

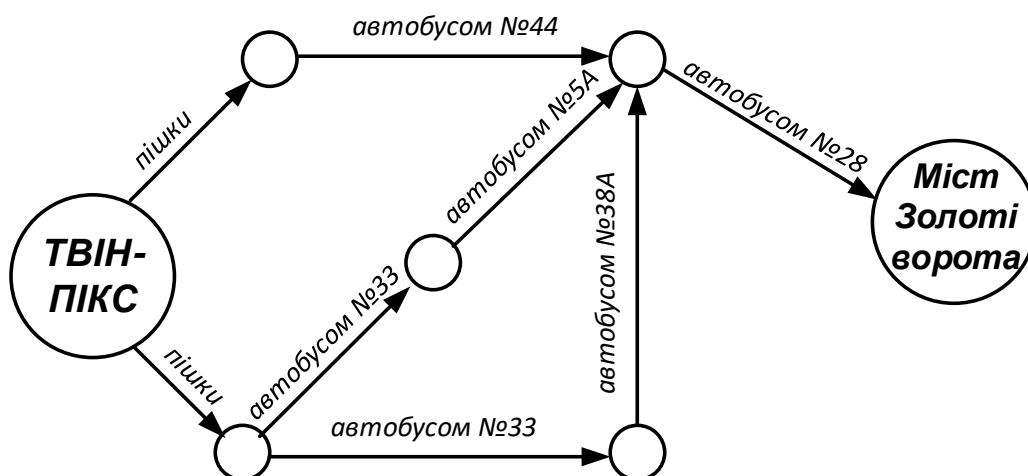
## Знайомство з графами

**Дорожня мережа.** Коли ПЗ смартфона обчислює маршрути руху, воно виконує пошук в графі, що представляє дорожню мережу, вершини якої відповідають перетину, а ребра – окремими ділянками дороги.

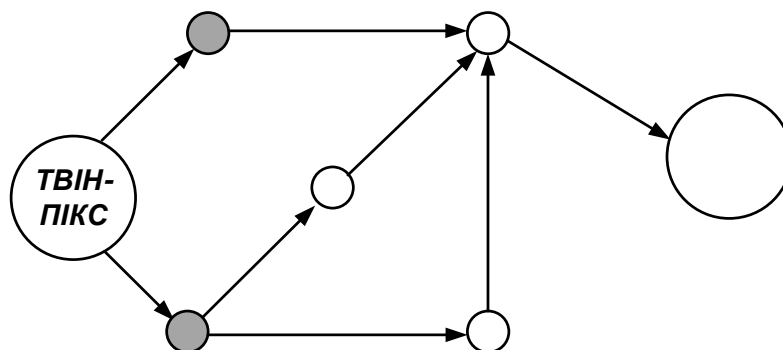
**Всесвітня павутина.** Інтернет може бути змодельований як орієнтований граф; його вершини будуть відповідати окремим веб-сторінкам, ребра – гіперпосиланнями, спрямованим зі сторінки, що містить гіперпосилання, на цільову сторінку.

**Соціальні мережі.** Соціальну мережу можна представити у вигляді графа, вершини якого відповідають окремій особистості та ребра якого відповідають деякому типу зв'язків. Наприклад, ребро може вказувати на дружбу між кінцевими точками або на те, що одна з кінцевих точок «читає» іншу. Які з найбільш популярних соціальних мереж змодельовані як неорієнтований граф, а які – як орієнтований?

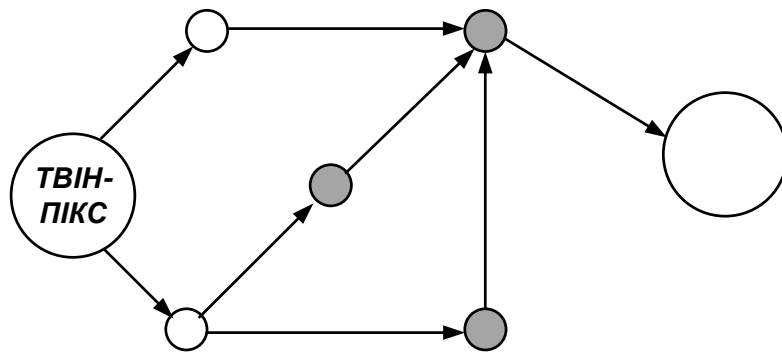
Припустимо, ви перебуваєте в Сан-Франциску і хочете дістатися з «Твін-Пікс» до мосту «Золоті Ворота». Ви хочете доїхати на автобусі з мінімальною кількістю пересадок. Можливі варіанти:



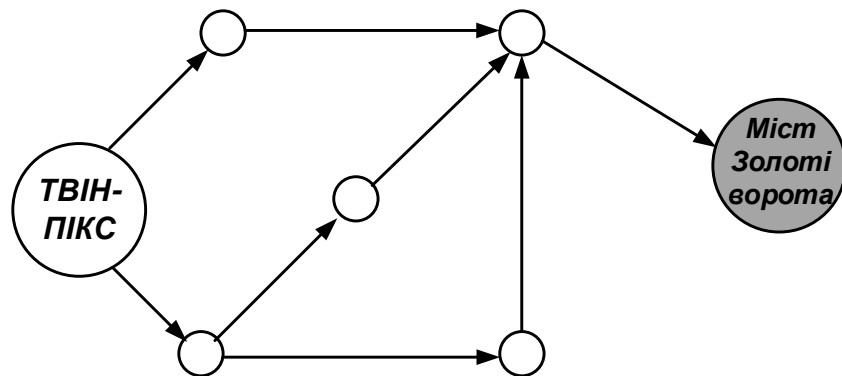
Який алгоритм ви б використовували для пошуку шляху з найменшою кількістю кроків? Чи можна зробити це за один крок? На наступному малюнку виділені всі місця, в які можна дістатися за один крок.



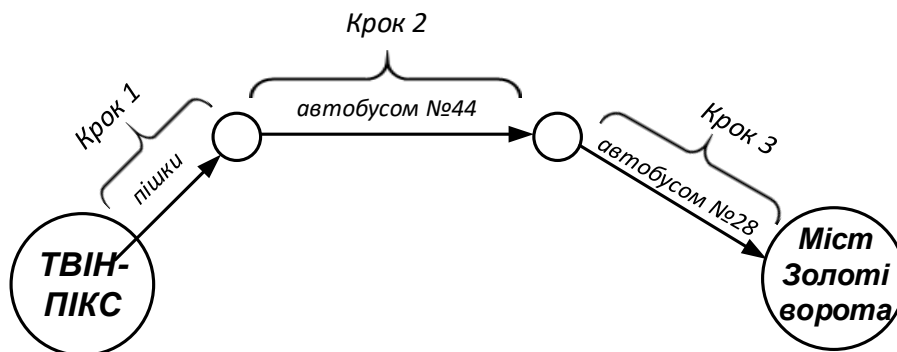
Міст на цій схемі не виділено; до нього неможливо добратися за один крок. А чи можна дістатися до нього за два кроки?



І знову міст не виділено, а значить, до нього неможливо дістатися за два кроки. Як щодо трьох кроків?



На цей раз міст Золоті Ворота виділено. Отже, щоб дістатися з Твін-Пікс до мосту по цьому маршруту, необхідно зробити три кроки.



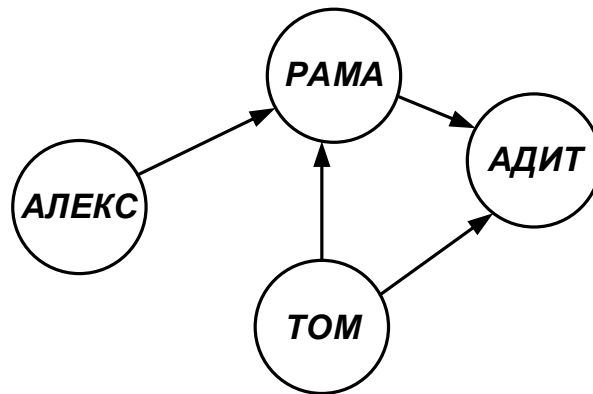
Алгоритм виявив, що найкоротший шлях до мосту складається з трьох кроків. Завдання такого типу називаються завданням пошуку найкоротшого шляху.

