

# **ЛЕКЦІЯ 10**

**Дерева, ліс**

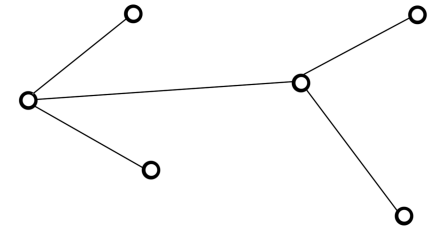
**Остовні дерева, остовний ліс**

**Алгоритми на графах**

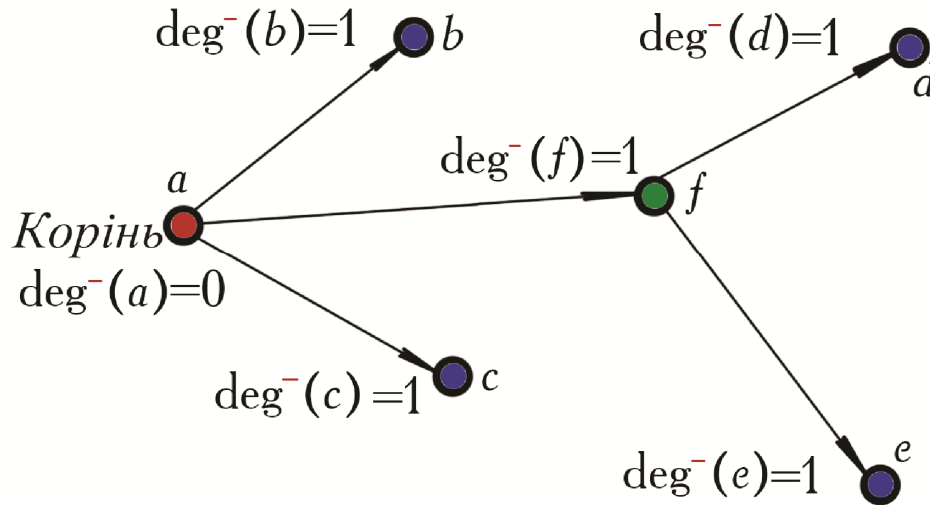
## Визначення дерева. Властивості дерев

**Неорієнтованим деревом** називають зв'язний неорієнтований граф без циклів.

**Кореневим деревом** називають таке дерево, у якому існує виділена вершина, яку називають *коренем*.



**Корінь** у неорієнтованому графі – це одна з вершин, обрана за бажанням спостерігача.

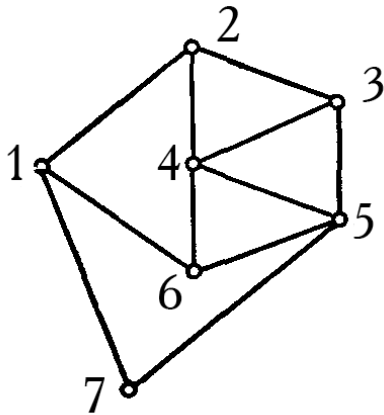


**Орієнтованим деревом** називають зв'язний орієнтований граф без циклів, у якому напівстепінь **входу** кожної вершини, за винятком кореневої, дорівнює **1**, а напівстепінь **входу**

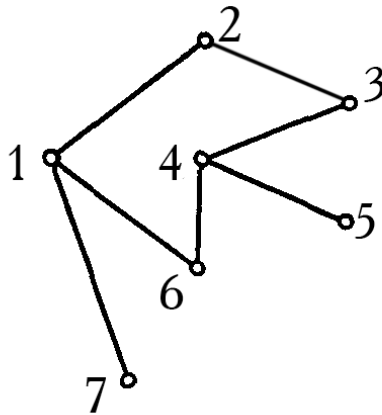
кореневої вершини дорівнює **0**.

**Остовним підграфом** називають такий підграф, у якому множина його вершин збігається з множиною вершин самого графа.

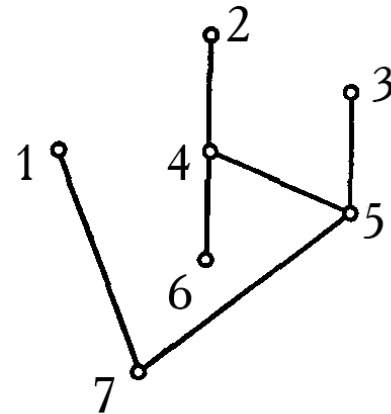
**Остовним деревом** графа  $G$  називають **ОСТОВНИЙ підграф** графа  $G$  без циклів.



Початковий  
граф



Остовний  
підграф



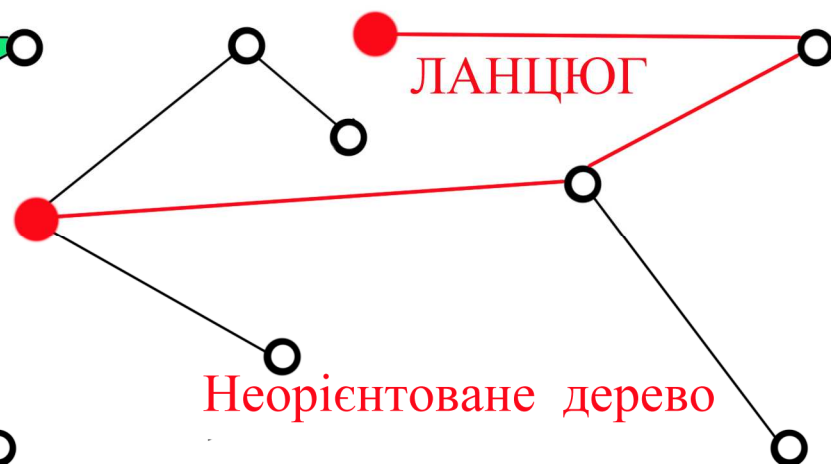
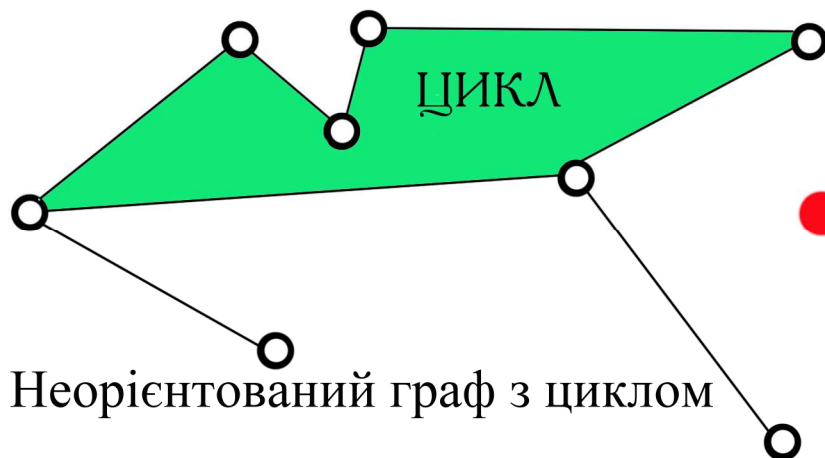
Остовне  
дерево

**Теорема 1.** Граф є деревом тоді і тільки тоді, коли будь-які дві його вершини зв'язані єдиним ланцюгом.

**Доведення.** Нехай граф є деревом. Якщо припустити існування більш ніж одного ланцюга, що зв'язує будь-які дві його вершини, то в такому графі існує цикл, тобто граф не може бути деревом.

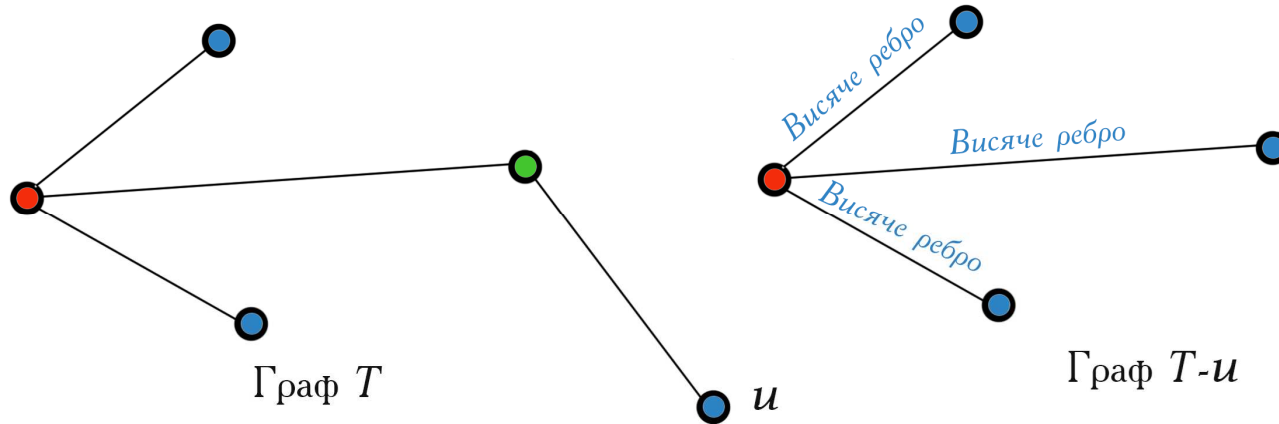
**Навпаки,** оскільки будь-які дві вершини графа з'єднані ланцюгом, то граф зв'язний, а в силу того, що цей ланцюг єдиний, він не має циклів. Отже, граф є деревом.

*Теорема доведена.*



**Наслідок 1.** Якщо  $T$  – дерево і  $u$  – його кінцева вершина (листок), то граф  $T - u$  ( $T$  мінус  $u$ ) – дерево.

Дійсно, граф  $T - u$  – **правильний підграф** дерева  $T$ , для якого виконуються всі умови теореми.



