# Лекція 11.

## Алгоритми пошуку остовних дерев

**Що таке обхід графа?**

Простими словами, обхід графа — це перехід від одніє його вершини до іншої в пошуку властивостей зв`язків цих вершин. Зв`язки (лінії, що з`єднують вершини) називаються напрямками, шляхами, гранями або ребрами графа. Вершини графа також називаються вузлами.

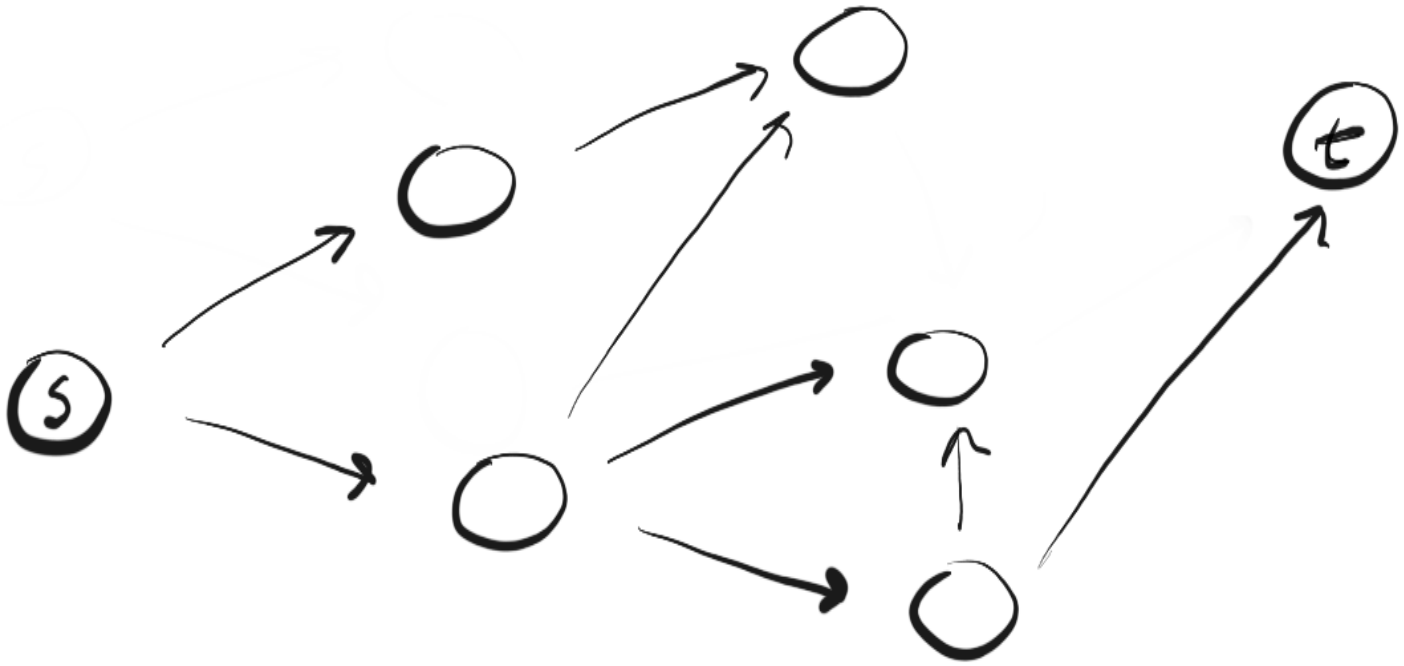
Двома основними алгоритмами обходу графа є пошук в глибину (Depth-FirstSearch, DFS) і пошук в ширину (Breadth-FirstSearch, BFS).

Не дивлячись на те, що обидва алгоритми використовуються для обходу графа, вони мають деякі відмінності. Почнемо з DFS.