Алгоритм для вирішення завдання пошуку найкоротшого шляху називається **пошуком в ширину**. Щоб знайти найкоротший шлях з Твін-Пікс до мосту «Золоті Ворота», довелося виконати два кроки:

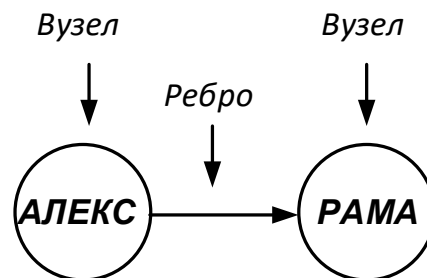
1. Змоделювати завдання у вигляді графа.
2. Вирішити задачу методом пошуку в ширину.

## Що таке граф?

Граф моделює набір зв'язків.



Кожен граф складається з вузлів та ребер.



Графи складаються з вузлів та ребер. Вузол може бути безпосередньо з'єднаний з декількома іншими вузлами. Ці вузли називаються сусідами. На цьому графі Рама є сусідом Алекса. З іншого боку, Адит є сусідом Алекса не є, тому що вони не з'єднані безпосередньо. При цьому Адит є сусідом Рами і Тома. Графи використовуються для моделювання зв'язків між різними об'єктами.

## Пошук в ширину

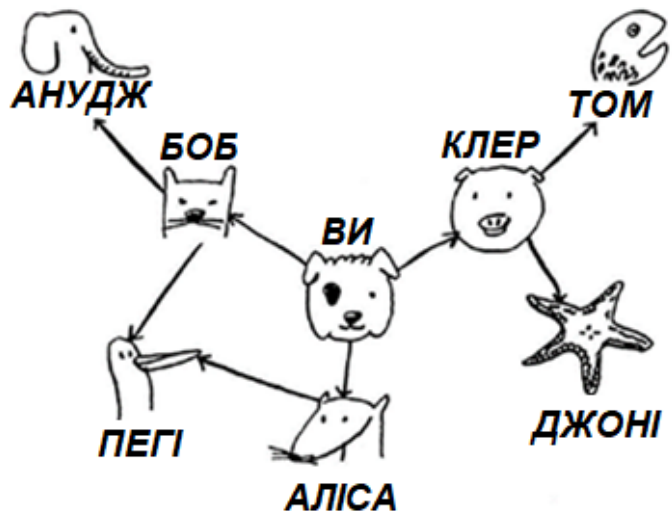
Пошук в ширину також відноситься до категорії алгоритмів пошуку, але цей алгоритм працює з графами. Він допомагає відповісти на питання двох типів:

- тип 1: чи існує шлях від вузла А до вузла В?
- тип 2: як виглядає найкоротший шлях від вузла А до вузла В?

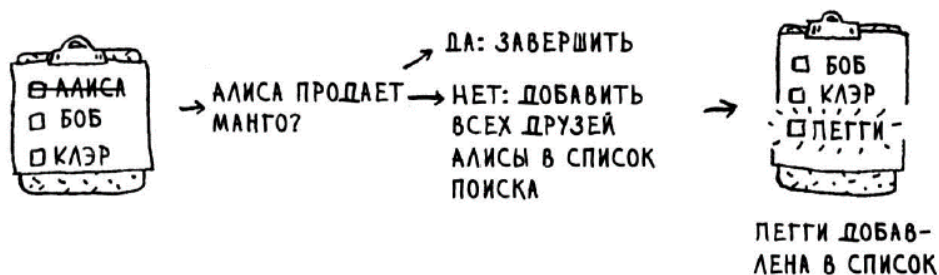
Прикладом пошуку в ширину є визначення найкоротшого шляху з «Твін-Пікс» до мосту «Золоті Ворота». Це було питання типу 2: як виглядає найкоротший шлях? Тепер розберемо роботу алгоритму більш детально з питанням типу 1: чи існує шлях?