**Наслідок 2.** Усяке непусте дерево має принаймні **дві висячі вершини** і одне висяче ребро.

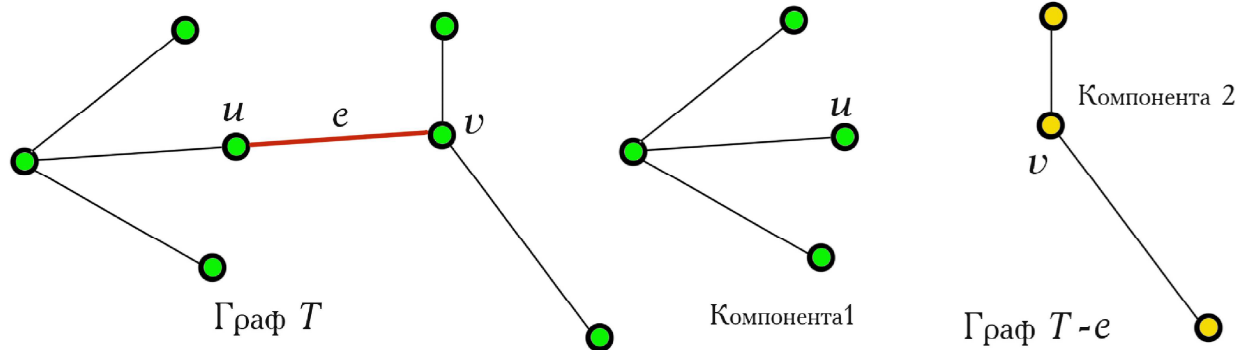
1. **Висяча вершина в неорієнтованому графі** – це вершина степеня 1.
2. **Висяча вершина в орграфі** – вершина з напівстепенем входу, що дорівнює 1, і напівстепенем виходу, що дорівнює 0.
3. **Висяче ребро** – це ребро, інцидентне вершині зі степенем 1.

**Визначення.** Ребро зв'язного графа називають *істотним*, якщо його видалення веде до порушення зв'язності цього графа.

**Наслідок.** У неорієнтованому графі істотним ребром є міст.

**Теорема.** У дереві кожне ребро істотне.

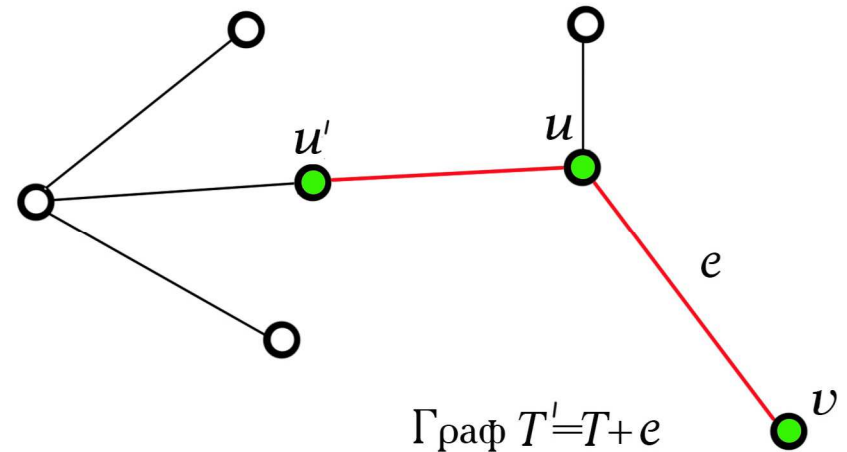
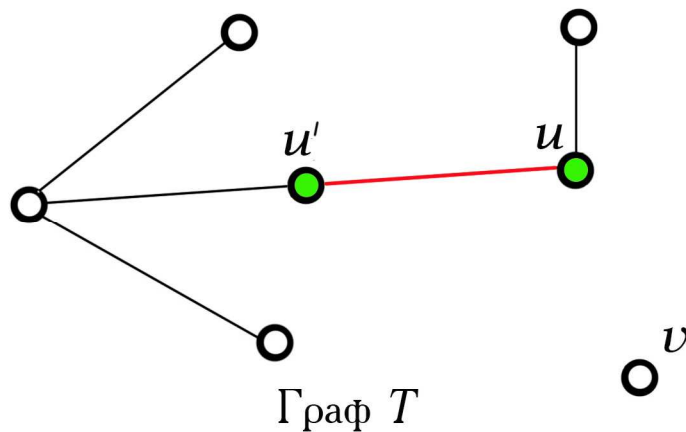
**Доведення.** Доведення випливає з того, що видалення ребра  $e = (u, v)$  в дереві  $T$  через наявність єдиного ланцюга, який з'єднує вершини  $u$  і  $v$ , веде до появи двох компонент зв'язності: одна компонента містить вершину  $u$ , а інша – вершину  $v$ .



Отже, граф  $T - e$  не є зв'язним.

**Теорема 2.** Якщо  $T = (V, E)$  – дерево і  $v \notin V$  – вершина,  $u$  – довільна вершина  $u \in V$ , то граф  $T' = (V \cup \{v\}, E \cup \{(u, v)\})$ , теж є деревом.

**Доведення.** Оскільки  $T$  – дерево, то існує єдиний ланцюг, що з'єднує будь-яку вершину  $u'$  з вершиною  $u$ . Оскільки вершина  $v \notin V$ , то додавання одного кінцевого ребра  $(u, v)$  приводить до того, що з кожної вершини  $u'$  маємо лише єдиний ланцюг, який з'єднує вершини  $u'$  і  $v$ . Виходячи з теореми 1 граф  $T'$  є деревом.

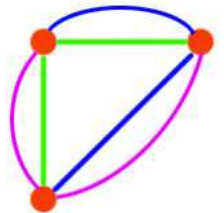


**Теорема 3.** Нехай дерево  $T$  має  $n$  вершин. Тоді еквівалентними є такі твердження:

1.  $T$  не має циклів і має  $n - 1$  ребро.
2.  $T$  – зв'язний граф і має  $n - 1$  ребро.
3.  $T$  – зв'язний граф і кожне його ребро є мостом.
4. Будь-які дві вершини графа  $T$  з'єднані тільки одним простим ланцюгом.
5.  $T$  не має циклів, але додавання будь-якого нового ребра в  $T$  сприяє виникненню тільки одного циклу.

**Теорема Келі.** Кількість різних дерев  $S$ , які можна побудувати на  $n$  вершинах, дорівнює  $n^{n-2}$ .

**Приклад.**  $n = 3$   $S = n^{n-2} = 3^{3-2} = 3$





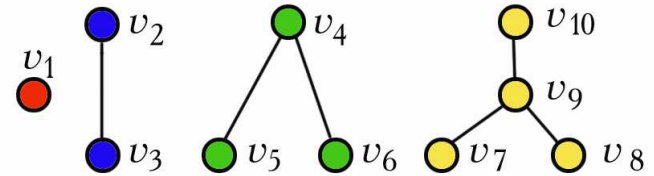
## Визначення. Ліс –

незв'язний  $n$ -граф без циклів.

1. Зв'язні компоненти лісу є деревами.

2. Будь-яка зв'язна компонента лісу або дерева також не має циклів.

3. Будь-яка частина лісу є лісом або деревом.



**Теорема.** Нехай ліс  $G$  містить  $n$  вершин і  $k$  компонентів. Тоді ліс  $G$  має  $n - k$  ребер.

**Доведення.** Оскільки кожне дерево має  $n - 1$  ребро, то кожний компонент лісу  $G_i$  має  $(n_i - 1)$  ребро. Але тоді число ребер у  $G$  дорівнює

$$(n_1 - 1) + (n_2 - 1) + \dots + (n_k - 1) = n_1 + n_2 + \dots + n_k - k = n - k,$$

що і потрібно довести.

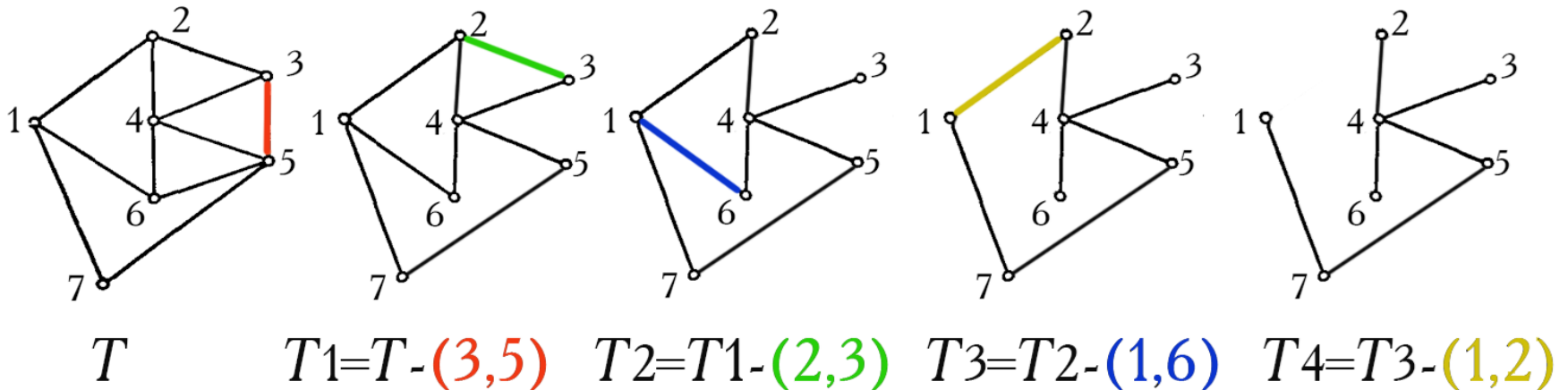
На рисунку  $n = 10$ ,  $k = 4$ ,  $|E| = 10 - 4 = 6$

## Процедура побудови остовного дерева

1.Видалимо зі зв'язного графа  $G$  одне ребро, що належить деякому циклу і не порушує зв'язності графа  $G$ .

2.Будемо повторювати видалення доти, поки в  $G$  не залишиться жодного циклу.

3.У результаті одержимо дерево, що містить усі вершини графа  $G$ . Це дерево називають *остовним деревом графа  $G$* .

