# Графи. Частина 2

Алгоритми і структури даних
М2 Лекція 5

Михайло Пастула
2019

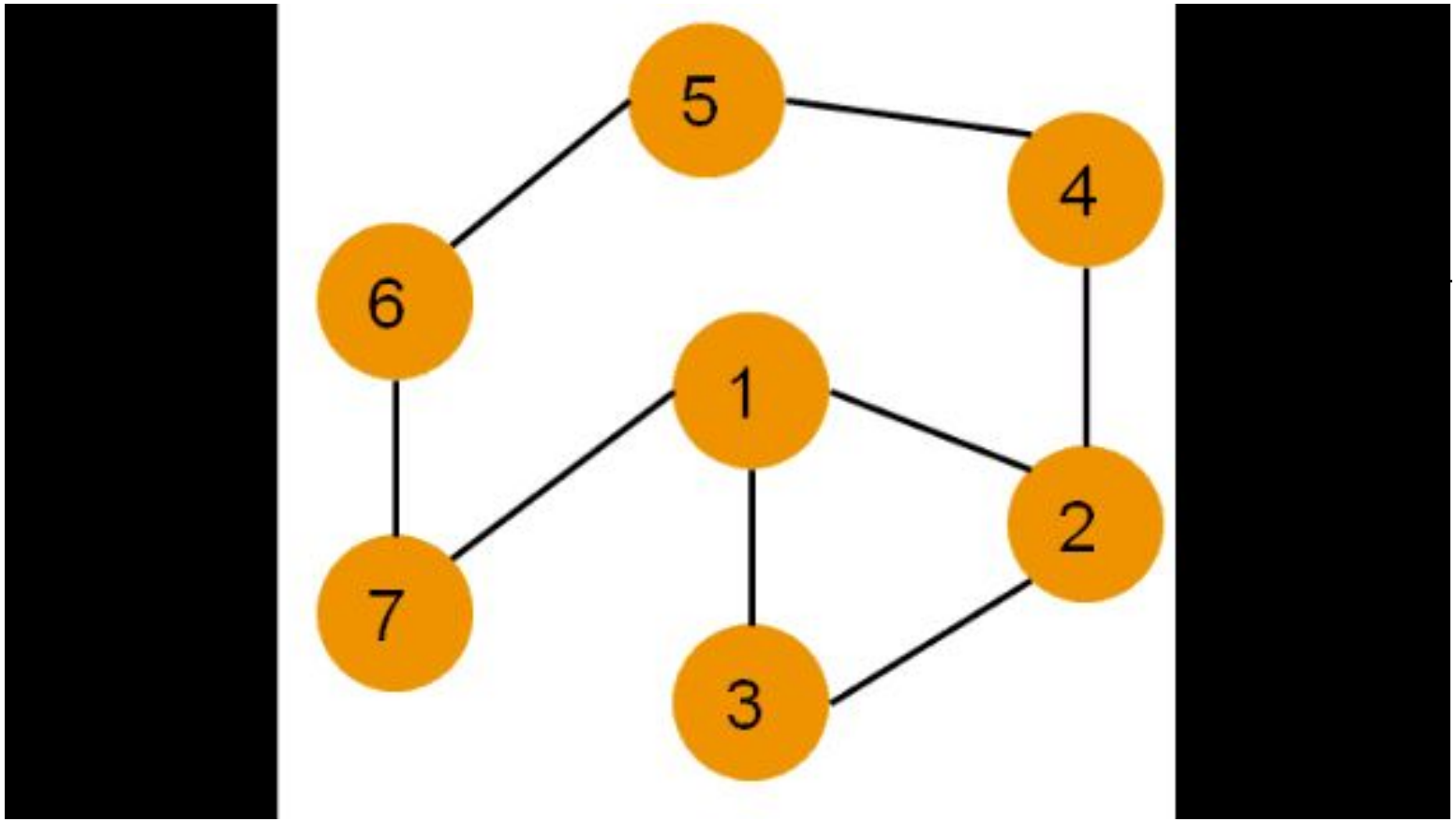
# I.Пошук в глибину

Пошук в глибину - це алгоритм обходу вершин графа.

Пошук в ширину проводиться симетрично (вершини графа проглядалися за рівнями). Пошук в глибину передбачає просування вглиб до тих пір, поки це можливо. Неможливість просування означає, що наступним кроком буде перехід на останній, який має кілька варіантів руху (один з яких досліджений повністю), раніше відвіданий вузол (вершина).

Відсутність останнього свідчить про одну з двох можливих ситуацій:

- всі вершини графа вже переглянуті,
- переглянуті вершини доступні з вершини, взятої в якості початкової, але не всі (незв'язні і орієнтовані графи допускають останній варіант).




Кожна вершина може перебувати в одному з 3 станів:

- 0 - помаранчевий - невиявлення вершина;
- 1 - зелений - виявлена, але не відвідана вершина;
- 2 - сірий - оброблена вершина;

Фіолетовий - розглянута вершина.

# Програмна реалізація

```
#include <iostream>
using namespace std;
int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 },// матриця суміжності
{ 1, 0, 1, 1, 0, 0, 0 },
{ 1, 1, 0, 0, 0, 0, 0 },
{ 0, 1, 0, 0, 1, 0, 0 },
{ 0, 0, 0, 1, 0, 1, 0 },
{ 0, 0, 0, 0, 1, 0, 1 },
{ 1, 0, 0, 0, 0, 1, 0 } };
int nodes[7]; // вершини графа
int main()
{
  for (int i = 0; i < 7; i++) // спочатку всі вершини рівні 0
    nodes[i] = 0;
  search(0, 7);
  cin.get();
  return 0;
}
```

```
void search(int st, int n)
{
  int r;
  cout << st + 1 << " ";
  nodes[st] = 1;
  for (r = 0; r < n; r++)
    if ((mas[st][r] != 0) && (nodes
[r] == 0))
      search(r, n);
}
```


C:\Projects\MyProgram\Debug\MyProgram.exe
1 2 3 4 5 6 7

# Пошук шляху в графі

```
#include <iostream>
#include <stack> // стек
using namespace std;
struct Edge {
  int begin;
  int end;
};
int main()
{
  stack<int> Stack;
  stack<Edge> Edges;
  int req;
  Edge e;
  //Уведення таблиці суміжності графа та створення масиву вершин, з їх початковим зануленням
```

```
cout << "N = ";
  cin >> req;
  req--;
  Stack.push(0); // заносимо в стек першу вершину
  while (!Stack.empty()){ // поки стек не пустий
    int node = Stack.top(); // дістаємо вершину
    Stack.pop();
    if (nodes[node] == 2) continue;
    nodes[node] = 2; // відмічаємо її як пройдену
    for (int j = 6; j >= 0; j--)
    { // переглядаємо усі суміжні до неї вершини
      if (mas[node][j] == 1 && nodes[j] != 2)
      { //якщо вершина суміжна і не знайдена досі
        Stack.push(j); // додаємо її в стек
        nodes[j] = 1; // відмічаємо вершину як пройдешу
        e.begin = node; e.end = j;
        Edges.push(e);
        if (node == req) break;
      }
    }
    cout << node + 1 << endl; // виводимо номер вершини
  }
```

```
cout << «Шлях до вершини " << req + 1 << endl;
  cout << req + 1;
  while (!Edges.empty())
  {
    e = Edges.top();
    Edges.pop();
    if (e.end == req)
    {
      req = e.begin;
      cout << " <- " << req + 1;
    }
  }
  cin.get(); cin.get();
  return 0;
}
```

