Graphes

Algorithme des graphes: le parcours en largeur

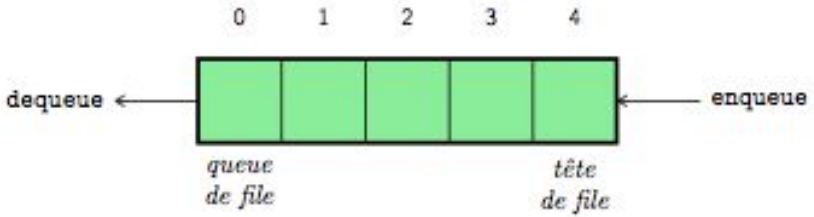
Anicet E. T. Ebou, ediman.ebou@inphb.ci

01

File, Pile et Deque

File

Une file (*queue* en anglais) est une séquence ordonnée d'éléments, c'est-à-dire une liste, tel que l'ajout d'un nouvel élément (*enqueue*) se fait à sa fin et le retrait d'un élément (*dequeue*) à sa tête. Une file est une liste de type FIFO (First-in First-out), c'est-à-dire que le premier élément qui y a été ajouté sera aussi le premier qui en sortira.



File

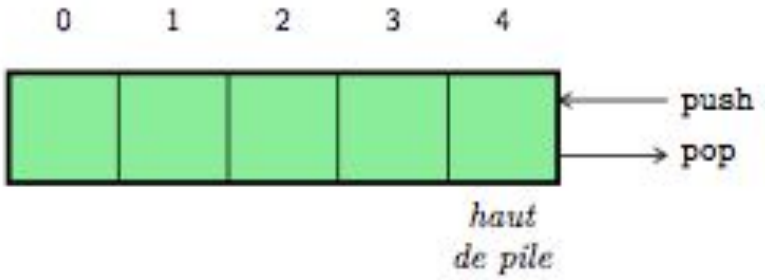
En Python, on utilise la fonction `append` appliquée sur la liste pour ajouter un élément à sa fin et on utilise la fonction `pop` pour retirer un élément à son début. Voici un exemple qui crée une file vide, lui ajoute deux éléments, puis en retire un:

```
queue = []              # la file est vide
queue.append(1)         # la file contient [1]
queue.append(2)         # la file contient [1, 2]
queue.append(3)         # la file contient [1, 2, 3]
result = queue.pop(0)   # la file contient [2, 3]

print(result)
print(queue)
```

Pile

Une pile (*stack* en anglais) est également une séquence ordonnée d'éléments, mais tel que l'ajout d'un nouvel élément (*push*) et le retrait d'un élément (*pop*) se fait toujours du même côté, en haut de la pile. Une pile est une liste de type LIFO (Last-in First-out), c'est-à-dire que l'élément qui en sortira sera toujours celui qui y est entré en dernier.



Pile

En Python, il va falloir utiliser la fonction `append`, à appliquer sur la liste, pour lui ajouter un élément à sa fin et la fonction `pop` pour en retirer un élément à la fin.

```
stack = []              # la pile est vide
stack.append(1)         # la pile contient [1]
stack.append(2)         # la pile contient [1, 2]
stack.append(3)         # la pile contient [1, 2, 3]
result = stack.pop()    # la pile contient [1, 2]

print(result)
print(stack)
```

Deque

Les *deques* sont une généralisation des piles et des files d'attente (le nom se prononce "deck" et est l'abréviation de "*double-ended queue*"). Les deques supportent les ajouts et retraits sécurisés et efficaces en mémoire de chaque côté de la deque avec approximativement la même performance $0(1)$ dans les deux sens.

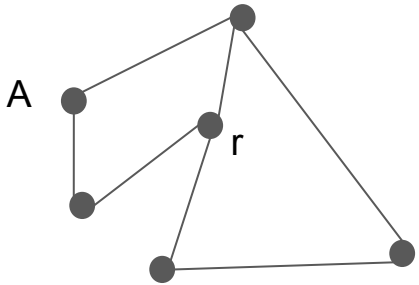
Nous utiliserons les deques pour l'implémentation de l'algorithme du parcours en largeur.

02

Parcours d’un graphe en largeur

Plus court chemin

Donnons nous pour but de trouver le plus court chemin entre le sommet r et le sommet A.