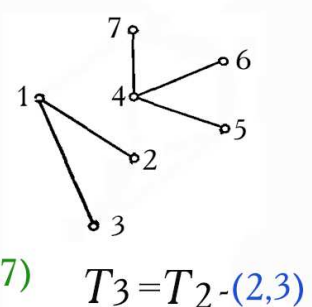
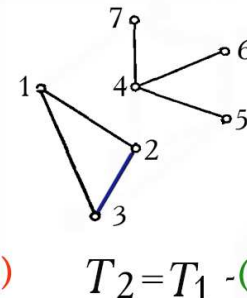
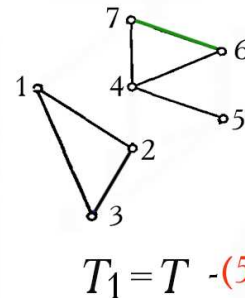
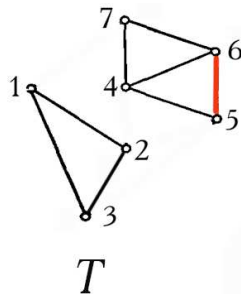


## Процедура побудови остовного лісу

Нехай  $G$  являє собою незв'язний граф з  $n$  вершинами,  $m$  ребрами і  $k$  компонентами.

1. Застосуємо процедуру видалення ребра до одного з циклів у кожній компоненті зв'язності графа  $G$ .
2. Будемо повторювати видалення доти, поки в кожній компоненті  $G$  не залишиться жодного циклу.
3. У результаті одержимо граф, який називають **остовним лісом**.
4. Число ребер, які при цьому видаляються, називають **цикломатичним числом** або **циклічним рангом графа**  $G$  і позначають  $C(G)$ . Таким чином, цикломатичне число є мірою зв'язності графа.

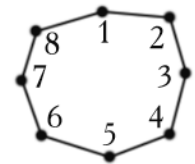
$$C(G) = 3$$



## Властивості циклічного рангу (цикломатичного числа)

1. **Циклічний ранг дерева** дорівнює нулю.
2. **Циклічний ранг циклічного графа** дорівнює одиниці.

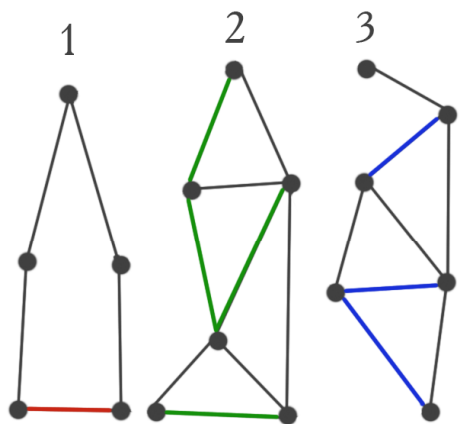
**Циклічний граф** – зв'язний регулярний граф степеня 2, єдина компонента зв'язності циклічного графа є простим циклом.



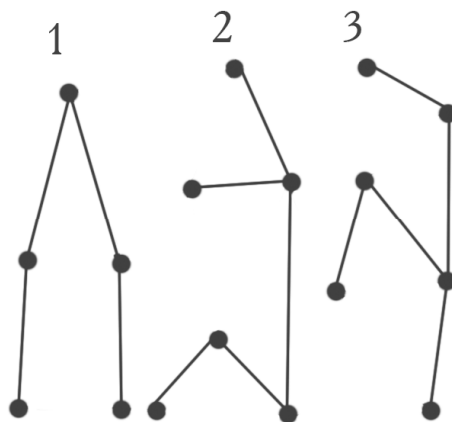
**Остовне дерево графа** – це дерево, що містить усі вершини графа.

**Остовним лісом** називають незв'язний граф, що складається з компонентів, які є остовними деревами.

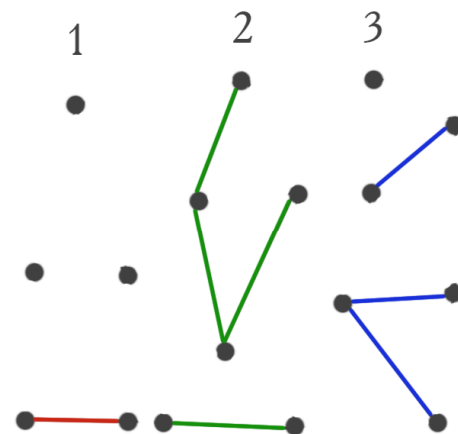
**Теорема.** Нехай  $T$  – остовний ліс графа  $G$ . Граф  $G'$ , отриманий із графа  $G$  шляхом видалення всіх ребер графа  $T$ , називають **доповненням остовного лісу**  $T$  графа  $G$ .



Незв'язний граф  
 $G$



Остовний ліс  
 $T$

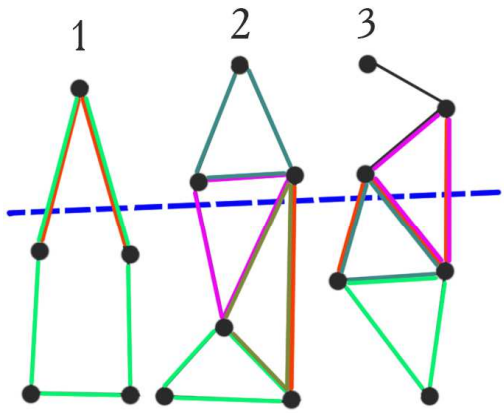


$G'$ -доповнення  
остовного лісу  $T$

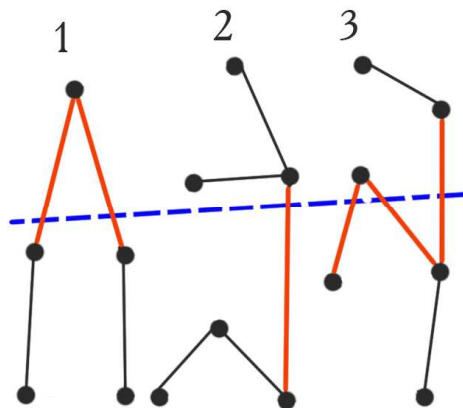
Не плутати з доповненням графа

**Теорема 4.** Якщо  $T$  – остовний ліс графа  $G$ , то

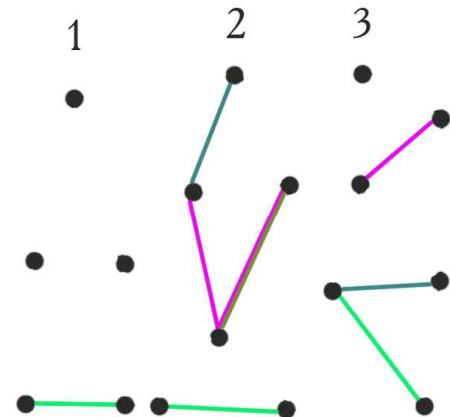
- а) усякий перетин в  $G$  має хоча б одне спільне ребро з  $T$  ;
- б) усякий цикл у  $G$  має спільне ребро з доповненням  $T$  .



Незв'язний граф  
 $G$



Остовний ліс  
 $T$



Доповнення  
остовного лісу  $G'$

## Фундаментальна система циклів графа

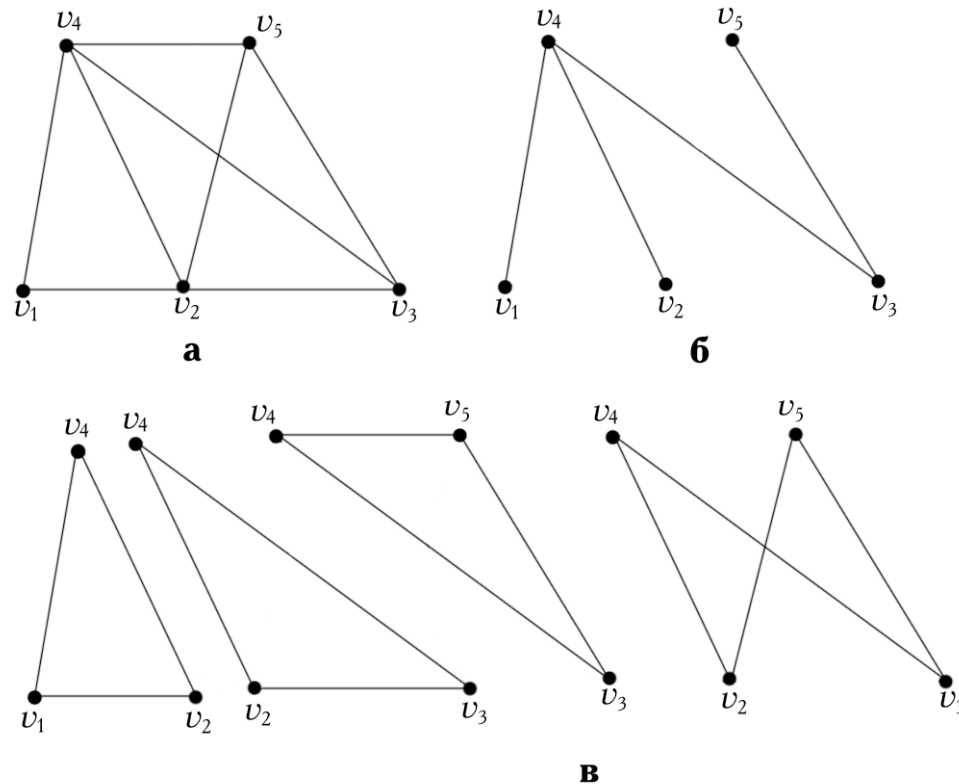
Нехай дано  $T$  – остовний ліс графа  $G$  .

Якщо додати до  $T$  будь-яке ребро графа  $G$ , що не входить до нього, то за п.5 теореми 3 одержимо один цикл.

- **Визначення.** Множину всіх циклів, які одержують шляхом додавання окремо кожного ребра з  $G$ , що не входить в  $T$ , називають фундаментальною системою циклів, асоційованою з  $T$ .

Цикли даної фундаментальної системи будуть різними, але їх кількість дорівнює циклічному рангу графа  $G$ .

На рисунку дано графи, які показують фундаментальну систему циклів.



**Рисунок:** **а** – граф  $G$ ; **б** – остовне дерево графа  $G$ ; **в** – фундаментальна система циклів, асоційована з остовним деревом графа  $G$ .