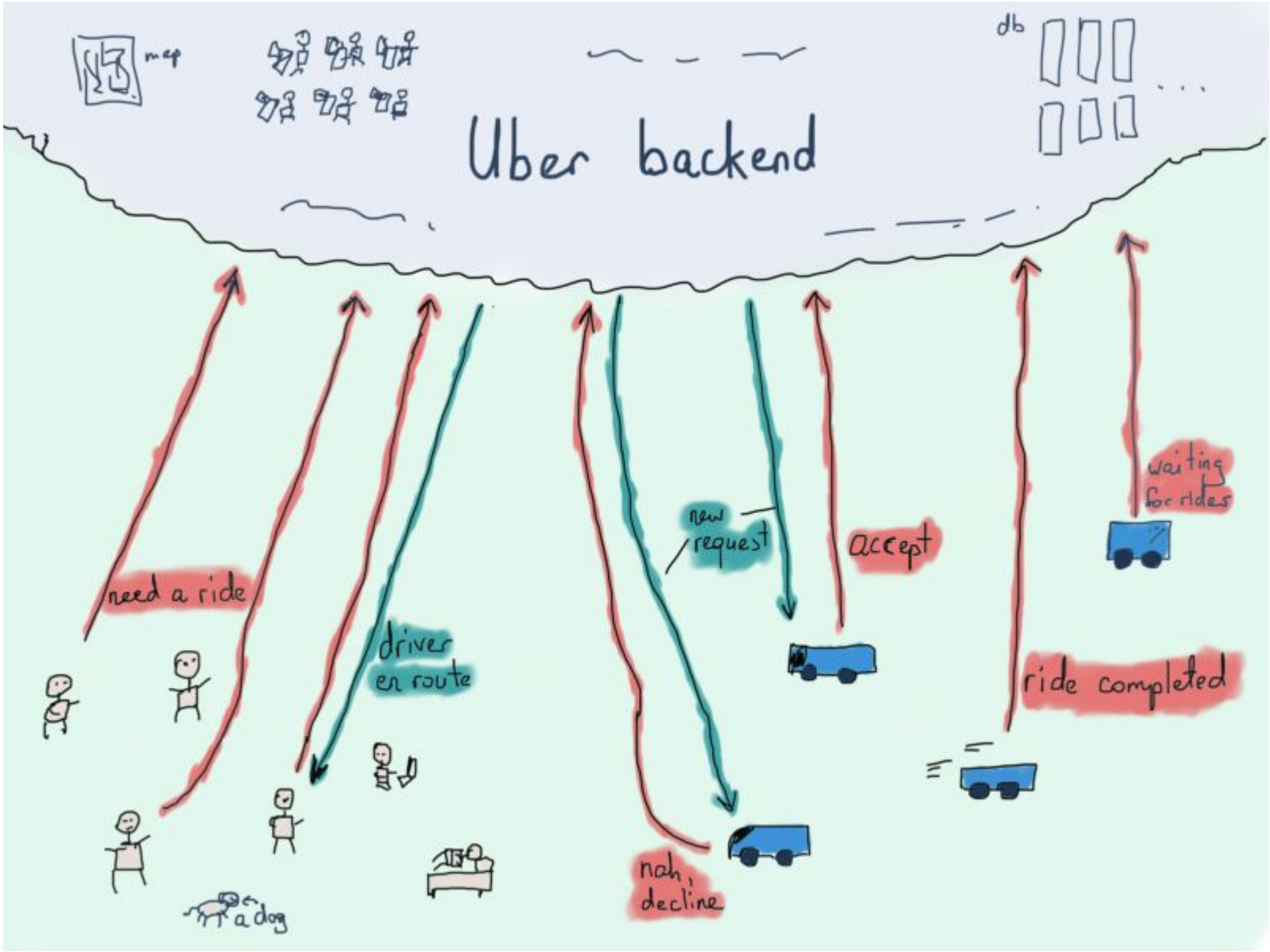
# Застосуваня пошуку в глибину

- Пошук будь-якого шляху в графі.
- Пошук лексикографічно першого шляху в графі.
- Перевірка, чи є одна вершина дерева предком інший.
- Пошук найменшого спільного предка.
- Топологічна сортування.
- Пошук компонент зв'язності.
- Алгоритм пошуку в глибину працює як на орієнтованих, так і на неорієнтованих графах.
- Застосовність алгоритму залежить від конкретного завдання.

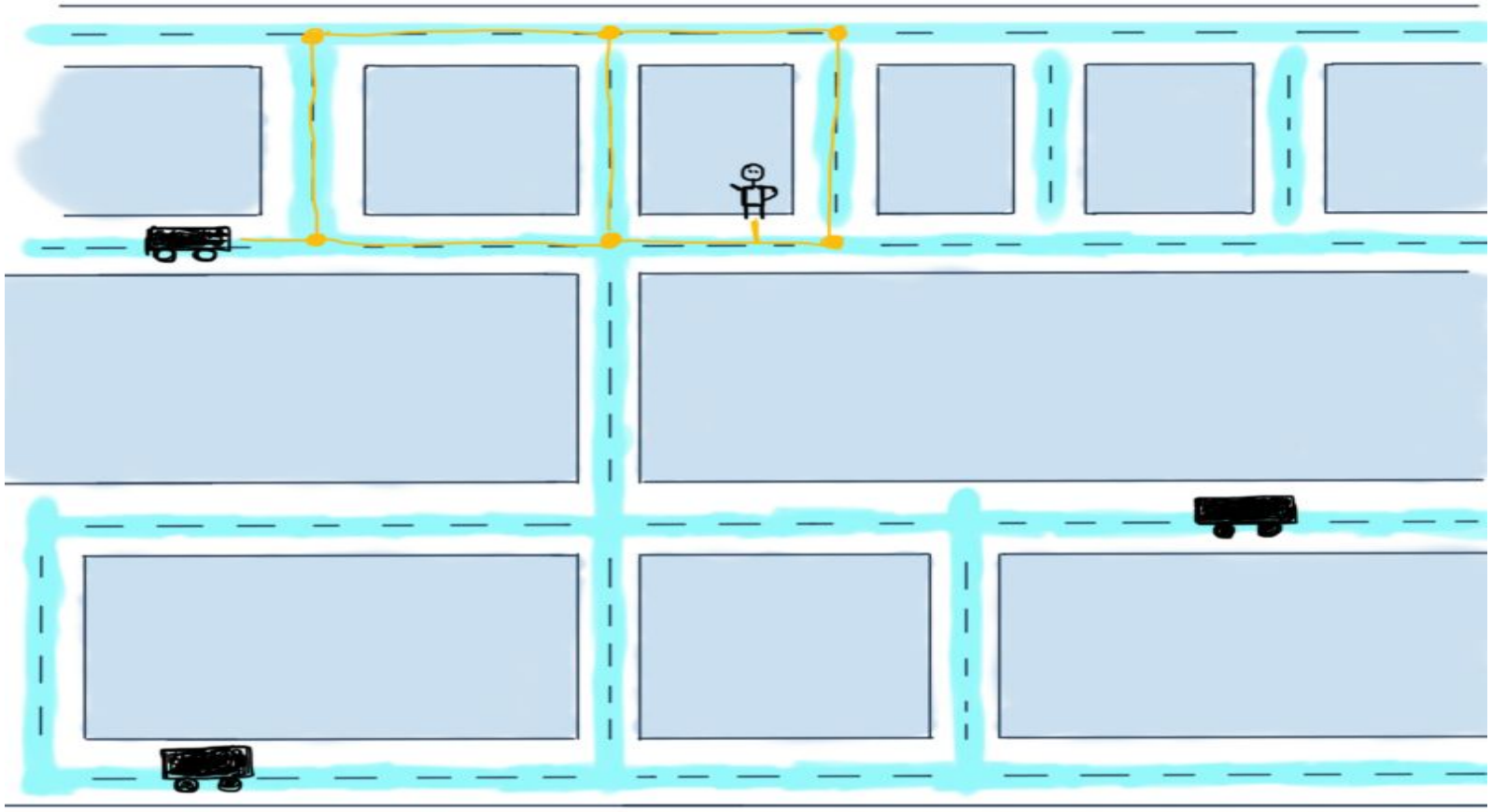
# II.Алгоритм Дейкстри

З його 50 мільйонами користувачів і 7 мільйонами водіїв (джерело), одна з найважливіших речей, яка має вирішальне значення для нормального функціонування Uber - це здатність ефективно поєднувати водіїв з користувачами. Проблема починається з розташування.

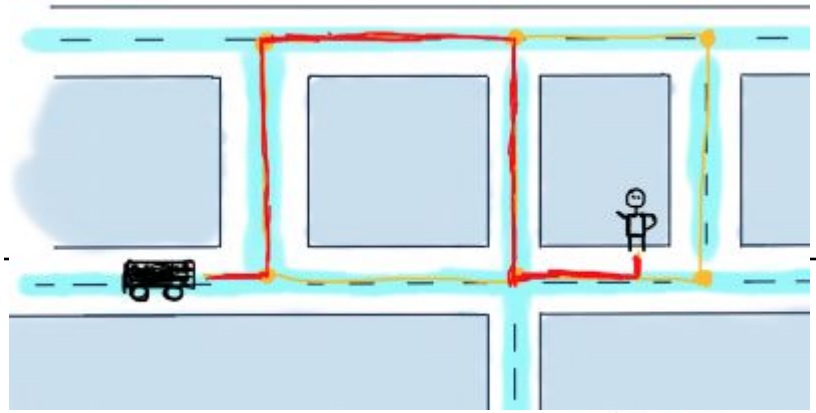
Серверна частина повинна обробляти мільйони призначених для користувача запитів, відправляючи кожен із запитів одному або декільком водіям поблизу. Хоч простіше і іноді навіть раціональніше відправляти запит користувача всім водіями, які перебувають поблизу, все ж буде потрібно попередня обробка.

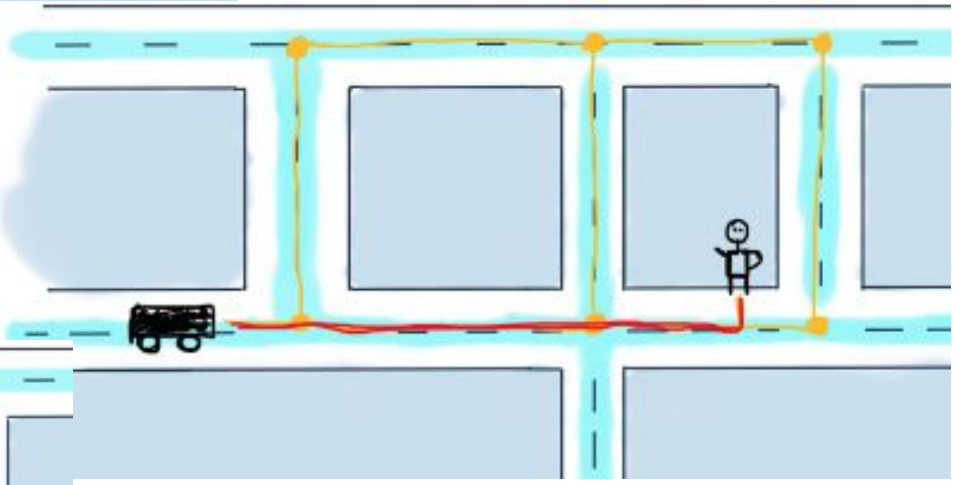

map
Uber backend
db
need a ride
driver en route
new request
accept
waiting for rides
ride completed
nah, decline
a dog

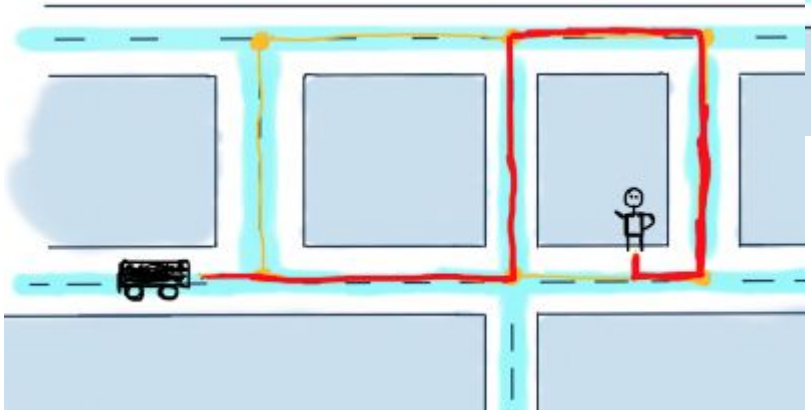
Крім обробки вхідних запитів і знаходження області розташування на основі координат користувача, а потім знаходження водіїв з найближчими координатами, нам також потрібно знайти правильного водія для поїздки. Щоб уникнути обробки геопросторових запитів (отримання прилеглих автомобілів шляхом порівняння їх поточних координат з координатами користувача), припустимо, у нас вже є сегмент карти з користувачем і декількома автомобілями поблизу.



