])) res += (*iter)[2];
  printf("%d\n",res);
}
```

Приклад . Алгоритм Крускала. В результаті виконання коду для графа зображеного на Рисунку 20 буде отримано послідовність графів зображену на Рисунку 21.

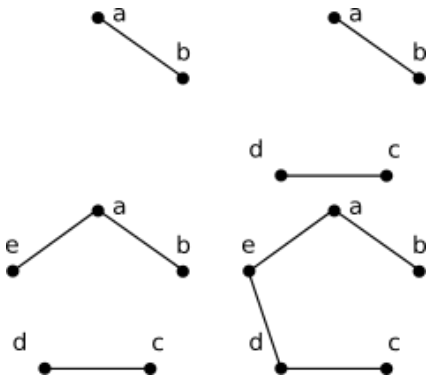


Рисунок 21. Послідовність кістякових дерев утворених алгоритмом Крускала.

Алгоритмічні властивості:

Часова складність O(|E|log |E|). Складність по пам’яті O(|V|).

5.2 Алгоритм Прима

Нехай V={1,2,...,*n*} – множина вершин графа G=(V,E). Побудуємо множину U, з якої будему отримувати кістякове дерево. Спочатку покладемо U={1} (кістякове дерево починає будуватися з першої вершини). На кожному кроці алгоритму знаходиться ребро найменшої вартості (*u*,*v*) таке, що *u* є U і *v* є V\U, після чого вершина v переноситься з множини V\U в U. Цей процес триває до тих пір, поки множина U не стане рівна множині V.

Псевдокод до алгоритму виглядає так:

Prim(G)

1. T = ∅;

2. U = {1};

3. поки U ≠ V

4. знаходимо таке ребро (*u*,*v*) наименшої вартості, що *u*∈U, *v*∈V\U;

5. T = T∪{(*u*, *v*)};

6. U = U ∪ {*v*};

7. Вивести T;

Розглянемо програму на C++, що реалізує алгоритм Прима. Вектор *inMST* вказує на те, чи входять дві вершини в мінімальне кістякове дерево.

```
#include <bits/stdc++.h>
using namespace std;
#define V 5
bool createsMST(int u, int v, vector<bool> inMST)
{
   if (u == v)
      return false;
   if (inMST[u] == false && inMST[v] == false)
      return false;
   else if (inMST[u] == true && inMST[v] == true)
      return false;
   return true;
}
void printMinSpanningTree(int cost[][V])
{
   vector<bool> inMST(V, false);
   inMST[0] = true;
   int edgeNo = 0, MSTcost = 0;
   while (edgeNo < V - 1) {
      int min = INT_MAX, a = -1, b = -1;
      for (int i = 0; i < V; i++) {
         for (int j = 0; j < V; j++) {
            if (cost[i][j] < min) {
```

```
                if (createsMST(i, j, inMST)) {
                    min = cost[i][j];
                    a = i;
                    b = j;
                }
            }
        }
        if (a != -1 && b != -1)
        {
            cout<<"Ребро "<<edgeNo++<<" : ("<<a<<" , "<<b<<"
) : cost = "<<min<<endl;
            MSTcost += min;
            inMST[b] = inMST[a] = true;
        }
    }
        cout<<"Ціна мінімального кістякового дерева
="<<MSTcost;
}
int main()
{
    int cost[][V] = {
        { INT_MAX, 12, INT_MAX, 25, INT_MAX },
        { 12, INT_MAX, 11, 8, 12 },
        { INT_MAX, 11, INT_MAX, INT_MAX, 17 },
        { 25, 8, INT_MAX, INT_MAX, 15 },
        { INT_MAX, 12, 17, 15, INT_MAX },
    };
    cout<<"Мінімальне кістякове дерево:\n";
    printMinSpanningTree(cost);
```