**Теорема 5.** Нехай  $G = (V, E)$  – довільний скінченний граф. Число ребер графа  $G$ , які необхідно вилучити для одержання остовного лісу  $T$ , не залежить від порядку їх видалення і дорівнює  $C(G) = |E| - |V| + k$ , де  $k$  – число компонент зв'язності графа  $G$ .

**Наслідок 1.** Граф  $G$  є остовним лісом тоді і тільки тоді, коли  $C(G) = 0$ .

**Наслідок.** Граф  $G$  має єдиний цикл тоді і тільки тоді, коли  $C(G) = 1$ .

**Наслідок 2.** Граф  $G$ , у якого число ребер перевищує число вершин, має цикл.

**Наслідок 3.** Будь-яке дерево порядку  $n \geq 2$  має принаймні дві кінцеві вершини. 

# Остовне дерево найменшої ваги

## Постановка задачі

Нехай кожному ребру графа  $G = (V, E)$  поставлена у відповідність деяка вага

$$d_i = d(e_i), \quad e_i \in E, \quad i = 1, 2, \dots, |E|.$$

Необхідно в графі  $G$  знайти остовне дерево, сума ваг  $d_i$  у якому найменша.

$$S = \min \sum_{e_i \in E} (d(e_i))$$

Число  $d_i$  називають вагою ребра  $e_i$ , а сам граф  $G$  – таким, що має вагу (зваженим).

**Отже, завдання полягає в тому, щоб знайти остовне дерево з найменшою вагою.**

## Розв'язок, що впливає з теореми Келі

1. Розглянемо всі можливі остовні дерева повного графа  $G$  з  $n$  вершинами.

Згідно з теоремою Келі число різних дерев, які можна побудувати на  $n$  вершинах, дорівнює  $n^{n-2}$ .

2. При виборі кожного дерева визначимо суму його ваг.

3. Застосувавши сортування за величиною суми ваг, на початку списку одержимо ті остовні дерева, які мають мінімальну суму ваг.

*Але оскільки число  $n^{n-2}$  навіть при невеликому  $n$  буде дуже великим, то такий шлях розв'язування даної задачі досить трудомісткий.*

*Тому актуальним є використання більш ефективних алгоритмів.*

## Попередні зауваження

1. **Порядок графа** – число, яке дорівнює **кількості вершин** графа.
2. **Порожній граф** – граф, що не містить ребер або регулярний граф степеня 0.

## Алгоритм Краскала

*Алгоритм Краскала* полягає у виконанні такої послідовності дій:

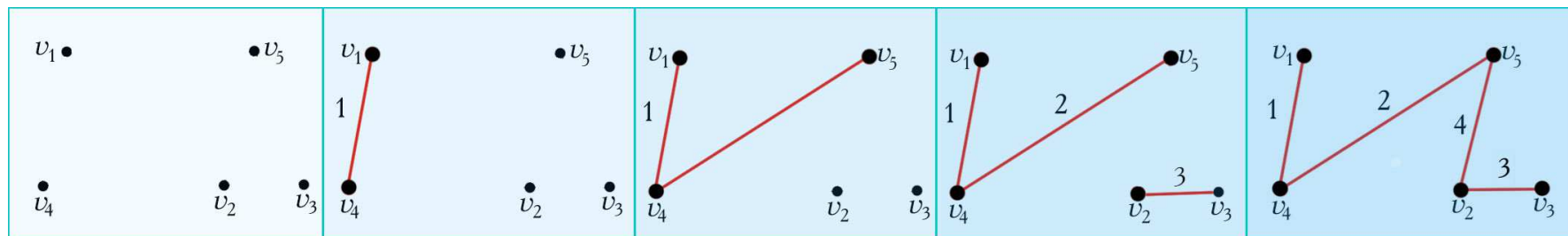
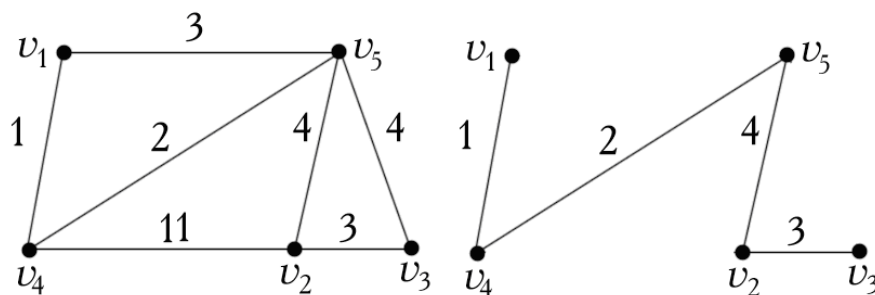
1. **Беремо порожній граф**  $O$  і будуємо граф  $T_1 = O + e_1$ , де  $e_1$  – ребро графа  $G = (V, E)$  мінімальної ваги.
2. Якщо граф  $T_k$  уже побудований і  $k < n - 1$ , то **будуємо граф**  $T_{k+1} = T_k + e_{k+1}$ , де  $e_{k+1}$  – ребро мінімальної ваги серед ребер графа  $G$ , що не ввійшли в граф  $T_k$ , **яке не становить циклу з ребрами графа**  $T_k$ .

Цей алгоритм базується на спеціальній теоремі.

**Теорема 7.** Нехай  $G$  – зв'язний граф порядку  $n$  і  $T$  – підграф графа  $G$ , отриманий у результаті виконання кроків 1 і 2 алгоритму Краскала. Тоді підграф  $T$  – остовне дерево графа  $G$  мінімальної ваги.

**Приклад.** Нехай дано граф  $G$ , показаний на рисунку ліворуч. Необхідно знайти остовне дерево мінімальної ваги цього графа за допомогою алгоритму Краскала.

**Розв'язок.**



## Властивості алгоритму Краскала

1. Алгоритм використовується для зважених неорієнтованих графів.
2. Алгоритм може бути використаний для побудови остовного лісу в багатокомпонентних незв'язних графах. У цьому випадку **структури даних**, що описують кожну з компонент зв'язності, **повинні бути окремими**.
3. Обчислювальна складність алгоритму Краскала  $O(e \cdot \log e)$ , де  $e$  – кількість ребер у даному графі.

## Алгоритм Прима

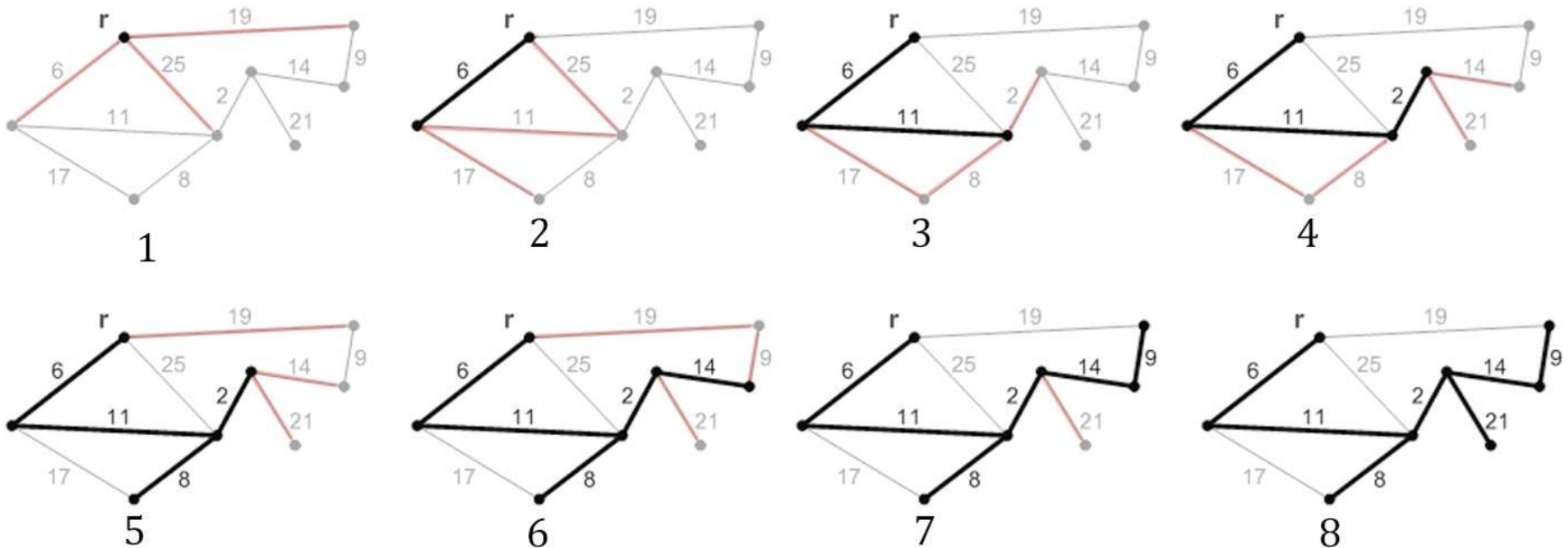
Алгоритм Прима полягає у виконанні такої послідовності дій:

1. У порожньому графі  $O$  **вибираємо довільну вершину**.
2. Вибираємо ребро  $e_1$  мінімальної ваги до суміжної вершини і будуємо підграф  $T_1 = O + e_1$ .
3. Для побудованого графа  $T_k$ , за умови  $k < n - 1$ , будуємо граф  $T_{k+1} = T_k + e_{k+1}$ , де  $e_{k+1}$  – ребро мінімальної ваги, що з'єднує вершину графа  $T_k$  із суміжною вершиною, яка не включена в множину вершин графа  $T_k$ .

**Приклад.** Нехай даний граф  $G$ . Необхідно знайти остовне дерево мінімальної ваги цього графа за допомогою алгоритму Прима.

**Розв'язок.** 1. **Вибираємо довільну вершину  $r$**  і проводимо інцидентне ребро мінімальної ваги.

2. Потім, переглядаючи інцидентні ребра на кожній з кінцевих вершин, знаходимо ребро мінімальної ваги.