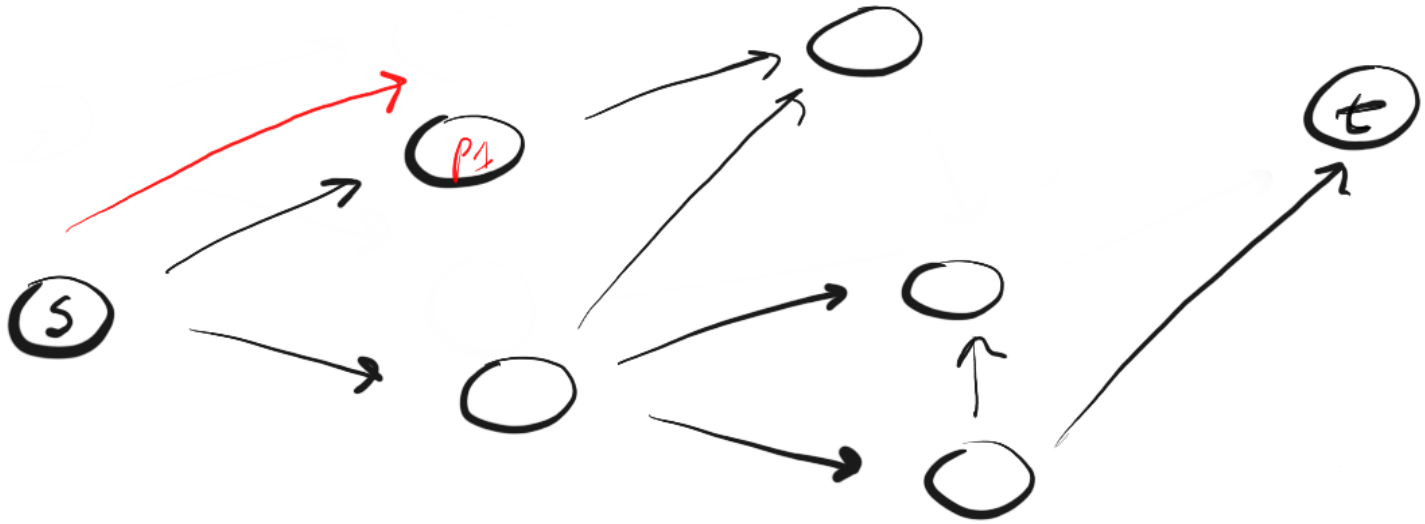
**Пошук в глибину**

DFS слідує концепції «поринай глибже, головою вперед» («godeep, headfirst»). Ідея полягає в тому, що ми рухаємося від початкової вершини (точки, місця) в заданому напрямку (за заданим шляхом) до тих пір, поки не досягнемо кінця шляху або пункту призначення (шуканої вершини). Якщо ми досягли кінця шляху, але він не є пунктом призначення, то ми повертаємося назад (до точки розгалуження шляхів) і йдемо по іншому маршруту.

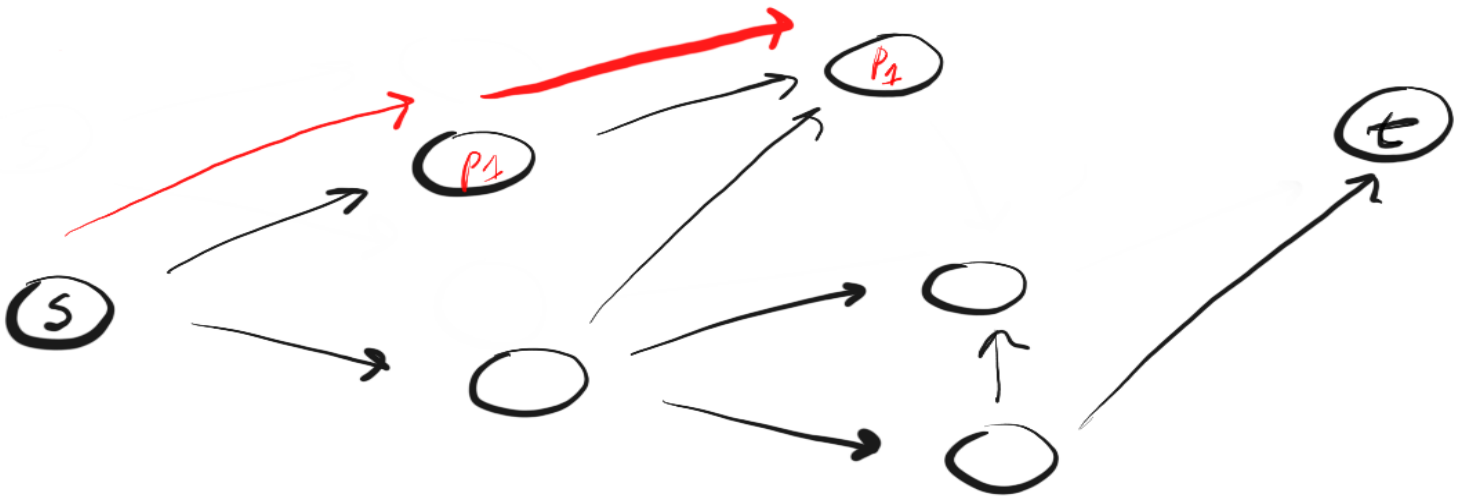
Давайте розглянемо приклад. Нехай, в нас є орієнтований граф, який має наступний вигляд:



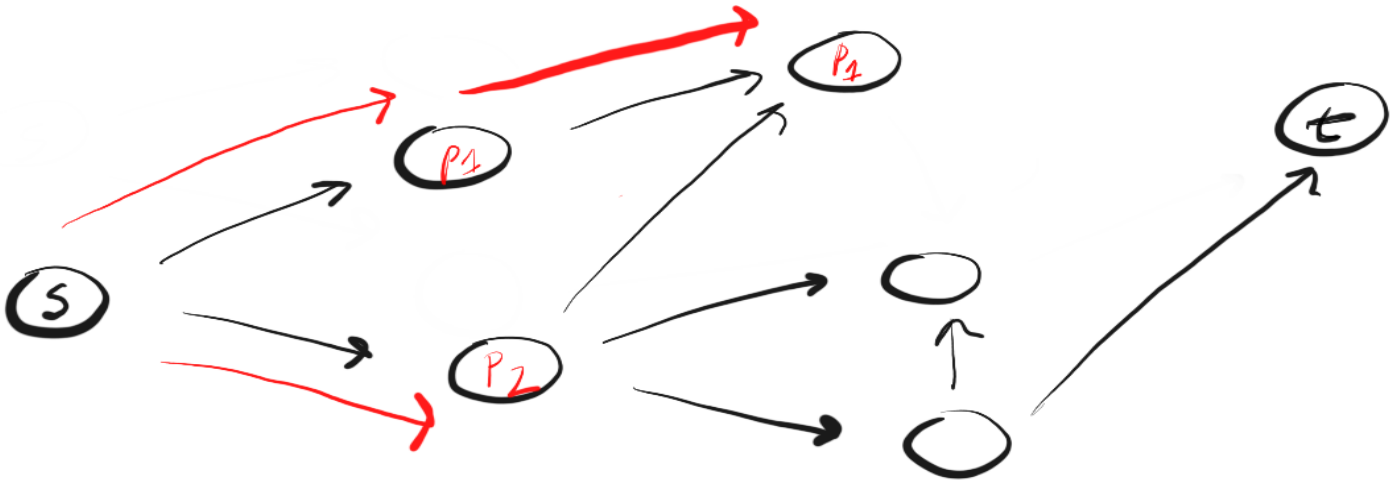

Ми знаходимося в точці «s» і нам потрібно знайти вершину «t». Застосовуючи DFS, ми досліджуємо один з можливих шляхів, рухаємося по ньому до кінця і, якщо не знайшли t, повертаємося і досліджуємо інший шлях. Ось як виглядає процес:



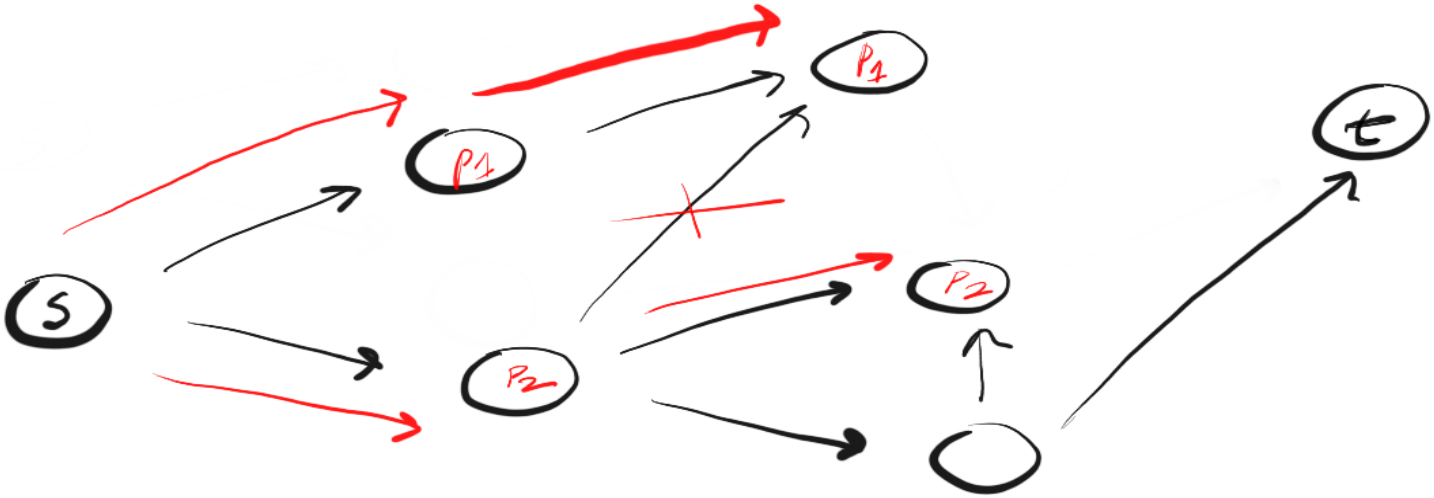

Тут ми рухаємося по маршруту (p1) до найближчої вершини і бачимо, що це не кінець шляху. Тому ми переходимо до настуної вершини.