```
    return 0;
}
```

Приклад. Алгоритм Прима. В результаті виконання коду для графа зображеного на Рисунку 20 буде отримано послідовність графів зображену на Рисунку 22.

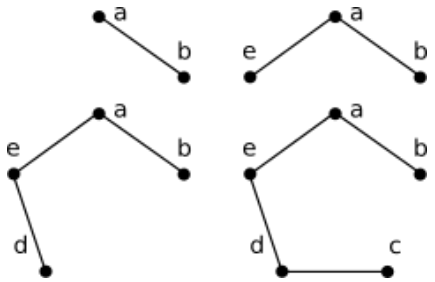


Рисунок 22. Послідовність кістякових дерев утворених алгоритмом Прима.

Алгоритмічні властивості:

Часова складність при використанні матриці суміжності $O(|V^2|)$, тому рекомендується користуватися купою та переліком ребер. Внутрішній цикл можна розпаралелити.

5.3 Матрична формула Кірхгофа, теорема Келі

Означення. Матрицею ступенів графа G=(V,E) називається матриця D розміром n x n (n = | V |), задана таким чином: Dij =0, для елементів не на головній діагоналі, та Dij = ступінь вершини v_i для елементів на головній діагоналі.

Теорема. Матрична формула Кірхгофа.

Нехай G – зв'язний неорієнтований граф з розміченимми вершинами. Нехай K=D-C, де C – матриця суміжності, а D – матриця ступенів графа G. Тоді кількість кістякових дерев графа G дорівнює будь-якому з алгебраїчних доповнень матриці K.

Приклад. На Рисунку 23 зображено граф, треба обчислити кількість кістякових дерев для нього.

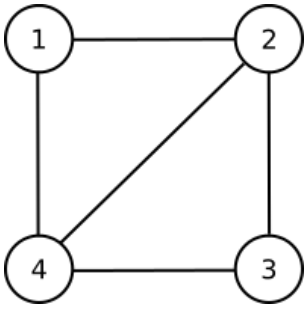


Рисунок 23.

Матриця суміжності C и матриця степенів D мають вигляд:

$$C = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}, D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

$$K = D - C = \begin{pmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{pmatrix}$$

Обчислимо визначник алгебраїчного доповнення K_{11}:

$$\det(K_{11}) = \begin{vmatrix} 3 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 3 \end{vmatrix} = 3 \begin{vmatrix} 2 & -1 \\ 1 & 3 \end{vmatrix} - (-1) \begin{vmatrix} -1 & -1 \\ -1 & 3 \end{vmatrix} + (-1) \begin{vmatrix} -1 & 2 \\ -1 & -1 \end{vmatrix} = 3*5 - 4 - 3 = 8$$

Таким чином граф має 8 кістякових дерев.

Теорема Келі. Кількість різних кістяків повного зв'язного неорієнтованого розміченого графа з n вершинами складає n^{n-2}.

Доведення. Матриця суміжності C і матриця ступенів D мають вигляд:

$$C = \begin{pmatrix} 0 & 1 & 1 & \dots & 1 & 1 \\ 1 & 0 & 1 & \dots & 1 & 1 \\ 1 & 1 & 0 & \dots & 1 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 1 & 1 & \dots & 0 & 1 \\ 1 & 1 & 1 & \dots & 1 & 0 \end{pmatrix}, D = \begin{pmatrix} n-1 & 0 & 0 & \dots & 0 & 0 \\ 0 & n-1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & n-1 & 0 \\ 0 & 0 & 0 & \dots & 0 & n-1 \end{pmatrix}$$