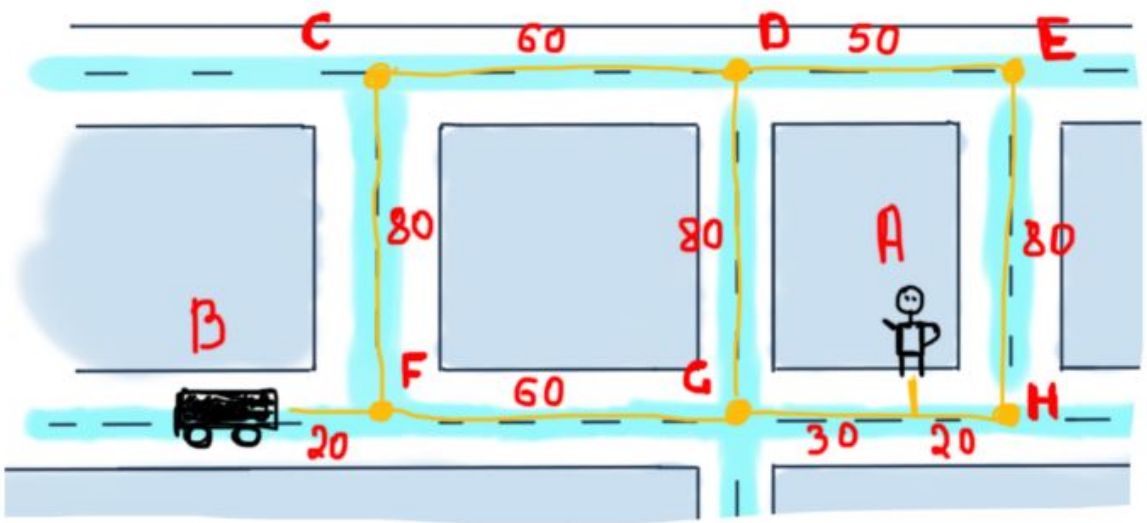
Можливі шляхи від автомобіля до користувача позначені жовтим. Завдання полягає в тому, щоб розрахувати мінімальну відстань між автомобілем та користувачем, іншими словами, знайти найкоротший шлях між ними.

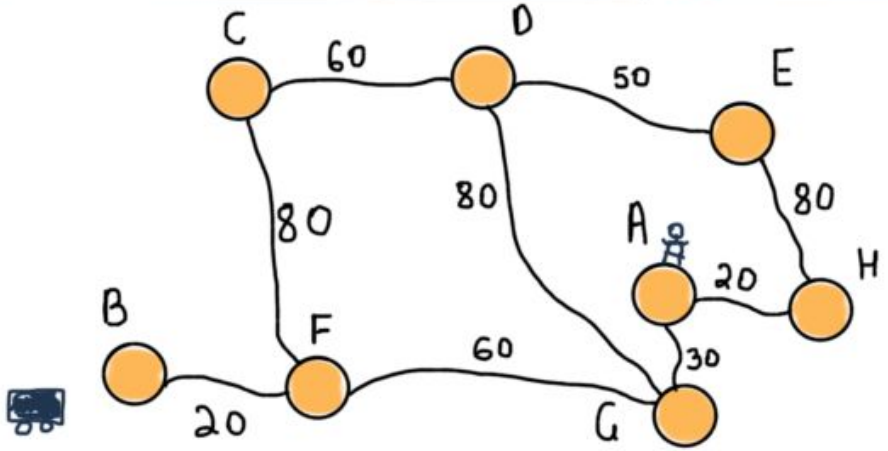






Уявімо цей сегмент у вигляді графа:



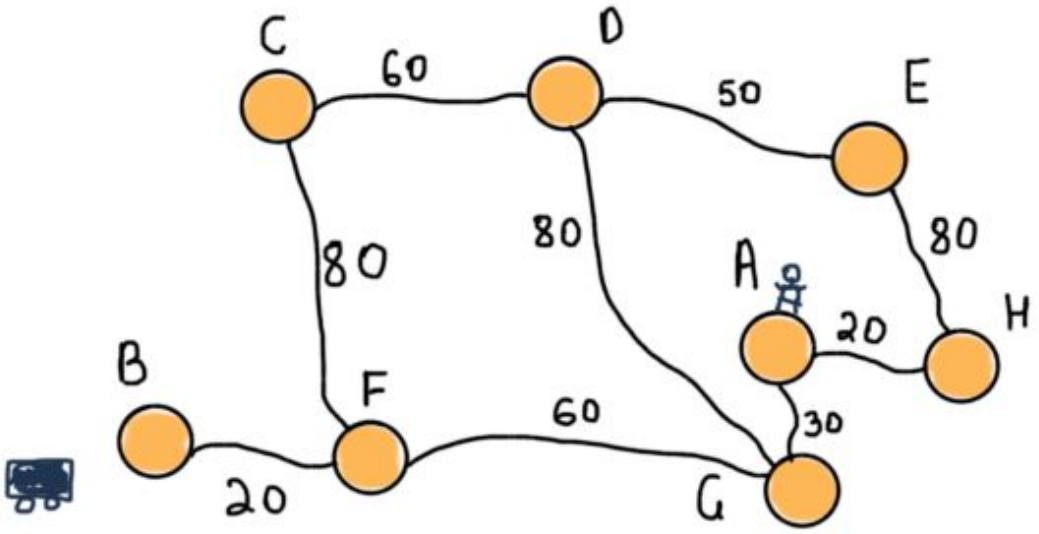



# Алгоритм дій

1. Відзначте всі вершини як ще невідвідані. Створіть структуру всіх невідвіданих вершин.
2. Дайте кожній вершині значення відстань до цієї вершини. Для першої вершині надайте нуль.
3. Для поточної вершини розгляньте невідвідані сусідні вершини і обчисліть відстані до кожної з урахуванням поточної вершини. Порівняйте нове обчислене відстань з поточним призначеним значенням і призначте менше. *(Наприклад, якщо поточна вершина A має вагу 6, а ребро, що пов'язує її з сусідом B, має довжину 2, то відстань до B через A буде 6 + 2 = 8.)*

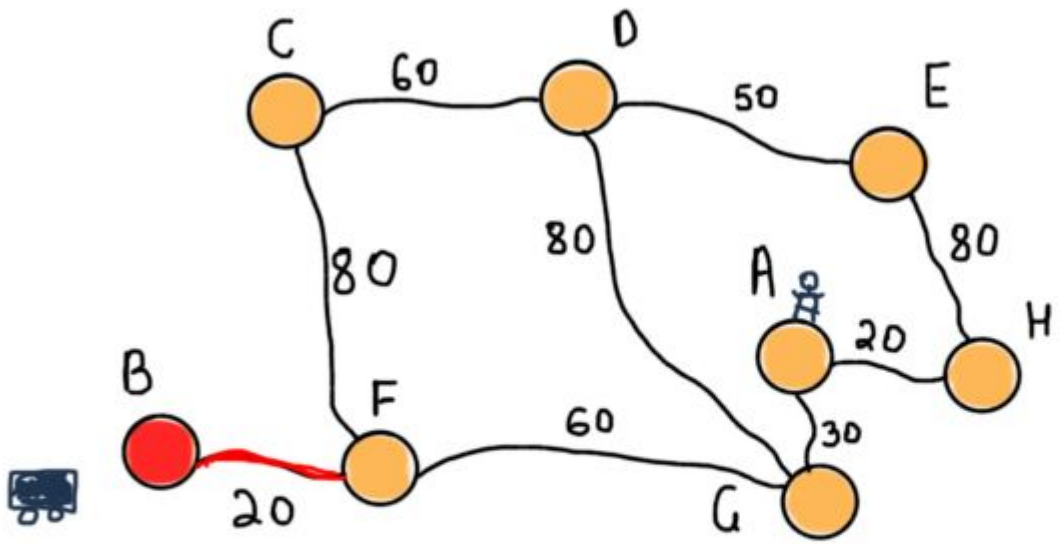
# Алгоритм дій

4. Коли ми закінчимо розглядати всіх сусідів поточної вершини, відзначте поточну вершину як відвідану і видаліть її зі структури невідвіданих вершин. Ця вершина більше ніколи не буде задіяна.
5. Якщо вершина призначення була відзначена як відвіданих (при плануванні маршруту між двома певними вершинами), зупиніться. Алгоритм завершено.
6. В іншому випадку виберіть невідвіданих вершину, зазначену найменшим попередніми відстанню, встановіть її в якості нової поточної вершини і поверніться до кроку 3



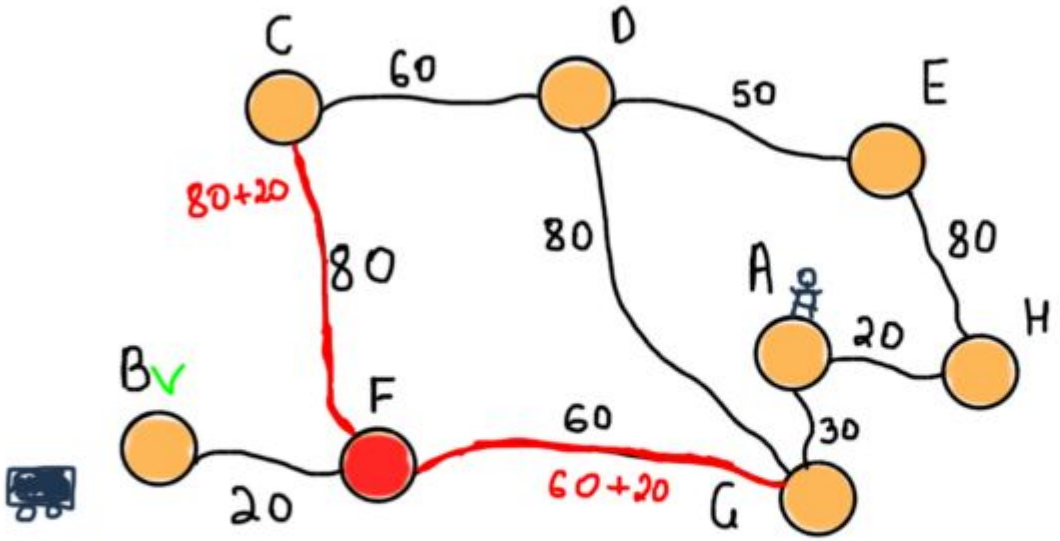

| Vertex | Dist | Prev |
|---|---|---|
| B | 0 | |
| C | ∞ | |
| F | ∞ | |
| D | ∞ | |
| G | ∞ | |
| A | ∞ | |
| E | ∞ | |
| H | ∞ | |