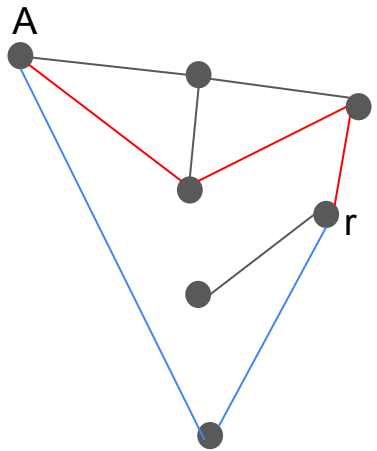


Plus court chemin

Donnons nous pour but de trouver le plus court chemin entre le sommet r et le sommet A.

Il existe plusieurs chemin entre r et A tel que le chemin 1 (de longueur 2) et le chemin 2 (de longueur 3).

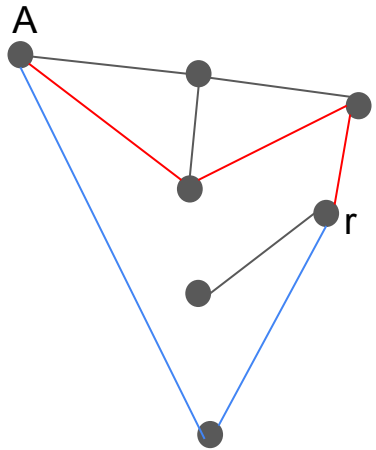
Remarquons qu'il peut exister plusieurs plus courts chemins entre r et A.



Plus court chemin

Comment trouver des plus courts chemins?

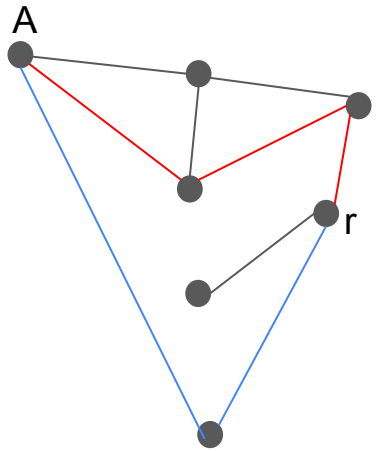
Une possibilité est l'**algorithme du parcours en largeur**.



Parcours en largeur

Le parcours en largeur est aussi appelé *BFS* pour *Breadth-First Search* en anglais.

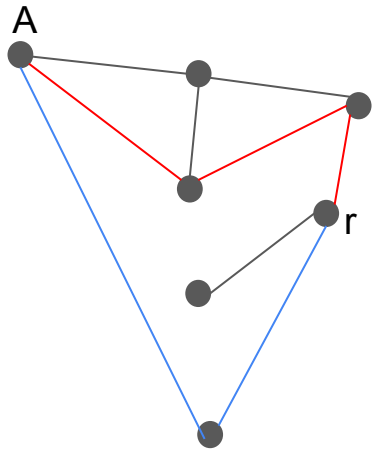
Principe: on explore le graphe à partir d'un sommet en visitant d'abord tous les sommets voisins (à une distance 1), puis tous les sommets voisins de ses voisins (à une distance 2), etc.



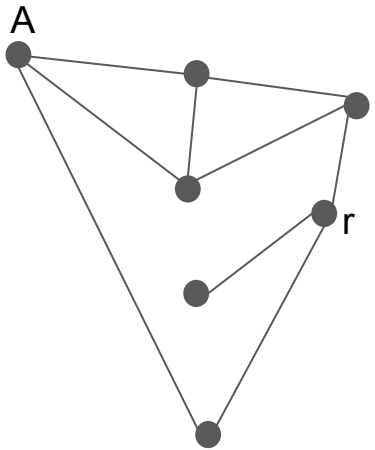
Parcours en largeur

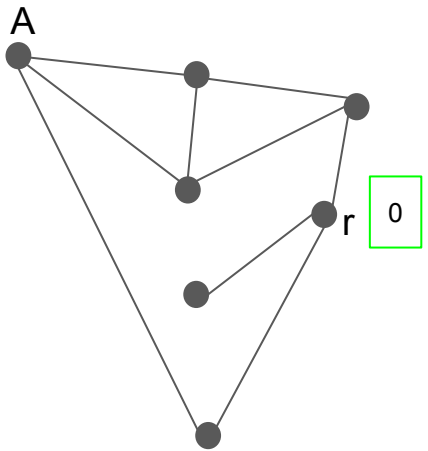
Principe:

- On part d'un sommet;
- On visite ses voisins directs et on les enfile s'ils ne sont pas déjà présents dans la file;
- On défile (c'est-à-dire qu'on enlève la tête de la file);
- On recommence à partir du point 2 tant que la file n'est pas vide.