Звідси

$$K = D - C = \begin{pmatrix} n-1 & -1 & -1 & \dots & -1 & -1 \\ -1 & n-1 & -1 & \dots & -1 & -1 \\ -1 & -1 & -1 & \dots & -1 & -1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -1 & -1 & -1 & \dots & n-1 & -1 \\ -1 & -1 & -1 & \dots & -1 & n-1 \end{pmatrix}$$

Обчислимо визначник алгебраїчного доповнення K_{11}:

$$\det(K11) = \begin{vmatrix} n-1 & -1 & -1 & \dots & -1 & -1 \\ -1 & n-1 & -1 & \dots & -1 & -1 \\ -1 & -1 & n-1 & \dots & -1 & -1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -1 & -1 & -1 & \dots & n-1 & -1 \\ -1 & -1 & -1 & \dots & -1 & n-1 \end{vmatrix}$$

додамо до першого рядка визначника всі інші, отримаємо в першому рядку всі одиниці.

$$= \begin{vmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ -1 & n-1 & -1 & \dots & -1 & -1 \\ -1 & -1 & n-1 & \dots & -1 & -1 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -1 & -1 & -1 & \dots & n-1 & -1 \\ -1 & -1 & -1 & \dots & -1 & n-1 \end{vmatrix} =$$

додамо перший рядок до всіх інших рядків, отримаємо визначник трикутного вигляду

$$= \begin{vmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 0 & n & 0 & \dots & 0 & 0 \\ 0 & 0 & n & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & n & 0 \\ 0 & 0 & 0 & \dots & 0 & n \end{vmatrix} = n^{n-2},$$

що і потрібно було довести.

5.4 Коди Прюфера та кістякові дерева

Означення. Кодом Прюфера довжини n-2 називається послідовність з чисел від 1 до n з повтореннями.

Лема. Кількість кодів Прюфера довжини n-2 рівна n^{n-2}.

Теорема. Існує взаємно однозначна відповідність між кістяковими деревами для графа з n вершин і кодами Прюфера довжини n-2. По кожному дереву з n вершинами можна побудувати код Прюфера довжини n-2 і навпаки.

Наслідок. Кількість пронумеровані дерев з n вершин рівна n^{n-2}.

Псевдокод алгоритму побудови коду Прюфера по дереву виглядає так:

Prüfer(T)

1.Задати пустий код *K*.

2. Повторити n-2 рази:

3. Вибрати вершину *v* – лист дерева T з найменшим номером;

4. Додати номер єдиного сусіда *v* в послідовність коду Прюфера *K*;

5. Видалити вершину v з дерева T;

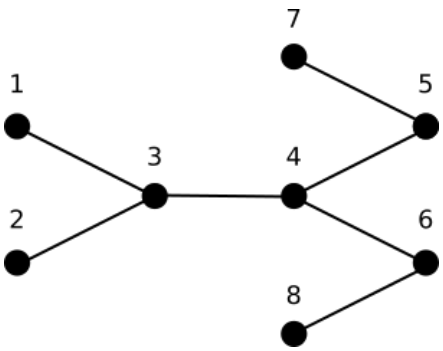
6. Вивести *K*

Нижче наведено код мовою C++ для побудови коду за деревом:

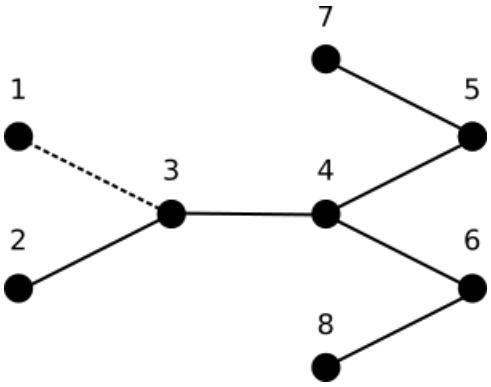
```
#include<iostream>
using namespace std;