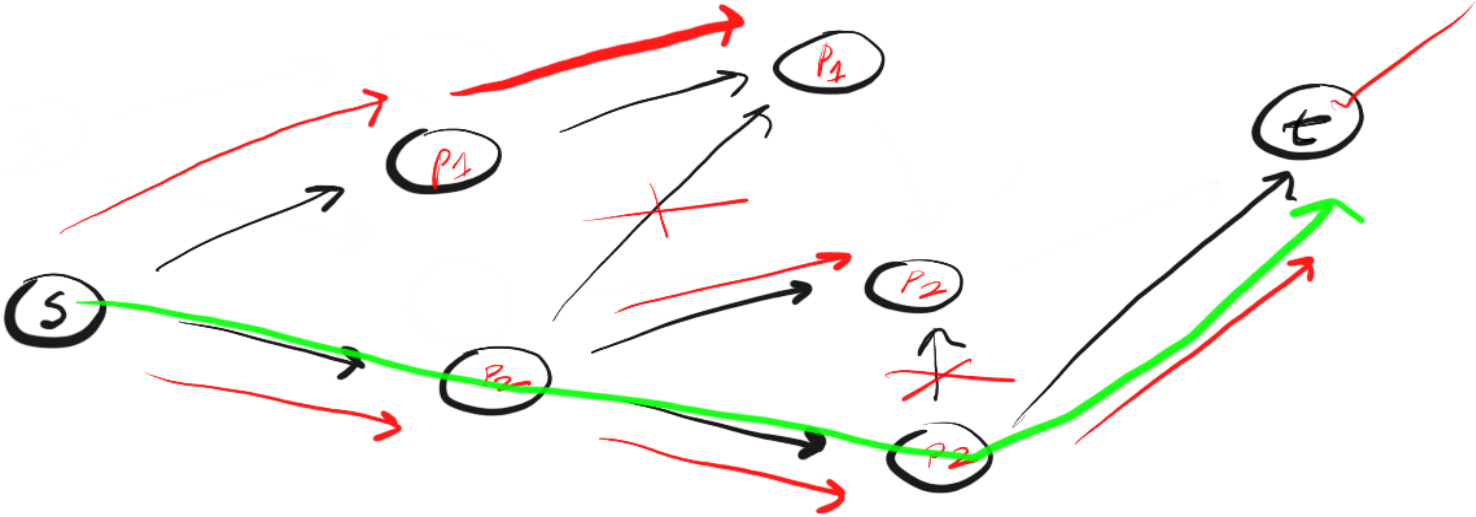

Ми досягли кінця p1, але не знайшли t, тому повертаємося в s і рухаємося по іншому маршруту.




Досягнувши найближчої до точки «s» вершини шляху «p2» ми бачимо три можливих напрямки для подальшого руху. Оскільки вершину, де закінчувався перший напрямок, ми вже відвідували, то рухаємося по другому.




Ми знову досягли кінця шляху, але не знайшли t, тому повертаємося назад. Слідуємо за третім шляхом і, нарешті, досягаємо шуканої вершини «t».




Так працює DFS. Рухаємося за заданим шяхом до кінця. Якщо кінець шляху — це шукана вершина, ми закінчили. Якщо ні, повертаємося назад і рухаємося іншим шляхом, доти, доки не дослідимо всі варіанти.

Ми слідуємо за цим алгоритмом, застосовуючи його до кожної відвідуваної вершини.

Необхідність багаторазового повторення процедури вказує на необхідність застосування рекурсії для реалізації алгоритму.

Вот JavaScript-код:

```
// за умови, що ми маємо справу з суміжним списком
// наприклад, таким: adj = {A: [B,C], B:[D,F], ... }
functiondfs(adj, v, t) {
        // adj - суміжний список
        // v - відвіданийвузол (вершина)
        // t - пункт призначення
        // це загальні випадки
        // абодосягли пунктупризначення, абовже відвідуваливузол
        if(v === t) return true
        if(v.visited) return false
        // помічаємовузол як відвіданий
        v.visited = true
        // досліджуємовсіх сусідів (найближчі сусідні вершини) v
        for(let neighbor of adj[v]) {
                // якщосусідневідвідувався
                if(!neighbor.visited) {i
                        // рухаємосяшляхомі перевіряємо, чи недосягли ми
пунктупризначення
                        let reached = dfs(adj, neighbor, t)
                        // повертаємо true, якщодосягли
                        if(reached) return true
                }
        }
        // якщо від v до t дістатися неможливо
        returnfalse
}
```

Примітка: цей спеціальний DFS-алгоритм дозволяє перевірити, чи можливо дістатися з одного місця в інше. DFS може застосовуватися з різною метою. Від цієї мети залежить те, як буде виглядати сам алгоритм. Тим не менше, загальна концепція виглядає саме так.

**Аналіз DFS**

Давайте проаналізуємо цей алгоритм. Оскільки ми обходимо кожного «сусіда» кожного вузла, ігноруючи тих, яких відвідували раніше ми маємо час виконання, що дорівнює O(V + E).

Коротке пояснення того, що означає V+E:

V — загальна кількість вершин. E — загальна кількість граней (ребер).

Може здатися, що правильніше застосувати V*E, однак давайте подумаємо, що означає V*E.