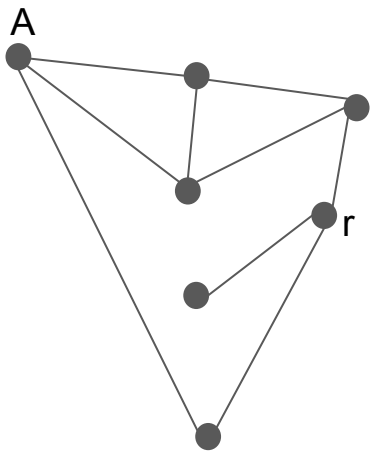
Parcours en largeur

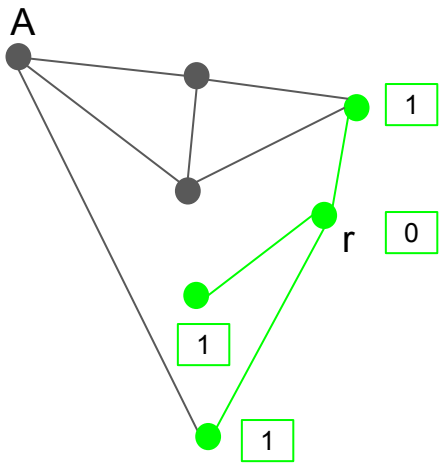




r est à distance 0 de lui-même

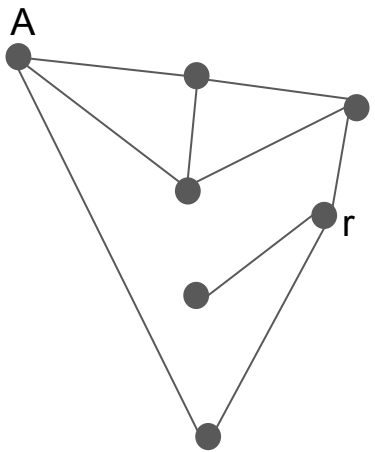
Parcours en largeur

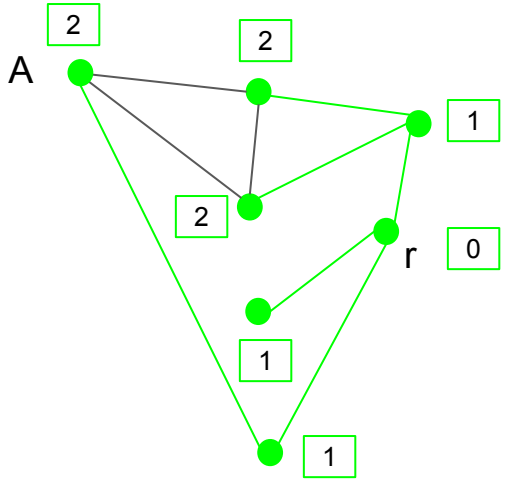




Les 3 voisins de r sont à distance 1 de r et on mémorise les arêtes et les sommets atteints qui constituent le plus court chemin entre r et ses arêtes en les colorants.