int main(){
    int i, j, v, e, min, x;
    cout<<"Введіть кількість вершин в дереві: ";
    cin>>v;
    e = v-1;
    int edge[e][2], deg[v+1] = {0};
    for(i = 0; i < e; i++){
        cout<<"Введіть пару вершин для ребра
"<<i+1<<":";
        cout<<"\nV(1): ";
        cin>>edge[i][0];
        cout<<"V(2): ";
        cin>>edge[i][1];
        deg[edge[i][0]]++;
        deg[edge[i][1]]++;
    }
    cout<<"\nКод Прюфера: { ";
    for(i = 0; i < v-2; i++){
        min = 10000;
```

```
        for(j = 0; j < e; j++){
            if(deg[edge[j][0]] == 1){
                if(min > edge[j][0]){
                    min = edge[j][0];
                    x = j;
                }
            }
            if(deg[edge[j][1]] == 1){
                if(min > edge[j][1]){
                    min = edge[j][1];
                    x = j;
                }
            }
        }
        deg[edge[x][0]]--;
        deg[edge[x][1]]--;
        if(deg[edge[x][0]] == 0)
            cout<<edge[x][1]<<" ";
        else
            cout<<edge[x][0]<<" ";
    }
    cout<<"}";
    return 0;
}
```

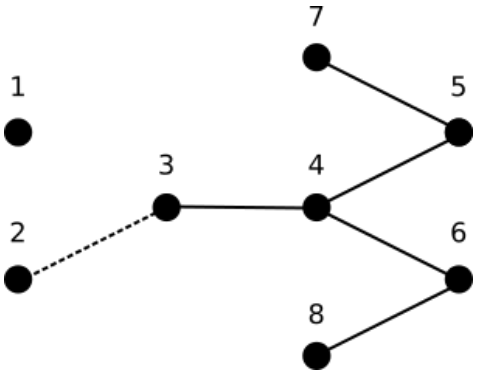
Приклад. Побудувати код Прюфера по дереву:



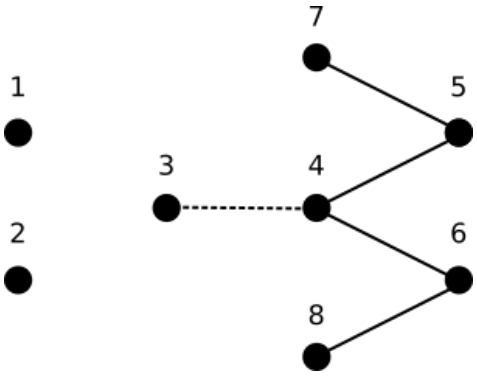
1. Лист з найменшим номером: $v = 1$, сусід: 3. Код (3).



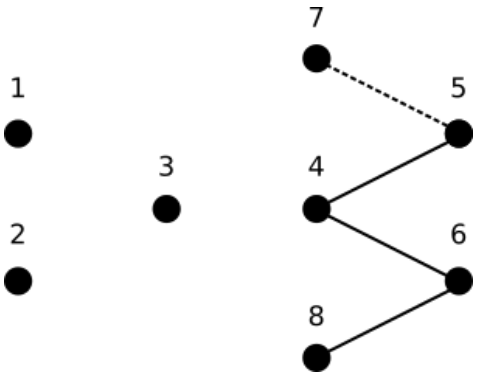
2. Лист з найменшим номером: $v = 2$, сусід: 3. Код (3, 3).



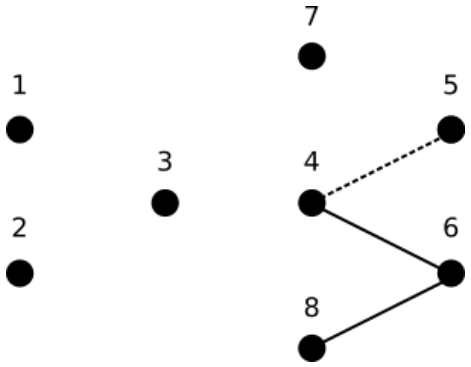
3. Лист з найменшим номером: $v = 3$, сусід: 4. Код (3, 3, 4).



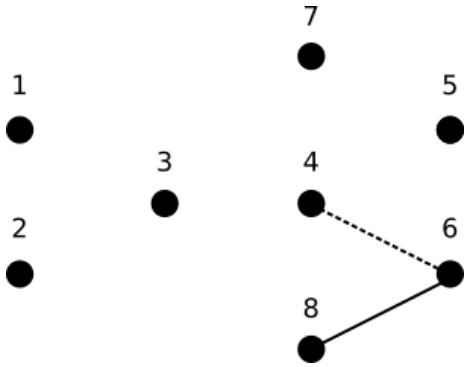
4. Лист з найменшим номером: $v = 7$, сусід: 5. Код (3, 3, 4, 5).



5. Лист з найменшим номером: $v = 5$, сусід: 4. Код (3, 3, 4, 5, 4).



6. Лист з найменшим номером: $v = 4$, сусід: 6. Код (3, 3, 4, 5, 4, 6).



Лишилося 2 вершини 6 и 8.

Результат: (3, 3, 4, 5, 4, 6).

Псевдокод алгоритму побудови дерева по коду Прюфера наведено нижче, при цьому дерево має *n* вершин, що пронумеровані від 1 до *n*.

TreeFromPrüfer(code)

1. $n \leftarrow |P| + 2$;
2. $V \leftarrow \{1, 2, \ldots, n\}$;
3. Для цілого *i* від 1 до *n*-2:
4. $v \leftarrow$ найменший элемент V, якого немає в P;
5. обє'днати вершину *v* з вершиною з номером p_i;
6. видалити *v* из V;
7. видалити элемент p_i из P, $P = (p_{i+1}, p_{i+2}, \ldots, p_{n-2})$;
8. Обє'днати дві останні вершини в V.

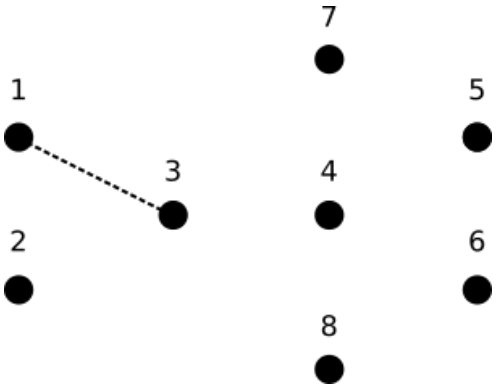
Нижче наведено код реалізації алгоритму побудови дерева за кодом Прюфера мовою C++.

