Algos basiques en python

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

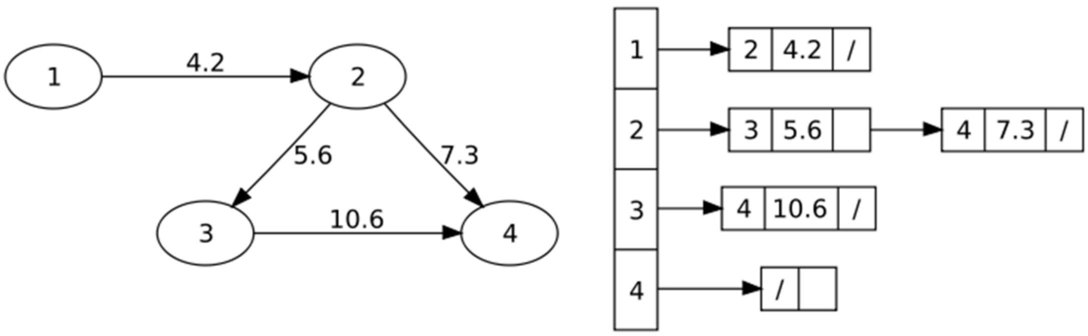

Figure 1: Exemple de liste d'adjacence

En python nous pouvons utiliser une liste de liste.

```
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:

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

On veut afficher:

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

Voici une implementation recursive.

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

```

    # Recursive function
    def rec(start):
        print(start, end=' ')
        for y in adj[start]:
            if not seen[y]:
                seen[y] = True
                rec(y)

    # Call rec on all connected component of the graph
    for start in range(n):
        if not seen[start]:
            seen[start] = True
            rec(start)
```

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

```
def dfs_iter(adj):
    n = len(adj)
    seen = [False] * n

    for start in range(n):
        if seen[start]:
            continue

        stack = [(start, 0)]
        while stack:
            (src, pos) = stack.pop()
            # First time on this node
            if not pos:
                print(src, end=' ')
                seen[src] = True
            # Last time on this node
            if pos == len(adj[src]):
                continue

            # Next call
            stack.append((src, pos + 1))

            # Go to successor
            succ = adj[src][pos]
```

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

BFS (Breadth-First Search)

Pour le graph suivant:

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

On veut afficher:

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

Voici l'implementation:

```
def bfs(adj):
    n = len(adj)
    seen = [False] * n

    for start in range(n):
        if not seen[start]:
            continue
        seen[start] = True
        queue = [start]
        while queue:
            src = queue.pop()
            print(src, end=' ')

            for succ in adj[src]:
                if not seen[succ]:
                    seen[succ] = True
                    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. $|E| \leq \frac{|V| \times (|V| - 1)}{2} < \frac{|V|^2}{2}$

$\boxed{|E| = O(|V|^2)}$

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

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

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

Distmap

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

Pour le graph suivant:

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

On veut avoir:

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

```
from collections import deque

def distmap(adj, start):
    n = len(adj)
    dist = [None] * n
    q = deque([start]) # Utilisation d'un deque :D
    dist[start] = 0
    while q:
        src = q.popleft()
        d = dist[src]
        for dst in adj[src]:
            if dist[dst] is None:
                dist[dst] = d+1
                q.append(dst)
    return dist
```

Dijkstra

Changons de graph (pondéré cette fois):

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

This code is not real python because of the heap.

```
def djikstra(graph, start):
    dist = [float('inf')] * len(graph)
    dist[start] = 0
    h = heapify([start]) # Min Heap sort with  dist
    while h:
        src = h.pop()
        for (dst, w) in graph[src]:
            d = dist[src] + w
            old = dist[dst]
            dist[dst] = min(old, d)
            if old == float('inf'):
                h.push(dst)
            else:
                h.update(dst)
    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**.