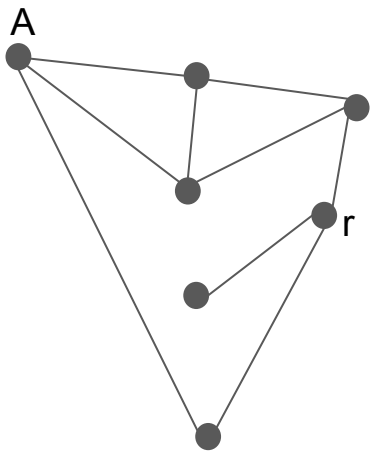
Parcours en largeur

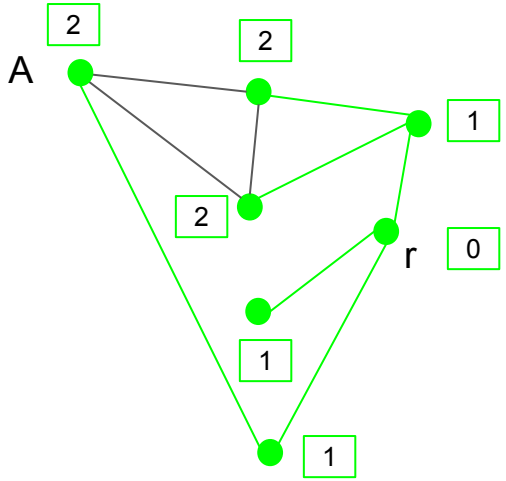




On explore ensuite les voisins des 3 sommets précédents. On mémorise les sommets et arêtes atteints comme précédemment.

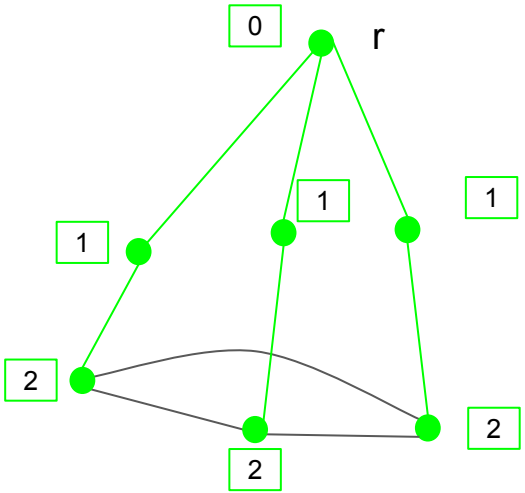
Parcours en largeur

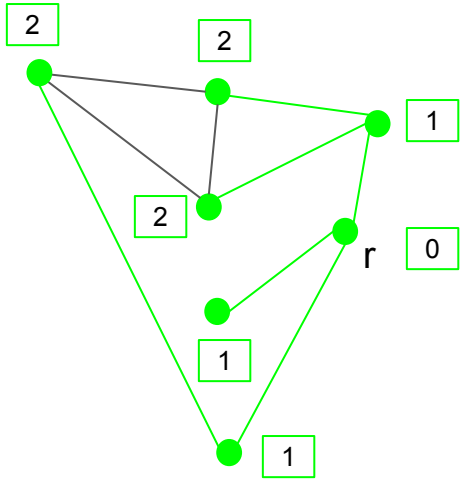




On continue l'exploration en explorant à partir des sommets de la couche 2. On remarque qu'on ne découvre aucun sommet qui n'avait pas encore été exploré. On est donc à la fin de l'algorithme puisque tous les sommets ont été visités.

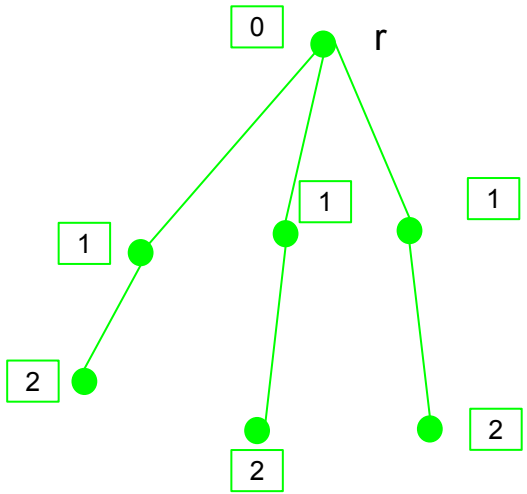
Parcours en largeur

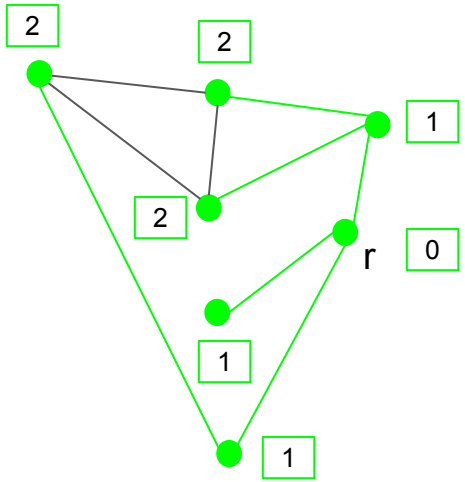




Représentons ce résultat sous une autre forme (figure à gauche) par les distances par rapport à r.