Уявіть, що ви вирощуєте манго. Ви шукаєте продавця, який буде продавати ваші чудові манго. А може, продавець знайдеться серед ваших контактів на Facebook? Для початку варто пошукати серед друзів.

Спочатку потрібно побудувати список друзів для пошуку. Тепер потрібно звернутися до кожної людини в списку і перевірити, чи продає ця людина манго.



Припустимо, жоден з ваших друзів не продає манго. Тепер пошук триває серед друзів ваших друзів. Кожен раз, коли ви перевіряєте когось з списку, ви додаєте до списку всіх його друзів.

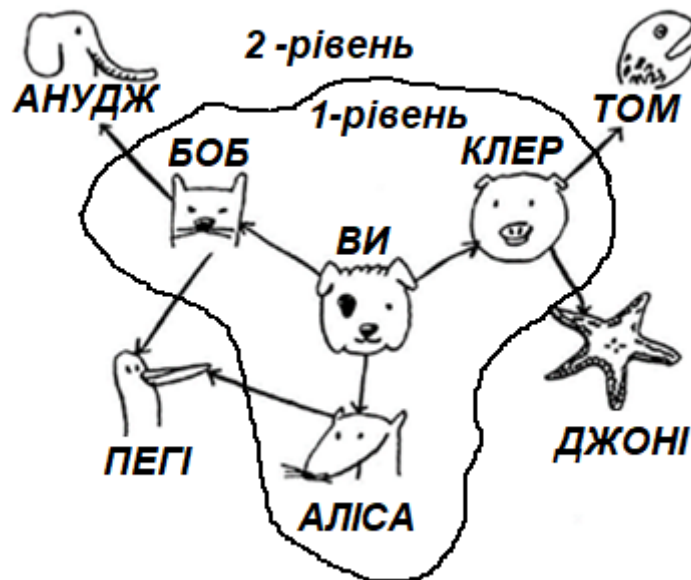


У такому випадку пошук ведеться не тільки серед друзів, але і серед друзів друзів теж. Якщо АЛІСА не продає манго, то до списку додаються усі її друзі. Це означає, що з часом Ви перевірите її друзів, їх друзів. За таким алгоритмом пошук пройде по всій мережі, поки не знайдете продавця манго. Такий алгоритм і називається пошук в ширину.

### Пошук найкоротшого шляху

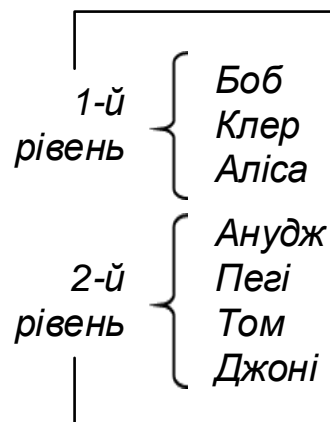
Два питання, на які може відповісти алгоритм пошуку в ширину:

- тип 1: існує чи шлях від вузла А до вузла В? (чи є продавець манго в вашій мережі?);
- тип 2: як виглядає найкоротший шлях від вузла А до вузла В? (хто з продавців манго знаходиться ближче всього до вас?)



Ви вже знаєте, як відповісти на питання 1; тепер спробуємо відповісти на питання 2. Чи вдасться чи вам знайти найближчого продавця манго? Будемо вважати, що ваші друзі – це зв'язку першого рівня, а друзі друзів – зв'язку другого рівня.

Зв'язки першого рівня краще ніж зв'язки другого рівня, зв'язку другого рівня краще зв'язків третього рівня і т. д. Звідси випливає, що пошук по контактам другого у рівня не має проводитися, поки ви не будете повністю впевнені в тому, що серед зв'язків першого рівня немає ні одного продавця манго. Пошук в ширину поширюється від початкової точки. А це означає, що зв'язки першого рівня будуть перевірені до зв'язків другого рівня. Контрольне питання: хто буде перевірений першим, Клер або Анудж? Відповідь: Клер є зв'язком першого рівня, а Анудж – зв'язком другого рівня. Отже, Клер буде перевірена першою. Також можна пояснити це інакше: з в'язі першого рівня додаються в список пошуку раніше зв'язків другого рівня.