# Властивості алгоритму Прима

1. Алгоритм використовується для зважених неорієнтованих зв'язних графів.
2. Обчислювальна складність алгоритму Прима  $O(n^2)$ , де  $n$  – кількість вершин графа. Якщо значення  $n$  достатньо велике, то використовувати цей алгоритм не раціонально.
3. Якщо кількість ребер  $e$  значно менша, ніж  $n^2$ , то алгоритм Краскала кращий, але якщо  $e$  близька до  $n^2$ , то рекомендують застосовувати алгоритм Прима.

## Структура алгоритму Прима

Даний граф  $G(V, E)$ , де множина вершин  $V = \{1, 2, \dots, i, \dots, n\}$

$V \neq \emptyset$  # множина вершин графа  $G$ .

$T = \emptyset$  # множина ребер остовного дерева.

$U = \emptyset$  # множина вершин остовного дерева.

**def Prim** ( $G$ ) :

$u_i = v_i$  # задаємо довільну вершину дерева

$U = \{u_i\}$

**while**  $U \neq V$  :

Знаходимо ребро  $(u_i, v_j)$  найменшої ваги і

таке, що  $u_i \in U$  і  $v_j \in V \setminus U$

$T = T \cup \{(u_i, v_j)\}$

$U = U \cup \{v_j\}$

print ( $T$ )

## Обхід графів. Основні положення

**Обійти граф** – це значить побувати у всіх вершинах точно по одному разу.

**Алгоритми обходу** складаються з:

1. **Послідовного відвідування вершин.** При цьому:

1.1. Вершину, яка ще **не відвідана, називають новою.**

1.2. **Факт відвідування вершини запам'ятовується**, так що з моменту відвідування і до кінця роботи алгоритму вона вважається відвіданою.

1.3. **У результаті відвідування вершина стає відкритою** і залишається такою, поки не будуть досліджені всі інцидентні їй ребра.

1.3. **Після дослідження всіх інцидентних ребер вона перетворюється в закриту.**

2. **Дослідження ребер.**

Які саме дії виконуються при відвідуванні вершини і дослідженні ребра – залежить від конкретної задачі

**Алгоритми обходу графа:** - обхід графа в глибину  
- обхід графа в ширину.

## Обхід у глибину

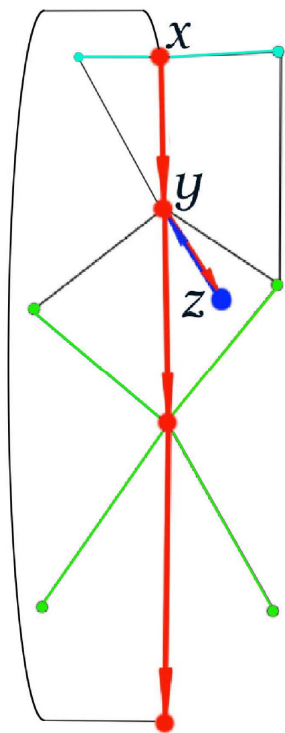
Обхід у глибину – це обхід графа за такими правилами:

1. Перебуваючи у вершині  $x$ , потрібно рухатися **в будь-яку іншу**, раніше не відвідану вершину  $y$  (якщо така знайдеться), одночасно **запам'ятовуючи ребро**, по якому ми вперше потрапили в дану вершину.

2. Якщо з вершини  $z$  ми **не можемо потрапити** в раніше **не відвідану** вершину або такої немає, то ми **повертаємося у вершину**  $y$ , з якої вперше потрапили в  $z$ , і

**продовжуємо обхід** у глибину з вершини  $y$ .

При виконанні обходу графа за цими правилами ми прагнемо проникнути "углиб" графа так далеко, як тільки це можливо, потім відступаємо на крок назад і знову прагнемо пройти вперед, і так далі.



## Приклад обходу графа в глибину

Пошук у глибину починаємо з будь-якої вершини. Рекурсивно застосуємо до всіх вершин, у які можна потрапити з поточної, таку послідовність дій.

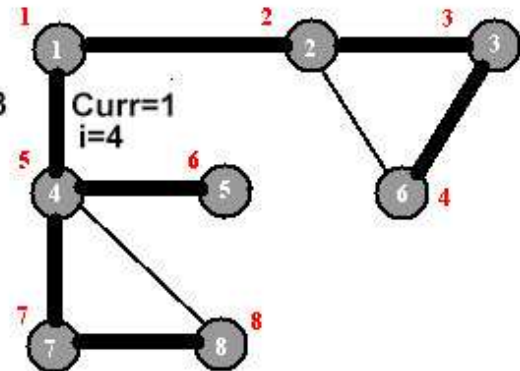
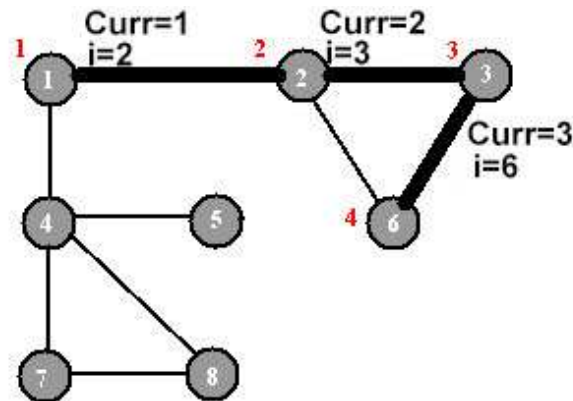
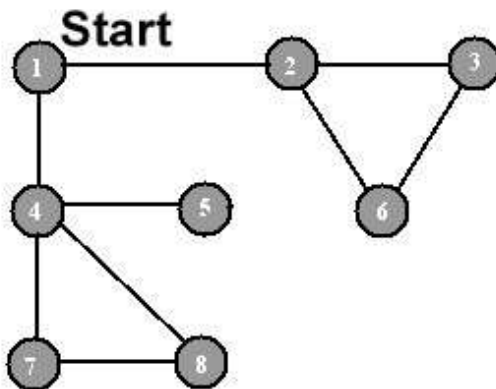
1. **Задамо граф** з  $V$  вершин матрицею суміжності  $A$  розміром  $V \times V$  у вигляді списку  $A[V, V]$ .
2. **Задамо одновимірний список**  $Visited[V]$  і заповнимо його нулями, вважаючи, що жодна вершина до початку виконання алгоритму не була відвідана.
3. **Задамо рекурсивну процедуру** обходу графа в глибину.

```

def go(Curr): #Curr - start node
    Visited[Curr]=1 # Check curr node
    for i in range(1,len(V)+1):
        If Visited[i]==0 AND (A[Curr,i]==1:
            go(i);
def Depth:
    go(Start)

```

	1	2	3	4	5	6	7	8
1	0	1	0	1	0	0	0	0
2	1	0	1	0	0	1	0	0
3	0	1	0	0	0	1	0	0
4	1	0	0	0	1	0	1	1
5	0	0	0	1	0	0	0	0
6	0	1	1	0	0	0	0	0
7	0	0	0	1	0	0	0	1
8	0	0	0	1	0	0	1	0



## Обхід в ширину

Обхід в ширину – це обхід графа за наступними правилами:

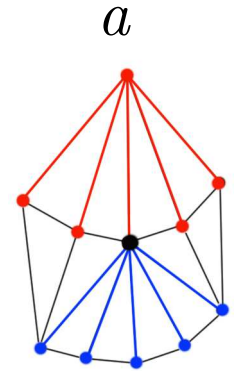
2. Нехай ми перебуваємо у початковій вершині  $a$ .

3. Із цієї вершини відбувається перегляд відразу всіх ще не переглянутих вершин, суміжних вершині  $a$ . Таким чином, пошук ведеться у всіх можливих напрямках одночасно.

4. Потім проглядаються вершини, що перебувають від  $a$  на відстані 2, і т.д.

Чим ближче вершина до стартової вершини, тем раніше вона буде відвідана.

Обхід в ширину шукає найкоротший можливий шлях.



## Приклад обходу графа в ширину

**При пошуку в ширину** замість стека рекурсивних викликів використовується черга, у яку записуються вершини в порядку віддалення від початкової.  $n = |V|$

1. **Задамо список**  $Visited[n]$  і заповнимо його нулями, вважаючи, що жодна вершина до початку виконання алгоритму не була відвідана.

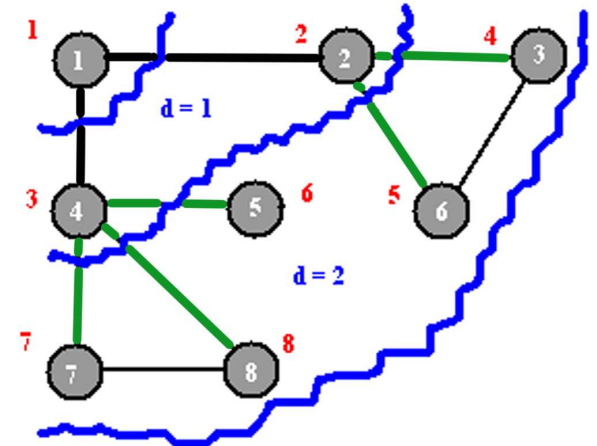
2. **Створимо чергу** для зберігання вершин у вигляді списку  $queue[n]$ . У початок черги запишемо початкову вершину.

$queue[1] = a$

3. **Змінна**  $rd$  вказує на позицію в черзі, з якої ми **читаємо** дані.

4. **Змінна**  $wt$  вказує на позицію в черзі, куди ми **будемо писати** дані.

5. Задамо процедуру обходу в ширину.





```
rd= 0, wt=1;
```

```
queue[1]=1
```

```
while (rd < wt):
```

```
    rd=rd+1
```

```
    Curr = queue[rd] #вибрали активну вершину
```

```
    for i in range(1, |V|+1): #відвідуємо всі суміжні вершини
```

```
        if ((Visited[i]=0) AND (A[curr,i]=1):
```

```
            Visited[i]= 1 #відзначаємо відвідану вершину
```

```
            wt=wt+1
```

```
            queue[wt]= i #ставимо її в чергу активних вершин
```

## Пошук шляхів у графі.

### Алгоритм Террі

Виходячи з початкової вершини й здійснюючи послідовний перехід від кожної досягнутої вершини до суміжної вершини, дотримуватися таких правил.