```
#include<bits/stdc++.h>
```

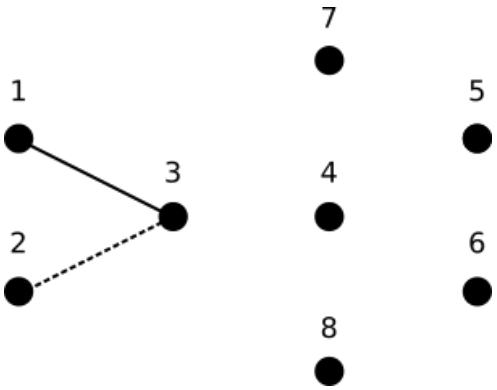
```
using namespace std;
void printTreeEdges(int prufer[], int m){
    int vertices = m + 2;
    int vertex_set[vertices];
    for (int i=0; i<vertices-2; i++)
       vertex_set[prufer[i]-1] += 1;
    cout<<"\nМножина ребер E(G) є:\n";
    int j = 0;
   for (int i=0; i<vertices-2; i++){
       for (j=0; j<vertices; j++)  {
           if (vertex_set[j] == 0) {
               vertex_set[j] = -1;
               cout << "(" << (j+1) << ","
                    << prufer[i] << ")  ";
               vertex_set[prufer[i]-1]--;
               break;
           }
       }
   }
   for (int i=0; i<vertices; i++ ) {
       if (vertex_set[i] == 0 && j == 0 ) {
           cout << "(" << (i+1) << ",";
           j++;
       }
       else if (vertex_set[i] == 0 && j == 1 )
           cout << (i+1) << ")\n";
   }
}

int main() {
   int prufer[] = {4, 1, 3, 4};
   int n = sizeof(prufer)/sizeof(prufer[0]);
   printTreeEdges(prufer, n);
   return 0;
}
```

Приклад. Інвертувати процес з попереднього прикладу, побудувати дерево по коду Прюфера (3, 3, 4, 5, 4, 6).

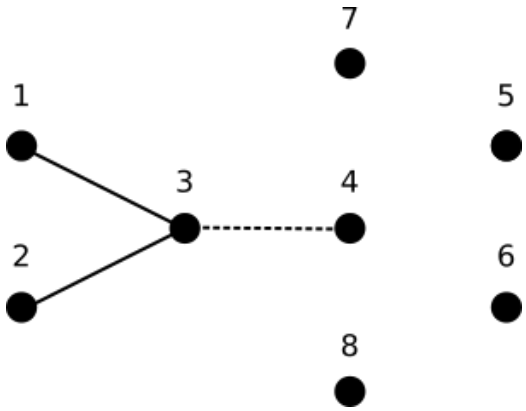
1. P = (3, 3, 4, 5, 4, 6), V = (1, 2, 3, 4, 5, 6, 7, 8). $v = 1$, додаєм ребро (1, 3):



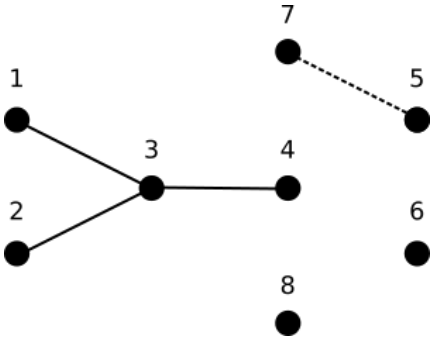
2. P = (3, 4, 5, 4, 6), V = (2, 3, 4, 5, 6, 7, 8). $v = 2$, додаємо ребро (2, 3):



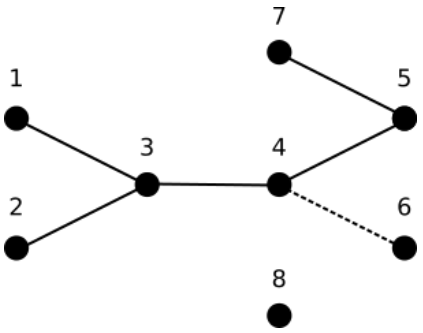
3. P = (4, 5, 4, 6), V = (3, 4, 5, 6, 7, 8). $v = 3$, додаємо ребро (3, 4):



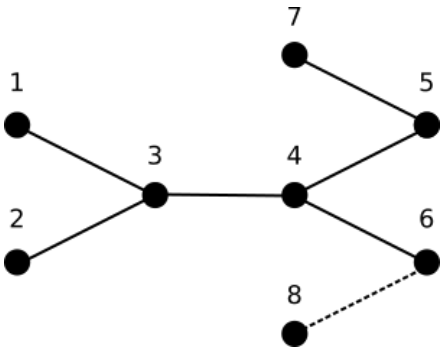
4. P = (5, 4, 6), V = (4, 5, 6, 7, 8). $v = 7$, додаємо ребро (7, 5):



5. P = (4, 6), V = (4, 5, 6, 8). v = 5, додаєм ребро (5, 4):



6. P = (6), V = (4, 6, 8). v = 4, додаєм ребро (4, 6):



7. P = (), V = (6, 8). V складається з 2 вершин, додаєм ребро (6, 8):

Результат: правильно побудоване дерево.

Питання для самоперевірки:

1. Що таке кістякове дерево?
2. Скільки їх можна нарахувати для повного зв'язного неорієнтованого розміченого граф?
3. Що таке код Прюфера.
4. Яка складність алгоритму Прима?
5. Яка складність алгоритму Крускала?

Розділ 6. ПОТОКИ В МЕРЕЖАХ

Для пошуку максимального потоку в графі з одним джерелом *s* і одним стоком *t* скористаємося алгоритмом Едмондса-Карпа. Ідея алгоритму спирається на побудову розрізів: множин дуг таких, що з'єднують дві множини вершин, що не перетинаються. Покладемо спочатку значення максимального потоку *Flow* рівним нулю. Шукаємо шлях з *s* в *t* пошуком в ширину. Якщо такого шляху немає, то закінчуємо алгоритм, повертаючи значення знайденого максимального потоку *Flow*. Інакше по знайденому шляху пропускаємо максимально можливий потік. Для цього шукаємо в цьому шляху ребро з найменшою пропускною здатністю *w*. Пропускаємо по шляху потік величини *w*, додаємо *w* до *Flow*. Далі слід перерахувати пропускні спроможності ребер на шляху з *s* в *t*.

Псевдокод алгоритму наведено нижче.

MaxFlow(G, *s*, *t*)

1. *Flow* = 0;
2. Поки (існує шлях з *s* в *t*, знайдений пошуком в ширину):
3. шукаємо в цьому шляху ребро з найменшою пропускною здатністю *w*;
4. *Flow* = *Flow* + *w*;
5. проводимо перерахунок пропускних здатностей ребер на шляху з *s* в *t*;
6. повернути *Flow*;

Код алгоритму написаний мовою C++ наведено нижче. Діють такі домовленості по змінним: максимально можливе число вершин в графі зберігаємо в MAX. Масив *m* містить матрицю суміжності, масиви *used* і *parent* будуть використовуватися при пошуку в ширину. Черга *q* використовується в пошуці в ширину.

Функція *Rebuild* здійснює перерахунок пропускних спроможностей ребер на шляху з *s* в *t*.

Функція *FindFlow* здійснює пошук максимальної пропускної здатності від *s* до *t* по шляху, знайденому пошуком в ширину.