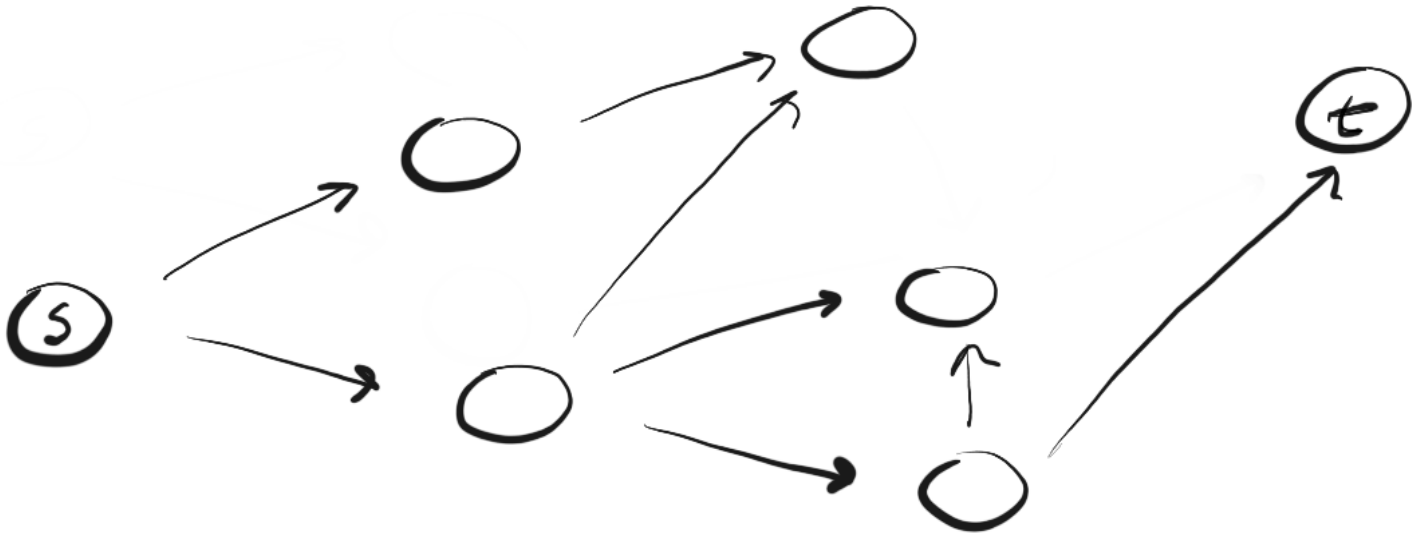
V*E означає,що стосовно кожної вершини, ми повинні дослідити всі грані графа безвідносно приналежності цих граней конкретній вершині.

З іншого боку, V+E означає, що для кожної вершини ми оцінюємо лише грані, що примикають до неї. Повертаючись до прикладу, кожна вершина має визначену кількість граней і, в гіршому випадку, ми обійдемо всі вершини (O(V)) і дослідимо всі грані (O(E)). Ми маємо V вершин і E граней, тому отримаємо V+E.

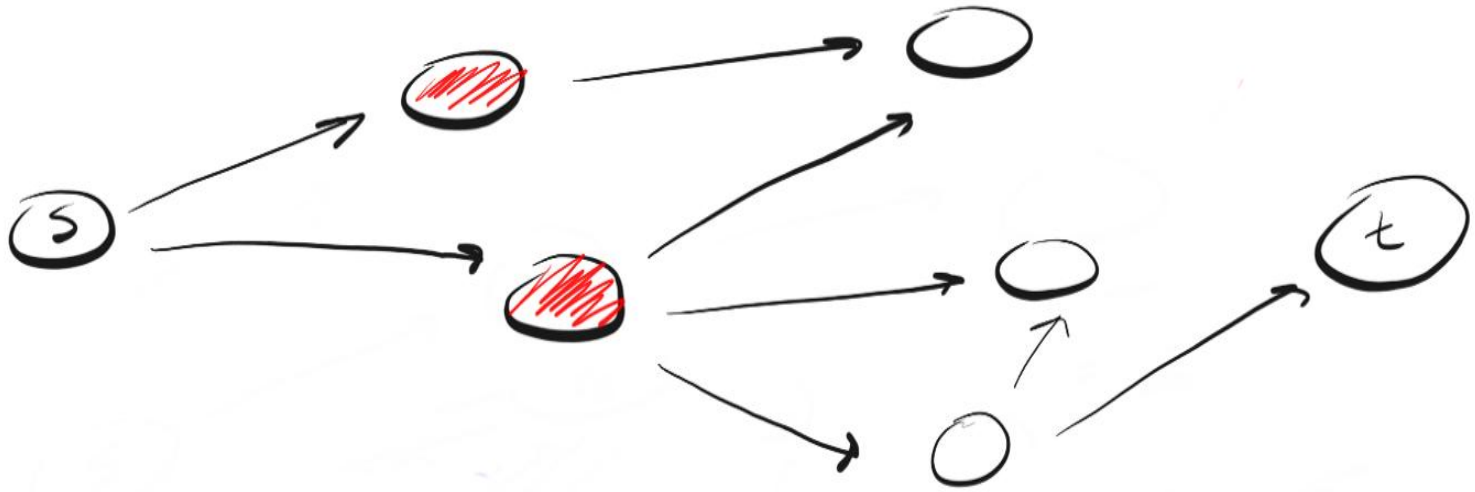
Далі, оскільки ми застосували рекурсію для обходу кожної вершини, цеозначає, що застосовується стек (безкінечна рекурсія приводить до помилки переповнення).