А що буде, якщо перевірите Ануджа раніше, ніж Клер, і обидва вони виявляться продавцями манго? Анудж є зв'язком другого рівня, а Клер – зв'язком першого рівня. В результаті буде знайдений продавець манго, чи не найближчий

до вас в мережі. Отже, перевіряти зв'язку потрібно в порядку їх додавання. Для операцій такого роду існує спеціальна структура даних, яка називається чергою. Чергу відноситься до категорії структур даних FIFO: First In, First Out, а стек належить до числа структур даних LIFO: Last In, First Out.

## Реалізація графа

Для початку необхідно реалізувати граф на програмному рівні. Граф складається з декількох вузлів. І кожен вузол з'єднується з сусідніми вузлами. Згадайте: хеш-таблиця пов'язує ключ зі значенням. У даному випадку вузол повинен бути зв'язаний з усіма його сусідами. На Python це записується:

```
graph= {}  
graph ["you"] = ["alice", "bob", "claire"]
```

Елемент «you» відображається на масив. Отже, результатом виразу `graph["you"]` є масив всіх ваших сусідів. Граф – це набір вузлів і ребер, тому для представлення графа на Python нічого більше не потрібно. Код на мові Python виглядає так:

```
graph = {}  
graph ["you"] = ["alice", "bob", "claire"]  
graph ["bob"] = ["anuj", "peggy"]  
graph ["alice"] = ["peggy"]  
graph ["claire"] = ["thom", "jonny"]  
graph ["anuj"] = []  
graph ["peggy"] = []  
graph ["thom"] = []  
graph ["jonny"] = []
```

Контрольне питання: чи має значення порядок додавання пар «ключ-значення»? Чи важливо, який запис ви будете використовувати:

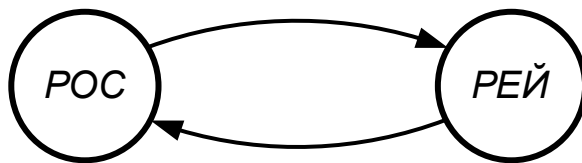
```
graph ["claire"] = ["thom", "jonny"]  
graph ["anuj"] = []
```

або

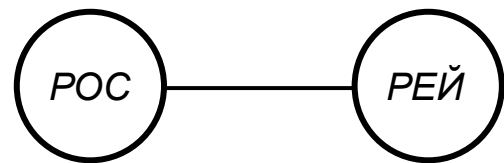
```
graph ["anuj"] = []  
graph ["claire"] = ["thom", "jonny"]
```

Відповідь: ні, не важливо. У хеш-таблицях елементи не впорядковані, тому додавати пари «ключ-значення» можна в будь-якому порядку.

У Ануджа, Пегі, Тома і Джонні сусідів немає. Лінії зі стрілками вказують на них, але не існує стрілок від них до інших вузлів. Такий граф називається спрямованим – відношення діють тільки в одну сторону. Отже, Анудж є сусідом Боба, але Боб не є сусідом Ануджа. У ненаправленому графі стрілок немає, і кожен з вузлів є сусідом по відношенню один до одного. Наприклад, обидва наступних графа еквівалентні.



Направлений граф



Ненаправлений граф

### Реалізація алгоритму



Усе починається зі створення черги. В Python для створення двосторонньої черги використовується функція `deque` :

```
from collections import deque
search_queue = deque () // Створення нової черги
search_queue += graph ["you"] //Усі сусіди додаються в чергу пошуку
```

А тепер розглянемо інше:

```

while search_queue : // Поки черга не порожня
    person= search_queue.popleft()//з черги витягується перша людина
    if person is seller(person): // Перевіряємо, є чи цей
        // людина продавцем манго
        print person + "is a mango seller!"
        // Так, це продавець манго
        return True
    else:
        search_queue += graph[person] // Ні, не є .
        // Всі друзі цієї людини додаються в чергу
return False // Якщо виконання дійшло до цього рядка,
//значить, в черзі немає продавця манго

def person_is_seller (name):
    return name [-1] == 'm'

```

Ця функція перевіряє, чи закінчується ім'я на букву «m», і якщо закінчується, ця людина вважається продавцем манго. Подивимося , як працює пошук в ширину.