```
#include <cstdio>
#include <memory>
#include <deque>
#include <limits>

using namespace std;
const int MAX = 20;
int m[MAX][MAX];
int used[MAX],parent[MAX];
int source,dest,n;
deque<int> q;


void bfs(int v)
{
  int i,First;
  q.push_front(v);
  while(!q.empty())
  {
     First = q[0];
     if (First == dest) return;
     q.pop_front();
     used[First] = 1;
     for(i=0;i<n;i++)
     if ((m[First][i] > 0) && (!used[i]))
     {
         parent[i] = First;
         q.push_back(i);
     }
  }
}


void Rebuild(int k, int flow){
  if (parent[k] == -1) return;
  Rebuild(parent[k],flow);
  m[parent[k]][k] -= flow;
  m[k][parent[k]] += flow;
}


int min(int i, int j){

     return (i < j) ? i : j;

}
```

```
int FindFlow(int k){
  int x;
  if (parent[k] == -1) return INT_MAX;
  x = FindFlow(parent[k]);
  return min(x,m[parent[k]][k]);
}


void main(void)
{
  int a,b,c;
  int MaxFlow,flow;

  memset(m,0,sizeof(m));
  scanf("%d %d %d\n",&n,&source,&dest);
  while (scanf("%d %d %d\n",&a,&b,&c)==3) m[a][b]=c;