1. Позначати напрямок, у якому проходимо ребро, як у прямому, так і у зворотному напрямках.
2. З будь-якої вершини просуватися тільки по тому ребру, яке ще не було пройдено або було пройдено в протилежному напрямку.
3. Для будь-якої вершини відзначати перше ребро, по якому до неї потрапили, якщо вершина зустрічається вперше.

4. Будь-яку вершину покидати по ребру, по якому прийшли в цю вершину (у зворотному напрямку) лише тоді, коли немає іншої можливості.

### Пояснення алгоритму Террі

Алгоритм Террі – це алгоритм пошуку шляху в зв'язному графі, з вершини  $v_i$  у вершину  $v_j$ , де  $v_i \neq v_j$ .

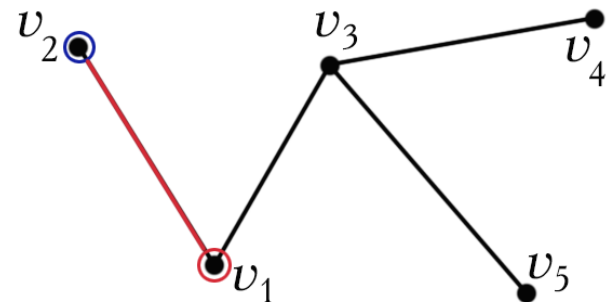
Приклад роботи алгоритму для пошуку шляху з  $v_1$  у  $v_5$ .

Переходимо до вершини з множини  $v_k \in \Gamma^+(v_i)$  за такими правилами:

**1.** Вибираємо довільну дугу, проходимо по ній і відзначаємо напрямок проходження.

Нехай початкова вершина  $v_1$

Вибрали першу суміжну  $v_2$



Як правило, вибираємо суміжну вершину з найменшим номером

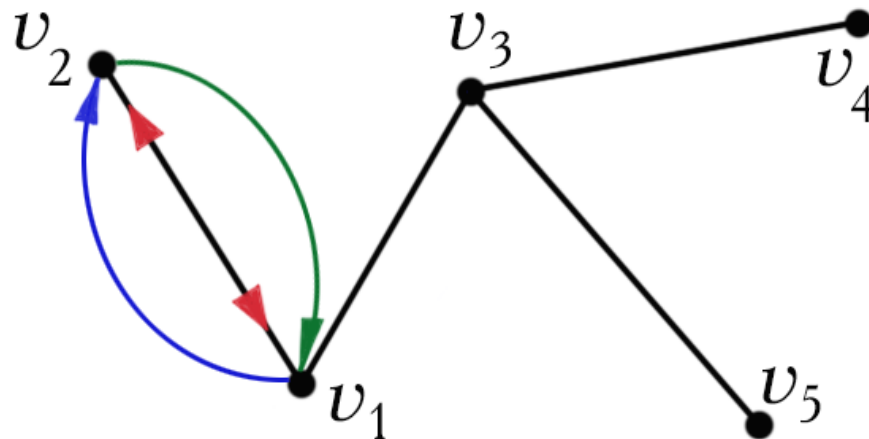
2. Виходимо з вершини  $v_k$  по дузі, яка ще не була вибрана, або була пройдена в протилежному напрямку.

3. Відмічаємо проходження ребра в прямому і протилежному напрямку

*Перейшли у вершину  $v_2$*

*Відмітили ребро  $(v_1, v_2)$*

*Вияснили, що іншого шляху, ніж  $(v_1, v_2)$  не існує*



4. Вибираємо те ребро, по якому прийшли, тільки у випадку, коли не існує іншого шляху.

Знову знаходимось  
у вершині  $v_1$

Відмічаємо ребро  $(v_1, v_3)$

Перейшли у вершину  $v_3$

---

Знаходимось у вершині  $v_3$

Відмічаємо ребро  $(v_3, v_4)$

Перейшли у вершину  $v_4$

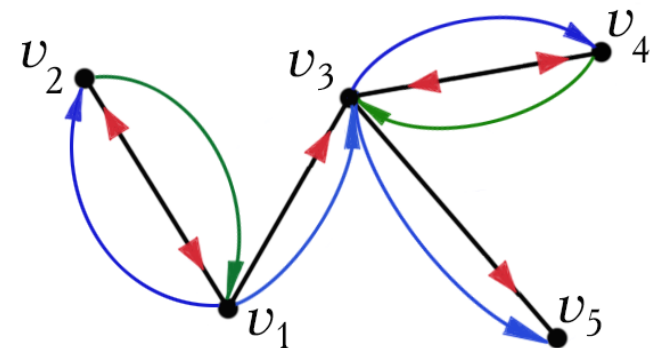
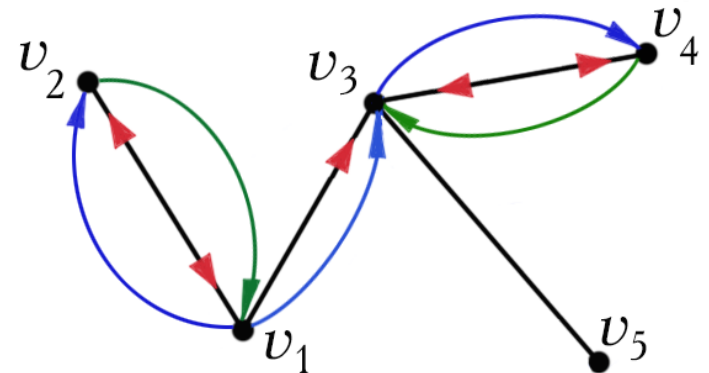
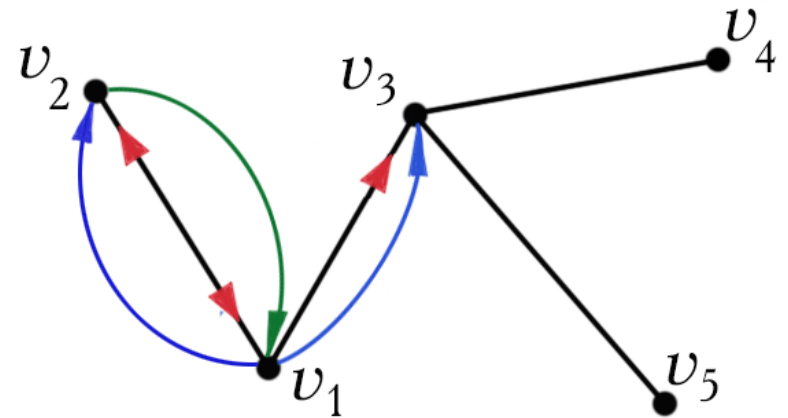
Відмічаємо ребро  $(v_4, v_3)$

---

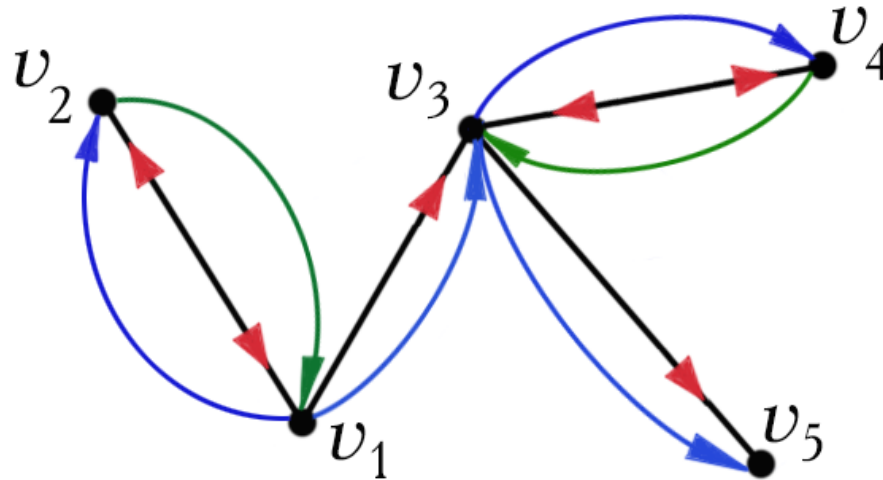
Знаходимось у вершині  $v_3$

Відмічаємо ребро  $(v_3, v_5)$

Перейшли у вершину  $v_5$



## ВИСНОВОК



За алгоритмом Террі одержали такий шлях:

$v_1, v_2, v_1, v_3, v_4, v_3, v_5$

Якщо виключити цикли, то одержимо простий ланцюг:

$v_1, v_3, v_5$

## Хвильовий алгоритм. Структура даних

Нехай  $G = (V, E)$  непустий граф. Потрібно знайти шлях між вершинами  $s$  і  $t$  графа ( $s \neq t$ ), який містить мінімальну кількість проміжних вершин (ребер).

### Структура даних

1.  $\text{Time}[i]$  – масив хвильових міток вершин графа  $G$ .

Початкова установка:

```
for i in range (1, len (V) +1) : Time [i] = -1
```

```
else Time [s] = 0
```

2.  $\text{OldFront}$  – множина вершин старого фронту хвилі.

Початкова установка:  $\text{OldFront} = \{s\}$ .

3.  $\text{NewFront}$  – множина вершин нового фронту хвилі.

Початкова установка:  $\text{NewFront} = \{\}$ .

4.  $T$  – змінна поточного часу.

Початкова установка:  $T = 0$ .