У Аліси і Боба є один спільний друг: Пегі. Отже, Пегі буде додана в чергу двічі: при додаванні і друзів Аліси і при додаванні друзів Боба. В результаті Пегі

появиться в черзі пошуку в двох примірниках. Але перевірити чи є Пегі продавцем манго, досить всього один раз. Перевіряючи її двічі, ви виконуєте зайву, непотрібну роботу. Отже, після перевірки людини необхідно помітити її як перевіреного, щоб не перевіряти його знову. Якщо цього не зробити, може виникнути нескінченний цикл. Перш ніж перевіряти людину, потрібно переконатися в тому, що він не був перевірений раніше. Для цього будемо вести список вже перевірених людей.

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []  # ←.....
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:  # ←.....
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)  # ←.....
    return False

search("you")
```

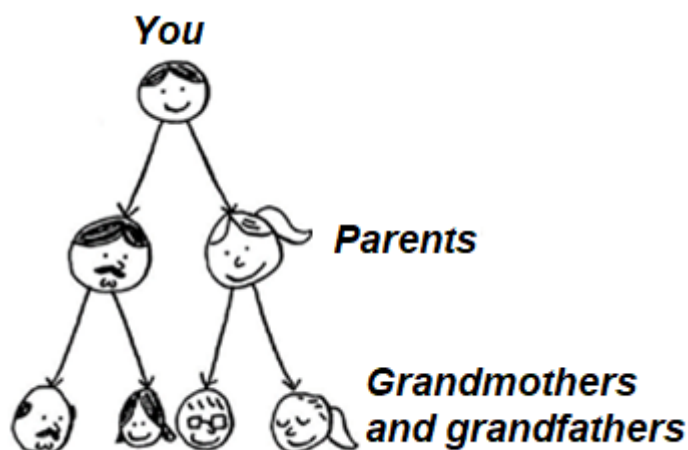
**Этот массив используется для отслеживания уже проверенных людей**

**Человек проверяется только в том случае, если он не проверялся ранее**

**Человек помечается как уже проверенный**

Пошук в ширину виконується за час  $O$  (кількість людей + кількість ребер), що зазвичай записується в формі  $O(V + E)$  ( $V$  - кількість вершин,  $E$  - кількість ребер).

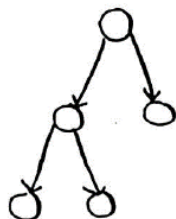
Мається генеалогічне древо.



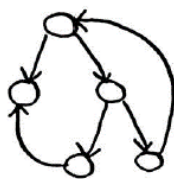
Генеалогічне древо – це граф, тому що в ньому є вузли (люди) та ребра. Ребра вказують на батьків людини. Природно, все ребра спрямовані вниз – в генеалогічному дереві ребро, яке вказує вгору, не має сенсу. Такий особливий різновид графа, в якому немає ребер, що вказують в зворотному напрямку, називається деревом.

Які з наступних графів також є деревами?

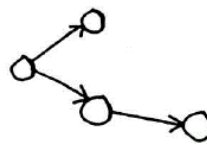
A.



B.



C.



### Висновки

1. Пошук в ширину дозволяє визначити, чи існує шлях з A в B .
2. Якщо шлях існує, то пошук в ширину знаходить найкоротший шлях.
3. У наведеному графі є стрілки, а відношення діють в напрямку стрілки.
4. У ненаправлених графах стрілок немає, а відношення йде в обидві сторони
5. Черги відносяться до категорії FIFO.
6. Людей слід перевіряти в порядку їх додавання до списку пошуку, тому список пошуку повинен бути оформлений у вигляді черги, інакше знайдений шлях не буде найкоротшим.
7. Подбайте про те, щоб уже перевірений людина не перевірявся заново, інакше може виникнути нескінченний цикл.

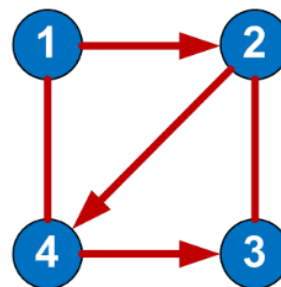
### Способи подання графа

Граф може бути представлений (збережений) декількома способами:

- матриця суміжності ;
- матриця інцидентності ;
- список суміжності (інцидентності);
- список ребер.

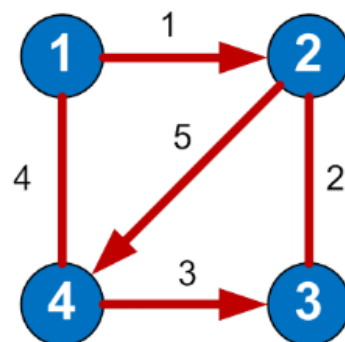
**Матриця суміжності графа** – це квадратна матриця, в якій кожен елемент приймає одне з двох значень: 0 або 1. Число рядків матриці суміжності дорівнює числу стовпців і відповідає кількості вершин графа. 0 – відповідає відсутності ребра, 1 – відповідає наявності ребра. Коли з однієї вершини в іншу прохід вільний (мається ребро ), в комірку заноситься 1, інакше – 0. Всі елементи на головній діагоналі рівні 0 якщо граф не має петель .

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