  MaxFlow = 0;
  while (1){
    q.clear();
    memset(parent,-1,sizeof(parent));
    memset(used,0,sizeof(used));
    bfs(source);
    flow = FindFlow(dest);
    if (flow == INT_MAX) break;
    MaxFlow += flow;
    Rebuild(dest,flow);
  }
  printf("Максимальний потік: %d\n",MaxFlow);
}
```

Алгоритмічні властивості: часова складність алгоритму складає $O(|V||E|^2)$

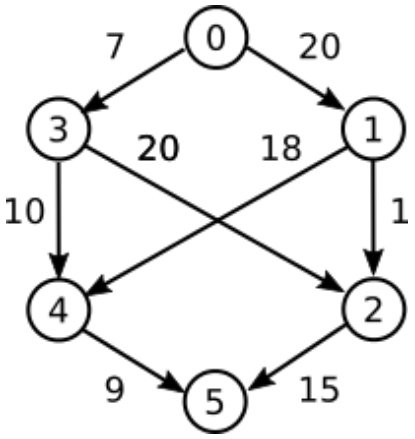


Рисунок 24. Мережа для потоку.

Приклад. Знайти величину максимального потоку в графі між вершинами *s* і *t*, для графу зображеного на Рисунку 24.

Вхід. Перший рядок містить кількість вершин *n* в графі, а також номери вершин *s* і *t*. Кожний наступний рядок містить три цілих числа *u*, *v* і *w*, описуючи ребро (*u*, *v*) графу з пропускною спроможністю *w*.

Вихід. Вивести величину максимального потоку в графі між вершинами *s* і *t* в форматі, представленому нижче.

Приклад входу	***Приклад виходу***
6 0 5 0 1 20 1 2 1 2 5 15 0 3 7 3 2 20 1 4 18 4 5 9 3 4 10	Максимальний потік: 17

Величина максимального потоку між вершинами 0 і 5 дорівнює 17. Розподіл потоку по ребрах представлено нижче, на Рисунку 25. Біля кожного ребра стоїть позначка «завантаження ребра/пропускна здатність ребра». Ребра, які входять до мінімального розріз, виділені жирним.

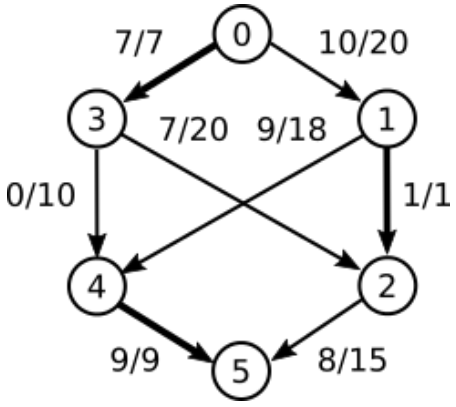


Рисунок 25. Потік та мінімальний розріз.

Алгоритм Едмондса-Карпа можна вживати для різних задач. Наприклад, розглянемо таку задачу.

Ейлерів шлях. Є граф, який має як орієнтовані, так і неорієнтовані ребра. Чи можна орієнтувати неорієнтовані ребра так, щоб отриманий граф мав Ейлерів цикл?

Для розв'язання цієї задачі перенумеруємо неорієнтовані ребра. Побудуємо двочастковий граф. Одній множині вершин відповідають вершини графа V_i, іншій – неорієнтовані ребра e_i. Від кожної вершини e_i проведемо ребра до тих вершин V_j, до яких вони можуть бути спрямовані. Пропускні спроможності цих ребер ставимо рівними 1.

Для існування ейлерову циклу в орієнтованому графі необхідно, щоб для кожної вершини кількість вхідних ребер дорівнювала кількості вихідних. Біля кожної вершини вихідного графа в дужках вкажем мітки m_i, рівні кількості неорієнтованих ребер, які повинні входити в цю вершину.

У дводольному графі додамо дві вершини – джерело S і стік D. Кожне ребро може бути орієнтоване тільки в одну сторону, тому пропускні спроможності ребер, що з'єднують вершини S і e_i, встановимо рівними 1. У вершину V_i має бути направлено m_i ребер, тому пропускні спроможності ребер, що з'єднують вершини Vi і D, встановимо рівними m_i.

Шукаємо максимальний потік в побудованому графі між вершинами S і D. Якщо його величина дорівнює кількості неорієнтованих ребер, то ейлерів цикл існує.

Розглянемо побудови на прикладі. На Рисунку 26 зображено граф, для якого ми спробуємо побудувати ейлерів цикл.

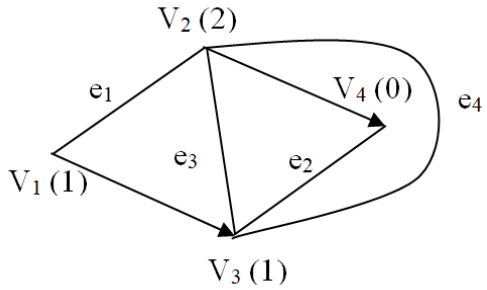