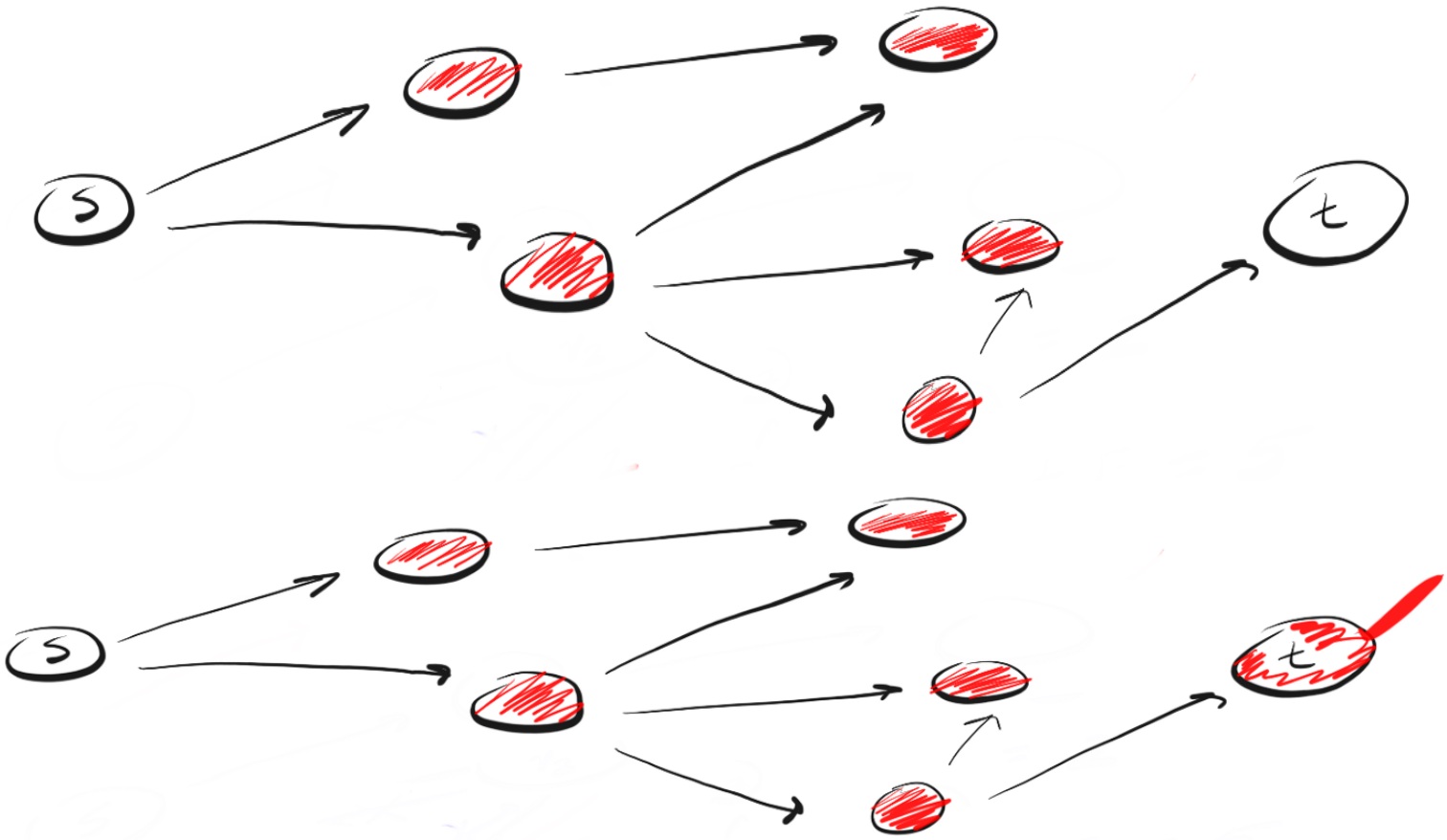
## Пошук в ширину

BFS слідує концепції «розширюйся, піднімаючись на висоту пташиного польоту» («gowide, bird'seye-view»). Замість того, щоб рухатися заданим шляхом до кінцяа, BFS передбачає рух вперед по одному сусіду за раз. Це означає наступне:



Замість слідування шляхом, BFS передбачає відвідування найближчих до s сусідів за один крок, потім відвідування сусідів і так далі до тих пір, поки не буде досягнута t.




s
t
s
t

Чим DFS відрізняється від BFS? Мені здається що, DFS йде напролом, а BFS не поспішає, а досліджує все в межах одного кроку.

Далі виникає питання: як дізнатися, яких сусідів слід відвідати першими?

Для цього ми можемо використати концепцію «першим зайшов, першим вийшов» (first-in-first-out, FIFO) з черги (queue). Ми заносимо в чергу спочатку найближчу до нас вершину, потім її невівіданих сусідів, і продовжуємо цей процес, доки черги не залишиться або доки ми не знайдемо шукану вершину.

Ось код:

```
// за умовою, що мимаємо справу з суміжним списком
// наприклад, таким: adj = {A:[B,C,D], B:[E,F], ... }
functionbfs(adj, s, t) {
	// adj - суміжний список
	// s - початкова вершина
	// t - пункт призначення
	// ініціалізуємо чергу
	letqueue = []
	// додаємо s в чергу
	queue.push(s)
	// помічаємо s як відвідану вершину, щоб не повторити додавання в чергу
	s.visited = true
	while(queue.length> 0) {
		// видаляємо перший (верхній) елемент з черги
		let v = queue.shift()
		// abj[v] - сусіди v
		for(let neighbor of adj[v]) {
			// якщо сусід не відвідувався
			if(!neighbor.visited) {
				// додаємойого вчергу
				queue.push(neighbor)
				// помічаємо вершину як відвідану
				neighbor.visited = true
				// якщо сусід є пунктом призначення, ми перемогли
				if(neighbor === t) return true
			}
		}
	}
	// якщо t не знайдена, значить пункта призначення досягти неможливо
	returnfalse
}
```

**Аналіз BFS**

Може здатися, що BFSпрацює повільніше. Однак,якщо уважно придивитися до візуалізацій, можно побачити, що вонимають однаковий час виконання.