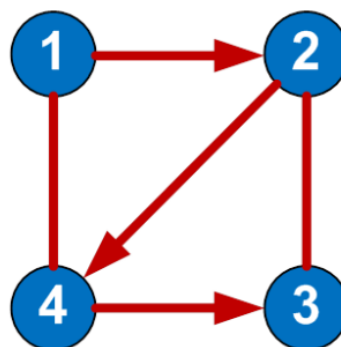
**Матриця інцидентності (інциденції)** графа – це матриця, кількість рядків в якій відповідає числу вершин, а кількість стовпців – числу ребер. У ній вказуються зв'язку між інцидентними елементами графа (ребро (дуга) і вершина). У неорієнтованому графі якщо вершина інцидентна ребру то відповідний елемент дорівнює 1, в іншому випадку елемент дорівнює 0. У орієнтованому графі якщо ребро виходить з вершини, то відповідний елемент дорівнює 1, якщо ребро входить в вершину, то відповідний елемент дорівнює – 1, якщо ребро відсутнє, то елемент дорівнює 0. Матриця інцидентності для свого уявлення вимагає нумерації ребер, що не завжди зручно.

	1	2	3	4	5
1	1	0	0	1	0
2	-1	1	0	0	1
3	0	1	-1	0	0
4	0	0	1	1	-1



**Список суміжності (інцидентності).** Якщо кількість ребер графа у порівнянні з кількістю вершин невелика, то значення більшості елементів матриці суміжності будуть рівні 0. При цьому використання даного методу недоцільно. Для подібних графів є більш оптимальні способи їх подання. Списки суміжності менш вимогливі до пам'яті, ніж матриці суміжності. Такий список можна представити у вигляді таблиці, стовпців в якій – 2, а рядків – не більше, ніж вершин в графі.

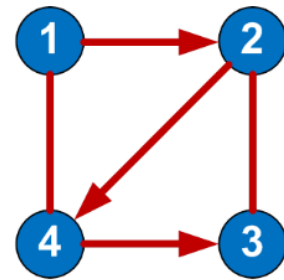
1	2, 4
2	3, 4
3	2
4	1, 3



У кожній рядку в першому стовпці вказана вершина виходу, а у другому стовпці - список вершин, в які входять ребра з поточної вершини.

**Список ребер.** У списку ребер в кожній рядку записуються дві суміжні вершини і вага з'єднує їх ребра (для зваженого графа). Кількість рядків в списку ребер завжди має бути одно величині, що виходить в результаті складання орієнтованих ребер з подвоєним кількістю неорієнтованих ребер.

	Начало	Кінець	Вес
1	1	2	
2	1	4	
3	2	3	
4	2	4	
5	3	2	
6	4	1	
7	4	3	

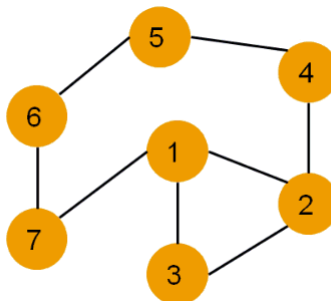


Графи з великим числом ребер називають щільними, з малим - розрідженими. Щільні граfi зручніше зберігати в вигляді матриці суміжності, розріджені – у вигляді списку суміжності.

**Пошук в ширину** передбачає дослідження графа:

- спочатку відвідується корінь – довільно обраний вузол;
- потім – усі нащадки даного вузла;
- після цього відвідуються нащадки нащадків і т.д.

Вершини проглядаються в порядку зростання їх відстані від кореня. Алгоритм припиняє свою роботу після обходу всіх вершин графа, або в разі виконання необхідно умови (наприклад, знайти найкоротший шлях з вершини 1 у вершину 6).



Кожна вершина може перебувати в одному з 3 станів:

- 1 помаранчева – не виявлена вершина;
- 2 зелений – виявлена, але не відвідана вершина;
- 3 сірий – оброблена вершина.

Фіолетовий – вершина, що розглядається.

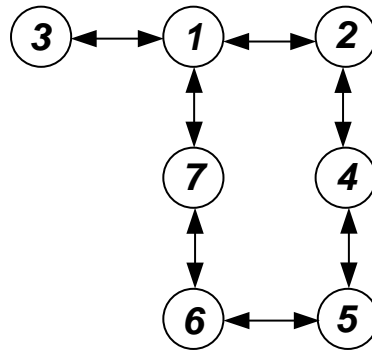
### Застосування алгоритму пошуку в ширину

- Пошук найкоротшого шляху в незваженому графі (орієнтованому або неорієнтованому).
- Пошук компоненту зв'язності.
- Знаходження рішення будь-якої задачі (гри) з найменшим числом ходів.

- Знайти всі ребра, що лежать на якомусь найкоротшому шляху між заданою парою вершин.
- Знайти всі вершини, що лежать на якомусь найкоротшому шляху між заданою парою вершин.

Для реалізації алгоритму зручно використовувати чергу.

Завдання пошуку найкоротшого шляху