Рисунок 26.

На Рисунку 27 зображено необхідний двочастковий граф. Наприклад, розглянемо вершину V_2. З неї виходить 4 ребра, одне з яких орієнтоване. Для існування ейлерову циклу в вершину V_2 має входити 2 ребра і 2 з неї виходити. Тобто два неорієнтованих ребра необхідно орієнтувати так, щоб вони входили в V_2. Вершина V_3 має мітку 1, так як тільки одне з неорієнтованих ребер має входити в V_3.

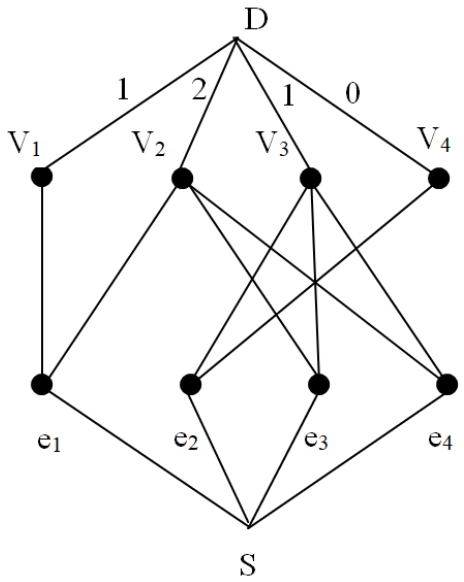


Рисунок 27. Двочастковий граф для орієнтування ребер.

Питання для самоперевірки:

1. Яка складність алгоритму Едмондса-Карпа?
2. Що таке розріз?

СПИСОК ЛІТЕРАТУРИ

1. Вступ до алгоритмів, Кормен, Томас; Лейзерсон, Чарльз; Рівест, Рональд; Стайн, Кліфорд. К.І.С. 2019 - 1296 с.. ISBN 978-617-684-239-2.
2. Алгоритмы построение и аналіз, Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К., – Москва, Санкт-Петербург, Киев, 2005 – 1292 с.
3. Искуство программирования, т3.Дональд Э. Кнут – Москва, Санкт-Петербург, Киев, 2004 - 712 с.
4. Практика и теория программирования, Книга 2. Винокуров Н.А., Ворожцов А.В., – М: Физматкнига, 2008 - 288 с.
5. Дискретна математика у прикладах і задачах. Р.М. Трохимчук, М.С. Нікітченко – Київ – 248 с. Електронний ресурс: http://csc.knu.ua/media/filer_public/89/10/89101127-5400-4d61-9840-7eab32caddab/discrete_mathematics.pdf
6. Теорія графів у задачах, Карнаух Т.О., Ставровський А.Б.— Київ:-90с. Електронний ресурс: http://www.cyb.univ.kiev.ua/library/books/karnaukh-23.pdf
7. Вступ до програмування мовою С++. Структури даних: навч. посіб. Р. А. Веклич, Т. О. Карнаух, А. Б. Ставровський – К. : ВПЦ "Київський університет", 2018. – 99 с. Електронний ресурс: http://csc.knu.ua/media/filer_public/69/0f/690f16f1-513d-4538-b438-82f1d15cafce/kkpsverstka02.pdf
8. Програмування. Поглиблений курс, Зубенко В.В., Омельчук Л.Л.— Київ: Видавничо-поліграфічний центр "Київський університет", 2011. — 623 с. Електронний ресурс: http://csc.knu.ua/uk/library/books/zubenko-omelchuk-2.pdf

Зміст

Передмова 1
Розділ 1. БАЗОВІ ЗАДАЧІ 2
1.1 Основні визначення 2
1.2 Пошук в глибину 5
1.2.1 Класифікація ребер графу 10
1.2.2 Властивості пошуку в глибину 11
1.3 Пошук в ширину 13
1.3.1 Пошук в ширину з двох та більше джерел 17
Розділ 2 ЗВ'ЯЗНІСТЬ 18
2.1 Побудова та підрахунок зв'язних компонент 18
2.2 Сильно зв'язні компоненти 22
Розділ 3. НАЙКОРОТШІ ШЛЯХИ 28
3.1 Алгоритм Дейкстри для пошуку шляхів у графі 28
3.2 Пошук найкоротших шляхів між парами вершин 34
3.3 Транзитивне замикання 37
Розділ 4. ЦИКЛИ 40
4.1 Ейлерів цикл 40
4.2 Гамільтонів цикл 42
Розділ 5. КІСТЯКОВІ ДЕРЕВА 46
5.1 Алгоритм Крускала 46
5.2 Алгоритм Прима 49
5.3 Матрична формула Кірхгофа, теорема Келі 52
5.4 Коди Прюфера та кістякові дерева 54
Розділ 6. ПОТОКИ В МЕРЕЖАХ 62
СПИСОК ЛІТЕРАТУРИ 68

Навчальне видання

Тарануха Володимир Юрійович

ЗАДАЧІ НА ГРАФАХ ДЛЯ КУРСУ АЛГОРИМІКА

Навчальний посібник