Unvisited : [B, C, F, D, G, A, E, H]



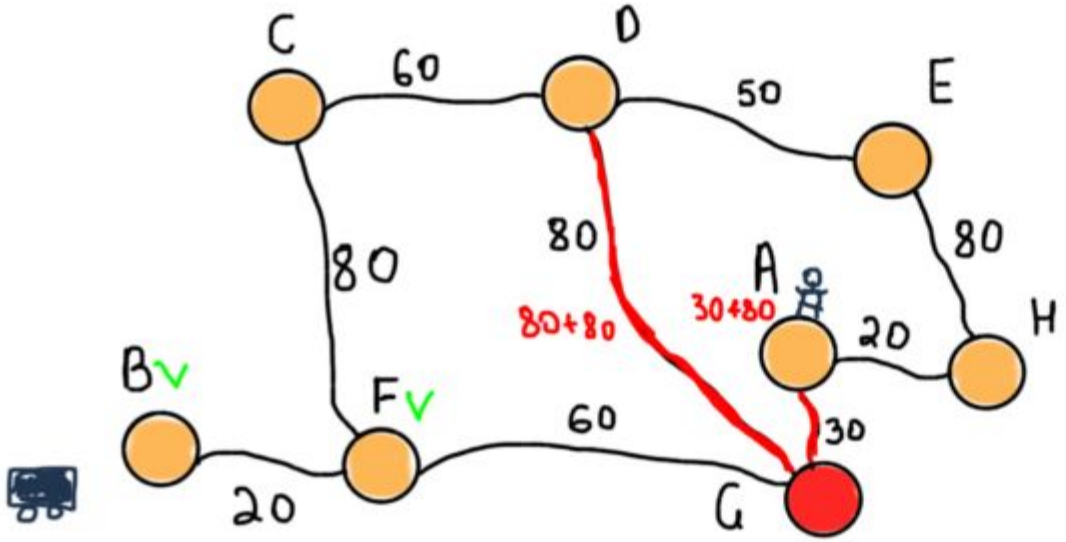

| Vertex | Dist | Prev |
|---|---|---|
| B | 0 | |
| C | ∞ | |
| F | 20 | B |
| D | ∞ | |
| G | ∞ | |
| A | ∞ | |
| E | ∞ | |
| H | ∞ | |

Unvisited : [B, C, F, D, G, A, E, H]



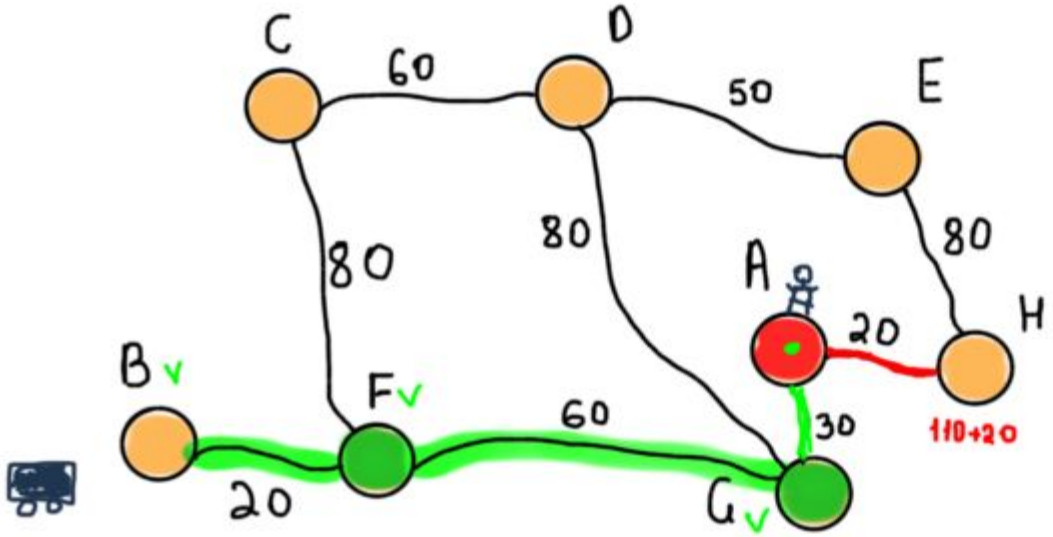

| Vertex | Dist | Prev |
|---|---|---|
| B | 0 | |
| C | 100 | F |
| F | 20 | B |
| D | ∞ | |
| G | 80 | F |
| A | ∞ | |
| E | ∞ | |
| H | ∞ | |

Unvisited : [~~B~~, C, F, D, G, A, E, H]



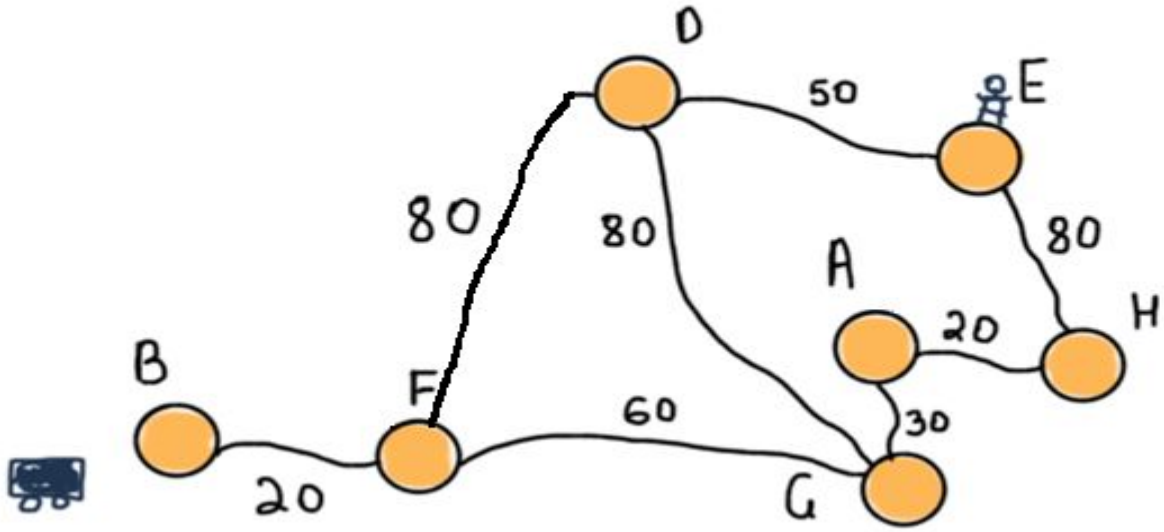

| Vertex | Dist | Prev |
|---|---|---|
| B | 0 | |
| C | 100 | F |
| F | 20 | B |
| D | 160 | G |
| G | 80 | F |
| A | 110 | G |
| E | ∞ | |
| H | ∞ | |

Unvisited : [~~B~~, C, ~~F~~, D, G, A, E, H]




| Vertex | Dist | Prev |
|---|---|---|
| B | 0 | |
| C | 100 | F |
| F | 20 | B |
| D | 160 | G |
| G | 80 | F |
| A | 110 | G |
| E | ∞ | |
| H | ∞ | |

Unvisited : [~~B~~, C, ~~F~~, D, ~~G~~, A, E, H]


D
E
50
80
80
80
A
20
H
B
F
60
30
20
G

# Програмна реалізація

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 6
int main(){
  int a[SIZE][SIZE]; // матриця суміжності
  int d[SIZE]; // мінімальна відстань
  int v[SIZE]; // відвідані вершини
  int temp, minindex, min;
  int begin_index = 0;
  // ініціалізація матриці відстаней
  for (int i = 0; i<SIZE; i++){
    a[i][i] = 0;
    for (int j = i + 1; j<SIZE; j++) {
      printf("Введіть відстань %d - %d: ", i + 1, j + 1);
      scanf("%d", &temp);
      a[i][j] = temp;
      a[j][i] = temp;
    }
  }
```

```
  for (int i = 0; i<SIZE; i++){
    for (int j = 0; j<SIZE; j++)
      printf("%5d ", a[i][j]);
    printf("\n");
  }
  for (int i = 0; i<SIZE; i++){
    d[i] = 10000;
    v[i] = 1;
  }
  d[begin_index] = 0;
  // Крок алгоритму
  do {
    minindex = 10000;
    min = 10000;
    for (int i = 0; i<SIZE; i++)
    { // якщо вешину ще не найшли і її вага менше min
      if ((v[i] == 1) && (d[i]<min))
      { //перевизначаємо значення
        min = d[i];
        minindex = i;
      }
    }
```