Черга передбачає обробку кожної вершини перед досягненням пункту приазначення. Це означає, що, в гіршому випадку, BFS досліджує всі вершини і грані.

Не дивлячись на те, що BFS може здаватися повільнішим, але він скоріший, оскільки при роботі з великими графами виявляється, що DFS витрачає багато часу на слідування шляхами, які в кінцевому рахунку виявляються хибними. BFS часто застосовують для знаходження найкоротшого шляху між двома вершинами.

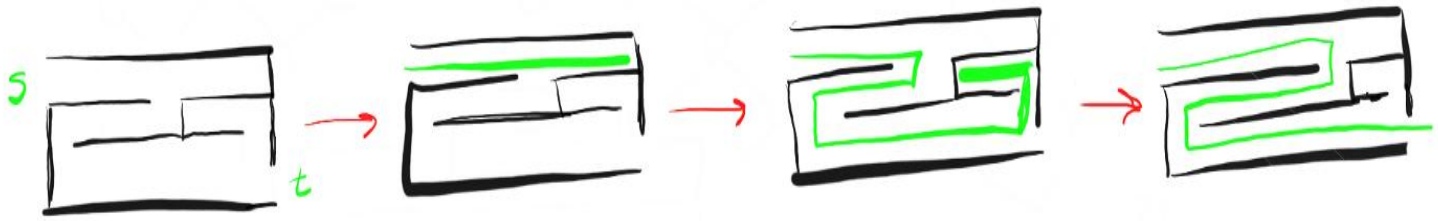
Таким чином, виконання BFS також складає O(V + E), а оскільки ми застосовуємо чергу, що вміщує всі вершини, його просторова складність складає O(V).

**Аналоги з реального життя**

Якщо наводити приклади з реального життя, то ось як я представляю собі роботу DFS і BFS.

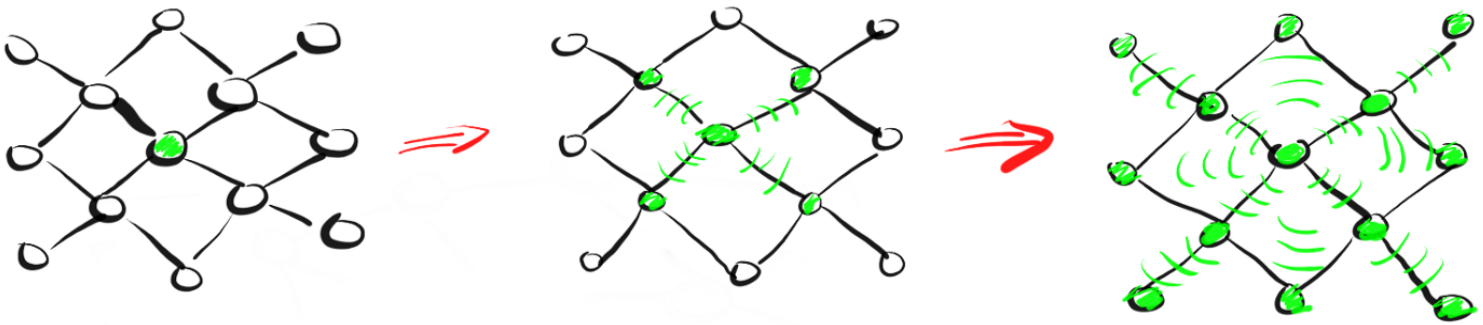
Коли я думаю про DFS, то представляю собі мишу в лабіринті в пошуку їжі. Для того, щоб потрапити до їжі миша змушена багато разів потрапляти в тупик, повертатися і рухатися в іншому напрямку, і так докивона не знайде вихід з лабіринту або їжу.

Спрощена версія виглядає так:




В свою чергу, коли я думаю про BFS, то представляю собі кола на воді. Падіння каменю в воду приводить до утворення кіл по всіх напрямках від центру.

Спрощена версія виглядає так:



**Висновки**

•Пошук в глибину і пошук в ширину використовують для обходу графа.

•DFS рухається по гранях (ребрах) туди і назад, а BFS розповсюджується по сусідах в пошуку призначення.

•DFS використовує стек, а BFS — чергу.

•Час виконання обох складає O(V + E), а просторова складність — O(V).

•Дані алгоритми мають різну філософію, але однаково важливі для роботи з графами.

**Алгоритм пошуку максимальної кількості остовних дерев**

Цей алгоритм застосовується для будь-якого зв'язного графа G(V,E) з поміченими вершинами, в якого необхідно знайти всі можливі остовні дерева.

**Кроки алгоритму пошуку максимальної кількості остовних дерев**

•Знайти степені вершин цього зв'язного графа;

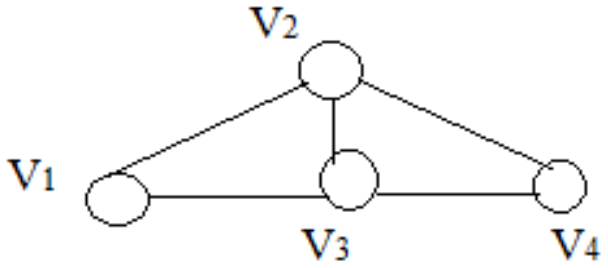
•Побудувати матрицю степенів вершин зв'язного графа;

•Побудувати матрицю суміжності зв'язного графа;

•Знайти різницю між матрицями степенів і суміжності зв'язного графа G(V,E);

•Знайти будь-яке з алгебраїчних доповнень отриманої матриці різниці на кроці 4 алгоритму, значення якого є кількісним значенням остовних дерев для графа G(V,E).

