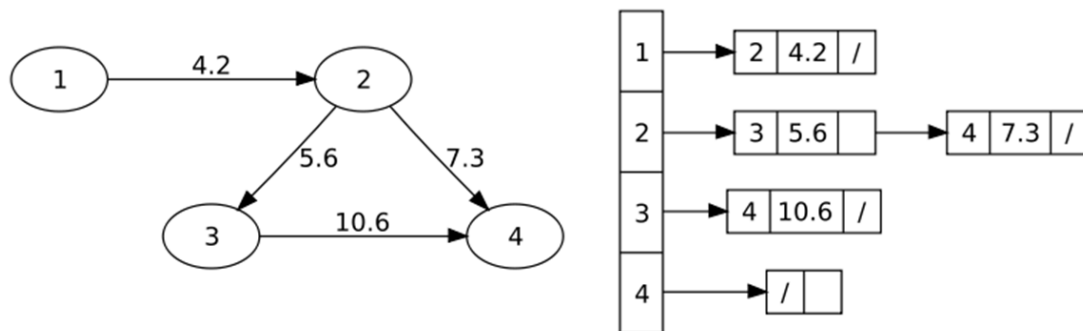


## Algos basiques en python

Pour l'implementation on peut utiliser une **liste d'adjacence**.



**Figure 1:** Exemple de liste d'adjacence

En python nous pouvons utiliser une liste de liste.

```
1 edges = [[(2, 4.2)], [(3, 5.6), (4, 7.3)], [(4, 10.6)], []]
```

### DFS (Depth First Search)

Il faut memoriser les noeuds sur lesquels ont passe.

Pour le graph suivant:

```
1 edges = [[1, 3], [0, 2, 5], [1], [0, 4], [3, 5], [1, 4], [7], [6]]
```

On veut afficher:

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

Voici une implementation recursive.

```
1 def dfs(adj):
2     n = len(adj)
3     seen = [False] * n
```

```
4
5     # Recursive function
6     def rec(start):
7         print(start, end=' ')
8         for y in adj[start]:
9             if not seen[y]:
10                seen[y] = True
11                rec(y)
12
13     # Call rec on all connected component of the graph
14     for start in range(n):
15         if not seen[start]:
16             seen[start] = True
17             rec(start)
```

---

Pour la version itérative avec une pile de (i,j)

```
1  def dfs_iter(adj):
2      n = len(adj)
3      seen = [False] * n
4
5      for start in range(n):
6          if seen[start]:
7              continue
8
9          stack = [(start, 0)]
10         while stack:
11             (src, pos) = stack.pop()
12             # First time on this node
13             if not pos:
14                 print(src, end=' ')
15                 seen[src] = True
16             # Last time on this node
17             if pos == len(adj[src]):
18                 continue
19
20             # Next call
21             stack.append((src, pos + 1))
22
23             # Go to successor
24             succ = adj[src][pos]
```

```

25         if not seen[succ]:
26             stack.append((succ, 0))

```

## BFS (Breadth-First Search)

Pour le graph suivant:

```

1 edges = [[1, 3], [0, 2, 5], [1], [0, 4], [3, 5], [1, 4], [7], [6]]

```

On veut afficher:

```

1 0 1 3 2 5 4 6 7

```

Voici l'implémentation:

```

1 def bfs(adj):
2     n = len(adj)
3     seen = [False] * n
4
5     for start in range(n):
6         if not seen[start]:
7             continue
8         seen[start] = True
9         queue = [start]
10        while queue:
11            src = queue.pop()
12            print(src, end=' ')
13
14            for succ in adj[src]:
15                if not seen[succ]:
16                    seen[succ] = True
17                    queue.insert(0, succ) # Couteux !! Il faut mieux
                                         utiliser un deque

```

La complexité est de  $\Theta(|V| + |E|)$  avec:

- $|V|$ : le nombre de noeud
- $|E|$ : le nombre d'arrete

## Parlons complexité

$$1. |E| \leq \frac{|V| \times (|V|-1)}{2} < \frac{|V|^2}{2}$$

$$|E| = O(|V|^2)$$

2. Sur un graph connexe  $|E| \geq |V| - 1$

$$|E| = \Omega(|V|) \text{ sur graphe connexe}$$

$$3. \sum_{v \in V} \deg(v) = 2|E| = \Theta(|E|)$$

## Distmap

Le but est de trouver la distance des noeuds par rapport a un noeud de référence.

Pour le graph suivant:

```
1 edges = [[1, 3], [0, 2, 5], [1], [0, 4], [3, 5], [1, 4], [7], [6]]
```

On veut avoir:

```
1 [0, 1, 2, 1, 2, 2, None, None]
```

```
1 from collections import deque
2
3 def distmap(adj, start):
4     n = len(adj)
5     dist = [None] * n
6     q = deque([start]) # Utilisation d'un deque :D
7     dist[start] = 0
8     while q:
9         src = q.popleft()
10        d = dist[src]
11        for dst in adj[src]:
12            if dist[dst] is None:
13                dist[dst] = d+1
14                q.append(dst)
15    return dist
```

## Dijkstra

Changons de graph (pondéré cette fois):

```
1 edges = [[(1, 8), (2, 2)], [(0, 8), (3, 2), (4, 1)], [(0, 2), (3, 2)],
2 [(1, 2), (2, 2), (4, 7)], [(1, 1), (3, 7)]]
```

This code is not real python because of the heap.

```
1 def djikstra(graph, start):
2     dist = [float('inf')] * len(graph)
3     dist[start] = 0
4     h = heapify([start]) # Min Heap sort with `dist`
5     while h:
6         src = h.pop()
7         for (dst, w) in graph[src]:
8             d = dist[src] + w
9             old = dist[dst]
10            dist[dst] = min(old, d)
11            if old == float('inf'):
12                h.push(dst)
13            else:
14                h.update(dst)
15     return dist
```

Complexité  $O((|E| + |V|)\log|V|)$

Sur *wikipedia* on trouve la complexité suivante:

$O(|E| + |V|\log|V|)$  Qui est mieux ! Pour ca il faut utiliser une structure de donnée qui **insert** et **update** en temps constant. C'est un **tas de fibonacci**.