```
// Додаємо знайдену мінімальну вагу до поточної ваги вершини
// і порівнюємо з поточною мінімальною вагою
    if (minindex != 10000)
    {
      for (int i = 0; i<SIZE; i++)
      {
        if (a[minindex][i] > 0)
        {
          temp = min + a[minindex][i];
          if (temp < d[i])
          {
            d[i] = temp;
          }
        }
      }
      v[minindex] = 0;
    }
  } while (minindex < 10000);
printf("\nНайкоротші відстані до вершин: \n");
  for (int i = 0; i<SIZE; i++)
    printf("%5d ", d[i]);
}
```

# III.Алгоритм Флойда-Уоршелла

**Алгоритм Флойда-Воршелла** використовується для розв'язання задачі про найкоротший шлях у зваженому графі з додатними або від'ємними вагами ребер (але без від'ємнозначних циклів). При звичайній реалізації алгоритм видасть довжини (сумарні ваги) найкоротших шляхів між *всіма* парами вершин, хоча він не видасть інформацію про самі шляхи.

Цей алгоритм був одночасно опубліковано в статтях Роберта Флойда (Robert Floyd) і Стівена Уоршелла (Stephen Warshall) в 1962 р, хоча в 1959 р Бернард Рой (Bernard Roy) опублікував практично такий же алгоритм, але це пройшло повз увагу.

Алгоритм Воршелла порівнює всі можливі шляхи в графі між кожною парою вершин. Він виконується за $\Theta(|V|^3)$ порівнянь. Це доволі примітивно, враховуючи, що в графі може бути до $\Omega(|V|^2)$ ребер, і кожну комбінацію буде перевірено. Він виконує це шляхом поступового поліпшення оцінки по найкоротшому шляху між двома вершинами, поки оцінка не стає оптимальною.

Розгляньмо граф *G* з ребрами *V*, пронумерованими від 1 до N. Крім того розгляньмо функцію shortestPath(*i*, *j*, *k*), яка повертає найкоротший шлях від *i* до *j*, використовуючи вершини з множини {1,2,…,*k*} як внутрішні у шляху. Тепер, маючи таку функцію, нам потрібно знайти найкоротший шлях від кожного *i* до кожного *j*, використовуючи тільки вершини від 1 до *k* + 1.

Для кожної з цих пар вершин, найкоротший шлях може бути або (1)- шлях, у якому є тільки вершини з множини {1, …, *k*}, або (2)-шлях, який проходить від *i* до *k* + 1 а потім від*k* + 1 до *j*.Найкоротший шлях від *i* to *j* that only uses vertices 1 через *k* визначається функцією shortestPath(*i*, *j*, *k*), і якщо є коротший шлях від*i* до (*k* + 1 до *j*), то довжина цього шляху буде сумою(конкатенацією) найкоротшого шляху від*i* до *k* + 1 (використовуючи вершини{1, …, *k*}) і найкоротший шлях від *k* + 1 до *j* (також використовуючи вершини з {1, …, *k*}).

w(i,j)— це вага ребра між *i* та *j*.Можна визначити shortestPath(*i*, *j*, *k* + 1) наступною рекурсивною формулою база:

$$\text{shortestPath}(i, j, 0) = w(i, j)$$

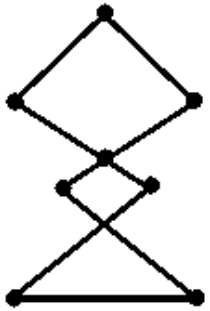
Рекурсивна частина:

$$\text{shortestPath}(i, j, k+1) = \min(\text{shortestPath}(i, j, k), \text{shortestPath}(i, k+1, k) + \text{shortestPath}(k+1, j, k))$$

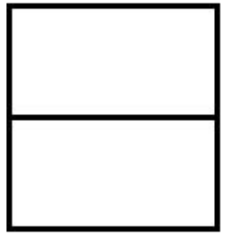
Ця формула є основною частиною алгоритму Флойда-Воршелла. Алгоритм спочатку обчислює shortestPath(*i*, *j*, *k*) для всіх пар(*i*, *j*) де *k* = 1, потім *k* = 2, і т. д. Цей процес продовжується, поки *k* = *N*, і поки не знайдено всі найкоротші шляхи для пар (*i*, *j*). Псевдокод для цієї версії алгоритму:

```
нехай |V| × |V| масив мінімальних відстаней
ініціалізований як ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // вага шляху (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
        if dist[i][j] > dist[i][k] + dist[k][j]
            dist[i][j] ← dist[i][k] + dist[k][j]
        end if
```

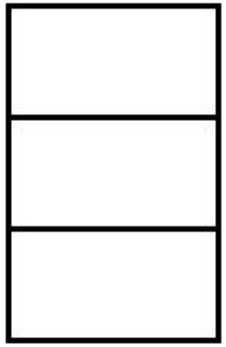
# IV. Ейлерів Граф



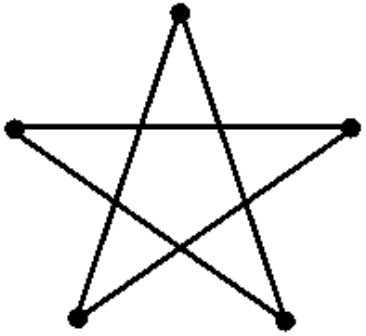
а



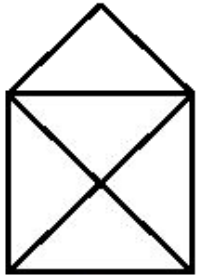
б



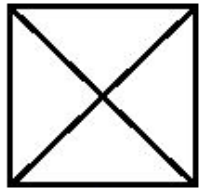
в



г



д



е

***Ейлеревим шляхом*** *у графі називається шлях, що містить всі ребра графа.*

***Ейлеревим циклом*** *у графі називається цикл, що містить всі ребра графа.*

*Граф, що має ейлеровий цикл, називається* ***ейлеревим графом.***

**Тв 1. Якщо граф має ейлерів цикл, то він зв'язний і всі його вершини парні.**
Справді, зв'язність графа випливає з означення ейлеревого циклу. Ейлерів цикл містить кожне ребро і притому тільки один раз, тому, скільки разів ейлерів шлях приведе кінець олівця у вершину, стільки і виведе, причому вже по іншому ребру. Отже, степінь кожної вершини графа повинна складатися з двох однакових доданків: один - результат підрахунку входів у вершину, другий - виходів.

Вірним і обернене твердження.

**Тв 2. Якщо граф зв'язний і всі його вершини парні, то він має ейлерів цикл.**
Якщо граф не містить ейлерів цикл, то можна поставити задачу про відшукання одного ейлеревого шляху чи декількох ейлеревих шляхів.

**Тв 3. Якщо граф містить ейлерів шлях з кінцями А та В (А не співпадає з В), то граф зв'язний та А і В - єдині непарні його вершини.**

Доведення :

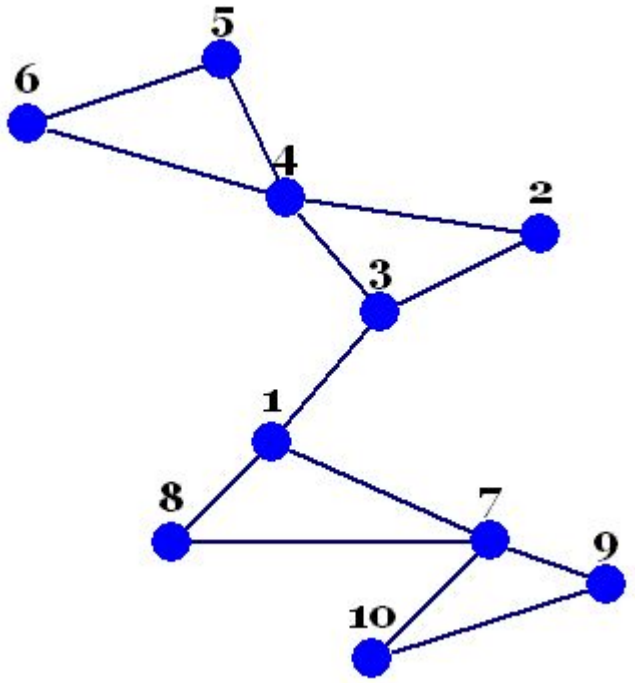
Зв'язність графа випливає з означення ейлеревого шляху.
Якщо шлях починається у А, а закінчується в іншій вершині , то А і В - непарні, навіть якщо шлях неодноразово проходив через А і В. У будь-яку іншу вершину графа шлях має привести і вивести з неї, тобто всі інші вершини мають бути парними.

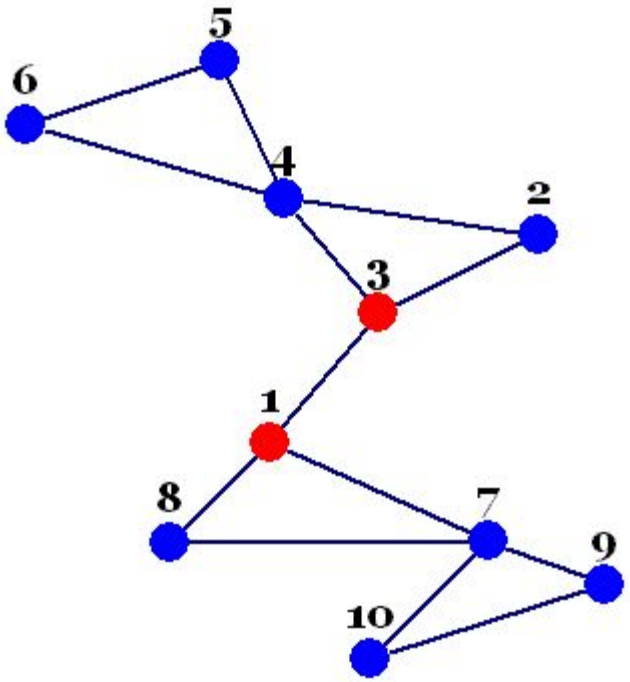
Вірним є і обернене твердження.