## Алгоритм полягає в наступному:

1. Для кожної з вершин, що входять в OldFront, переглядаємо суміжні вершини  $u_i$ )
2. if  $\text{Time}[u_i] == -1$  :  
     $\text{Time}[u_i] = T + 1$   
     $\text{NewFront} = \text{NewFront} \cup \{u_i\}$ ;
3. Якщо  $\text{NewFront} == \{ \}$ , то КІНЕЦЬ ("немає розв'язку");
4. Якщо  $t \in \text{NewFront}$  ( тобто одна з вершин  $u_i == t$  ), то знайдено найкоротший шлях між  $s$  і  $t$  з  $\text{Time}[t] = T + 1$  проміжними ребрами; КІНЕЦЬ ("розв'язок знайдений").  
Якщо ні одна з вершин  $u_i$  не співпадає з  $t$ , то goto 5
5.  $\text{OldFront} = \text{NewFront}$   
     $\text{NewFront} = \{ \}$ ;  $T = T + 1$ ; goto 1



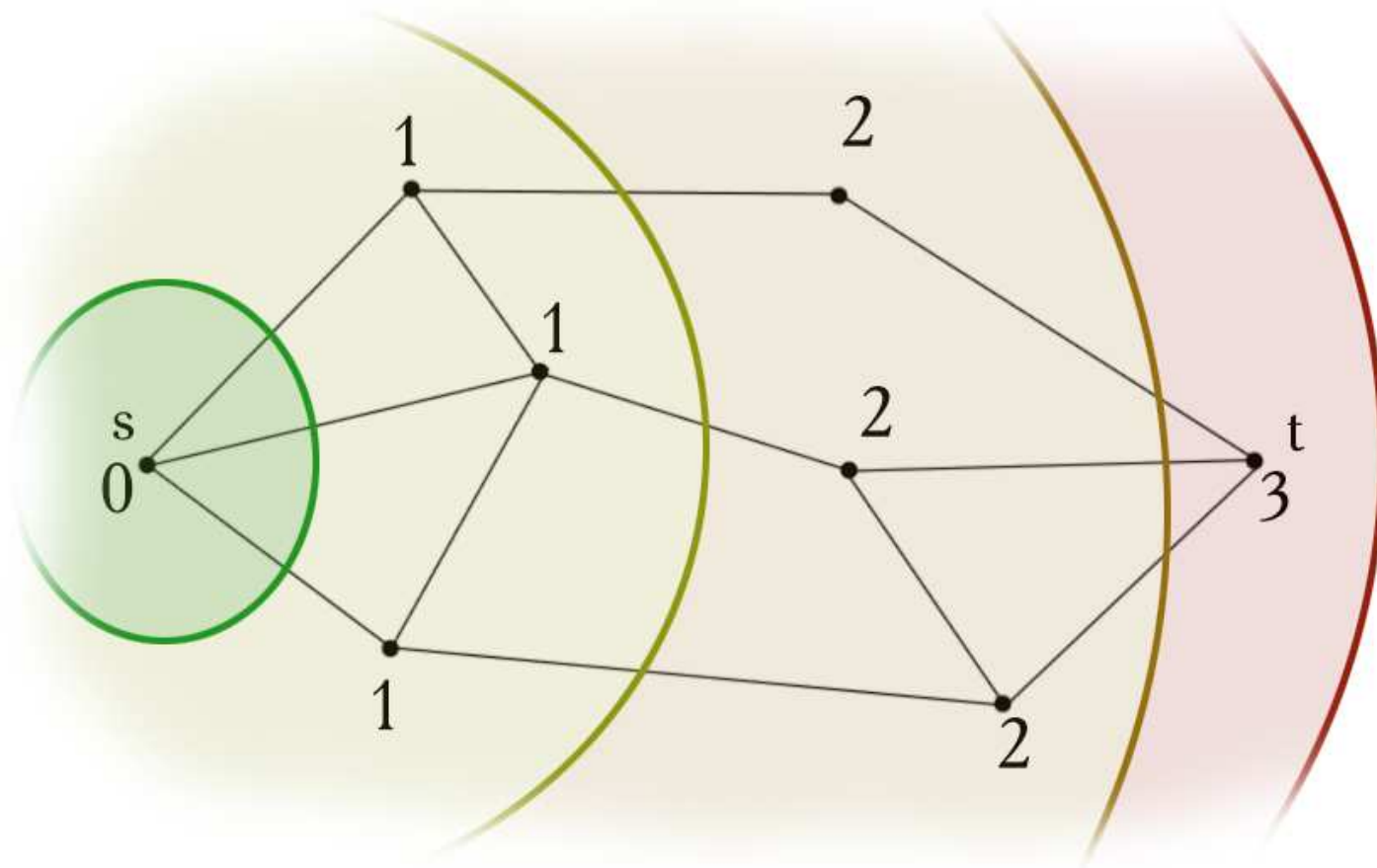
## Відновлення найкоротшого шляху.

Якщо на кроці (4) була досягнута вершина  $t$ , то алгоритм побудови найкоротшого шляху такий:

1. Серед сусідів вершини  $t$  знайдемо будь-яку вершину з хвильовою міткою  $\text{Time}[t] - 1$ .
2. Серед сусідів вже знайденої вершини знайдемо вершину з міткою  $\text{Time}[t] - 2$ , і т. д., поки не досягнемо  $s$ .

Знайдена послідовність вершин визначає один з найкоротших шляхів з  $s$  в  $t$

На практиці вигідно зберігати інформацію про те, з якої вершини "хвиля" прийшла у вершину  $u_i$  – тоді відновлення шляху відбувається швидше.



На малюнку показана нумерація вершин по  $\text{Time}[u_i]$ , отримана в результаті роботи хвильового алгоритму.

## Пошук найкоротшого шляху у зваженому графі

Нехай дано граф  $G$  з матрицею ваг  $C = [c(i, j)]$ .

**Задача про найкоротший шлях** полягає в знаходженні найкоротшого шляху від заданої **початкової вершини**  $s \in V$  до заданої **кінцевої вершини**  $t \in V$  за умови, що такий шлях існує.

Алгоритм **Дейкстри** знаходить найкоротший шлях **для випадку**  $c(i, j)$ .

1. Вершинам приписують **тимчасові позначки**. Нехай  $l(v_i)$  – позначка вершини  $v_i$ .
2. Кожна позначка вершини **дає верхню границю довжини шляху** від  $s$  до цієї вершини.
3. Величини цих **позначок зменшують ітераційно**.
4. На кожному кроці ітерації одна з тимчасових позначок **стає постійною**.
5. Величина цієї позначки є точною довжиною найкоротшого шляху від  $s$  до розглянутої вершини.

# Теоретичний опис алгоритму Дейкстри

## Присвоєння початкових позначок

**Крок 1.** Встановимо  $l(s) = 0$  і вважатимемо цю позначку постійною.

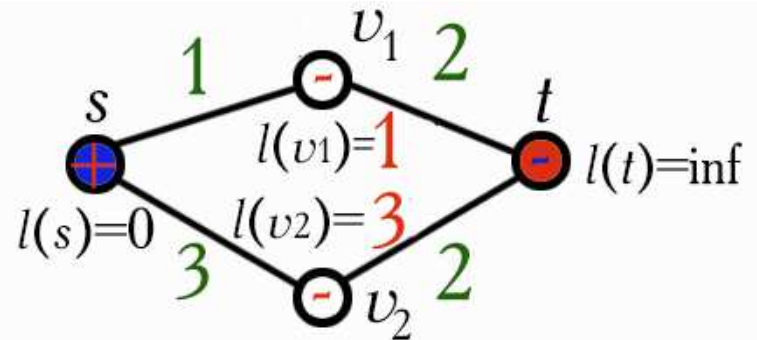
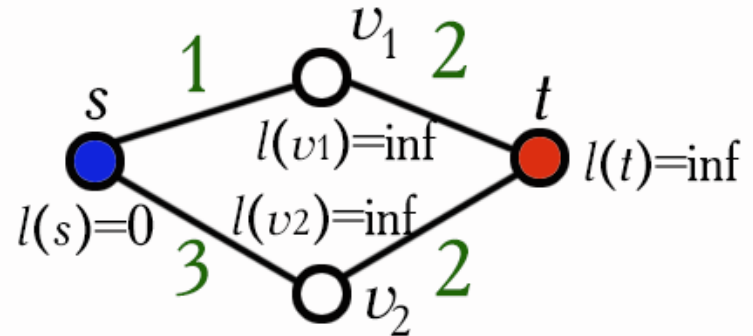
Встановимо  $l(v_i) = \infty$  для всіх  $v_i \neq s$  і вважатимемо ці позначки тимчасовими.

Встановимо  $cur = s$ .

## Встановлення нових позначок

**Крок 2.** Для всіх  $v_i \in \Gamma(cur)$  позначки яких тимчасові, змінімо позначки у відповідності з таким виразом:

$$l(v_i) = \min[l(v_i), l(cur) + c(cur, v_i)]$$



## Перетворення позначки в постійну

**Крок 3.** Серед вершин з тимчасовими позначками знайдемо таку, для якої

$$l(v_i^*) = \min(v_i), \quad v_i \in \Gamma(cur)$$

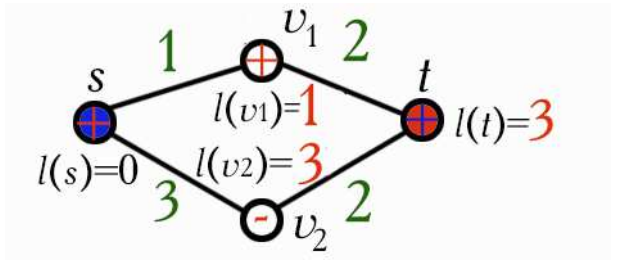
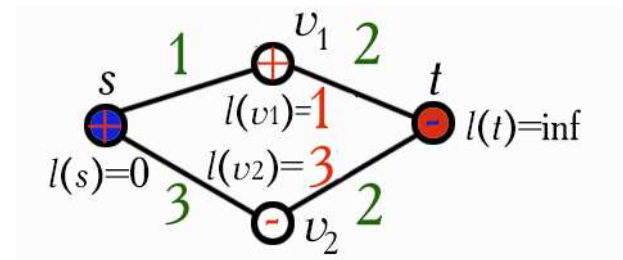
Вважаємо позначку  $l(v_i^*)$  постійною і встановлюємо  $cur = v_i^*$

**Крок 4.** Якщо потрібно знайти шлях від  $s$  до  $t$ . Якщо  $cur = t$ , то  $l(cur)$  є довжиною найкоротшого шляху від вершини  $s$  до вершини  $t$ . **Останов.**

**Крок 5.** Якщо  $cur \neq t$ , перейти до кроку 2.

**Крок 6.** Якщо потрібно знайти шляхи від  $s$  до всіх інших вершин. Якщо всі вершини позначені як постійні, то ці позначки дають довжини найкоротших шляхів. **Останов.**

**Крок 7.** Якщо деякі позначки залишаються тимчасовими, то перейти до кроку 2.



# Алгоритм Форда – Беллмана знаходження мінімального шляху

## Математичний опис

Нехай заданий граф  $G(V, E)$  з вагами ребер  $W(i, j)$  і виділеною початковою вершиною  $s$ . Позначимо через  $d(cur)$  найкоротшу відстань від джерела  $s$  до поточної вершини  $cur$ . Алгоритм Форда-Беллмана шукає функцію  $d(cur)$  як єдиний розв'язок рівняння

$$d(cur) = \min \{ d(i) + W(i, cur) \mid (i, cur) \in E \}, \forall cur \neq s$$

з початковою умовою  $d(s) = 0$ .