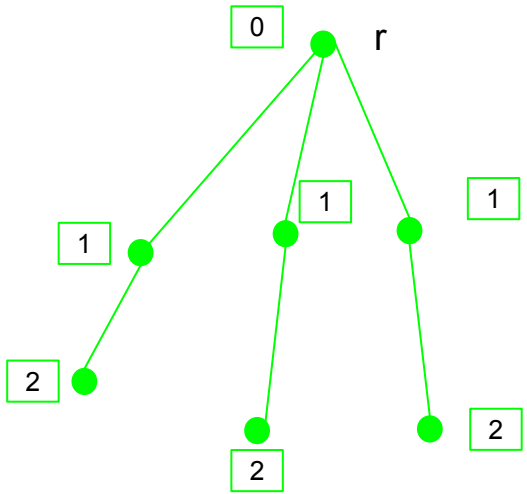
Parcours en largeur





Si on oublie les arêtes qui ne sont pas vertes, on obtient **un arbre couvrant du graphe**.

Parcours en largeur

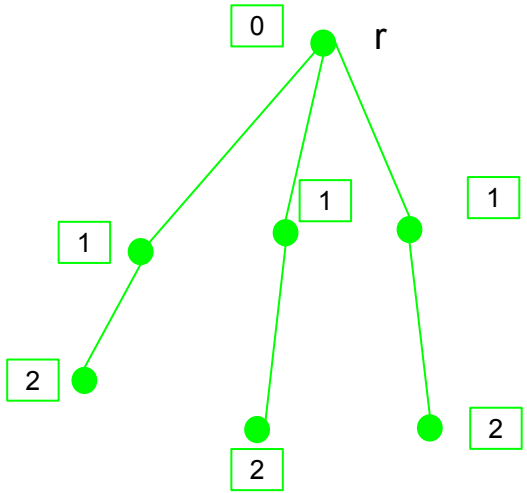


Distance entre u et v: longueur du plus court chemin entre u et v.

Le parcours en largeurs:

- Permet de construire un arbre couvrant (si le graphe est connexe).
- C’est un arbre de plus court chemin à partir du point de départ du parcours en largeur.

Parcours en largeur



Le parcours en largeur permet aussi de savoir si un graphe est connexe ou pas. En effet, en appliquant le parcours en largeur, si tous les sommets sont contenus dans l'arbre couvrant, alors le graphe est connexe.

```
from collections import deque
def bfs(graph, start, target):
       visited = set()
       queue = deque([(start, None)])  # Stock la paire (node, parent)
       parent_map = {}                 # Stock les informations sur le parent
       while queue:
           current, parent = queue.popleft()
           visited.add(current)
           if current == target:
               path = [current]        # Reconstruit le chemin de ‘start’ a ‘target’
               while parent is not None:
                   path.append(parent)
                   parent = parent_map.get(parent)  # Extrait le ‘parent’ a partir ‘parent_map’
               path.reverse()
               return path
           for neighbor in graph[current]:
               if neighbor not in visited:
                   queue.append((neighbor, current))
                   parent_map[neighbor] = current   # Stock l’information parent
                   visited.add(neighbor)
       return None  # Le noeud recherche n’a pas ete trouve
```

```
graph = {
      'A': ['B', 'C'],
      'B': ['A', 'D', 'E'],
      'C': ['A', 'F'],
      'D': ['B'],
      'E': ['B', 'F'],
      'F': ['C', 'E']
}

start_node = 'A'
target_node = 'F'

shortest_path = bfs(graph, start_node, target_node)

if shortest_path:
      print(f"Le chemin le plus court entre {start_node} et {target_node}: {' -> '.join(shortest_path)}")
else:
      print(f"Aucun chemin trouve entre {start_node} et {target_node}")
```