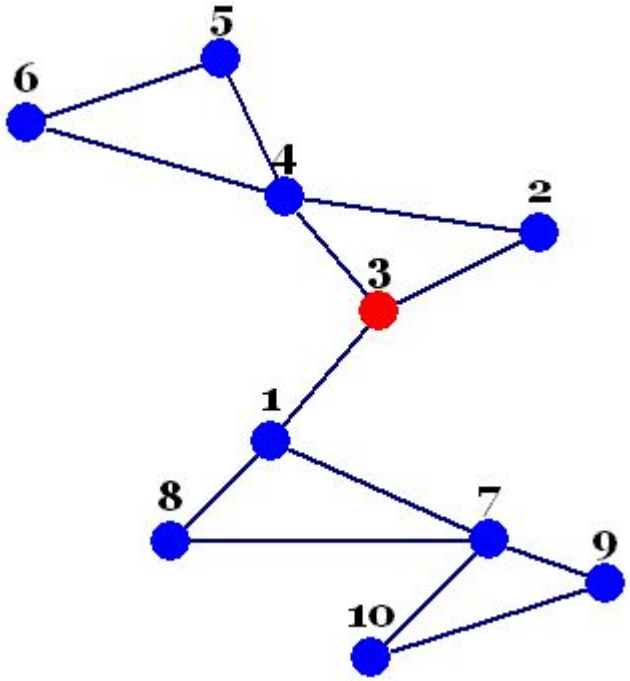
**Тв 4. Якщо граф зв'язний та А і В єдині непарні вершини, то граф має ейлерів шлях з кінцями А і В.**

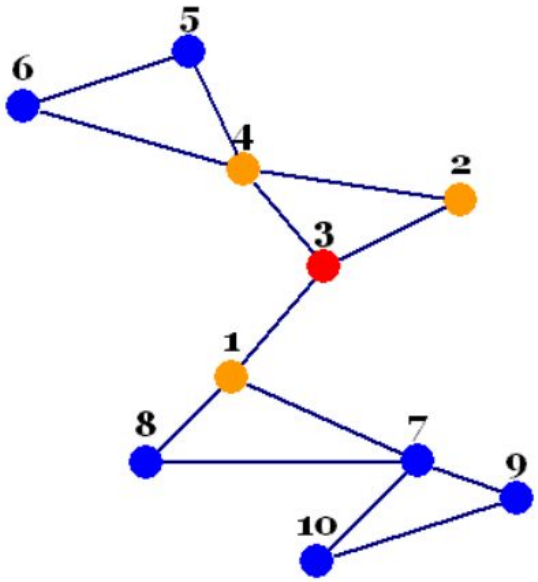
Доведення :
Вершини А і В можуть бути з'єднані ребром у графі, а можуть бути і не з'єднані.
Якщо А і В з'єднані ребром, то видалимо його; тоді усі вершини стануть парними. Новий граф, згідно твердження 2, має ейлерів цикл, причому початком і кінцем може слугувати будь-яка вершина. Почнемо ейлерів шлях у вершині А і закінчимо його у вершина А. Додамо ребро (А, В) та отримаємо ейлерів шлях з початком у А та кінцем у В.
Якщо А і В не з'єднані ребром, то до графа додамо нове ребро (А, В), тоді всі вершини графа стануть парними. Новий граф, згідно твердження 2, має ейлерів цикл. Почнемо його також у вершині А. Якщо тепер з отриманого циклу видалити ребро (А, В), то залишиться ейлерів шлях з початком у В та кінцем у А чи з початком у А і кінцем у В.

# Ейлерів Шлях

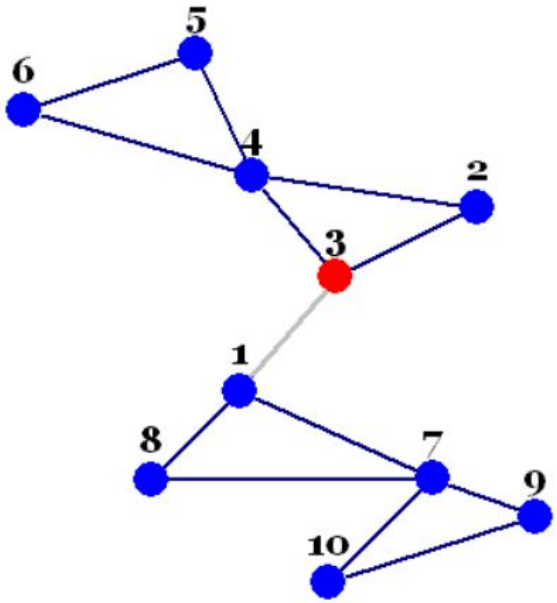



5
6
4
2
3
1
8
7
9
10

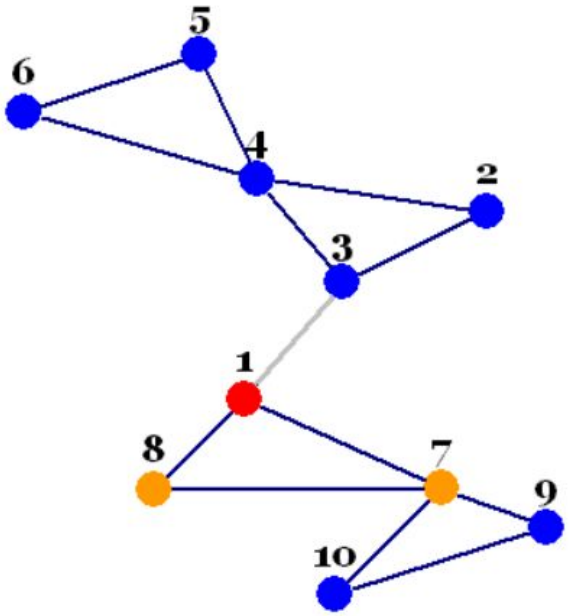

5
6
4
2
3
1
8
7
9
10



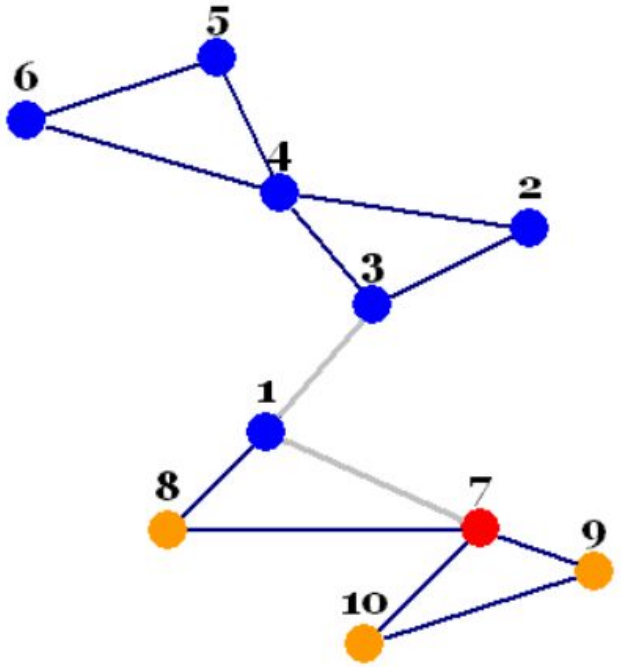

| |
|---|
| 1 |
| 3 |



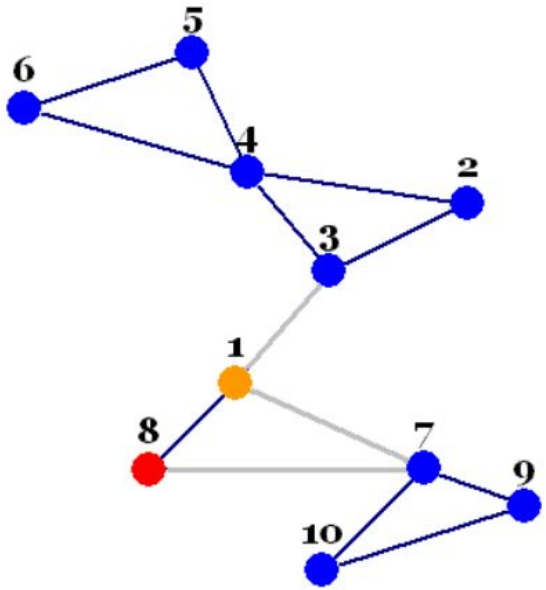

| |
|---|
| 1 |
| 3 |



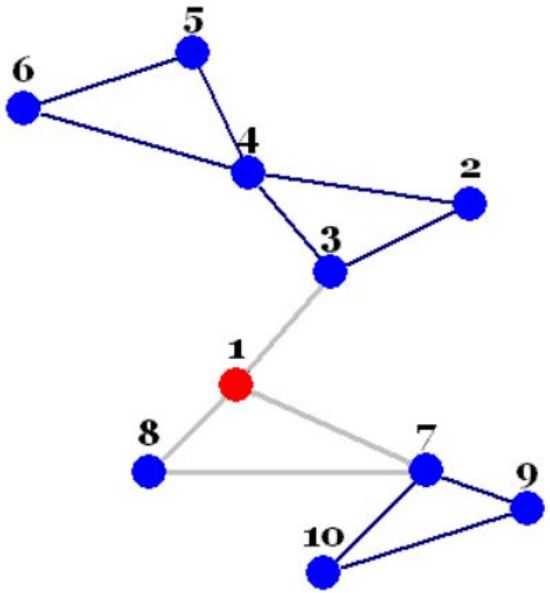

| 7 |
|---|
| 1 |
| 3 |




| 8 |
|---|
| 7 |
| 1 |
| 3 |

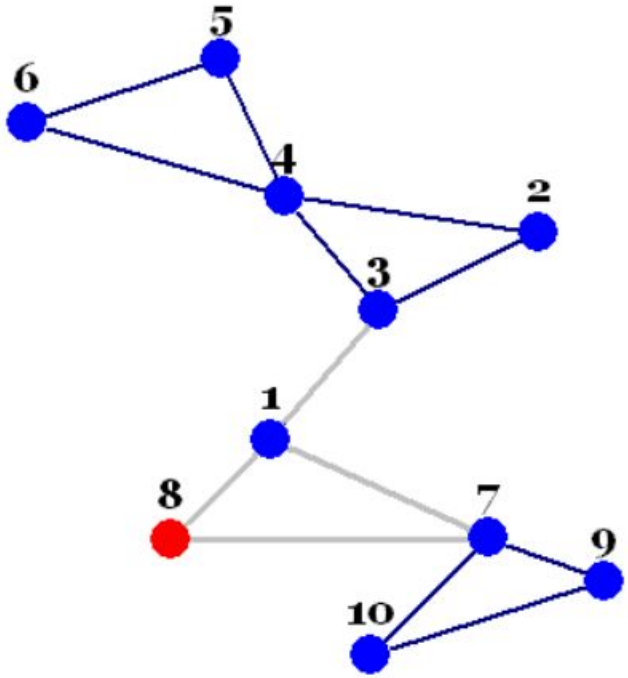



| |
|---|
| 1 |
| 8 |
| 7 |
| 1 |
| 3 |



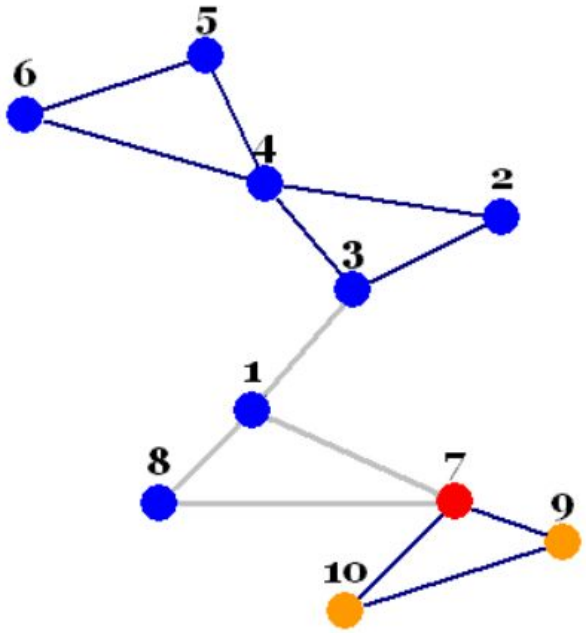

| |
|---|
| 8 |
| 7 |
| 1 |
| 3 |

| |
|---|
| 1 |



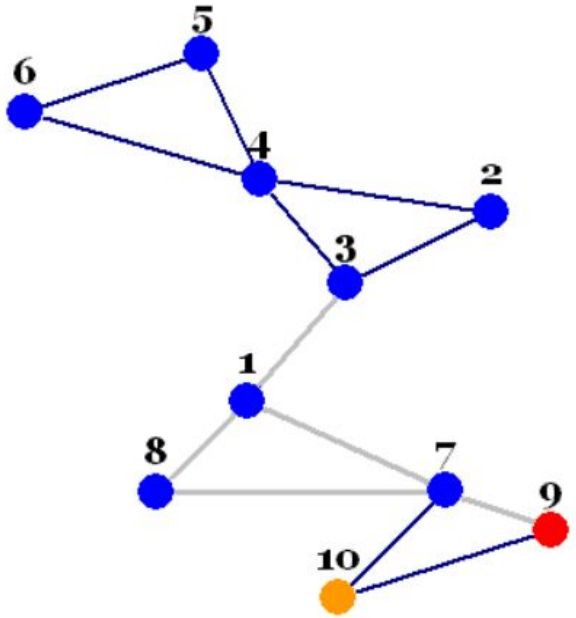

| 7 |
|---|
| 1 |
| 3 |

| 8 |
|---|
| 1 |



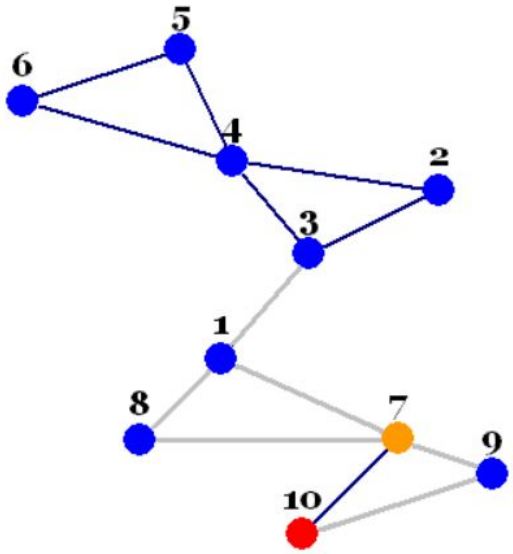

| 9 |
|---|
| 7 |
| 1 |
| 3 |

| 8 |
|---|
| 1 |



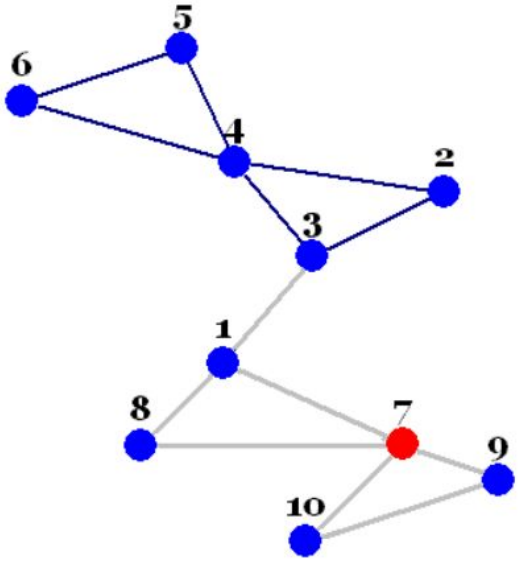

| |
|---|
| 10 |
| 9 |
| 7 |
| 1 |
| 3 |

| |
|---|
| 8 |
| 1 |



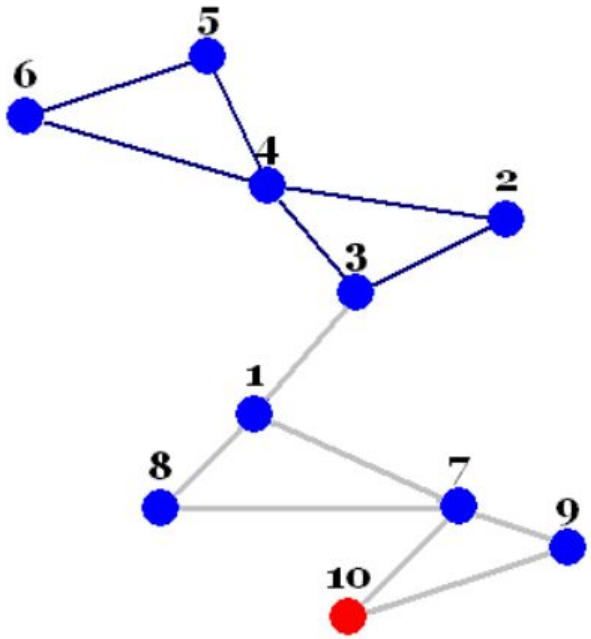

| |
|---|
| 7 |
| 10 |
| 9 |
| 7 |
| 1 |
| 3 |

| |
|---|
| 8 |
| 1 |



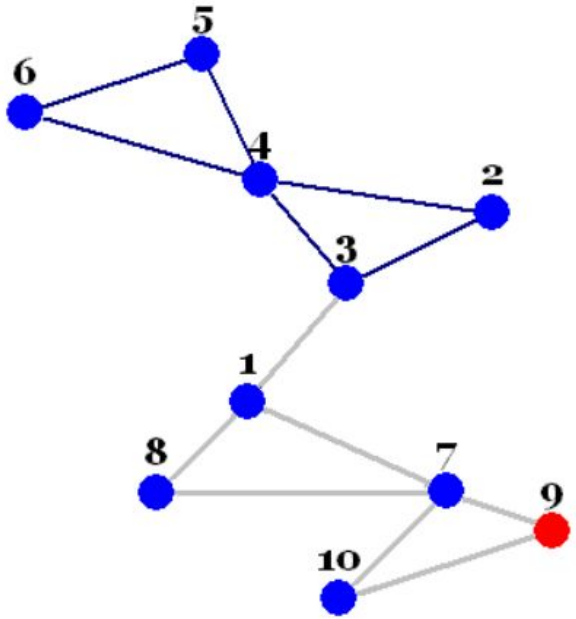

| 10 |
|---|
| 9 |
| 7 |
| 1 |
| 3 |

| 7 |
|---|
| 8 |