Основною операцією алгоритму є релаксація ребра.

Якщо  $(i, cur) \in E$  і  $d(cur) > d(i) + W(i, cur)$ , то виконують присвоєння нового значення  $d(cur) = d(i) + W(i, cur)$ .

## Макроструктура алгоритму

Алгоритм послідовно уточнює значення функції  $d(cur)$ .

1. На початку задаємо значення

$$d(s) = 0;$$

$$d(cur) = \infty \quad \forall cur \neq s.$$

2. Виконуємо  $n-1$  ітерацій під час яких виконуємо релаксацію всіх ребер графа.

### Вхідні дані:

Граф: вершини  $V$ , ребра  $(i, j) \in E$  з вагами  $W(i, j)$

Початкова вершина  $s$ .

**Вихідні дані:** відстані  $d(cur)$  до кожної вершини  $cur \in V$  від вершини  $s$ .

## Код алгоритму Форда – Беллмана

```
start=1  
  
ML = 10 ** 9  
d = [ML] * N  
d[start] = 0  
for k in range(1, N):  
    for i in range(N):  
        for j in range(N):  
            if d[j] + W[j][i] < d[i]:  
                d[i] = d[j] + W[j][i]
```



## Алгоритм Флойда-Уоршелла

Дано зважений граф  $G(V, E)$  з вершинами, що пронумеровані від 1 до  $n$ .

$$\text{Вага ребра } w_{uv} = \begin{cases} k, & (u, v) \in E, \\ \infty, & (u, v) \notin E. \end{cases}$$

Знаходимо матрицю найкоротших відстаней  $D$ , в якій елемент  $d_{ij}$  або дорівнює найкоротшій відстані від вершини  $i$  до вершини  $j$ , або дорівнює  $\infty$ , якщо вершина  $j$  не досяжна з  $i$ .

На кожному кроці беремо чергову вершину (нехай з номером  $i$ ) і для всіх пар вершин  $u$  і  $v$  будемо обчислювати  $d_{uv}^{(i)} = \min \left( d_{uv}^{(i-1)}, d_{ui}^{(i-1)} + d_{iv}^{(i-1)} \right)$

## Псевдокод

$$d_{uv}^{(0)} = w$$

*for*  $i$  *in*  $V$  :

*for*  $u$  *in*  $V$  :

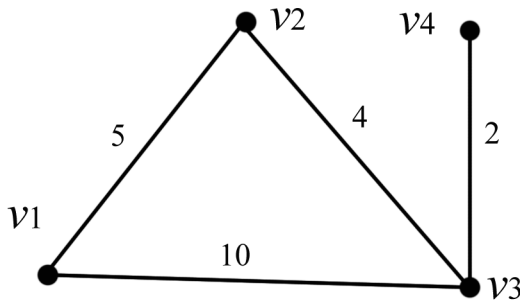
*for*  $v$  *in*  $V$  :

$$d_{uv}^{(i)} = \min \left( d_{uv}^{(i-1)}, d_{ui}^{(i-1)} + d_{iv}^{i-1} \right)$$

У підсумку отримуємо, що матриця  $D^{(n)}$  і є шуканою матрицею найкоротших шляхів, оскільки містить в собі довжини найкоротших шляхів між усіма парами вершин, що мають в якості проміжних вершин вершини з множини  $\{1, \dots, n\}$ , що є просто всі вершини графа.

На кожній ітерації перебирають всі пари вершин і шлях між ними скорочується за допомогою проміжної  $i$ -ї вершини.

**Приклад.** Розглянемо роботу алгоритму Флойда на прикладі графа  $G$ , що складається з чотирьох вершин.



1. Будуємо початкову матрицю  $T_0$ :

$$T_0 = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 5 & 10 & \infty \\ v_2 & 5 & 0 & 4 & \infty \\ v_3 & 10 & 4 & 0 & 2 \\ v_4 & \infty & \infty & 2 & 0 \end{pmatrix}$$

## Продовження прикладу роботи алгоритму Флойда

### 2. Модифікуємо матрицю $T_0$ при $k = 1$

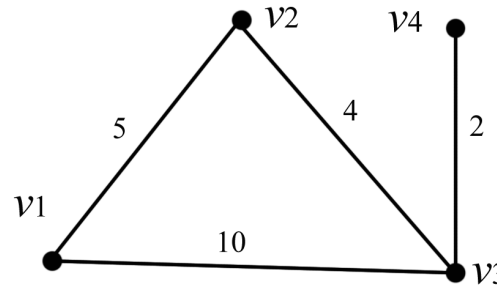
```
for i in range(1, 5):
```

```
    for j in range(1, 5):
```

```
        T[i, j] = min(T[i, j], T[i, 1] + T[1, j])
```

Замінюємо елемент матриці, якщо шлях між вершинами  $i$  та  $j$  виявиться коротшим через вершину  $k = 1$

$$T_1 = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 5 & 10 & \infty \\ v_2 & 5 & 0 & 4 & \infty \\ v_3 & 10 & 4 & 0 & 2 \\ v_4 & \infty & \infty & 2 & 0 \end{pmatrix}$$



На першому етапі в матриці немає змін, оскільки не знайдено більш коротких шляхів між вершинами через вершину 1

## 2. Модифікуємо матрицю $T_1$ при $k = 2$

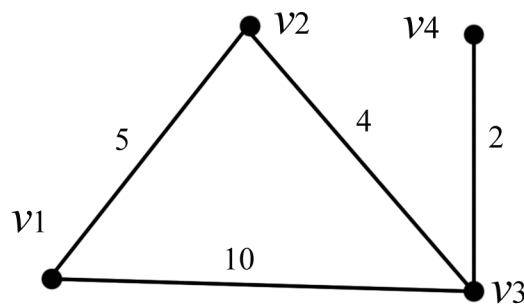
```
for i in range(1,5):
```

```
    for j in range(1,5):
```

```
        T[i,j] = min(T[i,j], T[i,2] + T[2,j])
```

Заміняємо елемент матриці, якщо шлях між вершинами  $i$  та  $j$  виявиться коротшим через вершину  $k = 2$

$$T_2 = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 5 & 9 & \infty \\ v_2 & 5 & 0 & 4 & \infty \\ v_3 & 9 & 4 & 0 & 2 \\ v_4 & \infty & \infty & 2 & 0 \end{pmatrix}$$



На другому етапі шлях з вершини 1 у вершину 3 виявився коротшим через вершину 2.

### 3. Модифікуємо матрицю $T_2$ при $k=3$

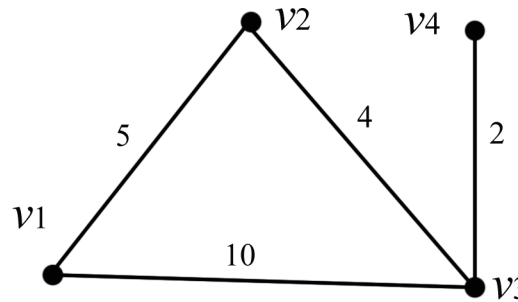
```
for i in range(1,5):
```

```
    for j in range(1,5):
```

```
        T[i,j] = min(T[i,j], T[i,3] + T[3,j])
```

Заміняємо елемент матриці, якщо шлях між вершинами  $i$  та  $j$  виявиться коротшим через вершину  $k=3$

$$T_3 = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 5 & \textcolor{red}{9} & \textcolor{blue}{11} \\ v_2 & 5 & 0 & 4 & \textcolor{violet}{6} \\ v_3 & \textcolor{red}{9} & 4 & 0 & 2 \\ v_4 & \textcolor{blue}{11} & \textcolor{violet}{6} & 2 & 0 \end{pmatrix}$$



На 3-му етапі встановили шлях у вершину 4 через вершину 3.

#### 4. Модифікуємо матрицю $T_3$ при $k = 4$

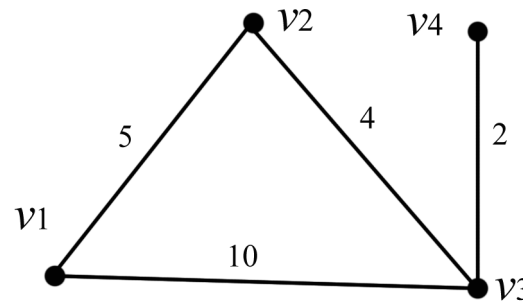
```
for i in range(1, 5):
```

```
    for j in range(1, 5):
```

```
        T[i, j] = min(T[i, j], T[i, 4] + T[4, j])
```

Заміняємо елемент матриці, якщо шлях між вершинами  $i$  та  $j$  виявиться коротшим через вершину  $k = 4$

$$T_4 = \begin{pmatrix} & v_1 & v_2 & v_3 & v_4 \\ v_1 & 0 & 5 & 9 & 11 \\ v_2 & 5 & 0 & 4 & 6 \\ v_3 & 9 & 4 & 0 & 2 \\ v_4 & 11 & 6 & 2 & 0 \end{pmatrix}$$



На 4-му етапі не знайшли оптимальних шляхів через вершину 4, оскільки вона висяча.