```

#include <iostream>
#include <queue> // чергу
#include <stack> // стек
using namespace std ;
struct Edge {
    int begin ;
    int end ;
};
int main ()
{
    system ( "chcp 1251" );
    system ( "cls" );
    queue < int > Queue ;
    stack < Edge > Edges ;
    int req ;
    Edge e;
    int mas [ 7 ] [ 7 ] = {{ 0 , 1 , 1 , 0 , 0 , 0 , 1 },
    { 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
    for ( int i = 0 ; i < 7 ; i ++ ) // початково все вершини рівні 0
        nodes [i] = 0 ;
    cout << "N =" ; cin >> req ; req --;
    Queue.push ( 0 ); // поміщаємо в чергу першу вершину
    while ( ! Queue.empty () )
    {
        int node = Queue.front () ; // витягаємо вершину
        Queue.pop () ;
        nodes [ node ] = 2 ; // відзначаємо її як відвідану
    }
}
  
```

```

for ( int j = 0 ; j < 7 ; j ++ )
{
    if ( mas [ node ] [j] == 1 && nodes [j] == 0 )
    { // якщо вершина суміжна і не виявлена
        Queue.push (j); // додаємо її в чергу
        nodes [j] = 1 ; // відзначаємо вершину як виявлену
        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 ;
}

```

```

$ clang++-7 -pthread -o main main.cpp
$ ./main
sh: 1: chcp: not found
sh: 1: cls: not found
N = 5
1
2
3
7
4
6
5
Путь до вершины 5
5 <- 4 <- 2 <- 1

```

Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. СПб.: Питер, 2017. 288 с.

<https://prog-cpp.ru/data-graph/>