| 1 |



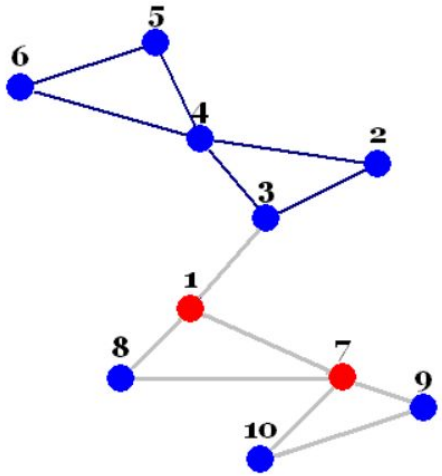

| 9 |
|---|
| 7 |
| 1 |
| 3 |

| 10 |
|---|
| 7 |
| 8 |
| 1 |



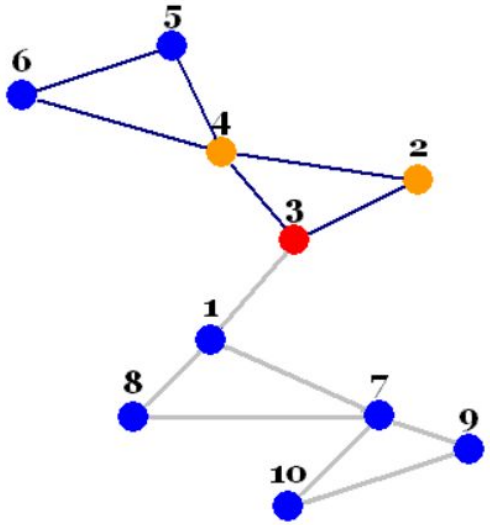

| 7 |
|---|
| 1 |
| 3 |

| 9 |
|---|
| 10 |
| 7 |
| 8 |
| 1 |



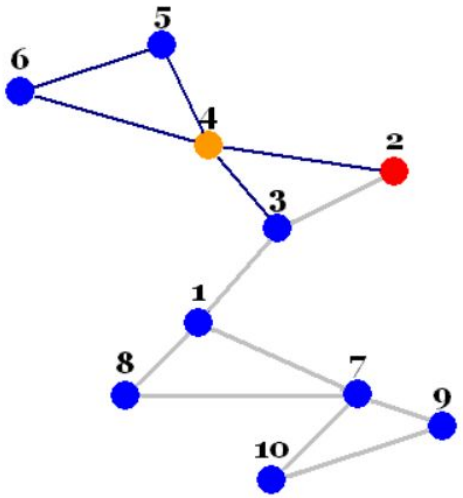

| 3 |
|---|

| 1 |
|---|
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



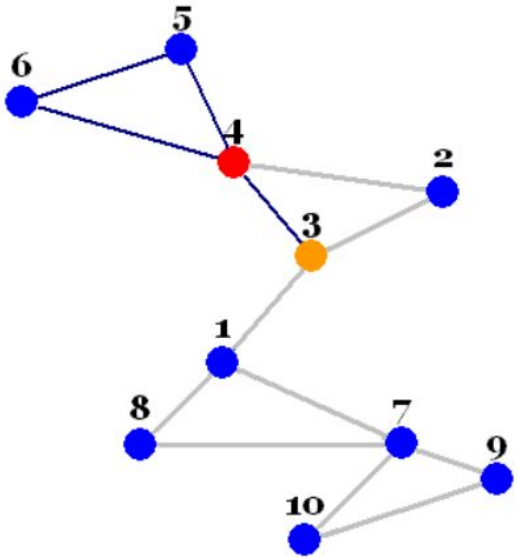

| |
|---|
| 2 |
| 3 |

| |
|---|
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



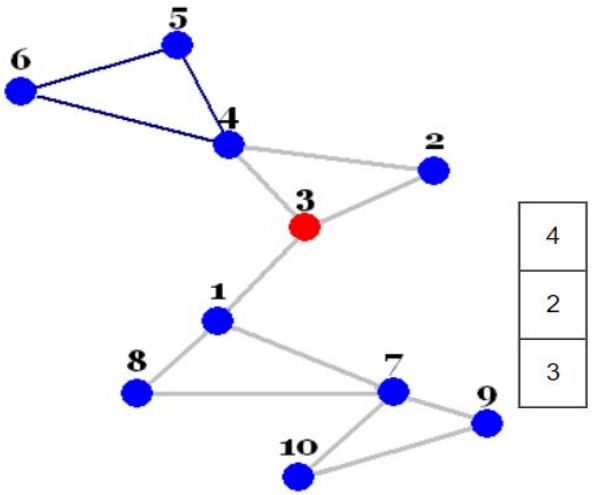

| |
|---|
| 4 |
| 2 |
| 3 |

| |
|---|
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



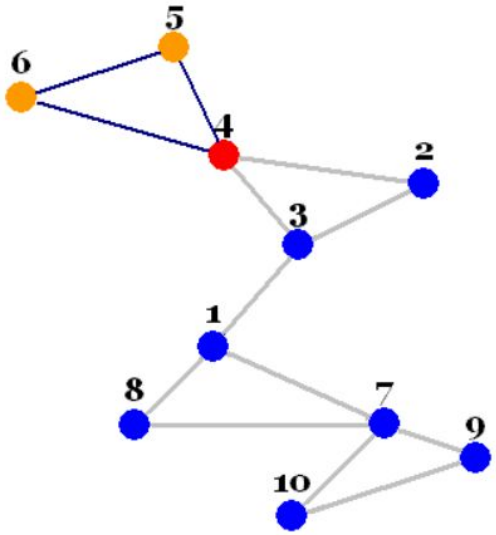

| |
|---|
| 3 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



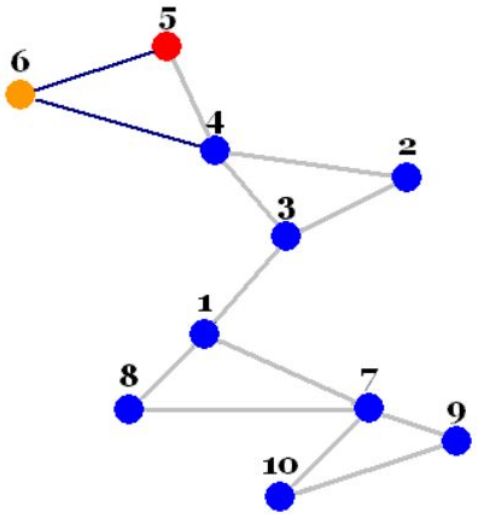

| |
|---|
| 4 |
| 2 |
| 3 |

| |
|---|
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



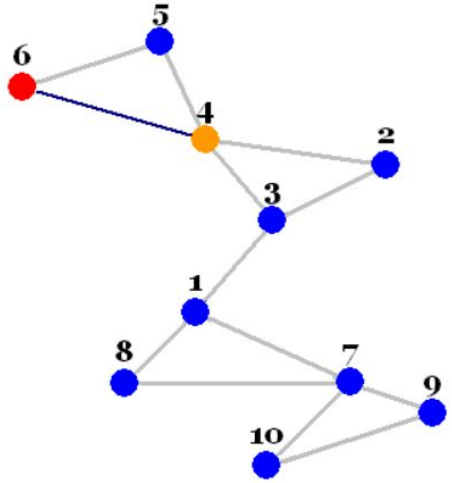

| |
|---|
| 5 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



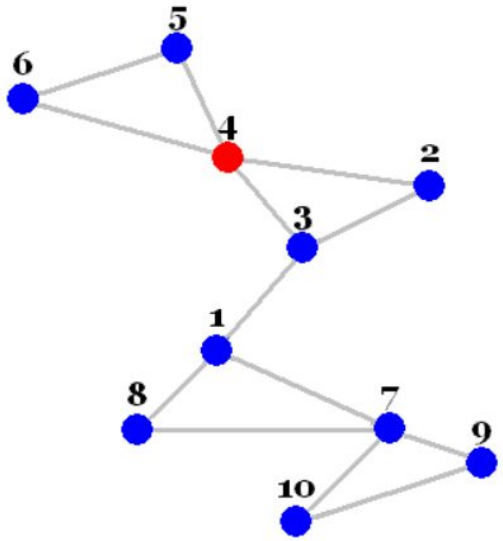

| |
|---|
| 6 |
| 5 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



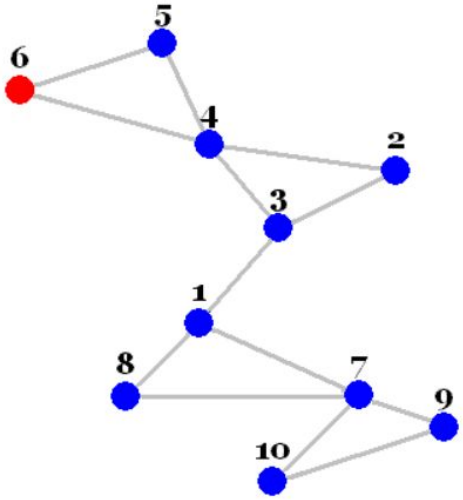

| |
|---|
| 4 |
| 6 |
| 5 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



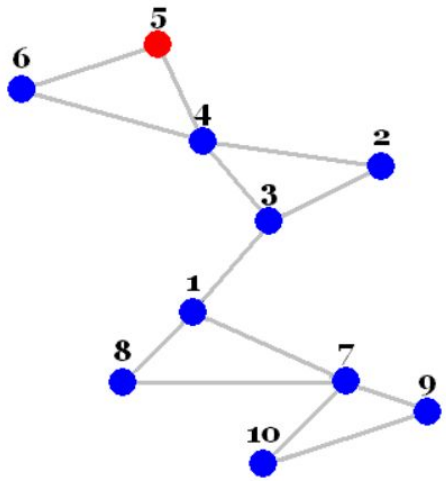

| |
|---|
| 6 |
| 5 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



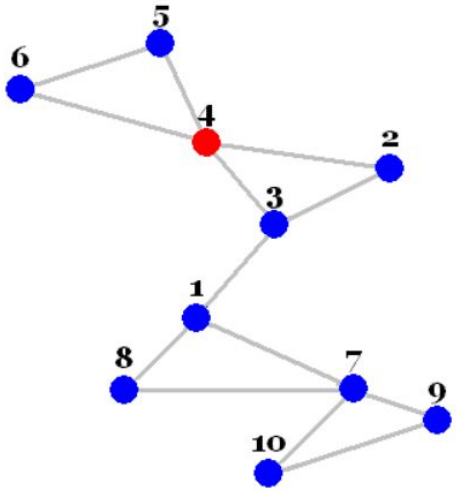

| |
|---|
| 5 |
| 4 |
| 2 |
| 3 |

| |
|---|
| 6 |
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



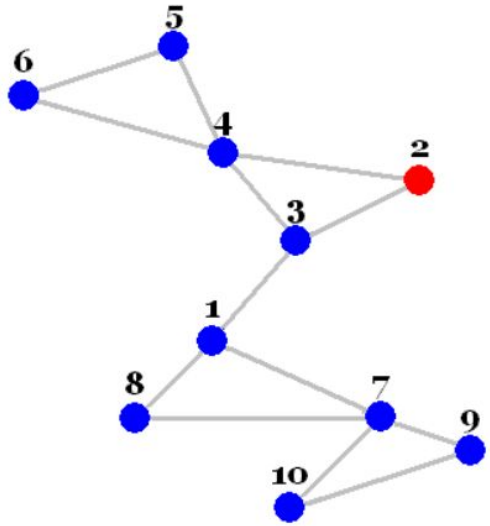

| |
|---|
| 4 |
| 2 |
| 3 |

| |
|---|
| 5 |
| 6 |
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |



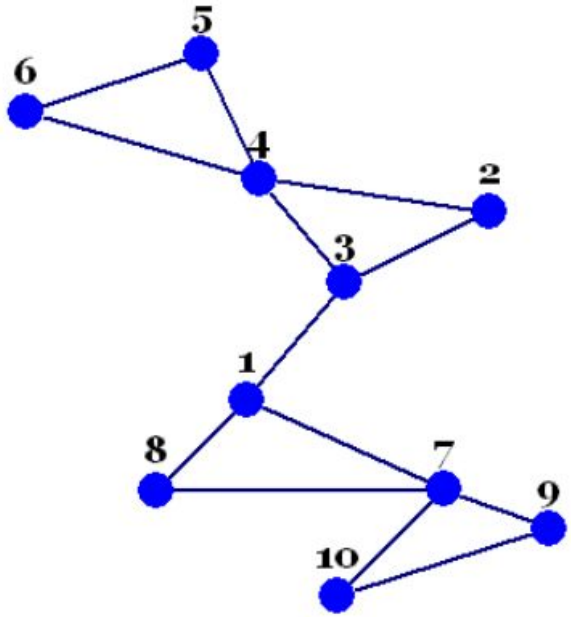

| |
|---|
| 2 |
| 3 |

| |
|---|
| 4 |
| 5 |
| 6 |
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |




| |
|---|
| 3 |

| |
|---|
| 2 |
| 4 |
| 5 |
| 6 |
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |




| |
|---|
| 3 |
| 2 |
| 4 |
| 5 |
| 6 |
| 4 |
| 3 |
| 1 |
| 7 |
| 9 |
| 10 |
| 7 |
| 8 |
| 1 |

Дякую за увагу