**Приклад.** Використовуючи розглянутий алгоритм, знайти максимальну кількість остовних дерев для зв’язного графа G(V,E) , наведеного на рисунку:




$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \qquad C = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

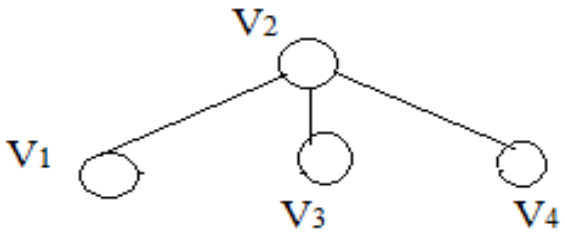
*Розв'язання:* Відповідно до кроку 1 алгоритму, знайдемо ступені вершин графа G(V,E), які дорівнюють: deg(V1) = deg(V4) =2; deg(V2) = deg(V3) . Згідно з кроком 2 алгоритму будуємо степенну вершинну матрицю С і кроком 3 – матрицю суміжності А графа G(V,E). Користуючись кроком 4 алгоритму, знаходимо різницю між двома матрицями С і А, внаслідок чого отримуємо третю матрицю К:

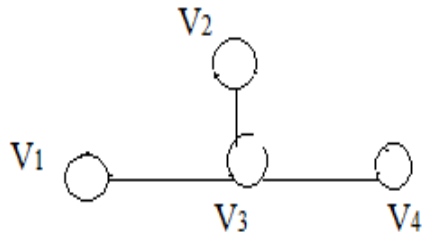
$$K = C - A = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}$$

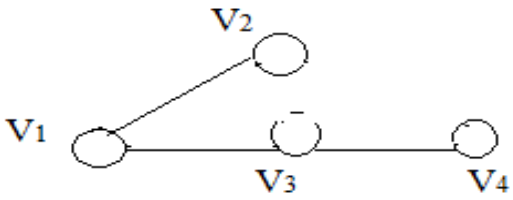
Використовуючи крок 5 алгоритму, знаходимо алгебраїчне доповнення до матриці K, наприклад K11, яке дорівнює

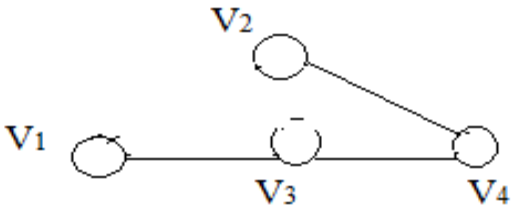
$$K_{11} = \det\begin{pmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 2 \end{pmatrix} = 3 \cdot \det\begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix} - (-1)\begin{pmatrix} -1 & -1 \\ -1 & 2 \end{pmatrix} + (-1)\begin{pmatrix} -1 & 3 \\ -1 & -1 \end{pmatrix} = 3 \cdot 5 - 3 - 4 = 8$$

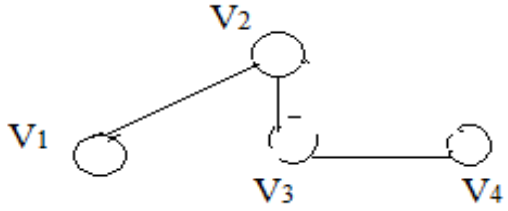
Виходячи із значення цього доповнення, можна сказати, що граф G(V,E), який наведений на рисунку, має вісім остовних дерев:

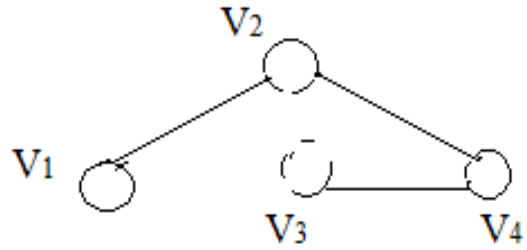


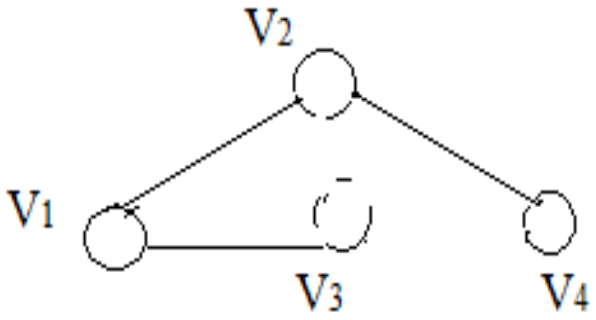

V2
V1
V3
V4


V2
V1
V3
V4


V2
V1
V3
V4


V2
V1
V3
V4


V2
V1
V3
V4


V2
V1
V3
V4